PAPER Special Section on Concurrent Systems Technology

Computing the Stabilization Times of Self-Stabilizing Systems

Tatsuhiro TSUCHIYA[†], ***Regular Member***, **Yusuke TOKUDA**[†], ***Nonmember***, ***and*** **Tohru KIKUNO**[†], ***Regular Member***

SUMMARY A distributed system is said to be self-stabilizing if it converges to some legitimate state from an arbitrary state in a finite number of steps. The number of steps required for convergence is usually referred to as the *stabilization time*, and its reduction is one of the main performance issues in the design of self-stabilizing systems. In this paper, we propose an automated method for computing the stabilization time. The method uses Boolean functions to represent the state space in order to assuage the state explosion problem, and computes the stabilization time by manipulating the Boolean functions. To demonstrate the usefulness of the method, we apply it to the analysis of existing self-stabilizing algorithms. The results show that the method can perform stabilization time analysis very fast, even when an underlying state space is very huge.

key words: *self-stabilization, stabilization times, automated analysis, symbolic representation, distributed algorithms*

1. Introduction

A distributed system is said to be *self-stabilizing* if it converges to some legitimate state from an arbitrary state. Self-stabilizing systems are thus inherently tolerant to transient faults that may change their state arbitrarily. The notion of self-stabilization was first introduced to computer science by Dijkstra [5]. This notion, which originally had a very narrow scope of application, has attracted much research interest in recent years (e.g., [13]). Practical applications include, for example, Internet servers [15] and FDDI media access control protocols [4].

The number of steps required for reaching a legitimate state from an illegitimate state is usually called the *stabilization time*. The reduction of the stabilization time is one of the main performance issues in the design of self-stabilizing algorithms, which specify self-stabilizing systems, for several reasons. First, since being in illegitimate states means malfunction, it is naturally desirable to reduce the time interval in which the system remains in such states. Moreover, for algorithms for some kinds of problems, such as ring orientation, leader election, and synchronization, the stabilization time is the time complexity itself. It is therefore desirable to design algorithms that are not only self-stabilizing but also have the ability to reach a legitimate state very fast. Research in this direction includes [1], [8], [16].

In this paper, we propose an automated method for computing the worst stabilization time, aiming at supporting algorithm designers. In [6], Dolev et al. proposed a theoretical method, called the *scheduler luck game*, for stabilization time analysis. This method can deal with randomized algorithms only, and it is not automated. Some work addresses the issues of automatic verification of self-stabilizing systems (e.g., [11], [12], [14]). To the best of our knowledge, however, there has been no previous research that deals with stabilization times.

In general, the major problem with analysis of concurrent systems is that the state space becomes very huge since it grows exponentially in the number of components. This problem, usually referred to as the *state explosion* problem, is serious especially in modeling self-stabilizing systems. Since any state can be the initial state in a self-stabilizing system, the set of the reachable states is exactly the same as the Cartesian product of sets of states of all components.

The use of *partial order* techniques is one of the most promising approaches to this problem (e.g., [7], [17]). This approach is based on the observation that the validity of a given correctness property is often insensitive to the order in which independently executed events are interleaved. These techniques generate a reduced set of reachable states that is indistinguishable for the given property, instead of generating the whole reachable state space. Although this approach has proven to be successful in verifying concurrent systems in a large class (e.g., [9]), it is not useful in analyzing self-stabilizing algorithms which can start their execution from any state.

In this paper we adopt *symbolic representation* of the state space, which is another approach to the state explosion problem. In the approach, 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. This approach has already proven to be effective in *model checking* [3], [10], and model checking that uses this approach is called *symbolic model check-*

Manuscript received April 6, 2000.

[†]The authors are with the Department of Informatics and Mathematical Science, Osaka University, Toyonaka-shi, 560-8531 Japan.

ing. In the proposed method, the stabilization time is computed by manipulating the OBDDs. As will be described later, this can be done by using *preimage computation*, which is also a technique used in symbolic model checking.

In spite of the similarities between the proposed analysis method and symbolic model checking, symbolic model checking algorithms cannot be used for the stabilization time analysis due to the following reason. In general, model checking algorithms are used to determine whether or not a given property holds in a system in question. In contrast, the stabilization time is a quantitative attribute of the system, and it cannot be represented as a property that can be verified by model checking. Therefore we need a different method that is tailored to this analysis.

The remainder of this paper is organized as follows. In the next section, we describe the system model and the concept of self-stabilizing algorithms. In Sect. 3, we present the proposed method. In Sect. 4, we explain a prototype system that implements the proposed method, and show the results of applying the method to several algorithms. We conclude this paper with a brief summary in Sect. 5.

2. Self-Stabilizing Algorithms

2.1 Models and Definitions

We consider a distributed system that consists of m processes, $p_0, p_1, p_2, \cdots, p_{m-1}$. The topology of the system is modeled by an undirected graph each of whose vertices 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. Two commonly used communication models in the self-stabilization literature are the *state-reading model* and the *link-register model*. The former model assumes that each process can directly read the internal state of its adjacent processes, while the latter assumes that processes can communicate with other processes only by using separate registers. In this paper we assume the state-reading model for brevity[†].

A distributed algorithm specifies a transition relation for each process p_i. In each step of execution of process p_i, p_i reads the states of its adjacent processes, calculates the next local state based on the transition relation. A distributed algorithm thus specifies the behavior of the system.

To describe distributed algorithms, we use a guarded command language in this paper, since it is commonly used. A process has a set of finite domain variables, and the local state of the process is the vector of the current values of the variables. Hence, the set of local states is the Cartesian product of domains of all the variables. The transition is specified by a list of actions, and a list has the form

$$\text{begin} < \text{action} > [] \cdots [] < \text{action} > \text{end}.$$

The symbol "[]" is a separator that separates the different actions. Each action is of the form

$$< \text{guard} > \longrightarrow < \text{statement} >$$

where the guard is a Boolean expression over the variables of the process and the adjacent processes, and the statement updates at least one variable of the process.

The *global state* of the system is the vector of the states of all processes. Therefore the set of all global states, denoted by $\mathcal{G}$, is given as

$$\mathcal{G} = Q_0 \times Q_1 \times \cdots \times Q_{m-1}$$

where Q_i $(0 \leq i \leq m-1)$ denotes the set of states of p_i.

An action is said to be *enabled* at a global state iff its guard holds at that state. A process is *enabled* iff some action in the process is enabled.

We assume the following semantics. In each step, an arbitrary set of enabled processes is selected to change their state. Each of the selected processes executes an enabled action. If there is more than one enabled action, exactly one action is selected nondeterministically.

We denote by $g \xrightarrow{U} g'$ $(g' \neq g)$ the fact that processes in $U(\subseteq \{p_0, p_1, \cdots, p_{m-1}\})$ are all enabled at $g \in \mathcal{G}$ and their parallel execution yields $g' \in \mathcal{G}$. We say that $g \rightarrow g'$ holds if there exists $U(\subseteq \{p_0, p_1, \cdots, p_{m-1}\})$ such that $g \xrightarrow{U} g'$. A sequence of global states $g_0 g_1 g_2 \cdots$ is a *computation* iff for every i (≥ 0) $g_i \rightarrow g_{i+1}$ holds. A maximal computation is one which is either infinite or ends with a state g where no process is enabled.

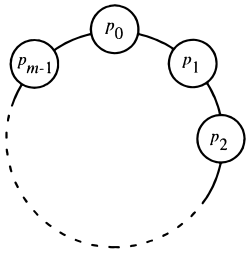
Self-stabilization is defined as follows. Let P be a predicate that identifies the correct execution of the system. We assume that P is given in the form of a Boolean expression over the variables of the processes in the system. We say global states satisfying (not satisfying) P to be legitimate (illegitimate) states, respectively. Let $\mathcal{L}$ denote the set of the legitimate states. 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 maximal computation $g_0 g_1 g_2 \cdots$ starting with g_0, there is an integer k (≥ 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 to be self-stabilizing in a corresponding manner. Thus, a self-stabilizing algorithm specifies a self-stabilizing system.

[†]The results shown here, however, can be easily extended to the the link-register model, by considering a global state to be the vector of the states of processes and shared registers.



(a)

p_0	p_1	p_2	enabled processes	selected processes
2	1	2	p_0, p_1, p_2	p_0
0	1	2	p_1, p_2	p_1, p_2
0	0	1	p_2	p_2
0	0	0	p_0	p_0
1	0	0	p_1	p_1
1	1	0	p_2	p_2
1	1	1	p_0	p_0
2	1	1	p_1	—

(b)

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

The stabilization time, r, of a self-stabilizing system is the number of steps required for reaching a legitimate state in the worst case, i.e.,

$$r = \max_{g_0 g_1 g_2 \cdots \in \mathcal{M}} \left\{ \min_{i \geq 0} \{ i : g_i \in \mathcal{L} \} \right\}$$

where $\mathcal{M}$ is the set of all maximal computations.

2.2 Illustrative Example

Here we take Dijkstra's K-state mutual exclusion algorithm as an illustrative example [5]. Consider a distributed system consisting of m processes connected in the form of a ring, as shown in Fig. 1(a).

We then define the legitimate states as those in which exactly one process is enabled. This corresponds to a form of mutual exclusion, because the enabled 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 m. Process p_0 is treated differently from all other processes. This algorithm is described below.

Process p_0
Variables $S_0 \in \{0, 1, \cdots, K-1\}$
begin
$S_{m-1} = S_0 \longrightarrow S_0 := (S_0 + 1) \bmod K$
end

Process $p_i (i = 1, 2, \cdots, m-1)$
Variables $S_i \in \{0, 1, \cdots, K-1\}$
begin
$S_{i-1} \neq S_i \longrightarrow S_i := S_{i-1}$
end

The predicate P that represents the legitimate states is

$$\begin{aligned} P \;=\; & \Big((S_{m-1} = S_0) \wedge \bigwedge_{1 \leq j \leq m-1} (S_{j-1} = S_j) \Big) \\ & \vee \bigvee_{i=1}^{m-1} \Big((S_{i-1} \neq S_i) \wedge (S_{m-1} \neq S_0) \\ & \wedge \bigwedge_{\substack{1 \leq j \leq m-1 \\ j \neq i}} (S_{j-1} = S_j) \Big). \end{aligned}$$

Figure 1(b) shows a part of a computation of the system with three processes and $K = 3$. Although every process is enabled initially, after two steps the system reaches a state where only one process is enabled.

3. Proposed Method

3.1 Symbolic Representation

The major problem of automated analysis of concurrent systems is that the state spaces arising from practical problems are often huge, generally making exhaustive exploration infeasible. To cope with the difficulty, we symbolically represent the state space by using Boolean functions. Since Boolean functions can be often represented by Ordered Binary Decision Diagrams (OBDDs) very compactly, this approach can reduce the memory and time required for analysis.

Let $\mathcal{S}$ be the set of Boolean vectors $\{0, 1\}^n$. Then any subset $\mathcal{A}$ of $\mathcal{S}$ can be uniquely represented by a Boolean function $f(\boldsymbol{x})$ with n Boolean variables ($\boldsymbol{x} = (x_1, x_2, \cdots, x_n)$), such that

$$f(\boldsymbol{a}) = \begin{cases} 1 & \boldsymbol{a} \in \mathcal{A} \\ 0 & \boldsymbol{a} \notin \mathcal{A}. \end{cases}$$

We call f the *characteristic function* of $\mathcal{A}$. Since $\mathcal{S} = \{0, 1\}^n$ has 2^n elements, up to 2^n states can thus be handled by using n Boolean variables. The transition relation is also represented by a Boolean function $R(\boldsymbol{x}, \boldsymbol{x}')$ with $2n$ variables ($\boldsymbol{x} = (x_1, x_2, \cdots, x_n)$, $\boldsymbol{x}' = (x'_1, x'_2, \cdots, x'_n)$), such that

$$R(\boldsymbol{a}, \boldsymbol{b}) = \begin{cases} 1 \text{ there is a transition from } \boldsymbol{a} \text{ to } \boldsymbol{b}. \\ 0 \text{ otherwise.} \end{cases}$$

($\boldsymbol{a}, \boldsymbol{b} \in \mathcal{S}$). We call Boolean function R the *transition relation function*. In the rest of the paper, we use $\boldsymbol{x} (= (x_1, x_2, \cdots, x_n))$ and $\boldsymbol{x}' (= (x'_1, x'_2, \cdots, x'_n))$ only to specify variables in Boolean functions. In other words, these symbols never signify values assigned to these variables.

Using the above idea, we represent the state space using Boolean functions as follows. We encode each variable in a distributed algorithm by using $\lceil \log_2 d \rceil$

Boolean variables, where d is the number of elements in the domain of the variable. For example, global states of the K-state algorithm are represented by using $n = \lceil \log_2 K \rceil * m$ Boolean variables. In the sequel, we identify a global state in $\mathcal{G}$ with its corresponding vector in $\mathcal{S}$. Then $\mathcal{G} \subseteq \mathcal{S}$ holds.

The transition relation function R for a given algorithm is the one such that $R(\boldsymbol{a}, \boldsymbol{b}) = 1$ iff $\boldsymbol{a}, \boldsymbol{b} \in \mathcal{G}$ and $\boldsymbol{a} \rightarrow \boldsymbol{b}$ holds. In addition to R, we need to obtain characteristic functions for $\mathcal{G}$ and $\mathcal{L}$. Let G and L denote these functions.

These necessary Boolean functions can be obtained directly from the given algorithm. For example, consider the K-state algorithm. When $K = n = 3$, $G(\boldsymbol{x}) = \neg((x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee (x_5 \wedge x_6))$ if $(x_{2i-1}, x_{2i}) = (0,0), (x_{2i-1}, x_{2i}) = (0,1), (x_{2i-1}, x_{2i}) = (1,0)$ specify $S_i = 0, S_i = 1, S_i = 2$, respectively $(i = 1, 2, 3)$. R and L can also be obtained without analyzing the state space of the algorithm, because the guards for actions and the predicate for legitimate states are given in the form of Boolean expressions over variables in the algorithm.

3.2 Computing Stabilization Time

Given the transition relation function R, the characteristic function G of $\mathcal{G}$, and the characteristic function L of the set of all legitimate states $\mathcal{L}$, the proposed method computes the maximum number of steps needed for the system to reach some state in $\mathcal{L}$ from an arbitrary state. The number is equal to the stabilization time, if the system is self-stabilizing. We assume that there is no state $\boldsymbol{a} \in \mathcal{G}$ such that $\boldsymbol{a} \notin \mathcal{L}$ and $\neg\exists \boldsymbol{b}, \boldsymbol{a} \rightarrow \boldsymbol{b}$, since it is easy to check whether such states exist or not[†]. Note that if such states exist, then the system is not self-stabilizing since these states never reach a legitimate state.

In the method, the state space is traversed backward from $\mathcal{L}$, by using (implicit) breadth-first search. The breadth-first search is performed by iterating *preimage computation*, which is an important technique used in symbolic model checking.

Given a characteristic function C and a transition relation function R, the *preimage* of C according to R is defined as

$$\mathcal{P}(R, C) = \{\boldsymbol{a} \in \mathcal{S} : \exists \boldsymbol{b}, R(\boldsymbol{a}, \boldsymbol{b}) = 1 \wedge C(\boldsymbol{b}) = 1\}.$$

In other words, $\mathcal{P}(R, C)$ is the set of states that can reach $\mathcal{C}$ in one step where $\mathcal{C}$ is the set of states represented by C; i.e.,

$$\mathcal{P}(R, C) = \{\boldsymbol{a} \in \mathcal{S} : \exists \boldsymbol{b}, \boldsymbol{a} \rightarrow \boldsymbol{b} \wedge \boldsymbol{b} \in \mathcal{C}\}.$$

The characteristic function of $\mathcal{P}(R, C)$, denoted by P, can be obtained as follows. First let

$$\exists x_i.f = f_{x_i=1} \vee f_{x_i=0}$$
$$\exists(x_{i1}, x_{i2}, \cdots, x_{il}).f = \exists x_{i1}.\exists x_{i2}. \cdots \exists x_{il}.f$$

where $f_{x_i=b} (b \in \{0,1\})$ is obtained by substituting b for x_i, and let $\{1, 2, \cdots, n\} - \{i_1, i_2, \cdots, i_l\} = \{j_1, j_2, \cdots, j_{n-l}\}$. Then $\exists(x_{i_1}, x_{i_2}, \cdots, x_{i_l}).f$ is evaluated to true for input $(a_{j_1}, a_{j_2}, \cdots, a_{j_{n-l}})$ iff f has at least one truth assignment $(b_1, b_2, \cdots, b_n)$ such that $(a_{j_1}, a_{j_2}, \cdots, a_{j_{n-l}}) = (b_{j_1}, b_{j_2}, \cdots, b_{j_{n-l}})$. Thus

$$P(\boldsymbol{x}) = \exists \boldsymbol{x'}.(R(\boldsymbol{x}, \boldsymbol{x'}) \wedge C(\boldsymbol{x'})).$$

Preimage computation means to compute this formula. This can be done very efficiently by using an algorithm proposed by Bryant [2].

Suppose that $\mathcal{L} \subseteq \mathcal{C} \subseteq \mathcal{G}$ holds. Using the above technique, the set of states that either necessarily stay in $\mathcal{C}$ or necessarily move to a state in $\mathcal{C}$ in exactly one step in any computation, i.e.,

$$\{\boldsymbol{a} \in \mathcal{S} : (\boldsymbol{a} \in \mathcal{C} \wedge \neg\exists \boldsymbol{b}, \boldsymbol{a} \rightarrow \boldsymbol{b}) \vee ((\exists \boldsymbol{b}, \boldsymbol{a} \rightarrow \boldsymbol{b}) \wedge (\boldsymbol{a} \rightarrow \boldsymbol{b} \Rightarrow \boldsymbol{b} \in \mathcal{C}))\}$$

can be computed. Specifically, this set of states is represented by

$$(\mathcal{S} - \mathcal{P}(R, \neg C)) \cap \mathcal{G},$$

since $\mathcal{P}(R, \neg C)$ represents a set of states that can move to some state in $\mathcal{S} - \mathcal{C}$ in one step, and by assumption any state in $\mathcal{G} - \mathcal{C}$ can reach some other states. Therefore its characteristic function is computed as

$$(\neg\exists \boldsymbol{x'}.(R(\boldsymbol{x}, \boldsymbol{x'}) \wedge \neg C(\boldsymbol{x'}))) \wedge G(\boldsymbol{x}).$$

The set of states that necessarily reach a state in $\mathcal{C}$ in at most one step in any computation is the union of $\mathcal{C}$ and the set of states any of whose next states is in $\mathcal{C}$; i.e.,

$$\mathcal{C} \cup ((\mathcal{S} - \mathcal{P}(R, \neg C)) \cap \mathcal{G}).$$

It is then clear that this set can be represented by characteristic function

$$C(\boldsymbol{x}) \vee ((\neg\exists \boldsymbol{x'}.(R(\boldsymbol{x}, \boldsymbol{x'}) \wedge \neg C(\boldsymbol{x'}))) \wedge G(\boldsymbol{x})).$$

As can be seen, the above formula contains the OR, AND, and NOT operators. These operations can also be efficiently performed by using algorithms proposed in [2].

Based on the above techniques, the method works in the following four steps. Here C is the characteristic function for the set of states visited, and c is the number of iterations of preimage computation.

Step 1. $C := L$. $c := 0$.
Step 2. Check whether the whole state space is explored; i.e., check whether $C = G$ or not. If $C = G$, then c, which is the number of iterations of preimage computation, is the stabilization time r. Then return c and stop; otherwise go to Step 3.

[†] $(\neg\exists \boldsymbol{x'}.R(\boldsymbol{x}, \boldsymbol{x'})) \wedge G(\boldsymbol{x}) \wedge \neg L(\boldsymbol{x})$ is the characteristic function for these states. If this function is not the constant FALSE, then deadlock occurs in some states in $\mathcal{G} - \mathcal{L}$.

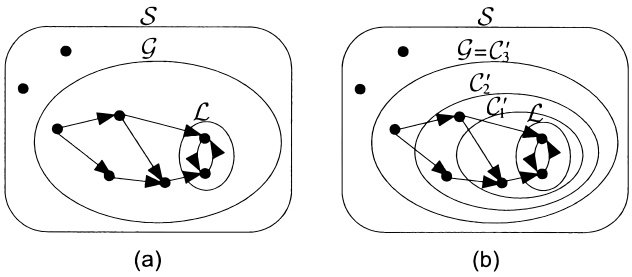


Fig. 2 Schematic overview of the proposed method.

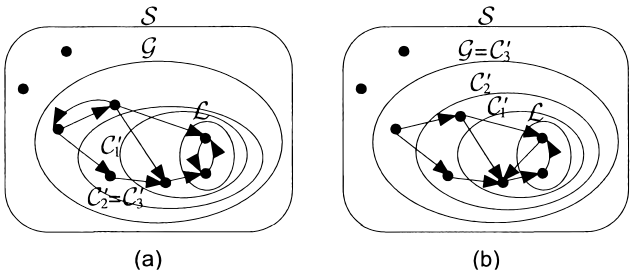


Fig. 3 The cases where the system is not self-stabilizing.

Step 3. By computing preimage, make the search go backwards in one step; i.e., $C'(\boldsymbol{x}) := C(\boldsymbol{x}) \vee ((\neg\exists \boldsymbol{x}'.(R(\boldsymbol{x},\boldsymbol{x}') \wedge \neg C(\boldsymbol{x}'))) \wedge G(\boldsymbol{x}))$. $c := c + 1$.

Step 4. If $C' \neq C$, then $C := C'$ and go to Step 2. Otherwise, stop. In this case, there is a state from which the system can never reach any legitimate state.

Figure 2 schematically explains how the algorithm works. In the figure, each dot represents a state in $\mathcal{S}$ and an arc from dots A to B means that $g_i \rightarrow g_j$ holds where g_i and g_j are the states represented by A and B, respectively. Suppose that the state space defined by a given algorithm is as shown in Fig. 2(a). Then the search goes as shown in Fig. 2(b). $\mathcal{C}'_i$ is the set of states represented by C' obtained in Step 3 at the ith iteration. In this example, $\mathcal{C}'_i$ becomes the same as $\mathcal{G}$ at the third iteration. Thus the stabilization time is computed at three.

If the system does not meet the convergence property, then the method can successfully detect that fact. In that case, since there are some states that cannot be guaranteed to reach a legitimate state in finite steps, C converges before it becomes G. Then the fact that the convergence property does not hold is detected in Step 4. Figure 3(a) schematically depicts this case. In the figure, there is a loop between two illegitimate states. Hence there is a infinite computation that does not reach a legitimate state.

On the other hand, whether or not the closure property holds cannot be determined by the proposed method. Figure 3(b) illustrates this. In this case, any state must reach some state in $\mathcal{L}$ in at most three steps, and the proposed method computes this number correctly. However, the algorithm is not self-stabilizing, since it does not satisfy the closure property.

Using another procedure, we can determine whether or not the algorithm meets the closure property. Interested readers are referred to the appendix.

4. Case Studies

4.1 Prototype System

We developed a prototype system that implements the proposed method. The system was written in the C language. A BDD library[†] from Carnegie Mellon University was used for manipulating OBDDs. The system was built on a Linux workstation with a 500 MHz pentium III processor and 256 Mbyte memory.

At the moment, we have not fully implemented the procedure for deriving the transition relation function from a given concurrent program written in the guarded command language. In the case studies, therefore, we made transition relation function partly by hand.

4.2 Algorithms Tested

We used the prototype system for analyzing two algorithms. The first algorithm is the K-state algorithm, while the second one is another mutual exclusion algorithm on rings, which was also proposed by Dijkstra [5]. The second algorithm uses only three-state processes and the state of a process is in $\{0, 1, 2\}$. The algorithm is presented below.

Process p_0
Variables $S_0 \in \{0, 1, 2\}$
begin
$(S_0 + 1) \bmod 3 = S_{m-1} \longrightarrow S_0 := (S_0 - 1) \bmod 3$
end

Process $p_i (i = 1, 2, \cdots, m-2)$
Variables $S_i \in \{0, 1, 2\}$
begin
$(S_i + 1) \bmod 3 = S_{i-1} \longrightarrow S_i := S_{i-1}$
[] $(S_i + 1) \bmod 3 = S_{i+1} \longrightarrow S_i := S_{i+1}$
end

Process p_{m-1}
Variables $S_{m-1} \in \{0, 1, 2\}$
begin
$(S_{m-2} = S_0) \wedge ((S_{m-2} + 1) \bmod 3 \neq S_{m-1})$
$\longrightarrow S_i := S_{i-1}$
end

As in the K-state algorithm, legitimate states are the ones where exactly one process is enabled in this

[†]http://www.cs.cmu.edu/afs/cs.cmu.edu/project/modck/pub/www/bdd.html

Table 1 Analysis results and performance for the K-state algorithm ($K = m$).

	$m=3$	$m=4$	$m=5$	$m=6$	$m=7$	$m=8$
Stabilization time	3	13	24	38	55	75
Execution time (in seconds)	0.01	0.02	0.16	1.09	6.87	28.11
Memory used (in Kbytes)	100	180	559	853	1725	2712

Table 2 Analysis results and performance for the 3-state algorithm.

	$m=3$	$m=4$	$m=5$	$m=6$	$m=7$	$m=8$
Stabilization time	1	10	22	39	57	79
Execution time (in seconds)	0.01	0.01	0.03	0.09	0.24	0.71
Memory used (in Kbytes)	96	153	283	330	618	851

algorithm. Hence the predicate P that represents these legitimate states is given as

$$P = \bigvee_{i=0}^{m-1} \left(E_i \wedge \bigwedge_{\substack{0 \le j \le m-1 \\ j \ne i}} \neg E_j \right)$$

where

$$E_i = \begin{cases} ((S_0 + 1) \bmod 3 = S_{m-1}) \\ \qquad\qquad i = 0 \\ ((S_i + 1) \bmod 3 = S_{i-1}) \\ \quad \vee ((S_i + 1) \bmod 3 = S_{i+1}) \\ \qquad\qquad 1 \le i \le m-2 \\ (S_{m-2} = S_0) \\ \quad \wedge ((S_{m-2} + 1) \bmod 3 \ne S_{m-1}) \\ \qquad\qquad i = m-1 \end{cases}$$

4.3 Results

Varying the number of processes, we computed the stabilization time of each of the two algorithms by the proposed method. We also recorded the time required for the computation and the size of memory used. Tables 1 and 2 show the results.

From the results, it can be seen that the two algorithms have almost the same stabilization time when the number of processes is modest. An interesting finding, however, is that the stabilization time of the K-state becomes slightly smaller than the 3-state algorithm when the number of processes exceeds six and the difference becomes larger as the number of processes grows. This is rather counter-intuitive, since the number of states of each process required by the K-state algorithm is larger than the 3-state algorithm and the difference increases with the number of processes.

The results also show that the proposed method completed analysis very fast, even when the state space was very huge. For example, when the number of the processes, m, is eight, the number of global states of the K-state algorithm is in excess of 10 million. (More precisely, it is $K^m = 16,777,216$ since $K = m = 8$.) Even in this case, the amount of time required for analysis was only around 30 seconds. The size of memory used was also very small, and it was less than three megabytes.

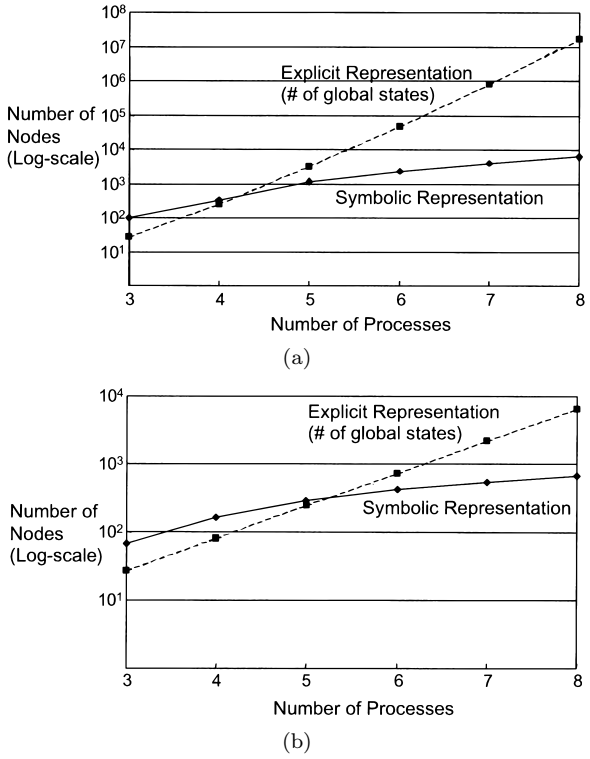


Fig. 4 Effect of symbolic representation. (a) the K-state algorithm and (b) the 3-state algorithm.

It seems rather clear that the good performance stems from the fact that the state space was represented very compactly using Boolean functions. This fact is well illustrated by the two graphs in Fig. 4. In the graphs, we plotted the number of nodes in the OBDD for the transition relation function and the number of global states. Note that the latter would be the same as the number of nodes when an explicit adjacency list were used for representing the transition relation. As shown in these graphs, the reduction ratio increases very rapidly with the number of processes. For instance, for the case of the K-state algorithm, the number of nodes of the OBDD was 6189 when $m = 8$. This is only 0.037 percent of the total number of global states.

5. Conclusions

Reduction of the stabilization time is one of the main performance issues in the design of self-stabilizing systems. In this paper, we proposed an automatic method for computing the stabilization time, aiming at supporting algorithm designers. To circumvent the state explosion problem, which often occurs in automated analysis of concurrent systems, the proposed method uses OBDDs to symbolically represent state spaces and computes the stabilization time by manipulating the OBDDs. Using a library for handling OBDDs, we developed a prototype system that implemented the proposed method. To demonstrate the applicability of the proposed method, we applied it to the analysis of existing self-stabilizing algorithms. In the case studies, the method required a very little amount of time and memory even when an underlying state space was very huge. From the results obtained, it was found that this was due to the fact that state spaces were expressed very compactly by means of symbolic representation.

References

[1] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese, "Time optimal self-stabilizing synchronization," Proc. 25th ACM Symp. on Theory of Computing, pp.652–661, 1993.
[2] R.E. Bryant, "Graph-based algorithms for boolean function manipulation," IEEE Trans. Comput., vol.C-35, no.8, pp.677–691, 1986.
[3] J.R. Burch, E.M. Clarke, and K.L. McMillan, "Symbolic model checking: 10^{20} states and beyond," Information and Computation, vol.98, pp.142–170, 1992.
[4] A.M. Costello and G. Varghese, "The FDDI MAC meets self-stabilization," Proc. 3rd Workshop on Self-Stabilizing Systems, pp.1–9, 1999.
[5] E.W. Dijkstra, "Self-stabilizing systems in spite of distributed control," Commun. ACM, vol.17, no.11, pp.643–644, Nov. 1974.
[6] S. Dolev, A. Israeli, and S. Moran, "Analyzing expected time by scheduler-luck games," IEEE Trans. Software Eng., vol.21, no.5, pp.429–439, May 1995.
[7] P. Godefroid and P. Wolper "A partial approach to model checking," Information and Computation, vol.110, no.2, pp.305–326, May 1994.
[8] S. Ghosh and A. Gupta, "An exercise in fault-containment: Self-stabilizing leader election," Inf. Process. Lett., vol.59, pp.281–288, 1996.
[9] G.J. Holzmann, "The model checker SPIN," IEEE Trans. Software Eng., vol.23, no.5, pp.279–295, May 1997.
[10] K.L. McMillan, Symbolic Model Checking, Kluwer Academic, 1993.
[11] I.S.W.B. Prasetya, "Mechanically verified self-stabilizing hierarchical algorithms," Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97), ed. E. Brinksma, LNCS, vol.1217, pp.399–415, April 1997.
[12] S. Qadeer and N. Shankar, "Verifying a self-stabilizing mutual exclusion algorithm," Proc. IFIP Working Conf. on Programming Concept and Methods (PROCOMET'98), pp.424–443, June 1998.
[13] M. Schneider, "Self-stabilization," ACM Computing Surveys, vol.25, no.1, pp.45–67, March 1993.
[14] S. Shukla, D.J. Rosenkrantz, and S.S. Ravi, "Simulation and validation of self-stabilizing protocols," Proc. SPIN96 Workshop, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol.32, pp.1052–1798, 1997.
[15] A. Singhai, S.-B. Lim, and S.R. Radia, "The SunSCALR framework for Internet servers," Proc. 28th Symp. on Fault-Tolerant Computing (FTCS-28), pp.108–117, 1998.
[16] G. Tel, "Maximal matching stabilizes in quadratic time," Inf. Process. Lett., vol.49, no.6, pp.271–272, 1994.
[17] A. Valmari, "A stubborn attack on state explosion," Proc. Second Workshop on Computer Aided Verification, LNCS, vol.531, pp.156–165, June 1990.

Appendix: Verifying the Closure Property

By using preimage computation, it is also possible to check whether or not a given algorithm meets the closure property. By definition, the closure property is met if and only if no legitimate state exists that can reach an illegitimate state in one step. Note that the set of illegitimate states is $\mathcal{G} - \mathcal{L}$ and its characteristic function is $G \wedge \neg L$. Let $\mathcal{N}$ be the set of states that can reach an illegitimate state in one step; i.e.,

$$\mathcal{N} = \{\boldsymbol{a} \in \mathcal{S} : \exists \boldsymbol{b}, \boldsymbol{a} \to \boldsymbol{b} \wedge \boldsymbol{b} \in \mathcal{G} - \mathcal{L}\}.$$

Clearly, $\mathcal{N}$ is the preimage of $G \wedge \neg L$ according to R; i.e.,

$$\mathcal{N} = \mathcal{P}(R, G \wedge \neg L).$$

Since $\mathcal{N} \cap \mathcal{L}$ is the set of legitimate states that can reach an illegitimate state in one step, the closure property can be verified by checking the emptiness of this intersection. This observation leads to the verification procedure depicted below.

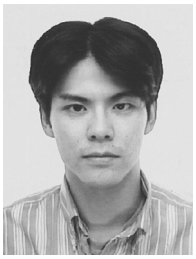
Step 1. $L' := G \wedge \neg L$.
Step 2. Perform preimage computation to obtain the characteristic function N for $\mathcal{N}$; i.e., $N(\boldsymbol{x}) := \exists \boldsymbol{x}'.(R(\boldsymbol{x}, \boldsymbol{x}') \wedge L'(\boldsymbol{x}'))$.
Step 3. Compute the characteristic function I for $\mathcal{N} \cap \mathcal{L}$; i.e., $I := N \wedge L$.
Step 4. Check whether I is the constant FALSE or not. If so, then the closure property is met since $\mathcal{N} \cap \mathcal{L} = \emptyset$; otherwise, the property does not hold, because there is a legitimate state that can reach an illegitimate state.

If we use the above procedure to restrict given distributed algorithms to those that meet the closure property, then the proposed analysis method can be simplified as follows. When the closure property holds, $C(\boldsymbol{x}) \vee ((\neg\exists \boldsymbol{x}'.(R(\boldsymbol{x}, \boldsymbol{x}') \wedge \neg C(\boldsymbol{x}'))) \wedge G(\boldsymbol{x})) = (\neg\exists \boldsymbol{x}'.(R(\boldsymbol{x}, \boldsymbol{x}') \wedge \neg C(\boldsymbol{x}'))) \wedge G(\boldsymbol{x})$ holds in Step 3 of the proposed analysis method. Thus $C'(\boldsymbol{x}) := C(\boldsymbol{x}) \vee ((\neg\exists \boldsymbol{x}'.(R(\boldsymbol{x}, \boldsymbol{x}') \wedge \neg C(\boldsymbol{x}'))) \wedge G(\boldsymbol{x}))$ in this step can be modified as $C'(\boldsymbol{x}) := (\neg\exists \boldsymbol{x}'.(R(\boldsymbol{x}, \boldsymbol{x}') \wedge \neg C(\boldsymbol{x}'))) \wedge G(\boldsymbol{x})$.

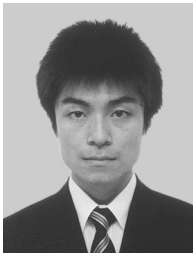
The rationale of this simplification is as follows. Suppose $\forall \boldsymbol{a} \in \mathcal{C}, \boldsymbol{a} \rightarrow \boldsymbol{b} \Rightarrow \boldsymbol{b} \in \mathcal{C}$. Then, by definition $\mathcal{C} \subseteq (\mathcal{S} - \mathcal{P}(R, \neg C)) \cap \mathcal{G}$ holds (i.e., $C(\boldsymbol{x}) \vee ((\neg\exists \boldsymbol{x}'.(R(\boldsymbol{x}, \boldsymbol{x}') \wedge \neg C(\boldsymbol{x}'))) \wedge G(\boldsymbol{x})) = (\neg\exists \boldsymbol{x}'.(R(\boldsymbol{x}, \boldsymbol{x}') \wedge \neg C(\boldsymbol{x}'))) \wedge G(\boldsymbol{x})$).

In this case, $\forall \boldsymbol{a} \in ((\mathcal{S} - \mathcal{P}(R, \neg C)) \cap \mathcal{G}) - \mathcal{C}, \boldsymbol{a} \rightarrow \boldsymbol{b} \Rightarrow \boldsymbol{b} \in \mathcal{C}$ also holds by definition. Hence $\forall \boldsymbol{a} \in \mathcal{C}', \boldsymbol{a} \rightarrow \boldsymbol{b} \Rightarrow \boldsymbol{b} \in \mathcal{C}'$, where $\mathcal{C}' = (\mathcal{S} - \mathcal{P}(R, \neg C)) \cap \mathcal{G}$.

Note that $\mathcal{C}$ is equal to $\mathcal{L}$ at the first iteration of the proposed method and $\forall \boldsymbol{a} \in \mathcal{L}, \boldsymbol{a} \rightarrow \boldsymbol{b} \Rightarrow \boldsymbol{b} \in \mathcal{L}$ holds due to the closure property. Therefore, for $\mathcal{C}$ at any iteration, $\forall \boldsymbol{a} \in \mathcal{C}, \boldsymbol{a} \rightarrow \boldsymbol{b} \Rightarrow \boldsymbol{b} \in \mathcal{C}$.



Tatsuhiro Tsuchiya received M.E. and Ph.D. 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 fault-tolerant computing. He is a member of IEEE.



Yusuke Tokuda is currently a master course student in the Department of Informatics and Mathematical Science at Osaka University. He has been engaged in research on self-stabilizing systems.

Tohru Kikuno was born in 1947. He received M.S. and Ph.D. degrees from Osaka University in 1972 and 1975, respectively. He joined Hiroshima University from 1975 to 1987. Since 1990, he has been a Professor of the Department of Informatics and Mathematical Science at Osaka University. His research interests include the quantitative evaluation of software development processes and the analysis of fault-tolerant systems. He served as a program co-chair of the 1st International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '98) and the 5th International Conference on Real-Time Computing Systems and Applications (RTCSA'98). He is a member of IEEE and ACM.