Distributed Execution Model for Self-Stabilizing Systems*

Shing-Tsaan Huang, Lih-Chyau Wuu, Ming-Shin Tsai

Dept. of Computer Science, National Tsing Hua University
HsinChu, TAIWAN, 30043 R.O.C.
Email: sthuang@nthu.edu.tw

Abstract

There are several execution models for self-stabilizing systems discussed in the literature. Among them the distributed model is a more realistic one in the sense that it makes the weakest assumption about the execution environment; whereas the serial model is a less realistic one in the sense that it makes the strongest assumption. In this paper we first discuss how to convert a self-stabilizing system operating with the serial model into a system operating with the distributed model, but such a conversion does not guarantee that the converted system is self-stabilizing. Then we propose a transform technique which makes the proof whether or not the converted system is self-stabilizing much easier.

1. Introduction

A distributed computing system can be considered as a network of nodes. On each node there are variables as well as program code and constants. The program code and constants can be stored in Read-Only-Memory, and hence are resilient to transient faults. On the other hand, those variables which are used to maintain the state of a node must be kept in Random-Access-Memory, and hence are vulnerable to transient faults: any transient fault of the node may make its variables perturbed to be any possible value. For example, a variable of 2 bits may be any value between 0 and 3 after a transient fault.

Self-stabilizing systems, which were originally introduced by Dijkstra [7], are able to tolerate such perturbations on the variables of the nodes. Since the system state can be any possible state after a transient fault, a self-stabilizing system is modeled as one which starting from any possible initial state can reach a legitimate (or *stable*) state in finite time.

According to the semantics of concurrency (the extent to which process execution may overlap) and the program atomicity (the granularity of process executions), there are several execution models discussed in the literature for self-stabilizing systems. Here in this paper we classify them into four categories: *serial model*, *synchronous model*, *synchronized distributed model*, and *distributed model*. In the serial model [7], only one node executes an atomic step at a time, and an atomic step consists of (1) reading the states of its neighbors, and (2) modifying its state (by *making a move*), if necessary, according to the read states of its neighbors and its own state. Hence, the computing of the nodes is serialized and each node sees exactly the current states of its neighbors.

In the synchronous model [5], all the nodes simultaneously execute an atomic step as described above, and each node in this model also sees exactly the current states of its neighbors.

The computation in the synchronized distributed model [5] is like that of the synchronous model except that not all, but an arbitrary subset of the nodes synchronously execute the two substeps of an atomic step. Like the nodes in the previous two models, each node in the synchronized distributed model sees exactly the current states of ^Xxs neighbors.

As for the distributed model, an atomic step is refined to be either a reading step or a writing step [9, 2]. That is, in the distributed model, each node may read and record the states of its neighbors at a time, then modify its

*This work is supported by the National Science Council of R.O.C under the Contract NSC82-0408-E007-027.

state, if necessary, according to the recorded states of its neighbors and its own state at a later moment. This implies that the nodes in the distributed model do not see exactly the current states of their neighbors, which is more realistic in a distributed environment: A node knows the states of its neighbors by exchanging messages, and due to message delays the state of a node recorded by another node may not be its current state.

The distributed model is a more realistic one in the sense that it makes the weakest assumption about the execution environment; whereas the serial model is a less realistic one in the sense that it makes the strongest assumption. However, most proposed self-stabilizing systems [6, 7] assume the serial model because proving correctness in the serial model is easier than in other models. In this paper we first discuss how to convert such a serial system into a distributed one. As shown in [10], self-stabilization is, in principle, unstable across system classes. Such a conversion does not guarantee that the converted system with the distributed model is self-stabilizing. Then we propose a transformation technique which makes the proof whether or not a system operating with the distributed model is self-stabilizing much easier.

We demonstrate the applicability of the transformation technique by applying it to a number of protocols which are self-stabilizing in the serial model. These include Dijkstra's k-state protocol [7], the spanning tree protocol [6] and Dijkstra's 3-state protocol [7].

Related work

Proving correctness of self-stabilizing systems is not trivial [8, 12]. So far, the most frequently used technique is the *variant bounded function* method [12, 6]. By this method the system is first proven to be able to make a move as long as it is not stabilized, and a bounded function is then given whose value decreases (or increases) for each move. Although this technique is not restricted to the serial model, applying it on systems with other three models has difficulties. This is because in the serial model it is easier to find the bounded variant function since only one node may change its state at a time. Other models do not have this advantage.

Sur and Srimani [13] adopted a variant of bounded function to prove the correctness of self-stabilizing systems with the synchronous distributed model. The idea is to show that the number of possible system states is finite, and regardless of the execution sequence of the nodes the same system state can not be encountered twice. This means that the number of possible states is decreasing. Yet, the technique seems to have limited applications.

Burns, Gouda and Miller [5] proposed the *acyclic dependency graph* method for proving correctness of self-stabilizing systems with the synchronized distributed model. Node A depends on node B in a system configuration if the movement of node B can affect the movement of node A. A self-stabilizing system is correct with the synchronized distributed model if both of the following conditions hold: (1) it is correct with the serial model, and (2) the system can eventually reach a configuration after which its dependency graph is always acyclic. This method can also be applied on the synchronous model because by definition it can be easily verified that a system is correct with the synchronous model provided that it is correct with the synchronized distributed model. However, it cannot be directly applied on the distributed model since each node in the distributed model sees the recorded states of its neighbors, not the current states.

Brown, Gouda, and Wu [4] designed protocols that have the *noninterfering* property: Once a node can move from one state to another, it will make that move regardless of other nodes' movements. The noninterfering property allows a protocol to be executed in distributed environment; but, it is difficult to design protocols with such a property.

2. The Distributed Model

A distributed system considered in this paper is represented by a network of nodes. Two nodes are each other's neighbor if there exists a communication link between them. The only way to pass information from one node to another is via messages. It is assumed that the communication link is a reliable FIFO channel, i.e., messages arrive error free, without duplication or loss, and in the order sent. How to implement such a reliable FIFO channel in a self-stabilizing fashion were discussed in [1, 3, 11]. Those self-stabilizing data-link protocols [1, 3, 11] can serve as a basic building block when we design self-stabilizing systems with the distributed model.

In the distributed model, each node not only maintains its own state in some variables but also records the states of its neighbors in some other variables. Due to message

delay, the recorded state of a node by its neighbor may not be its current state.

A self-stabilizing protocol with the distributed model consists of several rules for each node. Each rule has the syntax

if <guard> **then** <sequence of *local* or *read* statements>

where a *local* statement is one that involves only the local variables of its node, and a *read* statement has the syntax <local variable>:= **read** <neighbor's state>. The <guard> is anyone of the following two forms:

<*local* guard>

<**True**>

where a *local* guard is a predicate that involves only the local variables of its node. A rule is said to be applicable when the guard is true. Though more than one rule may be applicable at the same time, it is assumed that a node can only execute one rule atomically at a time. Also, a *fairness scheduling* is assumed: each rule is scheduled infinitely often.

When an applicable rule is scheduled, the execution time is assumed instantaneous unless the rule includes **read** statements. A **read** statement sends acquiring message to one of the node's neighbors and waits for the reply. During the waiting period, the node does nothing except replying the acquiring messages from its neighbors. Such an implementation avoids deadlock which may happen when nodes in a cycle execute **read** statements at the same time.

Most self-stabilizing protocols are designed to work with the serial model. It is easy to convert them to be able to operate on the distributed model. First, each node must keep extra variables to record the states of its neighbors. Second, the current states of the neighbors in a rule in the serial model must be replaced with the corresponding recorded states. Finally, some extra rules with the syntax "if **True** then <extra recoding variable> := **read** <neighbor's state>" are added to update the recorded state.

In the following, Dijkstra's k-state protocol [7] for token circulation on a unidirectional ring is used as an example to first describe how to convert the serial protocol into a distributed one. Then the implementation of the converted protocol in a distributed environment is discussed.

There are n nodes connected as a unidirectional ring in the Dijkstra's k-state protocol. A ring is unidirectional in the sense that each node on the ring only communicates with its left (or right) node. The original protocol is as follows. The state of node i is denoted by S_i, where $0 \le S_i \le k\text{-}1$. The protocol works with the serial model when $k \ge n\text{-}1$, where n is the number of nodes on the ring.

Serial k-state protocol:

For node 0:

(R0) if $S_0 = S_{n-1}$ then $(S_0 := S_0 + 1) \bmod k$

For other node i $(1 \le i \le n\text{-}1)$:

(R1) if $S_i \ne S_{i-1}$ then $S_i := S_{i-1}$

We say that node 0 has a token when $S_0 = S_{n-1}$, and node i $(1 \le i \le n\text{-}1)$ has a token when $S_i \ne S_{i-1}$. The system is in a legitimate state if one and only one token circulates on the ring.

To convert the serial k-state protocol into a distributed one, each node must maintain not only its current state but also a recorded state of its left neighbor. Here we use S_i to denote the current state of node i and L_i the recorded state of node i-1 at node i. Furthermore, the original rules (R0) and (R1) must be replaced by the following rules (A0) and (A1) respectively, and extra rules (B0) and (B1) are added.

Distributed k-state protocol:

For node 0:

(A0) if $S_0 = L_0$ then $(S_0 := S_0 + 1) \bmod k$

(B0) if **True** then $L_0 :=$ **read**(S_{n-1})

For other node i $(1 \le i \le n\text{-}1)$:

(A1) if $S_i \ne L_i$ then $S_i := L_i$

(B1) if **True** then $L_i :=$ **read**(S_{i-1})

In the distributed k-state protocol, it is said that node 0 has a token when $S_0 = L_0$, and node i $(1 \le i \le n\text{-}1)$ has a token when $S_i \ne L_i$. Besides, a token is in transit between node i-1 and node i if $L_i \ne S_{i-1}$. The system is in a legitimate state if one and only one token circulates on the ring.

The above rules (A0) and (A1) are the same as the original protocol except that the state of the left neighbor of node i (viz. S_{i-1}) is replaced by its recorded state (viz. L_i). Hence, node i modifies its state depending on its own state and the recorded state of its left neighbor only. Note that a node does not notify its neighbor when it changes its state. This may make the recorded state out of date. Furthermore, a recorded state may be different from its current state because some transient faults occur. Rules (B0) and (B1) are designed to update the recorded states and to cope with those faults. Since the guards of

Rules(B0) and (B1) are always true, by the fairness scheduling Rules(B0) and (B1) are executed infinitely often. That is, each node on the ring once a while sends a message to acquire the state of its left neighbor and modifies its own state accordingly.

Now let us consider how to implement the distributed k-state protocol in a distributed environment such that a rule can be executed atomically. It is without doubt that Rule (A0) or (A1) can be executed atomically since it accesses only local variables of node i.

As for Rule (B0) or (B1), the **implementation** can be as follows. Node i sends an acquiring message to its left neighbor i-1, and waits for a reply from it. No other rules can be applied by node i when it is waiting for a reply. During the waiting period, node i can not apply the other rule but is allowed to reply the acquiring message from node i+1. This avoids deadlock caused by all the nodes simultaneously sending an acquiring message to their left neighbors. Upon receiving the reply message, node i updates its recorded state of node i-1.

However, due to message delay S_{i-1} in the reply message may not be exactly the current state of node i-1 when node i updates its variable L_i. Yet, it makes no difference if we **consider** the updating is done at the moment when node i-1 makes the reply. This is because no rules can be applied by node i during the period from the time that node i sends an acquiring message to node i-1 till the time when node i receives the reply and updates L_i.

Under above **implementation** and **consideration**, Rule (B0) or (B1) can be treated as if it were executed atomically and each node had seen exactly the state of its neighbor while applying these rules.

Definition 1: The applied time of a **read** statement is defined to be the time when the neighbor makes the reply. For example, node i applies rule (B1) at time t_1, and its neighbor i-1 makes the reply at time t_2. Due to message delay, node i receives the reply and update L_i at time t_3. By Definition 1, the applied time of "L_i := **read**(S_{i-1})" is considered to be at time t_2.

3. The Transformation Technique

We let each node have several processes: one of them is called *central* process, and the others are called *peripheral* processes. Each peripheral process corresponds to a neighbor of the node, and maintains the recorded state of the neighbor via atomic read operations. The central process then maintains the state of the node and modifies the state by atomic write operation that depends on the state and the states of the peripheral processes only.

The rules executed by a node are distributed to its central and peripheral processes. The central process executes the rules which are used to maintain the state of the node, and the peripheral processes executes the other rules which are used to read the neighbors' states of the node.

Assumption 1: It is assumed that the central process cannot apply any rule while some peripheral processes in the same node are applying rules, and vice versa.

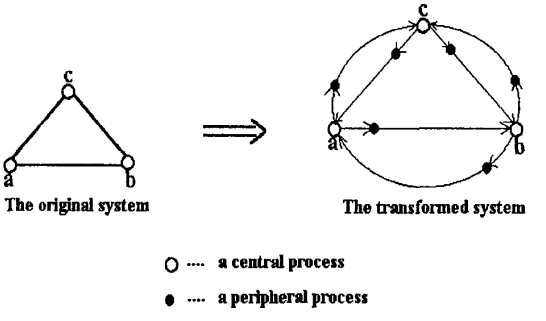
This assumption is reasonable since the node can only apply a rule at a time in the original system.

The system now consists of central and peripheral processes and can be modeled as a directed graph $G = (V, E)$, where V is the set of vertices and E is the set of directed edges. Every process in the system is represented by a vertex. A directed edge points from process i to process j if

(1) processes i and j reside on the same node and process i is the central process while process j is the peripheral process; or

(2) process i is a peripheral process which records the state of central process j.

Intuitively, a directed edge pointing from i to j means that the state of process i depends on that of process j. The following figure gives an example for transforming a system with three nodes a, b and c into a new system with three central processes and six peripheral processes.



In the transformed system, process j is a neighbor of process i if there is a directed edge pointing from vertex i to vertex j. Thus, the neighbors of a central process are

peripheral processes, and a peripheral process has one and only one neighbor which is a central process.

In what follows, we first show that the transformed system T is equivalent to its original system D. Next, we prove that any computation of the transformed system corresponds to a computation in the serial model. Consequently, a system with the distributed model is self-stabilizing if its transformed system is self-stabilizing with the serial model.

Lemma 1. System D is equivalent to its transformed system T.

Proof. As indicated in the above description, a system D consisting of n nodes is transformed into a system T which consists of n central processes and $n_1+n_2+...+n_n$ peripheral processes, where n_i is the number of node i's neighbors. By the transformation technique, the variables and the rules of a node are distributed into the central process and the peripheral processes at the same node. The transformation technique does not add any extra variable or rule on any process. That is, the tranformed system T is just a refinement mapping from the node level of system D to the process level. The behavior of a node can be emulated by its central process and peripheral processes, and vice versa. Thus, the lemma holds. □

Lemma 2. Any computation of system T corresponds to a computation in the serial model.

Proof. We first show that each process of system T sees the current states of its neighbors when it applies a rule. Then, we show that concurrent operations of system T can be regarded as being executed in some serial order.

A central process applies the rule with the syntax "**if** <local guard> **then** <sequence of local statements>". The variables involved in the rule are separated into the central process and the peripherial processes at the same node. By Assumption 1, it is obvious that a central process sees exactly the current states of its neighbors while it applies a rule.

A peripherial process applies the rule with the syntax "**if** <True> **then** <read statement>". By Definition 1, the applied time is the time when its neighbor replies. This makes a peripheral process *see* the current state of its neighbor when it applies a rule.

The system T can be modeled as a directed graph and the neighbors of a central process must be peripheral processes, and vice versa. Since the central process and peripheral processes of the same node cannot apply rules at the same time, the dependency graph [5] must be acylic. That is, the concurrent operations of any step in system T can be regarded as being executed in some serial order.

Since (1) each process of system T sees the current states of its neighbors when it applies a rule and (2) any step in a computation of system T can correspond to a sequence of serial operations, the lemma holds. □

Let us use the Dijkstra's k-state protocol for token circulation on a unidirectional ring as an example again. According to the transformation technique, each node has a central process and a peripheral process, and the transformed system is still a unidirectional ring. The central/peripheral process 0 executes Rule (A0)/(B0) of protocol α. The other central/peripheral processes execute Rule (A1)/(B1) of protocol α. For the convenience of proving, we hereafter replace the rule "**if** <True> **then read** statement" with "**if** <local variable for recording neighbor's state> $\neq$ <neighbor's state> **then** <assignment statement>". For example, we replace Rule (B0) with "if $L_0 \neq S_{n-1}$ then $L_0 := S_{n-1}$", and Rule (B1) with "if $L_i \neq S_{i-1}$ then $L_i := S_{i-1}$". Now we rename the processes in the ring as P_0, P_1, ..., P_{2n-1} and use S_i to denote the state of P_i. Then the rules of protocol α can be rewritten as follows.

For process P_0:

(A0) if $S_0 = S_{n-1}$ then $(S_0 := S_0 + 1) \bmod k$

For other process P_i $(1 \leq i \leq 2n-1)$:

(A1) if $S_i \neq S_{i-1}$ then $S_i := S_{i-1}$

The transformed protocol is exactly the original one reported in [7]. It has long been known that the original protocol is correct with the serial model when k is larger than or equal to the ring size minus one. Since the ring now has $2n$ processes, the protocol α is correct with the distributed model provided that $k \geq 2n-1$.

It is not always fortunate to have a transformed protocol being exactly the same as its original one. However, protocols based on the serial model are generally easier to be proven than those based on the distributed model. In the next section, we will give another example. First, we constructs the transformed system then we apply the variant function technique to prove the correctness of the transformed system.

4. The Spanning Tree Protocol

Let us use the spanning tree protocol reported in [6] as an example. That protocol works with the serial model; here it was converted into a protocol with the distributed model. Consider a connected graph $G(V, E)$ in which V is a set of nodes and E is a set of edges. Let $|V| = n$. Each node except a specific one which we call the *root* maintains two local variables: *level* and *parent*. The values of *level* range over {1, ..., n} and the *parent* of a node points to one of its neighbors. The root has a constant level of value 0 and no parent pointer. We use $L.i$ and $P.i$ to denote the level and parent of node i respectively, and use $N(i)$ to denote the set of the neighbors of node i. Note that the notation used in this section follows that used in [6], which uses $x.y$ to mean x of node y. Besides $L.i$ and $P.i$, for each neighbor node k, node i maintains an additional variable $L.k.i$ to record the level of node k.

The system is said to be in a *legitimate* state if the following three conditions hold. (1) The parent pointers constitute a spanning tree of G rooted at the root r. (2) For each node except the root r, its level is equal to the level of its parent plus one. (3) Each of the recorded levels of its neighbors is equal to the level of the corresponding neighbor. Otherwise, it is in an *illegitimate* state. When the system reaches a legitimate state, the following predicate is true:

$$GST \equiv (\forall\, i : i \neq r : (L.i = L.(P.i).i + 1) \wedge (\forall\, k : k \in N(i) : (L.k.i = L.k)))$$

The protocol consists of the following rules for each node i other than r:

(R0) if $(L.i \neq n \wedge L.i \neq L.(P.i).i + 1 \wedge L.(P.i).i \neq n)$ then $L.i := L.(P.i).i + 1$

(R1) if $(L.i \neq n \wedge L.(P.i).i = n)$ then $L.i := n$

(R2) if $(\exists\, k : k \in N(i) : L.i = n \wedge L.k.i < n - 1)$ then $L.i := L.k.i + 1$; $P.i := k$

(R3) if $(\exists\, k : k \in N(i) : L.k.i \neq L.k)$ then $L.k.i := L.k$

The above rules (R0), (R1) and (R2) are modified from those in [6] by replacing the label of a neighbor of node i with its recorded label of that neighbor. Rule (R3) is designed to read neighbors' states.

Now we first transform the original undirected graph into a directed one by using the transformation technique described in Section 3. Then the variant function technique is applied to prove the correctness of the transformed system.

The transformation is as follows. For each node i, we assume the existence of a central process i and several peripheral processes $(k.i)$, $k \in N(i)$. Each process is represented by a vertex. A directed edge points from the central process i to every peripheral process $(k.i)$ and a directed edge points from the peripheral process $(k.i)$ to the central process of node k.

By the above transformation, there exists peripheral process $(k.r)$. The existence of peripheral process $(k.r)$ is useless because the root r need not know its neighbor's state; however, its existence does not affect the proof of the correctness.

Every central process executes Rule (R0), (R1) and (R2) and every peripheral process executes Rule (R3) only. The variables maintained by node i originally for recording the state of its neighbor k, viz. $L.k.i$, is now assumed to be maintained by the peripheral process $(k.i)$. Thus, we can rewrite the above protocol as follows.

(R0) if $(L.i \neq n \wedge L.i \neq L.(P.i) + 1 \wedge L.(P.i) \neq n)$ then $L.i := L.(P.i) + 1$

(R1) if $(L.i \neq n \wedge L.(P.i) = n)$ then $L.i := n$

(R2) if $(\exists\, k : k \in N(i) : L.i = n \wedge L.k < n - 1)$ then $L.i := L.k + 1$; $P.i := k$

(R3) if $(L.i \neq L.(P.i))$ then $L.i := L.(P.i)$

The rewritten Rules (R0), (R1) and (R2) are exactly the ones in [6] except that here $P.i$ points to some peripheral process. Each peripheral process only needs to maintain one variable *level* since each peripheral process has a unique neighbor, thus its parent pointer is a fixed one. $P.i$ of Rule (R3) points to the central process whose state is recorded by peripheral process i.

When the system reaches a legitimate state, the following predicate is true:

$$GST \equiv (\forall \text{ central process } i : i \neq r : (L.i = L.(P.i)+1)) \wedge (\forall \text{ peripheral process } i : (L.i = L.(P.i)))$$

The transformation makes the correctness proof of the new system directly follow the proving steps in [6]. The proof is based on the variant function technique.

A central process i, $i \neq r$, is said to be *unstable* if $L.i \neq L.(P.i) + 1$, and a peripheral process i is said to be *unstable* if $L.i \neq L.(P.i)$.

Lemma 3. Before GST is true, the protocol does not terminate.

Proof. $\neg$GST implies that some process i is unstable. If process i is a peripheral one, then it can apply (R3).

There are two cases when process i is a central process.
Case 1: $L.i \neq n$. Process i can apply either (R0) or (R1).
Case 2: $L.i = n$. Let S be the maximum spanning tree induced by the parent pointers of stable processes rooted at r. Since central process i is not stable, the level of any process of S must be less than n-1. We can have some process k with a directed edge pointing to one process v of S because the directed graph is connected. If process k is a peripheral one, then it can apply (R3). If process k is a central one, then it can apply either (R0) or (R1) when $L.k \neq n$ as in Case 1. If $L.k = n$ then process k can apply (R2) because $L.v < n - 1$. □

Now let c_i, $1 \leq i \leq n$, be the number of unstable central processes that have level of value i, and p_i be the number of unstable peripheral processes that have level of value i. Then, define the bounded function F being a $2n$-ary vector as below.
$F \equiv (c_1, p_1, c_2, p_2, \ldots c_n, p_n)$. It can be easily seen that F is bounded below by (0, 0, ..., 0).

Lemma 4. F monotonically decreases each time when any of the rules is applied.
Proof. Case 1: A central process k with $L.k = i$ applies (R0). Then, c_i decreases by one because process k changes from unstable to stable, and p_i may increase because stable peripheral processes of k's neighbors becomes unstable. All others remains unchanged.
Case 2: A central process k with $L.k = i$, $i \neq n$, applies (R1). Then, c_i decreases by one, c_n may increase by one, and p_i may increase. All others remain unchanged.
Case 3: A central process k with $L.k = i$ applies (R2). Then, c_i decreases by one, and all others remain unchanged.
Case 4: A peripheral process k with $L.k = i$ applies (R3). Then, p_i decreases by one and c_{i+1} may increase. All others remain unchanged.

From the above four cases, according to lexicographical order, F monotonically decreases each time when any of the rules is applied. □

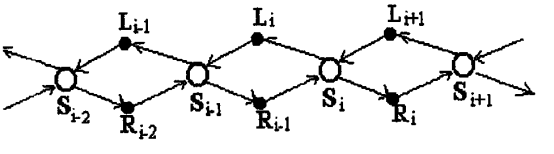
The following theorem is then a direct consequence of the above two lemmas.
Theorem 1. Eventually, the system reaches a legitimate state.

5. The 3-state Protocol

Dijkstra's 3-state protocol for token circulation on rings [7] is not self-stabilizing with the distributed model. This can be shown as follows. Let S_i denote the state of node i, and L_i and R_i denote the recorded states of its left and right neighbors respectively.

For node 0:
(A0) if $(S_0 + 1) \bmod 3 = R_0$ then $(S_0 := S_0 + 2) \bmod 3$
(B0) if $R_0 \neq S_1$ then $R_0 := S_1$
For other node i $(1 \leq i \leq n\text{-}2)$:
(A1) if $(S_i + 1) \bmod 3 = L_i$ then $S_i := L_i$
(A2) if $(S_i + 1) \bmod 3 = R_i$ then $S_i := R_i$
(B1) if $L_i \neq S_{i-1}$ then $L_i := S_{i-1}$
(B2) if $R_i \neq S_{i+1}$ then $R_i := S_{i+1}$
For node n - 1:
(A3) if $(L_{n-1} = R_{n-1}) \wedge (S_{n-1} \neq ((R_{n-1} + 1) \bmod 3))$
then $S_{n-1} := (R_{n-1} + 1) \bmod 3$
(B3) if $L_{n-1} \neq S_{n-2}$ then $L_{n-1} := S_{n-2}$
(B4) if $R_{n-1} \neq S_0$ then $R_{n-1} := S_0$

Node 0 is called the bottom node, node n - 1 is called the top node, and the other nodes are called the middle nodes. For simplicity, we are only concerned with the middle nodes and show that the system cannot stabilize under some initial condition. First we replace the nodes with central processes and peripheral processes, and the edges with the directed edges by using the transformation technique mentioned in Section 3. The transformed graph is depicted as follows. A central process can apply (A1) and (A2), whereas a peripheral process can only apply (B1) or (B2).



● --- a peripheral process

○ --- a central process

The computation given below is an example showing that the system cannot stabilize. The state of a process is underlined here to emphasize which process makes a move, and the symbol beside the underline indicates which rule is applied.

.... R_{i-1}	S_i	L_i	S_{i-1} ....
1	2	0	$\underline{0}_{A2}$
$\underline{1}_{B2}$	2	0	1
2	$\underline{2}_{A1}$	0	1
2	0	$\underline{0}_{B1}$	1
2	0	1	$\underline{1}_{A2}$
$\underline{2}_{B2}$	0	1	2
0	$\underline{0}_{A1}$	1	2
0	1	$\underline{1}_{B1}$	2
0	1	2	$\underline{2}_{A2}$
$\underline{0}_{B2}$	1	2	0
1	$\underline{1}_{A1}$	2	0
1	2	$\underline{2}_{B1}$	0
1	2	0	$\underline{0}_{A2}$
		:	
		:	
		:	

It is said that S_i has a token when $[(S_i + 1) \bmod 3 = R_i]$ or $[(S_i + 1) \bmod 3 = L_i]$, L_i has a token when $L_i \neq S_{i-1}$ and R_i has a token when $R_i \neq S_{i+1}$. The above computation shows that the system has three tokens in S_{i-1}, R_{i-1} and S_i initially, and these tokens may circulate on those processes forever so that the system cannot stabilize. Therefore, Dijkstra's 3-state protocol is not self-stabilizing with the distributed model.

6. Conclusion

We have proposed a transformation technique which can map a computation with the distributed model to a computation with the serial model so that the correctness proof of a self-stabilizing system working with the distributed model can be simplified. The utilization of the transformation technique has been demonstrated on Dijkstra's k-state protocol and the spanning tree protocol. Additionally, a counterexample shows that Dijkstra's 3-state protocol is not self-stabilizing with the distributed model. Such a technique would apparently be promising in proving the validity of other self-stabilizing systems working with the distributed model.

As shown in [10], self-stabilization is, in principle, unstable across system classes. Our method does not guarantee to preserve self-stabilization in simulating a serial system by a distributed system. Thus, it merits further investigation for finding some characteristics which make the serial system have a corresponding distributed system.

References

[1] Afek, Y. and Brown, G.M. "Self-Stabilization over unreliable communication media." *Distributed Computing*, 7(1993), pp27-34.

[2] Arora, A. and Gouda, M. "Distributed reset." *Proc. of 10th Conference on Foundations of Software Technology and Theoretical Computer Science, LNCS 472*(1990), pp.316-331, Springer-Verlag.

[3] Awerbuch, B., Patt-Shamir, B. and Varghese, G. "Self-stabilization By Local Checking and Correction." *Proc. of 32nd IEEE Symposium on Foundations of Computer Science*(1990), pp.268-277.

[4] Brown, G.M., Gouda, M.G. and Wu, C.L. "Token Systems that Self-Stabilize." *IEEE Trans. on Computers*, 38, 6(June 1989), pp.845-852.

[5] Burns, J.E., Gouda, M.G. and Miller, R.E. "On Relaxing Interleaving Assumptions." *Proc. MCC Worshop on Self-stabilization*(Nov. 1989).

[6] Chen, N.S., Yu, F.P. and Huang, S.T. "A Self-Stabilizing Algorithm for Constructing Spanning Trees." *Inf. Process. Lett.*, Vol. 39(Aug. 1991), pp.147-151.

[7] Dijkstra, E.W. "Self-Stabilizing Systems in Spite of Distributed Control." *Commun. ACM*, 17, 11(Nov. 1974), pp.643-644.

[8] Dijkstra, E.W. "A Belated Proof of Self-Stabilizing." *Distributed Computing*, 1,1(1986), pp. 5-6.

[9] Dolev, S., Israeli, A. and Moran, S. "Self-Stabilization of Dynamic Systems Assuming Only Read/Write Atomicity." *Distributed Computing*, 7(1993), pp.3-16.

[10] Gouda, M.G., Howell, R.R. and Rosier, L.E. "The Instability of Self-Stabilization." *Acta Informatica*(1990), pp.697-724.

[11] Gouda, M.G. and Multari, N.J. "Stabilizing Communication Protocols." *IEEE Trans.on Computers*, 40, 4(Apr. 1991), pp. 448-458.

[12] Kessels, J.L.W. "An Exercise in Proving Self-Stabilization with a Variant Function." *Inf. Process. Lett.*, Vol. 29(Sep. 1988), pp.39-42.

[13] Sur, S. and Srimani, P.K. "A Self-Stabilizing Distributed Algorithm to Construct BFS Spanning Trees of a Symmetric Graph." *Parallel Process. Lett.*, Vol. 2 No. 2 & 3 (1992), pp.171-179.