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

by

Hirotsugu Kakugawa
Masaaki Mizuno
Mikhail Nesterenko

Problem Statement

- development and verification of SS algorithms are complex and error prone;
- many SS algorithms proposed in the literature place strong assumptions on the execution environment (*serial model*);
- common distributed systems allow only minimum assumptions about the execution environment (*distributed model*). Serial model algorithms cannot run.

Solution

We presented a program that provides automatic translation of an SS algorithm from serial to distributed model (*WDAG'96*).

This transformation algorithm allows to execute many SS algorithms developed for serial model in distributed computing environment.

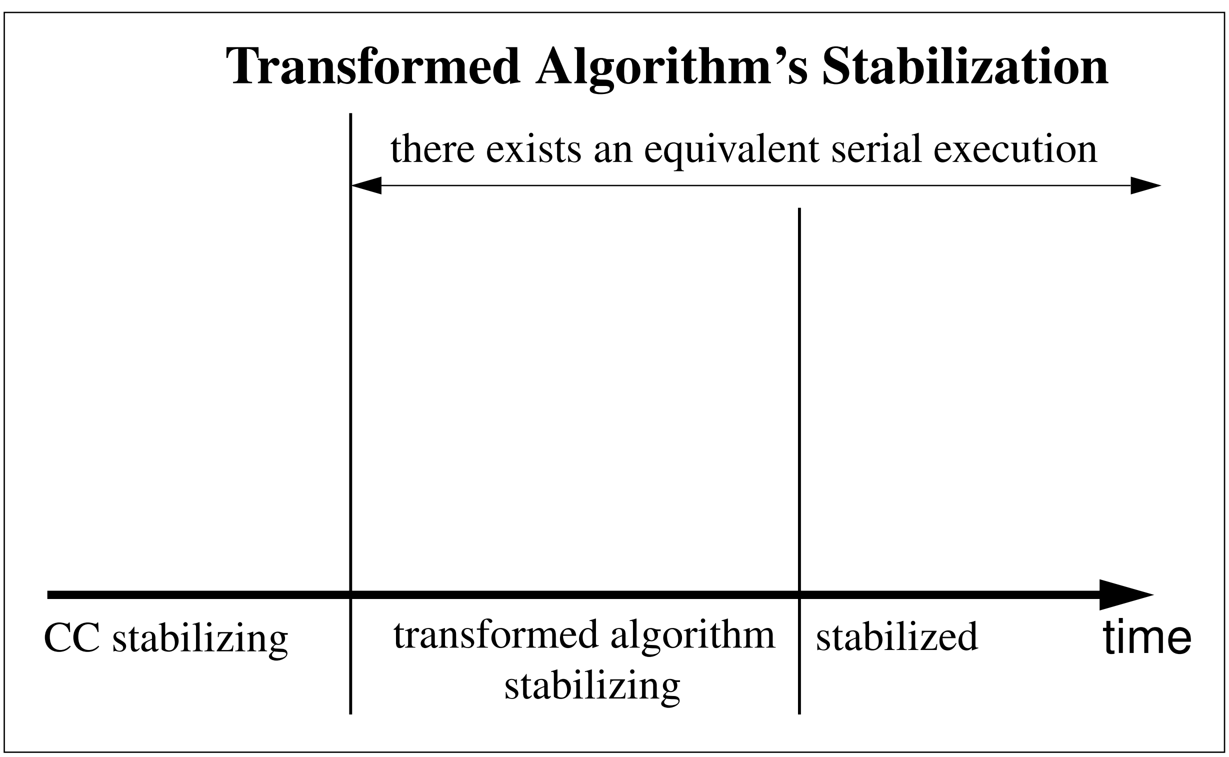
More importantly, the transformation helps to develop new practical SS algorithms. As an example we present lock-based distributed mutual exclusion algorithms (DMX) developed for serial model:

- Lamport's /Rikart-Agrawala's DMX;
- Maekawa's DMX.

Unlike many proposed SS token-circulation DMX algorithms, lock-based DMX algorithms provide constant synchronization delay and scale well.

Transformation Algorithm

- it provides automatic transformation of a self-stabilizing algorithm from serial to distributed model;
- CC is incorporated into target transformed algorithm: execution of guarded command of the transformed distributed model algorithm is considered a transaction; for every execution of the distributed transformed algorithm there is an equivalent serial execution of the original serial algorithm;
- CC mechanism is self-stabilizing



Computing Environment Models

Serial Model

- guarded commands of all processors in the system are executed serially (C-Daemon);
- each processor can see the variables of its neighbors.

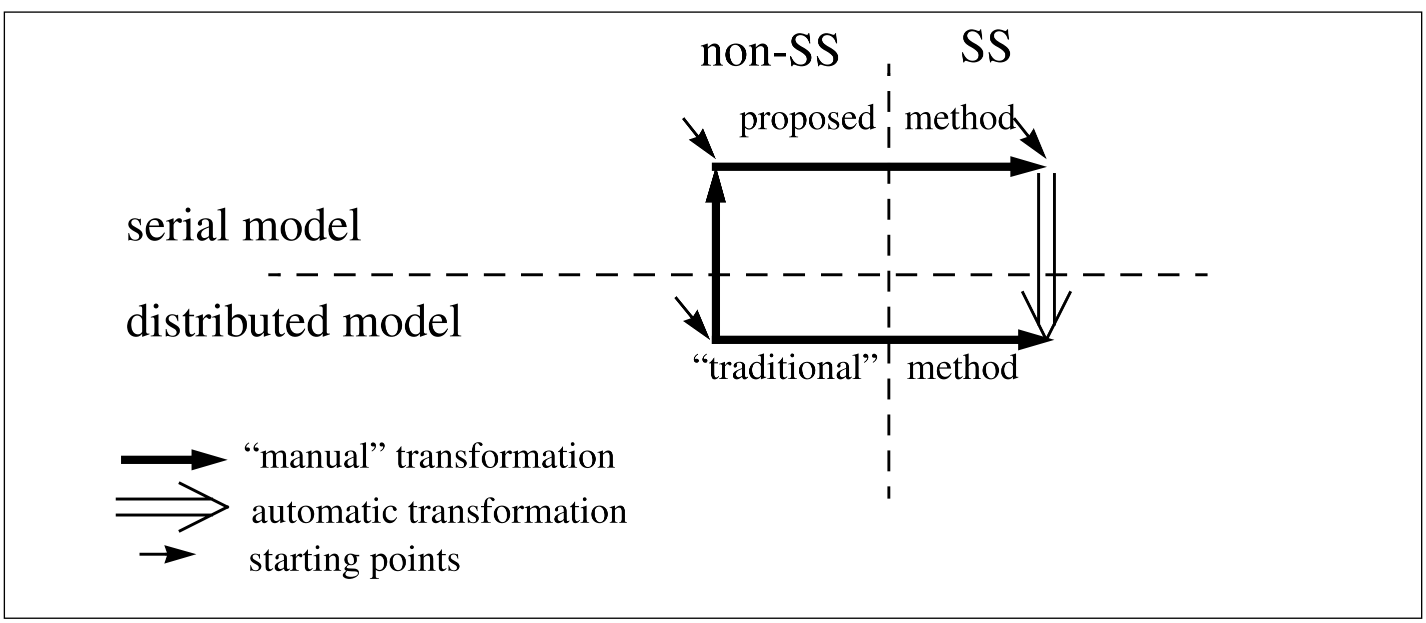
Transformed

$\Longrightarrow$

Distributed Model

- parallel asynchronous execution of guarded commands by different processors in the system;
- message passing through communication channels, no shared variables.

Algorithm Development Using Transformation



Outline

- transformation algorithm;
- SS serial model Lamport's/Rikart-Agrawala's algorithm;
- SS serial model Maekawa's algorithm;
- performance evaluation.
-

Self-Stabilizing Lamport's/Rikart-Agrawala's Algori •
for Serial Model (SSLR)

- application program requests CS entry by setting *A_request* to **true**;
- a Lamport's timestamp is associated with every CS request (it is stored in variable *ts*);
- a node with the smallest timestamp among the requesting nodes enters CS

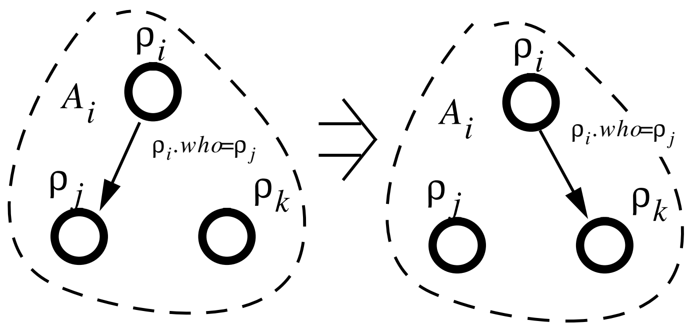
```
Node ρ_i
 *[
        A_request ∧ (mode = idle) →
        ts := new_timestamp()
        mode := waiting

   []   (mode = waiting) ∧
          (ts = min{ρ_j.ts | ∀ρ_j : ρ_j.mode ≠
        mode := locked

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

The Description of SSM

- each node maintains its lock, which can be granted to at most one node at a time;
- to enter CS each node ρ_i needs to obtain the locks from a set of nodes R_i called ρ_i 's r*equest set* b, the request sets of any two nodes intersect which guarantees mutual exclusion;
- since a node can see the timestamp values of only the nodes in its request set, there is a risk of a deadlock;
- to make sure that a circular wait (deadlock) is not formed a node can wait on the node with the smaller timestamp only if neither obtained all the necessary locks to enter CS.
-

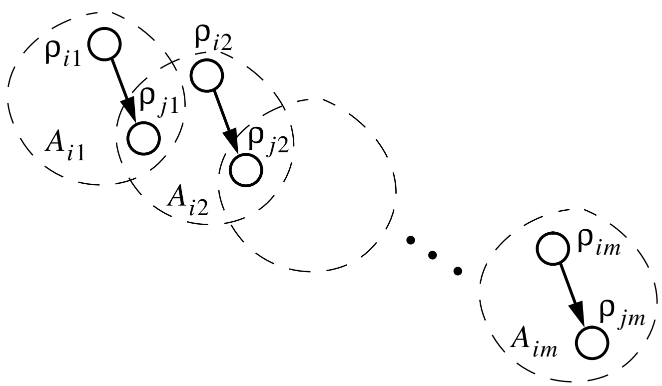
Self-Stabilizing Maekawa's Algorithm for Serial Model (SSM)

ρ_i grants its lock



$\rho_k.ts$ is the smallest in A_i

no circular wait possible



$\rho_{j1}.ts > \rho_{j2}.ts > \ldots > \rho_{jm}.ts < \rho_{j1}.ts$

Node ρ_i

const

R_i /* ρ_i's request set */

A_i /* ρ_i's acquired set $A_i \equiv \{\rho_j | (\rho_i \in R_j)$

*[

$A\_request \wedge (mode = \texttt{idle}) \wedge (\forall \rho_i \in R_j$

ts := **new_timestamp()**

$mode$:= `waiting`

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

$mode$:= `locked`

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

$mode$:= `idle`

[] $(who = \perp \vee (who = \rho_j \wedge \rho_j.mode \neq \texttt{locke}$

$(\exists \rho_k \in A_i$ such that $who \neq \rho_k$ and

$\rho_k.ts = \min\{\rho_l.ts | \forall \rho_l \in A_i : \rho_l.mode \neq \texttt{ic}$

who := ρ_k

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

who := $\perp$

]

Performance Evaluation

Lock based DMX algorithms provide constant synchronization delay and, therefore, scale well

- One of the major performance characteristics of such algorithms is the number of messages it takes a node to enter its critical section (MCS).

algorithm	MCS
non-SS Lamport's	3(N-1)
non-SS Rikart-Agrawala's	2(N-1)
Lamport's or Rikart-Agrawala's coded from scratch (ICDCS'96)	3(N-1)+
transformed SSLR	$3 \cdot 4 \cdot (N-1)$

Transformed and coded from scratch versions exhibit the same asymptotic complexity

Also in the paper:

- SSLR is proven correct using the variant function method introduced by Kessel;
- SS leader election algorithm based on non-SS Chang and Robert's algorithm.

Conclusion

In this paper we have shown the effectiveness of developing SS algorithms for distributed systems using transformation.

Transformed algorithms exhibit worse message complexity than the ones coded from scratch. Yet the asymptotic complexity of both types of algorithms was the same.

We believe the simplicity of the algorithm design using transformation outweighs the drawback of increased message complexity.

For applications sensitive to increase in message complexity the transformation method can be used to build a prototype for tuning and performance improvements.