Development of Self-Stabilizing Distributed Algorithms using Transformation: Case Studies

Hirotsugu Kakugawa*[1]
Masaaki Mizuno[2]
Mikhail Nesterenko[2]

[1] Research Institute for Information Science and Education, Hiroshima University, Higashi-Hiroshima, Japan. E-mail: kakugawa@se.hiroshima-u.ac.jp
[2] Department of Computing and Information Sciences, Kansas State University, Manhattan, KS 66506, USA. Email: {masaaki,mikhail}@cis.ksu.edu

Abstract. Many self-stabilizing (SS) algorithms have been developed for the serial model, which has much stronger assumptions on the execution environments than common distributed (*i.e.,* message-passing) systems provide. It seems that many of the serial model algorithms were developed for theoretical interest rather than practical application. In contrast, SS distributed model algorithms, which actually run in a real distributed computing environment, are more difficult to develop and verify than those for the serial model.

In an earlier paper, we presented a transformation algorithm that transforms SS serial model algorithms to equivalent distributed model algorithms. The transformation enables us to execute many existing SS serial model algorithms in a distributed environment. More importantly, the transformation also helps us develop new and more practical algorithms starting with the serial model. In this way, the task of developing SS distributed model algorithms is significantly simplified.

This paper demonstrates, through case studies, the effectiveness of the transformation to develop new SS distributed algorithms. We present SS versions of lock based distributed mutual exclusion (DMX) algorithms and a leader election algorithm for the serial model. For one of the DMX algorithms, we give a correctness proof and compare message complexity of its transformed algorithm to that of an algorithm developed from scratch; both algorithms exhibit the same asymptotic complexity.

1 Introduction

A system is defined to be *self-stabilizing (SS)*[3] with respect to a set of *legitimate* states if regardless of its initial state, the system is guaranteed to arrive at a legitimate state in a finite number of execution steps and will never leave legitimate states after that. Thus, SS system need not be initialized and is able to recover from transient failures by itself.

* This work is supported in part by the Ministry of Education, Science and Culture under Grant No. 09780289, and the Telecommunication Advancement Foundation.

An SS system consists of two components: a program and a run-time support system. A run-time system provides a program with an execution environment and services. A program is an implementation of an SS algorithm[3]. Many SS programs have been developed with various assumptions about their execution environments. These assumptions include the semantics of concurrency and communication primitives that the run-time support systems provide. Huang *et al.* classified the execution environments into four models based on these assumptions: serial model, synchronous model, synchronous distributed model, and distributed model [5].

Among the four models, the distributed model has the weakest assumptions about the execution environment. It accurately reflects a common message-passing based distributed computing environment. Thus, programs developed for the distributed model (called *distributed model programs*) should run on a real distributed system with only minor modifications, if any. On the other hand, the serial model has the strongest assumptions, and programs developed for the serial model (called *serial model programs*) do not run in a real distributed system without major modifications. Naturally, distributed model programs are the most difficult to develop and verify, and serial model programs are the easiest.

In the serial model, an atomic execution step consists of (1) a read sub-step, which reads the states of its neighbor nodes; followed by (2) a write sub-step, which modifies its own state (based on the neighbors' current states and its own state). Commonly, atomic execution steps are described in guarded commands. The communication and concurrency semantics are such that each node can always see the current states of its neighbors, and only one node at a time executes an atomic step. Thus, *writing serial model programs is like writing programs for a shared memory environment*, and *non-interference is guaranteed within guarded commands.* Furthermore, *channel states need not be considered* in the serial model. Since handling channel states is complex[4], absence of channel states is one of the major advantages of developing programs in the serial model. There are well-established verification methods for SS serial model programs [7] and such verification is much simpler than verification of SS distributed model programs.

We have developed an algorithm that transforms a serial model program to a distributed model program [11]. A timestamp based concurrency control (CC) protocol is incorporated in the transformed program. Since the CC protocol itself is self-stabilizing, after the CC protocol stabilizes, it guarantees that for each execution of the transformed distributed model program, there always exists an *equivalent* execution of the original serial model program. Therefore, if the original program is correct with respect to self-stabilization, the transformed program is also self-stabilizing and performs exactly like the original program. We have successfully implemented transformed versions of several existing serial model programs on our distributed/parallel program development system[10].

Many SS serial model algorithms have been proposed. It seems that many

[3] We do not distinguish between the terms "program" and "algorithm," and use them interchangeably.

of these algorithms were developed for theoretical interest rather than practical application. For example, many of these programs have unrealistically strong assumptions on which nodes each node can communicate with. Consider Dijkstra's 3-state and k-state mutual exclusion algorithms[3]. Each node can obtain states of only one or both of its neighbor nodes. The algorithms are theoretically interesting since even though each node has limited information, the system as a whole converges to states that satisfy mutual exclusion.

The transformation algorithm enables us to execute many of such existing SS serial model algorithms in a distributed computing environment. However, more importantly, it also helps us develop new and more practical SS distributed algorithms. For example, many (mainly non-SS) distributed algorithms have less restrictions on which nodes each node may directly communicate with. Most existing non-SS distributed mutual exclusion algorithms assume fully connected networks. In the state of current network technology, this assumption is natural. Under such assumption, the serial model still provides very effective environment to develop SS (as well as non-SS) distributed algorithms.

The goal of this paper is to demonstrate, through several case studies, the effectiveness of the transformation algorithm to develop new and practical SS distributed algorithms. We begin a development with a serial model algorithm, considering the execution environment, such as the connectivity of the network, and prove it correct using a well-established verification method for serial model algorithms. The resulting algorithm is then transformed to an equivalent SS distributed model algorithm by the transformation algorithm. Because of the freedom from interference within guarded commands and the absence of channel states in the serial model, design and verification of SS distributed algorithms become significantly simplified using this method, compared with development of SS distributed model algorithms from scratch. We present SS versions of lock-based mutual exclusion algorithms and a leader election algorithm in the serial model. Note how simple these algorithms are, compared with the original non-SS distributed algorithms.

One drawback of this approach is the efficiency of transformed algorithms. We have implemented several transformed algorithms on our distributed/parallel development system and measured their message complexity. We have also implemented the same SS distributed algorithms from scratch to compare the message complexity. One of the comparisons is given later in the paper. Understandably, the transformed algorithms require more messages than the ones coded from scratch. Yet, the asymptotic complexity of both types of the algorithms was the same. We believe that the simplicity of the algorithm design using the transformation outweighs the increase in message complexity. In particular, due to recent powerful CPUs and high speed networks, the increase in message complexity is not an obstacle in many applications. Certain applications are still sensitive to the increase in message complexity. For such applications, we suggest the use of the transformation method to do rapid prototyping and then by analyzing its execution to tune up the transformed program to reduce the message complexity.

The organization of this paper is as follows: In section 2, we formally define computational models and programs. Section 3 demonstrates development of distributed SS algorithms for the serial model. Lock-based SS distributed mutual exclusion algorithms are used as case studies. To demonstrate a verification method, the section also presents a correctness proof of one of the algorithms, by following the formal method presented in [7]. Section 4 gives an SS leader election algorithm, which is based on Chang and Robert's non-SS algorithm[2], as another case study.

2 Definitions

2.1 A program for the serial model

A serial model program is represented by a five-tuple $S_s = (n, \mathcal{P}_s, \mathcal{R}_s, \mathcal{Q}_s, \mathcal{A}_s)$, where

- n is the number of nodes;
- $\mathcal{P}_s = \{\rho_1, \rho_2, \cdots, \rho_n\}$ is a set of nodes;
- $\mathcal{R}_s \subseteq \mathcal{P}_s \times \mathcal{P}_s$ is a set of dependency relations such that $(\rho_i, \rho_j) \in \mathcal{R}_s$ if ρ_j refers to the state of ρ_i in its execution;
- $\mathcal{Q}_s = (Qs_1, Qs_2, \cdots, Qs_n)$ is an n-tuple of sets of states such that Qs_i is a set of states of ρ_i; and
- $\mathcal{A}_s = (As_1, As_2, \cdots, As_n)$ is an n-tuple of algorithms, where As_i is the algorithm executed by ρ_i.

Each node ρ_i maintains a set of variables, denoted by q_i; that is, the value in q_i is a member of Qs_i. A particular variable v of a node ρ_i is denoted by $\rho_i.v$.

Suppose that ρ_i accesses states of k other nodes, $\rho_{i_1}, \cdots, \rho_{i_k}$. Then, As_i is described in the following form:

$$
\begin{array}{l}
*[\\
\qquad g_{i_1}(q_i, q_{i_1}, q_{i_2}, \cdots, q_{i_k}) \rightarrow q_i := f_{i_1}(q_i, q_{i_1}, q_{i_2}, \cdots, q_{i_k}) \\
\qquad\qquad\qquad\qquad\qquad \vdots \\
\qquad [] \; g_{i_m}(q_i, q_{i_1}, q_{i_2}, \cdots, q_{i_k}) \rightarrow q_i := f_{i_m}(q_i, q_{i_1}, q_{i_2}, \cdots, q_{i_k}) \\
]
\end{array}
$$

where

- each $g_{i_j}(\cdots) \rightarrow q_i := f_{i_j}(\cdots)$ is called a *guarded command* ($g_{i_j} \cdots$ denotes ρ_i's j^{th} guarded command);
- g_{i_j} is a predicate $Q_i \times Q_{i_1} \times \cdots \times Q_{i_k} \rightarrow \{\mathbf{true}, \mathbf{false}\}$, and $g_{i_j}(\cdots)$ is called a *guard*; and
- f_{i_j} is a function $Q_i \times Q_{i_1} \times \cdots \times Q_{i_k} \rightarrow Q_i$, $q_i := f_{i_j}(\cdots)$ is called a *command*.

If a node ρ_i has a guard g_{i_j} which is true, ρ_i is said to have a *privilege*. At any point in time, more than one node may have privileges.

The system executes the following three steps forever:

1. All the nodes evaluate all of their guards.

2. The centralized scheduler, called the *c-daemon*, nondeterministically selects one node, say ρ_i, among all nodes that have privileges.
3. Node ρ_i nondeterministically chooses one guarded command out of all the guarded commands whose associated guards are true and executes this command.

2.2 A transformed distributed model program

The distributed model accurately reflects a real distributed computing environment, such as workstations connected via Ethernet. We assume the following for the distributed model:

- There is no shared memory or global scheduler such as the c-daemon.
- For each $(\rho_j, \rho_i) \in \mathcal{R}_s$ in the serial model, there is a bidirectional FIFO communication link between the nodes that implement ρ_i and ρ_j.

A distributed model program transformed from S_s consists of n nodes $\{P_1, P_2, \cdots, P_n\}$. P_i *simulates* behavior of ρ_i of S_s. P_i maintains q_i and an array variable of n entries, denoted nq_i. Entry $nq_i[j]$ stores the value of q_j obtained via message passing. Thus, the following guarded command in As_i

$$g_{i_j}(q_i, q_{i_1}, q_{i_2}, \cdots, q_{i_k}) \rightarrow q_i := f_{i_j}(q_i, q_{i_1}, q_{i_2}, \cdots, q_{i_k})$$

is modified in the transformed distributed model program to

$$g_{i_j}(q_i, nq_i[i_1], nq_i[i_2], \cdots, nq_i[i_k]) \rightarrow q_i := f_{i_j}(q_i, nq_i[i_1], nq_i[i_2], \cdots, nq_i[i_k]).$$

Due to message delay, $nq_i[i_j]$ does not always hold the current value of q_{i_j}. Yet, the incorporated distributed SS concurrency control (CC) algorithm guarantees that for each concurrent execution of the transformed program, there always exists an equivalent serial execution, which is an execution produced by the serial model program (such a concurrent execution is said to be *serializable*[1]). Thus, if the original serial model program is correct with respect to self-stabilization, the transformed program is also self-stabilizing and performs exactly like the original program. This is achieved by considering an execution of a guarded command to be a transaction as follows:

1. k read operations, via message passing, to read q_{i_l} into $nq_i[i_l] (1 \leq l \leq k)$,
2. an internal execution (checking guards and an execution of a command), and
3. a write operation into q_i.

At the end of the transaction, node P_i tries to determine, by exchanging messages with other nodes, whether the resulting execution including the transaction is serializable. If so, the node commits the transaction and the value in q_i becomes permanent; otherwise, the transaction is aborted and the previous value is restored in q_i, as if the transaction had never been executed at all. In this way, after the SS CC algorithm stabilizes, execution of each guarded command is guaranteed to be atomic.

Refer to [11] for more details of the transformation.

3 Case study: distributed mutual exclusion algorithms

This section presents development of SS versions of lock based distributed mutual exclusion (DMX) algorithms for the serial model.

Since Dijkstra presented a token circulation DMX algorithm as the first SS algorithm in 1974[3], many token circulation SS DMX algorithms have been proposed. Some of these algorithms, including Dijkstra's algorithm, assume the serial model. Therefore, using our transformation algorithm, these algorithms can be transformed into equivalent distributed model algorithms. However, token circulation algorithms do not scale well since their synchronization delay[4] is proportional to the number of nodes in the system.

Independently, non-SS DMX algorithms have been studied extensively in the last 15 years. Most of these algorithms impose synchronization delays independent of the number of nodes in the system and therefore scale well. Almost all proposed algorithms assume a fully connected network, and much effort has been made to improve two performance characteristics: (1) the number of messages required for a critical section (CS) entry and (2) synchronization delay. Non-SS DMX algorithms are classified into two categories: token based[5] and lock based. We present lock based SS DMX algorithms for the serial model. These algorithms are developed based on existing non-SS lock based DMX algorithms. Note how simple the resulting algorithms are, compared with the original algorithms.

3.1 Preliminaries

In lock based DMX algorithms, a node obtains a certain set of locks to enter its CS[12]. Three well-known Lock-based algorithms are Lamport's algorithm[8] (denoted "Alg L"), Ricart and Agrawala's algorithm[13] (denoted "Alg R"), and Maekawa's algorithm[9] (denoted "Alg M"). In Algs L and R, each node communicates with all other nodes in the system to obtain locks. In Alg M, a node communicates with a subset of all the nodes, called a *quorum* or a *request set*. Let N be the number of nodes in the system. Algs L, R, and M require $3 \times (N-1)$, $2 \times (N-1)$, and approximately $k \times \sqrt{N}$ messages, where $3 \leq k \leq 5$, respectively, for one node to enter its CS.

In all three algorithms, each node maintains a Lamport's logical clock [8]. A pair [logical clock value, a unique node identifier (ID)] makes up a timestamp. Timestamps are totally ordered[8]. Let C_1 and C_2 be clock values and ID_1 and ID_2 be IDs. Then, the relation on two timestamps is defined as: $[C_1, ID_1] < [C_2, ID_2]$ iff $C_1 < C_2$ or ($C_1 = C_2$ and $ID_1 < ID_2$). A node associates a timestamp value to each outstanding request.

Now, let us concentrate on development of SS versions of the above three algorithms in the serial model. In all three algorithms, each node maintains its timestamp value. The ID of a node is assumed to be stored in its read-only

[4] Synchronization delay is the number of rounds of message passing required for one node to leave the CS and for another to enter the CS.

[5] Token based algorithms are different from token circulation algorithms.

incorruptible memory. Each node can *see* the current timestamp values of (1) all other nodes in Algs L and R, and (2) other nodes in its request set in Alg M.

Interestingly, in the serial model, the essence of Algs L and R boils down to the following: *a node with the smallest timestamp value among all the requesting nodes enters the CS.* Thus, requests are serviced strictly in timestamp order, and this property guarantees deadlock freedom. Therefore, in the serial model, the difference between the original algorithms disappears. We denote an SS version of such an algorithm by Alg SSLR.

In Alg M, a node can see the timestamp values of only the nodes in its request set, not all other nodes. Due to this property, requests are not necessarily serviced in timestamp order, and there is a risk of deadlock. Therefore, a complex deadlock avoidance protocol is incorporated in Alg M. The deadlock avoidance protocol works as follows: Assume that a node P_i has locked a node P_j. Then, P_j receives a lock request from another node P_k with smaller timestamp than P_i's request; that is, P_k is waiting for P_i to release its lock on P_j. This wait may be a part of a circular wait. To make sure that a circular wait is not formed, P_j inquires P_i whether it has obtained all necessary locks. If so, there is no circular wait since P_i is not waiting; otherwise, P_i relinquishes P_j's lock so that P_j can grant a lock to P_k (then, P_i will wait for P_k). In this way, only a node with a larger timestamped request waits for a node with a smaller timestamped request if neither node has obtained all the necessary locks. This guarantees deadlock freedom. We denote an SS version of Alg M by Alg SSM.

3.2 Alg SSLR in the serial model

3.2.1 Algorithm: We assume that each node ρ_i consists of two threads: an application thread A_i and a synchronization thread S_i. A_i executes an application program that issues requests to enter its CSs. S_i interacts with other S_js to obtain necessary locks. S_i is the focus of attention of this section.

A_i and S_i communicate with each other using shared variables. A_i *maintains* and *exports* a boolean variable *A_request*. This means that *A_request* is only updated by A_i but can be seen by S_i. When A_i requests for a CS entry, it sets *A_request* to true, and when it leaves the CS, it resets *A_request* to false. S_i maintains and exports a variable *mode*, that takes one of the following values: `idle` (S_i is not trying to obtain locks), `waiting` (S_i is trying to obtain locks), and `locked` (S_i has obtained all the locks).

When there is no error, *A_request* and *mode* are used to coordinate handshaking synchronization between A_i and S_i:

1. When A_i is not requesting for a CS entry, *A_request* is false and *mode* is `idle`.
2. When A_i requests for CS, A_i sets *A_request* to true and S_i sets *mode* to `waiting`.
3. When S_i has obtains the locks, S_i sets *mode* to `locked` and A_i enters its CS.
4. When A_i leaves its CS, A_i sets *A_request* to false and S_i sets *mode* to `idle`.

S_i also maintains a variable *ts* to store its timestamp and a function called **new_timestamp()** that returns a timestamp value greater than the timestamp values of any other nodes that S_i communicates with. Recall that since timestamp values are states of nodes, S_i can *see* the current timestamp values of all other nodes. Thus, **new_timestamp()** is easily implemented.

```
1 Synchronization Thread S_i of Node ρ_i
2 var
3       A_request : bool
4       mode : {idle, waiting, locked}
5       ts: timestamp

6 *[
7       A_request ∧ mode = idle →
8           ts := new_timestamp()
9           mode := waiting

10      [] mode = waiting ∧ (ts = min{ρ_j.ts | ∀ρ_j : ρ_j.mode ≠ idle}) →
11          mode := locked

12      [] ¬A_request ∧ (mode ≠ idle) →
13          mode := idle
14 ]
```

Fig. 1. Alg SSLR

The essence of the algorithm is that S_i sets *mode* to `locked` (1) if it is `waiting` and (2) if its timestamp is the smallest among all the requesting (*i.e.*, their *mode*s are not `idle`) nodes (refer to lines 10 - 11).

3.2.2 Correctness Proof: We show the correctness of Alg SSLR in the serial model. We are interested only in the correctness of the synchronization threads and do not concern ourselves with the correctness of the application threads. For A_i, we simply assume that once it enters its CS, it leaves the CS in finite time. For S_i, we assume that the underlying scheduler conducts weakly fair scheduling in selection of guarded commands. That is, when ρ_i executes statement $*[g_1 \rightarrow C_1[] \cdots []g_m \rightarrow C_m]$, if a guard g_k becomes true and thereafter remains true, the associated command C_k will be executed in finite time.

[1] Safety property

First, we define a set: $\mathcal{M} \equiv \{\rho_j.ts \mid \forall \rho_j : \rho_j.mode \neq \texttt{idle}\}$. Then, a set of legitimate states conforms to the following predicate:

$$\forall \rho_i (\rho_i.mode = locked) \Rightarrow \rho_i.ts = \mathbf{min}(\mathcal{M})$$

We use the proof technique described in [7] to prove the safety property of SSLR. For any state s, let $F(s)$ denote the cardinality of the following set $\mathcal{F}$:

$$\mathcal{F} \equiv \{\rho_j \mid \rho_j.mode = locked \wedge \rho_j.ts > \mathbf{min}(\mathcal{M})\}$$

Lemma 1. *Let s_t be any (possibly illegitimate) configuration in which $F(s_t) > 0$ holds. Let s_{t+1} be a configuration after one node executes a guarded command in s_t. Then, $F(s_t) \geq F(s_{t+1})$.*

(Proof) Let ρ_i be a node that executes a command in s_t. Node ρ_i may execute any of the three guarded commands:

1. the first guarded command: It changes $\rho_i.ts$, which in turn would change the value of F if the new $\rho_i.ts$ is smaller than the smallest timestamp of all the nodes whose mode is `locked`. However, since **new_timestamp()** return a value larger than any timestamps, $F(s_t) = F(s_{t+1})$.
2. the second guarded command: In this case, $\rho_i.ts$ is the minimum value in $\mathcal{M}$. Thus, it does not change $\mathcal{F}$, and $F(s_t) = F(s_{t+1})$.
3. the third guarded command: One node in $\mathcal{M}$ is removed. Thus, $F(s_t) \geq F(s_{t+1})$.

□

Theorem 2. *Regardless of the initial state, every execution of SSLR eventually arrives at a legitimate state (convergence) and never leaves legitimate states afterwards (closure).*

(Proof) A legitimate state s conforms to $F(s) = 0$.

Recall that (1) once a node enters its CS, its application thread is assumed to set its $A\_request$ to false in finite time and that (2) weakly fair scheduling is assumed. Thus, any node ρ_j such that $\rho_j.mode =$ `locked` executes the third guard in finite time.

Every time a node in $\mathcal{F}$ executes the third guard, it is excluded from $\mathcal{F}$, and the value of F decreases by one. By Lemma 1, F is a non-increasing function. Therefore, F will eventually become 0 and continue to be 0 for the rest of the execution. □

[2] Liveness property:

Theorem 3. *Once the system reaches a legitimate state, any node requesting to enter its CS will eventually be allowed to do so.*

Since we assume that (1) an application thread in its CS will leave the CS in finite time; (2) function **new_timestamp()** returns a timestamp greater than timestamps of any other nodes; and (3) the requests are granted strictly in the timestamp order (obvious from the second guard), the theorem holds.

3.2.3 Performance: We evaluate the transformed version of Alg SSLR in terms of the average number of required messages per critical section entry. There are two factors to be considered in transformed programs with respect to the message complexity[11]:

- Each node sends its state to neighbor nodes at least once every τ time.
- For a node to change its state (to commit), at least three messages are exchanged between itself and each of other nodes.

Our transformation algorithm is based on optimistic concurrency control: if a node has a privilege, it executes a corresponding command as a transaction, assuming it would successfully commit. Then, it sends the new state to neighbor nodes asking that the update can actually be committed. Each neighbor node sends a reply that indicates whether or not it accepts the update. This decision is based on timestamp values that the nodes maintain. If all neighbor nodes accept the update, the transaction is committed and the node notifies its new state to its neighbor nodes. Otherwise, the transaction is aborted. Thus, at least 3 messages are necessary for each state change and extra messages are considered to be overhead. Since a node changes its *mode* three times per each critical section entry (`idle` to `waiting` to `locked` to `idle`) and it must exchange messages with $N-1$ nodes, the message complexity is approximately $3 \times 3 \times (N-1)$ per critical section entry in the optimal case.

We have implemented the transformed version of Alg SSLR on our distributed/parallel development system [10] to measure its message complexity. The parameters for the simulation of Alg SSLR is as follows: (1) the number of nodes N is 7, (2) transmission latency of communication links (denoted TL): 2 time units, (3) execution time of each guarded command: 0.2 time units, (4) execution time in critical section: 10 time units, and (5) execution time in non-critical section: 20 time units. Note that initial states of nodes are defined by uniform random numbers, and that parameters (2)–(5) use exponential distribution with the given average values.

Simulation is done for $\tau = 2 \times TL, 10 \times TL, 50 \times TL, 250 \times TL, 1000 \times TL$. The results are shown in Figure 3.2. We repeated the execution ten times for each condition and the results are the average of the ten trials. The figures show the average number of messages per critical section entry (y axis) for the duration of 5000 simulation time units (x axis). When τ is small ($\tau \leq 10 \times TL$), the number of messages is large since nodes frequently send their states to neighbor nodes. On the other hand, when τ is large ($\tau \geq 50 \times TL$), the system converges (after 3000 time units from the beginning of the simulation) to a configuration such that the number of messages per critical section entry is slightly over 70. Thus, nodes exchange averages of approximately 4 messages per state change (which is greater than the optimal case of 3).

We have developed SS versions of node locking and link locking DMX algorithms from scratch without using the transformation method[12]. These specialized algorithms require slightly more than $3 \times (N-1)$ messages per critical section entry, which are the same as the original Alg L requires (compared with measured $4 \times 3 \times (N-1)$ for Alg SSLR).

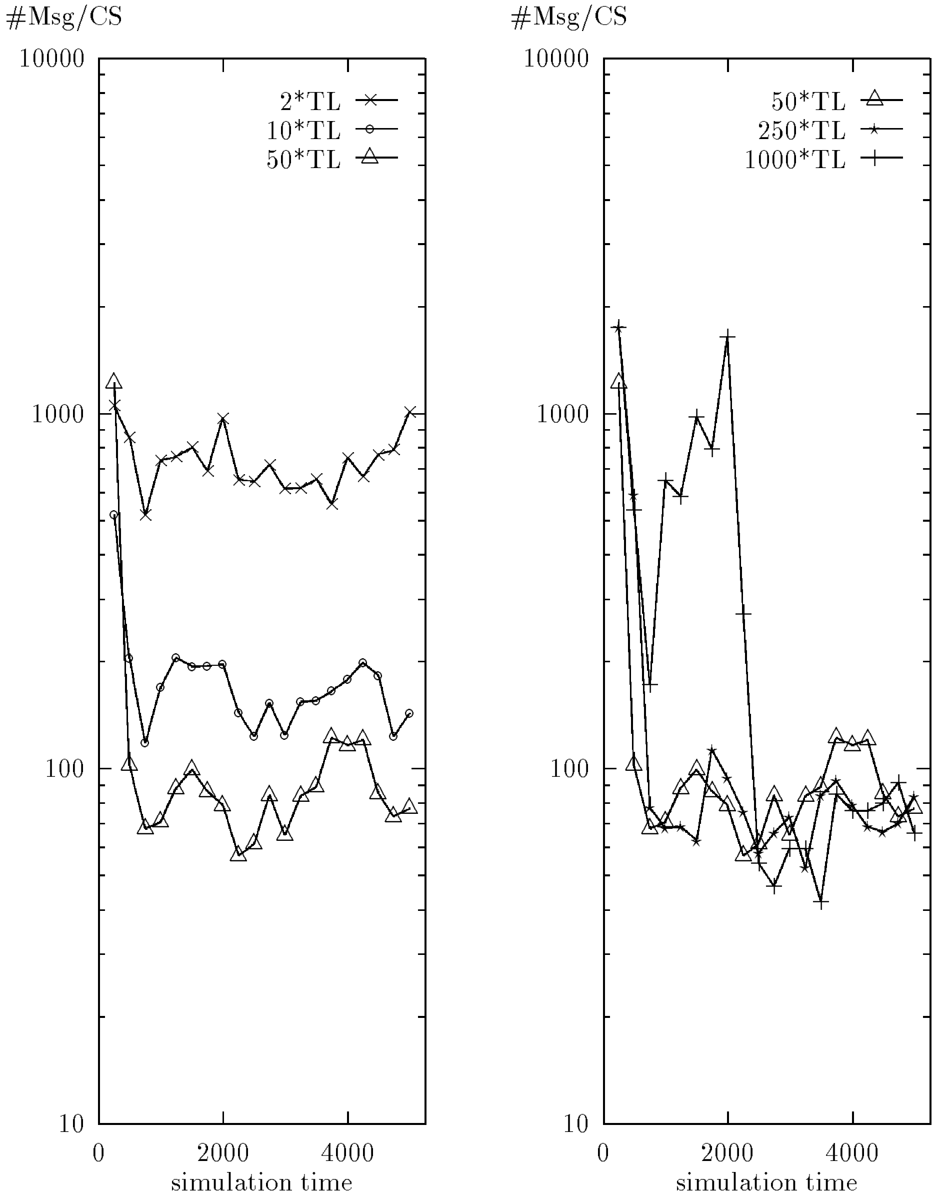

Fig. 2. Relationship between τ and the number of messages per CS entry

Even though the algorithms coded from scratch exhibit better performance in terms of message complexity, our experience suggests that they were much more difficult to develop, debug, and verify, compared with Alg SSLR. We believe that the simplicity and reliability of resulting code make the design of SS distributed algorithms using the transformation a preferable alternative.

3.3 Alg SSM

Synchronization Thread S_i **of Node** ρ_i
const
  R_i /* ρ_i's request set */
  A_i /* ρ_i's acquired set $A_i \equiv \{\rho_j \mid \rho_i \in R_j\}$ */

var
  $A\_request$: **bool**
  $mode$: {`idle`, `waiting`, `locked`}
  ts : **timestamp**
  who : **node_id** /* a node that obtained ρ_i's lock */

*[
  $A\_request \wedge (mode = \texttt{idle}) \wedge (\forall \rho_j \in R_i \; \rho_j.who \neq \rho_i) \rightarrow$
    ts:= **new_timestamp()**
    $mode$:= `waiting`

  [] $mode = \texttt{waiting} \wedge (\forall \rho_j \in R_i[\rho_j.who = \rho_i]) \rightarrow$
    $mode$:= `locked`

  [] $\neg A\_request \wedge (mode \neq \texttt{idle}) \rightarrow$
    $mode$:= `idle`

  [] $(who = \perp \vee (who = \rho_j \wedge \rho_j.mode \neq \texttt{locked})) \wedge$
  $(\exists \rho_k \in A_i$ such that $who \neq \rho_k$ and
  $\rho_k.ts = \mathbf{min}\{\rho_l.ts \mid \forall \rho_l \in A_i : \rho_l.mode \neq \texttt{idle}\}) \rightarrow$
    $who := \rho_k$

  [] $who = \rho_j \wedge \rho_j.mode = \texttt{idle} \rightarrow$
    $who := \perp$
]

Fig. 3. Alg SSM

In Alg SSM, in addition to *mode* and timestamp *ts*, each synchronization thread S_i maintains variable *who* that stores the identifier of the node that has locked S_i; or $\perp$ if S_i is not locked. The first three guards are similar to those of Alg SSLR. Instead of checking whether the node has the smallest timestamp among all the requesting nodes, S_i checks whether all the nodes in its request set are locked by ρ_i ($\forall \rho_j \in R_i[\rho_j.who = \rho_i]$).

The fourth guard handles lock requests from other nodes. Condition ($who = \rho_j \wedge \rho_j.mode \neq \texttt{locked}$)$\wedge$ ($\exists \rho_k \in A_i$ such that $who \neq \rho_k$ and $\rho_k.ts = \mathbf{min}\{\rho_l.ts \mid \forall \rho_l \in A_i : \rho_l.mode \neq \texttt{idle}\}$) is for deadlock avoidance. Even though the node itself is locked by a node ρ_j, if ρ_j has not obtained all the necessary locks yet ($\rho_j.mode \neq \texttt{locked}$), S_i may change who to the node with the smallest timestamped request.

Condition ($\forall \rho_j \in R_i \ \rho_j.who \neq \rho_i$) in the first guard (line 11) and condition ($who = \rho_j \wedge \rho_j.mode = \texttt{idle}$) in the fifth guard (line 20) are to prevent a situation described below: Assume that the condition in line 11 is removed and that right after a node ρ_i leaves its CS, it requests to enter the CS again. If none of the nodes in R_i have noticed the change of $\rho_i.mode$ from `locked` to `idle` and ρ_i executes the first guarded command to set $\rho_i.mode$ to `waiting` again, rho_i can immediately execute the second guarded command (lines 14 - 15) and enter the CS. The conditions in the first and the fifth guards prevent such a situation.

Finally, implementation of **new_timestamp()** deserves special attention. Unlike SSLR, each node in SSM communicates only with other nodes in its request and acquired sets (ref: Fig. 3) and does not communicate with the rest of the nodes. To guarantee propagation of timestamp values among nodes, **new_timestamp()** may be implemented as follows: every node ρ_i maintains a variable max_ts that holds the maximum value of the timestamps found in the nodes in its acquired set A_i. (Update of max_ts may be done in the third guarded command (lines 18-19) when ρ_i checks the minimum ts value in its A_i.) When a node ρ_i requests for its CS entry, **new_timestamp()** returns a value larger than any max_ts values found in the nodes in its request set R_i.

Alg SSM is very simple compared with the original non-SS Alg M. This is particularly true for the deadlock avoidance part.

4 Another Example: SS Leader Election

As another case study, this section presents an SS leader election algorithm (Alg SSCR). The algorithm is based on Chang and Robert's leader election algorithm[2] for unidirectional ring networks.

It is assumed that each node maintains a unique node identifier (ID) and integer N in its read-only incorruptible memory. N is chosen to be larger than the number of nodes in the system and is used to stabilize the system. In addition, each node maintains variable max to keep the largest node ID that it has seen. The largest max value is propagated along the ring. If node ρ_i finds that its neighbor holds ρ_i's ID in its max (*i.e.,* ρ_i's ID is propagated through all other nodes), ρ_i has the maximum ID and becomes the leader.

In our self-stabilizing version, each node also maintains a variable $dist$ which stores a distance to itself from the node with the maximum ID. The variable is used to eliminate a wrong *max* value from the system. If a node, say ρ_i, holds in its *max* a value *Wrong_ID* which is larger than any IDs, *Wrong_ID* is propagated to other nodes. However, eventually *dist* of each node will become larger than N and *Wrong_ID* will be discarded.

Node ρ_i
const
    ID : **node_id** /* node ID of ρ_i */
var
    max : **node_id**
    $dist$: **integer**
    $leader$: **bool**
[/ ρ_{i-1} denotes the left neighbor of node ρ_i in the ring */
    $(\rho_{i-1}.max > ID) \wedge (\rho_{i-1}.dist < N) \rightarrow$
        $max := \rho_{i-1}.max$
        $dist := \rho_{i-1}.dist + 1$
        $leader :=$ **false**
    $[](\rho_{i-1}.max \leq ID) \wedge (\rho_{i-1}.dist < N) \rightarrow$
        $max := ID$
        $dist := 0$
        $leader :=$ **true**
    $[]\ (\rho_{i-1}.dist \geq N) \rightarrow$
        $max := ID$
        $dist := 0$
        $leader :=$ **false**
]

Fig. 4. Alg SSCR

5 Conclusion

This paper demonstrated the effectiveness of our transformation algorithm to develop new and practical SS distributed algorithms. We begin a development with a serial model algorithm, by considering the execution environment, such as the connectivity of the network, and prove it correct using a well-established verification method for serial model algorithms. The resulting algorithm is then transformed to an equivalent SS distributed model algorithm by the transformation algorithm. This method significantly simplifies development of SS distributed algorithms since (1) developing an algorithm for the serial model is like developing an algorithm for a shared memory environment, (2) non-interference is guaranteed inside guarded commands, and (3) channel states do not have to be considered. As case studies, we presented SS versions of lock based DMX algorithms and a leader election algorithm. We implemented one of the SS DMX algorithms in two ways, one using the transformation anther directly from scratch, to conduct performance evaluation. The results indicate that both types of algorithms show the same asymptotic complexity.

References

1. Bernstein, P.A, Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Co., 1987.
2. E. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22:281–283, 1979.
3. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.
4. Gouda M.G. and Multari M.J. Stabilizing communication protocols. *IEEE Tran. on Computers*, 40(4):448–458, 1991.
5. S.-T. Huang, L.-C. Wuu, and M.-S. Tsai. Distributed execution model for self-stabilizing systems. In *Proceedings of the International Conference of Distributed Computing Systems (ICDCS '96)*, pages 432–439, 1994.
6. T. Kameda and M. Yamashita. *Distributed Algorithms.* Kindai Kagakusya, 1995. (in Japanese).
7. J. L. W. Kessels. An exercise in proving self-stabilization with a variant function. *Information Processing Letters*, 29(1):39–42, September 1988.
8. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
9. M. Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, March 1985.
10. M. Mizuno. A distributed/parallel algorithm development system. Kansas State University, 1994. (software).
11. M. Mizuno and H. Kakugawa. A transformation of self-stabilizing programs for distributed computing environments. In *Workshop on Distributed Algorithms (WDAG '96), LNCS*, pages 304–321. Springer Verlag, October 1996.
12. M. Mizuno, M. Nesterenko, and H. Kakugawa. Lock based self-stabilizing distributed mutual exclusion algorithms. In *Proceedings of the International Conference of Distributed Computing Systems (ICDCS '96)*, pages 708–716, May 1996.
13. G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer network. *Communications of the ACM*, 24(1):9–17, January 1981.