ソフトウェア工学 128－2
(2000. 7. 28)

自己安定分散アルゴリズムの自動検証システム[1]

角川裕次

広島大学工学部第二類 (電気系)

概要

自己安定分散システムとは分散システムの一種で、その初期状態に関わらず正常なネットワーク状態に収束するシステムである。自己安定システムは、任意の種類のそして任意の有限数の一時故障に耐えうるシステムである。何故なら、一時故障が終了した直後のシステム状態を新たな初期状態と考えれば、再び正常なネットワーク状態に回復するからである。

自己安定分散システムを設計する際には、その正しさの検証の困難さが問題となる。分散システムの非同期性に加えて、任意の初期状態から正常な状態への到達性の検証が必要だからである。

この論文では、自己安定分散システム記述用の言語を提案し、その自動検証システムとしての言語処理系を提案する。提案する検証システムは、正しくない自己安定分散システムに対しては正常なネットワーク状態に回復しないような実行系列を「反例」として提示する。幾つかの自己安定分散システムに対して本手法を適用し、その正当性の検証を行った。

Mechanical Verification of Self-Stabilizing Distributed Systems

Hirotsugu Kakugawa

Faculty of Engieering, Hiroshima University

Abstract

Self-stabilizing distributed systems are a class of distributed systems which converge to correct system states even if they start from arbitrary system states. A self-stabilizing system can recover from finite number of transient faults (e.g., message loss, memory corruption). Therefore, they are fault-tolerant systems. When we design a self-stabilizing system, its verification, such as convergence from arbitrary initial system state to a correct system state, is a difficult task; typically, a proof is long, complex and error-prone.

In this paper, we introduce a description language and its processor for self-stabilizing systems of arbitrary network topology to verify mechanically, i.e., a verification system. For an incorrect self-stabilizing system, the verification system outputs a counterexample which consists of an initial system state and an execution sequence which does not converge. Some distributed self-stabilizing systems have been verified by our verification system and their results are reported.

[1]本研究の一部は、文部省科学研究費補助金 (奨励研究 A, 課題番号 11780229) の援助を受けている。

1 Introduction

A distributed system consists of a set of processes and a set of communication links, each connecting a pair of processes. A distributed system is said to be *self-stabilizing* if it converges to a correct system state no matter which system state it starts with[5, 17]. A self-stabilizing system is considered to be a fault tolerant system, since it tolerates any kind and any finite number of transient failures. Consider transient faults (e.g., message loss, memory corruption) occur in a system, after which the system will be in an incorrect system state. Starting from this system state, self-stabilization property guarantees that the system converges to a correct system state again.

Generally speaking, verification of self-stabilizing system is not an easy task. The difficulty arises from following reasons:

- for any initial system state, convergence to correct system state must be guaranteed, and
- for any execution scheduling, convergence to correct system state must be guaranteed.

Many of self-stabilizing systems are proven by the *variant function* method proposed in [14]. A variant function, say f, is a function mapping from system states to non-negative integers. A system is verified by proving (1) a system state, say Q, is a correct state if and only if $f(Q) = 0$, (2) $f(Q) \geq f(Q')$, where Q is an arbitrary incorrect system state and Q' is an arbitrary system state that follows Q, and (3) for any execution scheduling, the value of f eventually decreases if the system is not in a correct system state. To use this method, we must find a variant function which satisfies above conditions. Self-stabilizing algorithms verified by this method are, for example, [13, 20].

A complex self-stabilizing system can be obtained by *composition* of several self-stabilizing subsystems. For example, composition of a spanning tree generation system on arbitrary network topology and a reset system on a spanning tree yields a reset system on arbitrary network topology[1]. Several methods to compose self-stabilizing systems are proposed, for example, in [6, 3]. But composed self-stabilizing subsystems must be verified by another method such as the variant function method.

Another verification method is reasoning on state transition rules. In [2], Beauquier et. al proposes a verification method based on *rewriting system*. State transition rules are considered as a set of rewriting rules. Although their method can be used to show correctness of correct self-stabilizing systems, it cannot be used to show incorrectness. In addition, network topologies of target distributed systems are limited only to ring and linear networks.

In [16], Lakhnech and Siegel discuss a deductive verification method of self-stabilizing systems based on *linear temporal logic*. In [11], Hsu and Huang propose a verification method based on *finite-state machine model*. As a case-study, they show correctness of a self-stabilizing maximal matching system [12], in which they show its correctness based on the variant function method.

By all methods above, we must interpret systems and operate manually to verify. An opposit approach is to describe a self-stabilizing system as a program in a description language and simulate the system by executing the program. To verify correctness of self-stabilizing systems, following conditions are important: (1) simulation should be started from each system state, and (2) every possible execution scheduling of processes should be considered.

In [18], Shukla et. al propose a method based on this scheme, except condition (1). Their verification system randomly selects an initial system state and check arbitrary execution scheduling of processes. Their verification system behaves as a preprocessor to SPIN which is a model verification system for concurrent event systems [7, 8, 19].

In this paper, we introduce a mechanical verification system of self-stabilizing systems. We define a description language of self-stabilizing systems, named SPR, and implement a language processor. Our verification system is based on exhaustive search of all possible system state transitions: for each (initial) system state, check every possible system state transition, and check some correctness criteria. Our verification system checks if a given self-stabilizing system is correct or not. If a given self-stabilizing system is incorrect, the verification system outputs an execution scheduling which causes an infinite loop (i.e., non-converging execution) as a counterexample.

Obviously, the number of possible system states exponentially explodes when the number of processes increases. Currently, we adopt SPIN as an underlaying verification system, since it provides strong techniques to save memory (state compression technique [10]) and execution time (partial oder reduction technique [9]), for example.

This paper is organized as follows. In section 2, we give a formal definition of self-stabilizing systems. In section 3, we propose a description language for our verification system. In section 4, we explain how self-stabilizing systems are mechanically verified. In section 5, we descrive experience of mechanical verification with our system. In section 6, we give concluding remarks.

2 Self-Stabilizing Systems

In this section, we describe self-stabilizing systems briefly. For details, please refer to [5, 17], for example.

A distributed system S consists of a set of n processes $P_0, P_1, ..., P_{n-1}$. A process P_i can refer to local states of neighbor processes. Neighboring relation is defined by a network topology of a given system.

An algorithm of a process, say P_i, is described as a set of *guarded commands*:

$$
\begin{aligned}
*[\quad & \\
& g_1^i(q_i, q_1^i, q_2^i, \cdots, q_k^i) \\
& \qquad \rightarrow q_1 := f_1^i(q_i, q_1^i, q_2^i, \cdots, q_k^i) \\
\Box\ & g_2^i(q_i, q_1^i, q_2^i, \cdots, q_k^i) \\
& \qquad \rightarrow q_i := f_2^i(q_i, q_1^i, q_2^i, \cdots, q_k^i) \\
& \quad \vdots \\
]\quad &
\end{aligned}
$$

where

- q_i is a local state (a tuple of local variables) of process P_i,
- q_j^i is a local state of P_j^i, which is a neighbor process of P_i (neighboring relation is determined by network topology of a system),
- each $g_j^i(\cdots) \rightarrow q_i := f_j^i(\cdots)$ is called a *guarded command*,
- g_j^i is a boolean function on local states of P_i and it's neighbors, called a *guard*, and
- $q_i := f_j^i(\cdots)$ is called a *command*, which updates next state of P_i from local states of P_i and it's neighbors.

The system is executed by repeating the following steps forever:

1. Evaluate each guard of each process. A process is *privileged* if and only if it has a guard evaluated to true.
2. A scheduler, called the *c-daemon*, arbitrarily selects one process, say P_i, among privileged processes.
3. Process P_i executes a command whose associated guard is true.

A *configuration* of S is a tuple of local states of all processes. Let Γ be a set of all configurations. A system S is *self-stabilizing* with respect to $\Lambda \subseteq \Gamma$ if and only if starting from any initial configuration, configuration reaches a configuration in Λ regardless how central daemon selects privileged process to execute. $\lambda \in \Lambda$ is called a *legitimate configuration*. A set of legitimate configurations Λ must be closed, i.e., for each $\lambda \in \Lambda$ and each possible next configuration λ' (if any), λ' must be in Λ.

If there is no privileged process at each legitimate configuration λ, then such a self-stabilizing system is called *silent*.

3 SPR: A Description Language

For mechanical verification, we define a description language, named SPR, for self-stabilizing systems. In this paper, we consider verification of *silent* self-stabilizing systems. Our language SPR adopts Lisp-like notation for its syntax. In SPR, integer is the only data type for variables and expressions. For boolean values, an integer zero is treated as boolean "false", and non-zero integers are treated as boolean "true". Each process is identified by a unique integer, called *id*.

The motivation to define a new language is to make high level description of self-stabilizing system possible. When we write a verification program in some existing programming languages, we must *explicitly* write a whole distributed system. This implies that such a verification model is heavily dependent on the number of processes and network topology. If we want to verify another network size or another network topology, we must re-write verification model from scratch.

In SPR language, such network parameters are parameterized; it is possible to write guarded commands which should not be modified for different network size and network topology. For example, it provides a predicate like "*there exists a neighbor such that F is true*", "*for each neighbor, F is true*", and a construct "*let x be a neighbour process such that F is true at x*", for example.

In SPR, we use following top level directives to define a self-stabilizing system to be verified.

- The number of processes in a system:

  `(the-number-of-processes` *n*`)`

- The minimum value of process id:

  `(process-id-base` *n*`)`

  This is an optional directive; default value is 1.

- Network topology:

  `(network-topology` ⟨*Name*⟩ [⟨*opt*⟩]`)`

  Supported topologies ⟨*Name*⟩ are: `linear`, `binary-tree`, `tree` (an optional parameter defines the number of children), `bidirectional-ring`, `unidirectional-ring`, `regular` (an optional parameter defines degree), `chordal` (an optional parameter defines degree), and `complete`.

- Local variables of a process:

```
(process-state
  ( ⟨var_1⟩ ⟨min_1⟩ ⟨max_1⟩ )
  ( ⟨var_2⟩ ⟨min_2⟩ ⟨max_2⟩ )
    ...  )
```

  By this directive, $\langle var_i \rangle$ is declared as a local variable of process and its range is $\langle min_i \rangle$ ... $\langle max_i \rangle$ for each $i = 1, 2.....$ Since supported data type is integer only, there is no declaration of data type.

- A set of guarded commands of a process:

```
(algorithm ⟨Process⟩
  (⟨Guard_1⟩ -> ⟨Command_1⟩)
  (⟨Guard_2⟩ -> ⟨Command_2⟩)
    ...  )
⟨Process⟩ ⟶ all | root | other | i
```

  We can define different or the same set of guarded commands for each process. By keyword `all`, all process have the same guarded commands. By keyword `root`, guarded commands of the root process (root is a process with minimum process id) is defined. Process can be selected by explicitly giving process id. By keyword `other`, a set of guarded commands is defined for processes which is not defined yet.

- Legitimate state:

```
(legitimate-state ⟨Expr⟩)
```

  $\langle Expr \rangle$ is an expression on local variables of processes. It evaluates to true if and only if a configuration is legitimate.

A self-stabilizing leader election algorithm proposed by Huang [13] is shown in Figure 3. It assumes bidirectional ring network and the number of processes is prime. This algorithm can be written in our SPR language. In Figure 4, his SPR version of his algorithm is shown (the number of processes is 7). For verifying networks of different sizes, only the number of processes should be modified.

Syntaxes for $\langle Guard \rangle$ and $\langle Expr \rangle$ are shown in Figure 1, and syntax for $\langle Command \rangle$ in Figure 2.

A self-stabilizing leader election algorithm proposed by Huang [13] is shown in Figure 3. It assumes bidirectional ring network and the number of processes is prime. This algorithm can be written in our SPR language, for the case of the number of processes is 7, as shown in Figure 4. For verifying networks of different sizes, only the number of process should be modified. Figure 5 shows an SPR version of Sur and Srimani's 2-coloring algorithm for bipartite network topology[20].

4 Verification Method

In this section, we explain how we can verify silent self-stabilizing systems.

Suppose that a description of a silent self-stabilizing system S is given. Let n be the number of processes of S, and let P_i be a process of S. For simplicity of explanation, we assume processes are numbered from 0 to $n-1$. Let $v_1^i, v_2^i, .., v_m^i$ be local variables of process P_i, and R_j be a range of v_j^i. (A range of v_j^i is the same for each i.) Let g_j^i be the j-th guard of process i, and let L be a predicate on configurations which evaluates to true if and only if a given configuration is legitimate.

Let G be a logical-OR of all guards of all processes, i.e., $G = \vee_{i,j} g_j^i$. By $G(\gamma)$, we denote that if there exists a g_j^i for some i and j such that g_j^i is true at configuration γ. In other words, $G(\gamma)$ is true if and only if there exists a privileged process.

Now we discuss our verification scheme in terms of $n, P_i, v_j^i, R_j, g_j^i, G$, and L. A set of all configurations Γ_S of S is defined as follows:

$$\Gamma_S = \{(v_1^0, ..., v_m^0, v_1^1, ..., v_m^1, ..., v_1^{n-1}, ..., v_m^{n-1}) \mid \forall i, j [v_j^i \in R_j]\}$$

A self-stabilizing system S is verified by the following algorithm:

```
var Visit : subset of $\Gamma_S$;
Verify($S$) {
  Visit := $\emptyset$;
  /* start from each configuration */
  for each $\gamma \in \Gamma_S$
    Traverse($\gamma$, $\varepsilon$);
  /* $S$ is verified */
}
Traverse($\gamma$, path) {
  if ($\gamma \in$ Visit)
    return;
  if ($\gamma$ appears in path)
    abort;  /* infinite loop exists */
  Visit := Visit $\cup \{\gamma\}$;
  if ($G(\gamma)$) {
    if ($L(\gamma)$)
        abort;    /* legitimate but non-stable */
    for each $\gamma'$ reachable by single step from $\gamma$
        Traverse($\gamma'$, path $\cdot \gamma'$);
  } else {
    if ($\neg L(\gamma)$)
        abort;  /* stable but non-legitimate */
```

$\langle \mathit{Guard} \rangle \longrightarrow \langle \mathit{Expr} \rangle$
$\langle \mathit{Expr} \rangle \longrightarrow$
`(not` $\langle \mathit{Expr} \rangle$`)` | `(and` $\langle \mathit{Expr}_1 \rangle$ ...`)` | `(or` $\langle \mathit{Expr}_1 \rangle$ ...`)` | `(=` $\langle \mathit{Expr}_1 \rangle$ $\langle \mathit{Expr}_2 \rangle$`)`
| `(<` $\langle \mathit{Expr}_1 \rangle$ $\langle \mathit{Expr}_2 \rangle$`)` | `(<=` $\langle \mathit{Expr}_1 \rangle$ $\langle \mathit{Expr}_2 \rangle$`)` | `(>` $\langle \mathit{Expr}_1 \rangle$ $\langle \mathit{Expr}_2 \rangle$`)`
| `(>=` $\langle \mathit{Expr}_1 \rangle$ $\langle \mathit{Expr}_2 \rangle$`)` | `(+` $\langle \mathit{Expr}_1 \rangle$ ...`)` | `(-` $\langle \mathit{Expr}_1 \rangle$ ...`)` | `(*` $\langle \mathit{Expr}_1 \rangle$ ...`)`
| `(/` $\langle \mathit{Expr}_1 \rangle$ $\langle \mathit{Expr}_2 \rangle$`)` | `(modulo` $\langle \mathit{Expr}_1 \rangle$ $\langle \mathit{Expr}_2 \rangle$`)`
| `(modulo-n-processes` $\langle \mathit{Expr} \rangle$`)` — $\langle \mathit{Expr} \rangle$ *modulo* the number of processes.
| `(cond-expr` $\langle \mathit{Expr}_1 \rangle$ $\langle \mathit{Expr}_2 \rangle$ $\langle \mathit{Expr}_3 \rangle$`)`
— Conditional expression. $\langle \mathit{Expr}_2 \rangle$ if $\langle \mathit{Expr}_1 \rangle$ is true; otherwise, $\langle \mathit{Expr}_3 \rangle$.
| `(state-ref` $\langle \mathit{var} \rangle$`)` — Reference to local variable $\langle \mathit{var} \rangle$.
| `(state-ref` $\langle \mathit{var} \rangle$ $\langle \mathit{Expr} \rangle$`)` — Reference to local variable $\langle \mathit{var} \rangle$ of process $\langle \mathit{Expr} \rangle$.
| `(root)` — Process id of the root.
| `(me)` — Process id of itself.
| `(right-process)` | `(left-process)` — Process id of right/left process. Ring networks only.
| `(itself)` — Process id that satisfys a condition, used inside of `for-each-process`, `for-each-neighbor`, etc.
| `(neighbor?` $\langle \mathit{Expr} \rangle$`)` — True if and only if $\langle \mathit{Expr} \rangle$ is a neighbor's process id.
| `(the-neighbor)` — Neighbor's process id that satisfys a condition, used inside of `let-neighbor` in $\langle \mathit{Command} \rangle$ part. See $\langle \mathit{Command} \rangle$.
| `(exists-process` $\langle \mathit{Expr} \rangle$`)` | `(exists-neighbor` $\langle \mathit{Expr} \rangle$`)`
— True if and only if there exists a (neighbor) process such that $\langle \mathit{Expr} \rangle$ is true.
| `(for-each-process` $\langle \mathit{Expr} \rangle$`)` | `(for-each-neighbor` $\langle \mathit{Expr} \rangle$`)`
— True if and only if $\langle \mathit{Expr} \rangle$ is true at each (neighbor) process.
| `(the-number-of-neighbors` $\langle \mathit{Expr} \rangle$`)` | `(the-number-of-processes` $\langle \mathit{Expr} \rangle$`)`
— The number of (neighbor) processes such that $\langle \mathit{Expr} \rangle$ is true.
| `(neighbor-with-max-id` $\langle \mathit{Expr} \rangle$`)` | `(neighbor-with-min-id` $\langle \mathit{Expr} \rangle$`)`
— Maximum (minimum) process id among neighbor processes such that $\langle \mathit{Expr} \rangle$ is true.
| `(neighbor-with-max-value` $\langle \mathit{Expr} \rangle$`)` | `(neighbor-with-min-value` $\langle \mathit{Expr} \rangle$`)`
— Process id such that $\langle \mathit{Expr} \rangle$ is maximum (minimum) among neighbor processes.
| `(max-value-among-neighbors` $\langle \mathit{Expr} \rangle$`)` | `(min-value-among-neighbors` $\langle \mathit{Expr} \rangle$`)`
— Maximum (minimum) of $\langle \mathit{Expr} \rangle$ among neighbor processes.
| `(sum-for-each-neighbor` $\langle \mathit{Expr} \rangle$`)` | `(product-for-each-neighbor` $\langle \mathit{Expr} \rangle$`)`
— Sum (product) of $\langle \mathit{Expr} \rangle$ for each neighbor process.

Figure 1: Syntax for $\langle \mathit{Guard} \rangle$ and $\langle \mathit{Expr} \rangle$

$\langle \mathit{Command} \rangle \longrightarrow$
`(begin` $\langle \mathit{Command}_1 \rangle$ $\langle \mathit{Command}_2 \rangle$ ...`)`
— Sequential execution of $\langle \mathit{Command}_1 \rangle$ ...
| `(state-set!` $\langle \mathit{variable} \rangle$ $\langle \mathit{Expr} \rangle$`)`
— Update value of local variable $\langle \mathit{variable} \rangle$ to $\langle \mathit{Expr} \rangle$.
| `(let-neighbor` $\langle \mathit{Expr} \rangle$ $\langle \mathit{Command}_1 \rangle$ $\langle \mathit{Command}_2 \rangle$ ...`)`
— Let the value of `(the-neighbor)` be process id of a neighbor such that $\langle \mathit{Expr} \rangle$ is true and execute $\langle \mathit{Command}_1 \rangle$ ...
| `(skip)`
— Empty statement. Do nothing.

Figure 2: Syntax for $\langle \mathit{Command} \rangle$

```
    }
}
```

By Traverse(γ, *path*), every configuration reachable from initial configuration γ is checked. To avoid checking a visited configuration more than once, a variable *Visited* is used to hold visited configurations. A variable *path* contains an execution history staring from an initial configuration. This is used to check a non-converging execution. A given self-stabilizing S is correct if and only if the following conditions *never* hold for every configuration γ reachable from arbitrary configuration. (recall that we are discussing *silent* self-stabilizing systems.)

- $G(\gamma) \wedge L(\gamma)$
  This implies that there exists a process which has a true guard but γ is legitimate; S is not silent.
- $\neg G(\gamma) \wedge \neg L(\gamma)$
  This implies that no process can make a move at a non-legitimate configuration γ. That is, S does not reach a legitimate configuration.
- γ is in an execution path from an initial configuration
  Suppose γ_0 be an initial configuration and an execution path is $\gamma_0 \cdot \gamma_1 \cdot \cdots \cdot \gamma_n$, where $\forall i[\gamma_i \not\in \Lambda]$, γ_{i+1} is a next configuration of γ_i by single step of move, and $\gamma_n = \gamma_0$. This is a counterexample of a non-converging execution.

We implemented a verification system of self-stabilizing system written in SPR language. Current implementation adopts SPIN for underlaying system, and our verification system consists of the following two components:

1. A SPR Compiler — We implemented a compiler, which generates a PROMELA code that implements an verification algorithm discussed above. It is written in a programming language SCHEME, a dialect of LISP. Our SPR compiler behaves as a preprocessor to SPIN.
2. A Model Checker SPIN — SPIN is a verifier for concurrent systems[7, 8, 19] and it adopts a language called PROMELA to model a system to be verified.

5 Verification Examples

We executed our verification system on a following plathome:

- Operating System: FreeBSD 3.2
- Hardware: IBM PC compatible computer with dual Pentium III 450Mhz and 1G byte of main memory
- Software: SCM scheme interpreter version 5d0, SPIN version 3.3.3, EGCS C compiler version 2.91.66

Verification process consists of following two phases:

- Checking of assertion $(G \wedge L) \vee (\neg G \wedge \neg L)$ is violated or not, and
- Checking of infinite loops

We used our verification system to verify the self-stabilizing leader election algorithm for unidirectional ring network by Huang [13]. (See Figure 3 and 4.) His algorithm assumes that the number of processes is prime. (It is known that there exists no deterministic algorithm which solves the leader election problem if the ring network size is composite.) Verification results for $n = 5, 6, ..., 9$ are shown in Table 1. The table shows consumed amount of memory and execution time to verify. For the cases of $n = 5, 7$, which are prime numbers, the verification system verified each systems. On the other hand, for the cases of $n = 6, 8, 9$, which are composite numbers, the verification system stops and outputs an initial configuration and a non-converging execution scheduling.

We also used our system to verify the self-stabilizing coloring algorithm for bipartite graph network by Sur et. al [20], and the self-stabilizing maximal matching algorithm for arbitrary network topology by Hsu et. al [12].

6 Conclusion

In this paper, we introduced a language SPR to model self-stabilizing distributed systems for mechanical verification. We developed a compiler from SPR to PROMELA which is an input language of SPIN system. We mechanically verified several self-stabilizing systems for some network sizes.

Our SPR language makes us possible to model self-stabilizing systems almost the same as original description (a set of guarded commands parameterized by the number of processes and network topology). This means that our system makes the verification process much easier.

Currently, our language processor does not implement a verification algorithm for non-silent self-stabilizing systems, such as mutual exclusion system. Implementation of verification systems for such systems is left as a future task. Current definition of SPR does not have arrays. Future tasks include extending the language and the language processor to handle arrays.

Table 1: Verification Results of the Leader Election Algorithm [13]

# Processes	Check of $(G \wedge L) \vee (\neg G \wedge \neg L)$		Check of non-convergence	
n	Memory (Mb)	Time (Sec)	Memory (Mb)	Time (Sec)
5	0.2	0.4	0.2	1.2
6	67.8	0.3	68.1	0.6
7	327.8	265.3	658.3	733.7
8	89.7	19.6	116.3	83.4
9	312.1	535.9	601.2	1340.7

References

[1] Anish Arora and Mohamed G. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43:1026–1038, 1994.

[2] J. Beauquier, B. Berard, and L. Fribourg. A new rewrite method for proving convergence of self-stabilizing systems. In *13th International Symposium on Distributed Computing (DISC), LNCS:1693*, pages 240–253. Springer-Verlag, 1999.

[3] J. Beauquier, M. Gradinariu, and C. Johnen. Randomized self-stabilizing optimal leader election under arbitrary scheduler on rings. Technical Report 1225, LRI, 1999.

[4] Ajoy K. Datta, Maria Grandinariu, and Sébastien Tixeuil. Self-stabilizing mutual exclusion using unfiar distributed schedular. Technical Report 1227, LRI, 1999.

[5] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[6] S. Dolev, A. Israeli, and S. Moran. Self stabilization of dynamic systems assuming only read/write atomicity. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, pages 103–117. ACM, 1990.

[7] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[8] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[9] Gerard J. Holzmann and D. Peled. An improvement in formal verification. In *FORTE*, 1994.

[10] Gerard J. Holzmann and Anuj Puri. A minimized automaton representation of reachable states. *International Journal on Software Tools for Technology Transfer*, 2(3):270–278, 1999.

[11] Su-Chu Hsu and Shing-Tsaan Huang. Analyzing self-stabilization with finite-state machine model. In *Proceedings of International Conference of Distributed Computing Systems*, pages 624–631, 1992.

[12] Su-Chu Hsu and Shing-Tsaan Huang. A self-stabilizing algorithm for maximal matching. *Information Processing Letters*, 43:77–81, 1992.

[13] Shing-Tsaan Huang. Leader election in uniform rings. *ACM Transactions on Programming Languages and Systems*, 15(3):563–573, July 1993.

[14] J. L. W. Kessels. An exercise in proving self-stabilization with a variant function. *Information Processing Letters*, 29(1):39–42, September 1988.

[15] S. S. Kulkarni, J. Rushby, and N. Shankar. A case-stydy in component-based mechanical verification of fault-tolerant programs. In *Proceedings of the Second Workshop on Self-Stabilizing Systems (WSS99)*, 1999.

[16] Y. Lakhnech and M. Siegel. Deductive verification of stabilizing systems. In *Proceedings of the Second Workshop on Self-Stabilizing Systems (WSS97)*, pages 201–216, 1997.

[17] M. Schneider. Self-stabilization. *ACM Computing Surveys*, 25(1):45–67, March 1993.

[18] Sandeep K. Shukla, Daniel J. Rosenkrantz, and S. S. Ravi. Simulation and validation tool for self-stabilizing protocols. In SPIN *Workshop*, 1996.

[19] The web page of SPIN. `http://netlib.bell-labs.com/netlib/spin/whatispin.html`.

[20] Sumit Sur and Pradip K. Srimani. A self-stabilizing algorithm for coloring bipartite graphs. *Information Sciences*, 69:219–227, 1993.

Processes: $P_0, P_1, ..., P_{n-1}$
Local variables of P_i: $b_i \in \{0, 1, ..., (n-1)\}$
Macro: $g(x,y) \;= n, \quad \text{if } x = y$
$\qquad = y - x(\bmod n), \quad \text{otherwise.}$
Rule 1: $(g(b_{i-1}, b_i) = g(b_i, b_{i+1})) \wedge (g(b_i, b_{i+1}) = n)$
$\qquad \rightarrow b_i = b_i + 1 \bmod n$
Rule 2: $(g(b_{i-1}, b_i) < g(b_i, b_{i+1}))$
$\qquad \rightarrow b_i = b_i + 1 \bmod n$

Figure 3: A Self-Stabilizing Leader Election Algorithm [13]

```
(the-number-of-processes  7)
(process-id-base  0)
(network-topology  bidirectional-ring)
(process-state  '(label 0 (- (the-number-of-processes) 1)))

(algorithm  all
  ;; Rule 1
  ((and (= (state-ref label) (state-ref label (left-process)))
        (= (state-ref label) (state-ref label (right-process))))
   -> (state-set! label (modulo-n-processes (+ (state-ref label) 1))))
  ;; Rule 2
  ((< (cond-expr (= (state-ref label (left-process)) (state-ref label))
                 (the-number-of-processes)
                 (modulo-n-processes
                  (- (state-ref label) (state-ref label (left-process)))))
      (cond-expr (= (state-ref label) (state-ref label (right-process)))
                 (the-number-of-processes)
                 (modulo-n-processes
                  (- (state-ref label (right-process)) (state-ref label)))))
   -> (state-set! label (modulo-n-processes (+ (state-ref label) 1)))))

(legitimate-state
  (and (for-each-process
        (= (modulo-n-processes
            (- (state-ref label) (state-ref label (left-process))))
           (modulo-n-processes
            (- (state-ref label (right-process)) (state-ref label)))))
       (= (the-number-of-processes (= (state-ref label) 0)) 1)))
```

Figure 4: A Self-Stabilizing Leader Election Algorithm [13] in SPR

```
(the-number-of-processes  7)
(process-id-base  1)
(network-topology  bipartite 2)
(process-state  (level 0 (the-number-of-processes)))

(algorithm  root
  ((!= (state-ref level) 0)
   -> (state-set! level 0)) )

(algorithm  other
  ((and (!= (state-ref level (neighbor-with-min-value (state-ref level)))
            (- (nprocs) 1))
        (!= (state-ref level)
            (+ (state-ref level (neighbor-with-min-value (state-ref level)))
               1)))
   -> (state-set! level
        (+ (state-ref level (neighbor-with-min-value (state-ref level)))
           1))) )

(legitimate-state
  (and (= (state-ref level (root)) 0)
       (for-each-non-root-process
        (= (state-ref level)
           (+ (state-ref level (neighbor-with-min-value (state-ref level)))
              1)))))
```

Figure 5: A Self-Stabilizing Coloring Algorithm [20] in SPR