A Transformation of Self-Stabilizing Serial Model Programs for Asynchronous Parallel Computing Environments

Masaaki Mizuno   Mikhail Nesterenko

Department of Computing and Information Sciences
Kansas State University
Manhattan, KS 66506, USA
{masaaki,mikhail}@cis.ksu.edu

Keywords: cuncrrency, fault tolerance, parallel algorithms, self-stabilization

1 Introduction

In 1974, Dijkstra presented the notion of self-stabilization in the context of distributed computing[5]. A system is *self-stabilizing (SS)* 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, an SS system need not be initialized and is able to recover from transient failures by itself.

Many SS programs have been developed for different models with various assumptions about their execution environments. These assumptions include the semantics of concurrency and communication primitives. Among these models, a serial model (C-daemon model) has the strongest assumptions. In the serial model, an atomic execution step consists of (1) a read sub-step, which reads the states of its neighbor processes; followed by (2) a write sub-step, which modifies its own state (based on the neighbors' current states and its own state). The communication and concurrency semantics are such that each process can always see the current states of its neighbors, and only one process at a time executes an atomic step.

As seen above, the assumptions on the execution environment of the serial model are much stronger than those of common parallel/distributed systems, and as a result, programs developed for the serial model (called *serial-model programs*) do not run directly on real parallel/distributed systems. One approach to execute serial-model programs on real hardware is to synchronize their executions using SS mutual exclusion (MX) algorithms; for example, [3, 4, 7] for parallel environments and [12] for distributed environments. A process holding an exclusive right executes its read and write sub-steps. In this way, the atomicity of execution steps is guaranteed. However, this approach forces execution to be serial and does not fully utilize the capability of parallel/distributed hardware.

Recently, work has been done to exploit concurrency in execution of serial-model programs. In 1996, as the first of such attempts, we presented an algorithm that transforms SS serial-model programs to equivalent SS programs that run concurrently in a message passing distributed computing environment[11]. For synchronization, an SS distributed concurrency control (CC) protocol is incorporated in the transformed program.

In 1997, Gouda and Haddix presented an algorithm that transforms SS serial-model programs for a shared memory parallel computing environment[8]. The algorithm uses linear alternators to synchronize concurrently running processes. At the time of the publication, they had completed a transformation for parallel systems with linear topology and were working on systems with more general topologies.

In this paper, we present an algorithm that transforms an SS serial-model program to an equivalent SS program that runs on an asynchronous shared memory parallel computing system, which provides essentially the same execution environment as the model proposed in [7]. Unlike Gouda and Haddix's transformation, our algorithm transforms a serial-model program with an arbitrary logical topology and preserves the *silence* property[1]; that is, if the original serial-model program is silent, the transformed program is also silent. However, parallel programs produced by our transformation require more memory than those by Gouda and Haddix's transformation.

The organization of this paper is as follows: Section 2 gives definitions of programs for the serial model and for the asynchronous parallel computing environment. Our transformation algorithm is presented in Section 3 and its correctness proof is given in Section 4. Some additional remarks are given in Section 5.

2 Definitions

2.1 A program for the serial model

A serial-model program S_s consists of n processes $\{\rho_1, \rho_2, \cdots, \rho_n\}$. Each process ρ_i has a unique identifier (i is assumed to be the identifier of ρ_i) and executes algorithm As_i. The identifiers and algorithms are assumed to be incorruptible.

[1] A stabilizing algorithm is *silent* if it converges to a global state after which the values stored in the communication registers are fixed [6]

Process ρ_i maintains a variable q_i to store its local state and updates q_i based on the local states of itself and other processes. If ρ_i reads q_j of ρ_j in its execution, ρ_i and ρ_j are said to be *neighbors*. We use N_i to denote a set of all neighbor processes of ρ_i. Note that $\rho_i \in N_i$ since ρ_i reads q_i.

Suppose that ρ_i reads local states of k other processes, $\rho_{i1}, \cdots, \rho_{ik}$, as well as its own state. Then, algorithm As_i is described in the following form of guarded commands:

$$
\begin{array}{l}
*[\\
\qquad\qquad\qquad\qquad \vdots \\
\quad [] g_{ij}(q_i, q_{i1}, q_{i2}, \cdots, q_{ik}) \rightarrow q_i := f_{ij}(q_i, q_{i1}, q_{i2}, \cdots, q_{ik}) \\
\qquad\qquad\qquad\qquad \vdots \\
]
\end{array}
$$

Execution of each guarded command is assumed to be completed in finite time.

If a process ρ_i has a guard g_{ij} which is true, ρ_i is said to have a *privilege*. The system executes the following three steps forever:

1. All the processes evaluate all of their guards.
2. The centralized scheduler, called the *C-daemon*, nondeterministically selects one process, say ρ_i, among all processes that have privileges.
3. Process ρ_i nondeterministically chooses one guarded command out of all of its guarded commands whose associated guards are true and executes its command.

2.2 A transformed asynchronous parallel program

A parallel program transformed from S_s consists of n processes $\{P_1, P_2, \cdots, P_n\}$. P_i *simulates* the behavior of ρ_i of S_s. N_i is defined in the same way as for the original serial-model program — just replacing ρ_j with P_j for any j.

We assume the following for the target environment:

- There is no global scheduler such as the C-daemon.
- Each process P_i has a set of *communication registers*, denoted C_i, which are written only by P_i but read by all processes. An atomic operation on communication registers is either a single read operation or a single write operation on one communication register (*i.e.,* the read/write atomicity[7] is assumed).
- Weakly fair scheduling is assumed within each process; that is, if a guard becomes true and remains true thereafter, the guarded command is eventually executed.

3 Transformation

3.1 Preliminaries

In order for the transformed program to work correctly, we must guarantee that each execution of the transformed program is *serializable* [2]; that is, for every execution of the transformed program, there always exists an *equivalent* execution of the original serial-model program. A concurrency control (CC) protocol guarantees that every execution of a given concurrent program is serializable.

Serialization of the transformed program is achieved by viewing an execution of a guarded command "$g_{ij}(q_i, q_{i1}, q_{i2}, \cdots, q_{ik}) \rightarrow q_i := f_{ij}(q_i, q_{i1}, q_{i2}, \cdots, q_{ik})$" to be a transaction, consisting of (1) $k+1$ read operations to read q_i in C_i and q_{ij} in C_{ij} $(1 \leq j \leq k)$, (2) an internal execution (checking guards and an execution of a command), and (3) a write operation into q_i in C_i.

The simplest form of CC is to guarantee that P_i and P_j do not execute their transactions (*i.e.*, guarded commands) simultaneously if they are neighbors. The algorithm presented in this paper is based on this principle. If other forms of CC are employed, it is possible that even neighbors execute concurrently[11].

Note that as shown in [11], the incorporated CC algorithm must also be SS. After the CC protocol stabilizes, for the rest of the execution of the transformed program, there exists an equivalent execution of the original SS serial-model program; thus, the execution of the transformed program as a whole is guaranteed to stabilize.

3.2 Transformation algorithm

In addition to state q_i, Each process P_i maintains the following three variables in its communication registers C_i: (1) q_i (state of P_i), (2) $ready_i$ (boolean), and (3) c_i (timestamp counter — integer). A pair (c_i, i) constitutes a timestamp ts_i, where i is the process identifier. When comparing two timestamps ts_i and ts_j, $ts_i < ts_j$ if $((c_i < c_j) \vee ((c_i = c_j) \wedge (i < j))$. Thus, timestamps are totally ordered. We assume that one timestamp counter is stored in a single communication register.

Note that since the initial state of an execution is arbitrary in an SS program, if a timestamp is used, it is not exempt from an overflow no matter how large a timestamp counter is. Several non-SS algorithms to implement bounded timestamps have been proposed [10]. However, SS versions of these algorithms are yet to be developed. Here, we use reset algorithms that work in read/write atomicity[1] as a solution for this problem. When a process detects an overflow, it triggers a reset algorithm to reset all the timestamp counters in the system. Section 4 discusses correctness of the execution after a reset is applied.

$$
\begin{array}{ll}
\text{t1} & \textbf{processor } P_i \\
\text{t2} & *[\\
\text{t3} & \qquad \vdots \\
\text{t4} & \qquad [\,]\, ready_i \wedge ts_i = \textbf{min\_ts}(X_i) \wedge \\
\text{t5} & \qquad\quad g_{ij}(q_i, q_{i1}, q_{i2}, \cdots, q_{ik}) \rightarrow \\
\text{t6} & \qquad\qquad q_i := f_{ij}(q_i, q_{i1}, q_{i2}, \cdots, q_{ik}) \\
\text{t7} & \qquad\qquad ready_i := \textbf{false} \\
\text{t8} & \qquad \vdots \\
\text{t9} & \qquad [\,](\cdots \vee g_{ij}(q_i, q_{i1}, q_{i2}, \cdots, q_{ik}) \vee \cdots) \wedge \neg ready_i \rightarrow \\
\text{t10} & \qquad\qquad ready_i := \textbf{true} \\
\text{t11} & \qquad\qquad ts_i := \textbf{new\_ts}(N_i) \\
\text{t12} & \qquad [\,]\neg(\cdots \vee g_{ij}(q_i, q_{i1}, q_{i2}, \cdots, q_{ik}) \vee \cdots) \wedge ready_i \rightarrow \\
\text{t13} & \qquad\qquad ready_i := \textbf{false} \\
\text{t14} &]
\end{array}
$$

Figure 1: transformed program

The algorithm transforms As_i (for $1 \leq i \leq n$) to the program presented in Figure 1. Each guarded command in As_i, "$g_{ij}(q_i, q_{i1}, \cdots, q_{ik}) \rightarrow q_i := f_{ij}(q_i, q_{i1}, \cdots, q_{ik})$", is replaced by a guarded command described in lines $t4 - t7$. Two new guarded commands given in lines $t9 - t11$ and $t12 - t13$ are added. We call a guarded command in As_i an "original guarded command" if we need to distinguish it from a guarded command in the transformed program. We say that P_i has a privilege if any of the original guards are true.

X_i denotes a set $\{P_k | P_k \in N_i \wedge ready_k\}$. Function $\textbf{min\_ts}(X_i)$ examines all ts_j such that $P_j \in X_j$ in arbitrary order and returns the smallest timestamp among them. Function $\textbf{new\_ts}(N_i)$ examines all ts_j such that $P_j \in N_i$ in arbitrary order and returns a value greater than all of them. More precisely, the function returns a new counter value to be stored in c_i; nonetheless, we use "$ts_i := \textbf{new\_ts}(N_i)$" to denote the update of c_i.

In an execution of the transformed program, if P_i has a privilege, guarded command $t9 - t11$ is selected for execution. It sets $ready_i$ to true to join X_j such that $P_j \in N_i$ and obtains a new timestamp. When P_i's timestamp becomes the smallest among those of its neighbors that have privileges and P_i still has a privilege, guarded command $t4 - t7$ is eventually selected (weakly fair scheduling is assumed). Then, P_i executes the associated command and resets $ready_i$ to false to leave X_j such that $P_j \in N_i$. This guarantees that only one process among all the neighbors executes its command. If none of its original guards are true and $ready_i = true$, P_i resets its $ready_i$ to false [lines $t12$ - $t13$]. Once $ready_i$ becomes false, it will not execute any command until at least one of its original guards becomes true. From this, it is clear that if the original serial-model algorithm is silent, the transformed algorithm is also silent.

4 Correctness

By following [9], we define a predicate $\mathcal{T}$ on the global states of an execution H of the algorithm such that the safety and liveness property holds if $\mathcal{T}$ is true. Then, we prove that the algorithm is self-stabilizing with respect to $\mathcal{T}$ by showing (1) *convergence*: $\mathcal{T}$ holds in finite time t in any execution H, and (2) *closure*: $\mathcal{T}$ remains true after t in H. In this paper, to simplify the proof, we allow $\mathcal{T}$ to include the notion of time.

Since the initial value in the program counter is arbitrary, the execution may begin in the middle of any guarded command. Since we assume that execution of any guarded command terminates in finite time, the control will go back to the beginning of the main loop [line $t2$] in finite time. Let $t_{\mathcal{T}}$ be the earliest time when control of every process has reached the beginning of its main loop at least once.

Definition $\mathcal{T}$: Once a process P_i has detected $ts_i = \mathbf{min\_ts}(X_i)$ [line $t4$] (let α_i denote the value of ts_i at this moment), no other process $P_j \in N_i$ adds ts_j that is smaller than α_i to the set of timestamps $\{ts_k | P_k \in X_i\}$ by executing $ts_j := \mathbf{new\_ts}(N_j)$ [line $t11$].

Lemma 1 A transformed program is self-stabilizing with respect to $\mathcal{T}$.

Proof: We show that $\mathcal{T}$ always holds after $t_{\mathcal{T}}$ (convergence and closure). Let α_j be the timestamp value assigned to ts_j by the execution of $\mathbf{new\_ts}(N_j)$ by P_j. Note that after $t_{\mathcal{T}}$, α_j is larger than any timestamps that $\mathbf{new\_ts}(N_j)$ observes in N_j (that includes P_i and P_j). There are two cases to consider:

1. When P_j starts executing $ts_j := \mathbf{new\_ts}\ (N_j)$ [line $t11$] (that will assign α_j to ts_j), P_i has completed execution of $ts_i := \mathbf{new\_ts}\ (N_i)$ [line $t11$] (that has assigned α_i to ts_i): In this case, α_j is larger than α_i. Therefore, $\mathcal{T}$ holds.

2. When P_j started executing $ts_j := \mathbf{new\_ts}\ (N_j)$ (*i.e.*, $ts_j = \alpha_j$), P_i had not completed execution of $ts_i := \mathbf{new\_ts}\ (N_i)$ (*i.e.*, $ts_i = \alpha_i$): α_j is either larger or smaller than α_i. When P_i checks the expression $ts_i = \mathbf{min\_ts}(X_i)$ [line $t4$],

   (a) P_j has not yet assigned α_j to ts_j: P_i reads the previous value, say α'_j, in ts_j. If $\alpha'_j < \alpha_i$, P_i does not find ($ts_i = \mathbf{min\_ts}(X_i)$) to be true, and $\mathcal{T}$ holds. If $\alpha_i < \alpha'_j$ and P_i finds ($ts_i = \mathbf{min\_ts}(X_i)$) to be true, the new ts_j value (α_j) is even larger than α'_j, and therefore $\mathcal{T}$ holds.

(b) P_j has already assigned α_j to ts_j: If $\alpha_i < \alpha_j$, $\mathcal{T}$ holds. If $\alpha_j < \alpha_i$, P_i does not find the expression $(ts_i = \mathbf{min\_ts}(X_i))$ to be true, and $\mathcal{T}$ holds.

From the above, after $t_{\mathcal{T}}$, once P_i has detected $ts_i = \mathbf{min\_ts}(X_i)$ (*i.e.* $ts_i = \alpha_i$), $\alpha_i < \alpha_j$ holds for any timestamp value α_j assigned after this. Therefore, the lemma holds. □

Theorem 1 (Safety) After $\mathcal{T}$ holds, no two processes P_i and P_j such that $P_j \in N_i$ (*i.e.,* $P_i \in N_j$) execute their original guarded commands [lines $t5-t6$] simultaneously.

Proof: On the contrary, assume that P_i and P_j execute their original guarded commands simultaneously. For process P_i to execute an original guarded command, it must check that expression "$ts_i = \mathbf{min\_ts}(X_i)$" [line $t4$] holds. While $\mathcal{T}$ holds, once P_i has detected the expression to be true, P_j cannot obtain ts_j smaller than ts_i. The same discussion applies to P_j. Since $\{P_i, P_j\} \subseteq X_i \cap X_j$ and $ts_i \neq ts_j$, this is a contradiction. □

As mentioned in Section 3, we assume that a process P_i triggers an SS reset algorithm (such as in [1]) when it detects an overflow in its timestamp counter c_i. When a process P_j responds to a reset request, it (1) clears c_j, (2) sets $ready_j$ to false, and (3) initializes its program counter to the beginning of the guarded command loop [line $t2$]. Then, $\mathcal{T}$ holds immediately after the reset algorithm terminates. During the reset process, $\mathcal{T}$ may be temporarily violated.

Theorem 2 (Liveness) After $\mathcal{T}$ holds, if any process P_i with a privilege has executed the guarded command in lines $t9 - t11$ and continues to keep the privilege, $ts_i = \mathbf{min\_ts}(X_i)$ will eventually hold.

Proof: Suppose $ts_i > \mathbf{min\_ts}(X_i)$. P_i is said to *wait for* processes $P_j \in X_i$ such that $ts_i > ts_j$. Each of such P_j in turn may wait for other processes $P_k \in X_j$ such that $ts_j > ts_k$. Note that X_i and X_j may not be the same. This relation forms P_i's *wait-for graph* (there is a wait-for edge from P_x to P_y in the graph if $ts_x > ts_y$). Since the timestamps are totally ordered, this graph is acyclic. No new wait-for edge can be created from any process P_j in the graph to any other process P_k which calls $\mathbf{new\_ts}(N_k)$ [line $t11$] after ts_j is set, since the function returns a value larger than ts_j. Therefore, a finite number of processes will be added to the graph.

Let process P_s be a sink in the graph. There are three cases to consider:

1. P_s has a privilege: Since P_s is a sink, $ts_s = \mathbf{min\_ts}(X_s)$ holds. Since weakly fair scheduling is assumed and $\mathcal{T}$ holds, P_s will eventually execute its original guarded command and then reset $ready_s$ to false to leave the graph;

2. P_s does not have a privilege: It will execute the guarded command $t12 - t13$ and reset $ready_s$ to false; or

3. P_s has just executed line $ready_s := true$ [line $t10$] to temporarily enter the graph: P_s will immediately execute line $t11$ to obtain a new timestamp ts_s to leave the graph.

The above cases 2 and 3 may also apply to any other non-sink process in the wait-for graph; if so, such a process leaves the graph.

Since there are a finite number of processes in P_i's wait-for graph, by repeating the above discussion on sink processes in the graph, $ts_i = \mathbf{min\_ts}(X_i)$ will eventually hold. □

5 Discussion

Even though our approach prevents only neighbor processes from executing concurrently, there are cases in which a process is blocked by a non-neighbor process. Let P_i and P_j be neighbors and P_j and P_k be neighbors (P_i and P_k are not neighbors). Assume (1) all three processes have privileges and (2) $ts_i < ts_j < ts_k$. If P_i is slow to complete its execution, then P_j has to wait and as a result, P_k has to wait.

It is possible to apply multiple execution instances of an existing SS MX algorithm [3, 4, 7] among different neighbors to allow concurrent executions. For example, the algorithm in [7] is flexible and may be suitable for this approach. Resulting algorithms may exhibit better memory complexity than algorithms produced by our transformation (which use timestamps). However, in such an approach, deadlock prevention may be difficult in an arbitrary logical topology, and the resulting programs could become more complex than those produced by our algorithm.

6 Conclusion

This paper presented an algorithm that transforms an SS serial-model program to an equivalent SS program that runs concurrently in an asynchronous shared memory parallel computing environment. Timestamps are used to guarantee mutually exclusive execution of guarded commands among neighbor processes. The algorithm preserves the silence property.

Acknowledgments

We would like to thank Professor Mohamed G. Gouda for his valuable comments and suggestions.

References

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

[2] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Co., 1987.

[3] G.M. Brown, M.G. Gouda, and C.L. Wu. Token systems that self-stabilize. *IEEE Transactions on Computers*, 38:845–852, 1989.

[4] J.E. Burns. Self-stabilizing rings without daemons. Technical Report GIT-ICS-87/36, Georgia Tech, 1989.

[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, M.G. Gouda, and M. Schneider. Memory requirements for silent stabilization. In *In Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC96)*, pages 27–34, 1996.

[7] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.

[8] M. Gouda and F. Haddix. The linear alternator. In *Proceedings of the 3rd Workshop on Self-Stabilization (WSS'97)*, pages 31–47, 1997.

[9] Gouda M.G. and Multari M.J. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4):448–458, 1991.

[10] A. Israeli and M. Li. Bounded time-stamps. In *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, pages 371–382, 1987.

[11] M. Mizuno and H. Kakugawa. A transformation of self-stabilizing programs for distributed computing environments. In *Workshop on Distributed Algorithms (WDAG '96), LNCS:1151*, 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.