JOURNAL OF AEROSPACE COMPUTING, INFORMATION, AND COMMUNICATION
Vol. 3, June 2006

Linear-State-Transition Protocols: Asynchronous Protocols with Ease of Efficiency Evaluation

Yoshihiro Nakaminami* and Toshimitsu Masuzawa†
Osaka University, Japan
and
Ted Herman‡
University of Iowa, U.S.A

Distributed systems are commonly modeled by asynchronous models where no assumption is made about process execution speed. The asynchronous model is preferable to the synchronous one because the model reflects the fact that a distributed system consists of heterogeneous computers and their performance varies with time depending on their loads. However, the asynchrony of the system makes it difficult to evaluate efficiency (performance) of distributed protocols. To diminish the difficulty, this paper introduces a class of distributed protocols called linear-state-transition protocols (LST protocols) in the shared-state model and shows that efficiency of the LST protocols in the asynchronous distributed model can be derived from analysis of their synchronous execution where all processes are synchronized in the lock-step fashion. This provides an effective method for evaluating efficiency of the LST protocols in the asynchronous distributed model. The paper also demonstrates the effectiveness of the method by applying it to the self-stabilizing alternator.

I. Introduction

A *distributed system* consists of processes connected by a communication network. In these days, many distributed systems are realized because the Internet makes it possible to use a large number of processes. The features that distributed systems potentially have are high performance processing, scalability and fault-tolerance. There are many brilliant works for the features.[1]

The *shared-state model* is one of the most commonly used models for designing distributed systems. Shared-state model allows a process to read states of its neighbors directly. As an execution model for the shared-state model, an *asynchronous model* that has no assumption on speed of processes is usually used. The asynchronous model is considered as a realistic model because it reflects properties of actual distributed systems: computers may have different processing power, and processing speed may change with time depending on their loads.

Time complexity is an important measure for distributed systems. In the asynchronous model, however, there is no assumption with process execution speed, it is impossible to estimate the time complexity. Therefore time complexity is estimated with some assumptions about process execution speed. *Step complexity* is one of the simplest measures. The step complexity is measured by the number of total actions that are executed in a distributed system. But the step complexity does not consider concurrent execution of the actions and thus it is not fit to evaluate the time complexity

Received 1 September 2005; revision received 00 Month 2006; accepted for publication 11 April 2006.

* Ph. D. Student, Dept. of Information Science and Technology, Osaka Univ., 1-3, Machikaneyama-cho, Toyonaka-city, Osaka-pref., Japan, 560-8531
† Professor, Dept. of Information Science and Technology, Osaka Univ., 1-3, Machikaneyama-cho, Toyonaka-city, Osaka-pref., Japan, 560-8531
‡ Associate Professor, Dept. of Computer Science, University of Iowa, Iowa City, Iowa, U.S.A., 52242

of distributed systems. Instead of the step complexity, *round complexity* is a commonly used measure for estimating time complexity. A round is an ideal time unit that each process has a chance to execute actions at least once. The round complexity is important estimation for the asynchronous model. However its estimation is generally hard because of indeterministic features of the asynchronous model: the asynchrony causes various executions even for a deterministic protocol, so it is hard to find the worst case execution.

On the other hand, analysis of the round complexity is easy in the *synchronous model* where all processes execute their actions in the lock-step fashion: execution of a deterministic protocol is uniquely determined from the initial configuration, and the uniqueness eases analysis of the round complexity.

In this paper we are interested in the following question: *What conditions should a protocol satisfy in order to simplify performance evaluation for asynchronous executions?*

We start our investigation with the observation that there are some protocols in the literature on *self-stabilization*, where the performance, measured in synchronous rounds, is the same as the performance measured in asynchronous rounds. This is true even though the asynchronous executions are nondeterministic, and include configurations that would not appear in a synchronous execution. We did not succeed to answer the general question in this paper, but we do present a sufficient condition called "*linear-state-transition*". The linear-state-transition protocols have enough inherent determinism so that performance in an asynchronous execution is approximately the same as that in the synchronous execution. One of our contributions is thus to facilitate analysis in the asynchronous model essentially by reduction to analysis for the (simpler case of) synchronous model. We have identified several works in the literature[3–6,9] that are linear-state-transition protocols, demonstrating applicability of our results. In this paper, we succeed in deriving new analysis of a synchronization protocol[3] using these results.

The remainder of the paper is organized as follows. Section II defines the system model and its complexity measures. Section III shows a gap between the step complexity and the round complexity. Section IV introduces the linear-state-transition criterion for protocols that we use in later sections. Section V presents theorems that relate synchronous and asynchronous executions of a linear-state-transition protocol and the section also discusses how the theorems can be applied to evaluate efficiency of the protocols. In Section VI, we use the results of Section V to analyze the alternator protocol.[3]

II. Model

A *distributed system* $\mathcal{S} = (P, L)$ consists of set $P = \{v_0, v_1, \ldots, v_{n-1}\}$ of processes and set L of bidirectional (communication) links. A link connects two distinct processes. When a link connects processes v and w, this link is denoted by (v, w). We say w is a *neighbor* of v if $(v, w) \in L$.

Each process v is a state machine and its state transition is defined by *guarded actions*:

$$\langle guard_v \rangle \longrightarrow \langle statement_v \rangle$$

The guard $\langle guard_v \rangle$ of process v is a boolean expression on its own state and its neighbors' states. When the guard is evaluated to be true, $\langle statement_v \rangle$ is executed to change the state of v. This model is called the shared-state model.

A *configuration* (i.e., a global state) of a distributed system is specified by an n-tuple $\sigma = (s_0, s_1, \ldots, s_{n-1})$ where s_i stands for the state of process v_i. A process v is said to be *enabled* at a configuration σ when v has a guarded action whose guard is true at σ. A process v is said to be *disabled* at σ when it is not enabled at σ.

Let $\sigma = (s_0, s_1, \ldots, s_{n-1})$ and $\sigma' = (s'_0, s'_1, \ldots, s'_{n-1})$ be configurations and Q be any set of processes. We denote the transition from σ to σ' by $\sigma \overset{Q}{\mapsto} \sigma'$, when σ changes to σ' by actions of every enabled process in Q. (All enabled processes in Q make actions but no disabled processes in Q make actions.) Reading states and executing actions are done atomically: each process that is included in Q and is enabled at σ reads its neighbors' states at σ, and changes its state by executing one of its enabled actions. The resultant state depends on its state at σ and the read states of the neighbors. Notice that $s_i = s'_i$ holds for a process v_i if $v_i \notin Q$ or v_i is disabled at σ. We sometimes simply denote $\sigma \mapsto \sigma'$ without specifying the set of processes Q.

A *schedule* is an infinite sequence $\mathcal{Q} = Q_1, Q_2, \ldots$ of nonempty sets of processes. In this paper, we assume that any schedule is *weakly fair*, that is, all processes appear infinitely often in any schedule. If an infinite sequence of configurations $E = \sigma_0, \sigma_1, \sigma_2, \ldots$ satisfies $\sigma_j \overset{Q_{j+1}}{\mapsto} \sigma_{j+1}$ $(j \geq 0)$, then E is called an *execution* starting from σ_0 by schedule $\mathcal{Q}$. For a process v_i, the *local history* of v_i in an execution E is the projection of E to process states of v_i with removal of the stuttering part.

A schedule specifies the order of processes that are activated to execute their guarded actions. In this paper, we consider an asynchronous distributed system where we make no assumption on the speed of processes. This implies that all weakly fair schedules are possible to occur. Among the schedules, we define the following two special schedules.

Synchronous schedule: When a schedule $\mathcal{Q} = Q_1, Q_2, \ldots$ satisfies $Q_j = P$ for each j $(j \geq 1)$, then we call $\mathcal{Q}$ a *synchronous schedule* and we call the corresponding execution a *synchronous execution*.

Sequential schedule: When a schedule $\mathcal{Q} = Q_1, Q_2, \ldots$ satisfies $|Q_j| = 1$ for each j $(j \geq 1)$, then we call $\mathcal{Q}$ a *sequential schedule* and we call the corresponding execution a *sequential execution*.

We consider schedules and executions to be infinite sequences. However sometimes, we consider finite parts of schedules and executions, and call them *partial schedules* and *partial executions*. Partial schedules and partial executions are defined as follows: A *partial schedule* is a finite sequence $\mathcal{Q} = Q_1, Q_2, \ldots, Q_m$ of nonempty sets of processes. If a finite sequence of configurations $E = \sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_m$ satisfies $\sigma_j \overset{Q_{j+1}}{\mapsto} \sigma_{j+1}$ $(0 \leq j \leq m-1)$, then E is called a *partial execution* starting from σ_0 by schedule $\mathcal{Q}$.

To evaluate the time complexity of protocols, we introduce *rounds* for an execution E. Let $E = \sigma_0, \sigma_1, \sigma_2, \ldots$ be an execution by a schedule $\mathcal{Q} = Q_1, Q_2, \ldots$. The first round of E is defined to be the minimal partial execution $\sigma_0, \sigma_1, \ldots, \sigma_k$ that satisfies $P = \cup_{1 \leq j \leq k} Q_j$. The second and later rounds of E are defined recursively for the execution $\sigma_k, \sigma_{k+1}, \sigma_{k+2}, \ldots$.

III. Step Complexity and Round Complexity

The *step complexity* is a measure commonly used to evaluate efficiency of asynchronous distributed systems. The step complexity is defined to be the total sum of executed actions at all processes. The step complexity is simple, however, does not consider concurrent execution of actions. Thus, it is not fit to evaluate the time complexity of distributed systems.

The *round complexity* is also a popular measure to evaluate the time complexity of asynchronous distributed systems. The round defined in the previous section can be considered as a time unit during which every process is activated at least once.

In this section, we clarify the difference between the step complexity and the round complexity by comparing them with an example: a self-stabilizing mutual exclusion on ring networks.[10] Mutual exclusion is one of the most fundamental problems in distributed systems. It requires that there is at most one token at any time in the system, and each process gets a token infinitely often.

Self-stabilizing protocols are defined by two properties, *convergence* and *closure*. The convergence requires that a self-stabilizing protocol eventually reaches a desired legitimate configuration. The closure requires that any configuration after a legitimate configuration is also in the set of legitimate configurations.

The ring consists of n processes $v_0, v_1, \ldots, v_{n-1}$ and n links where $(v_i, v_{(i+1) \bmod n})$ $(0 \leq i \leq n-1)$. The state of each process v_i is represented by a variable q_i that stores an integer in $\{0, 1, \ldots, n\}$. Figure 1 presents the protocol. Note that processes $v_1, v_2, \ldots, v_{n-1}$ have an identical action but v_0 has a distinct one.

In the rest of this section, we evaluate the step complexity and the round complexity of the protocol shown in Fig. 1. In this section, we consider only the synchronous schedule. Each process is considered to have a token when it has an enabled action. One of the legitimate configurations is that all processes have an identical value, where only process v_0 is enabled and has a token.[10]

```
$v_0$ :
  $q_0 = q_{n-1} \rightarrow q_0 := (q_0 + 1) \bmod (n+1)$
$v_i (i \neq 0)$ :
  $q_i \neq q_{i-1} \rightarrow q_i := q_{i-1}$
```

Fig. 1 Actions of the stabilizing mutual exclusion on a ring.

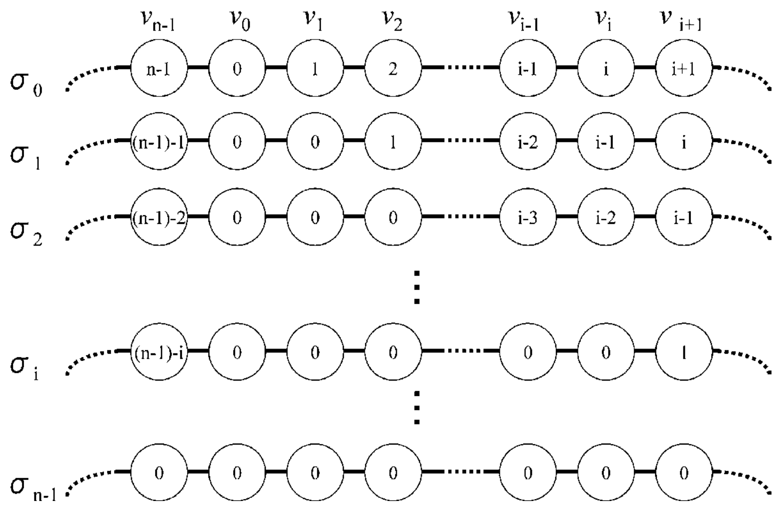

Fig. 2 Execution from σ_0 by the synchronous schedule.

To clarify the difference between the step complexity and the round complexity, we consider two cases, the *convergence time* and the *token circulation time*.

i) Convergence time: Convergence time is the time required to reach a legitimate configuration. We consider an initial configuration σ_0 where $q_i = i$ holds for each i $(0 \le i \le n-1)$. Note that all processes other than v_0 are enabled at σ_0. Let $E = \sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_{n-1}$ be the partial execution starting from σ_0 by the synchronous schedule.

We can easily show by induction on j that each configuration σ_j $(0 \le j \le n-1)$ satisfies $q_i = 0$ for each i $(0 \le i \le j)$ and $q_i = i - j$ for each i $(j+1 \le i \le n-1)$ (Fig. 2). It is also clear that σ_{n-1} is a legitimate configuration.

Now we analyze the step and round complexities of the execution from the initial configuration σ_0 to the legitimate configuration σ_{n-1} by the synchronous schedule. At each configuration σ_j $(0 \le j \le n-2)$, the number of enabled processes is $n-1-j$. So the step complexity is $\sum_{j=0}^{n-2}(n-1-j) = n(n-1)/2$. On the other hand, the round complexity is clearly $n-1$.

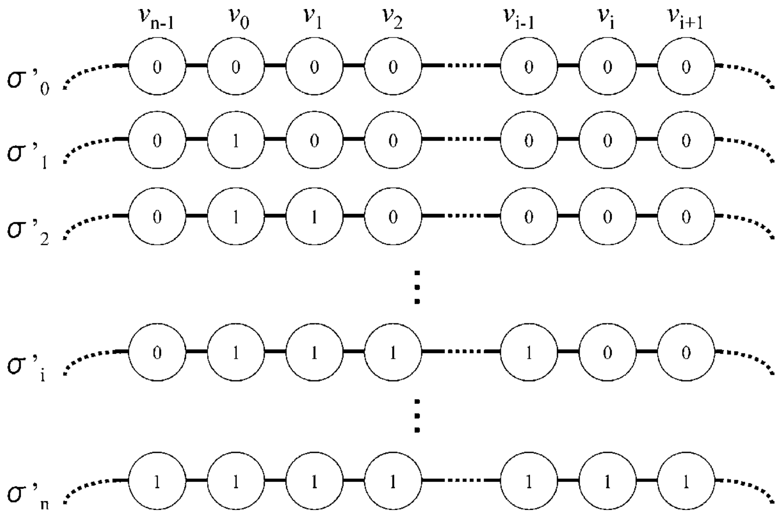

Fig. 3 Execution from σ'_0 by the synchronous schedule.

ii) Token circulation time: Let σ'_0 be a configuration where $q_i = 0$ holds for each i $(0 \le i \le n-1)$. The configuration σ'_0 is clearly legitimate. Let $E' = \sigma'_0, \sigma'_1, \ldots, \sigma_n$ be the partial execution starting from σ'_0 by the synchronous schedule (Fig. 3).

We can see that all the configurations $\sigma'_0, \sigma'_1, \ldots, \sigma'_n$ are legitimate and that only the process $v_{j \bmod n}$ is enabled at σ'_j $(0 \le j \le n)$. In the sequel, we can consider the execution E' as a token circulation along the ring: the token starts the circulation from v_0 and ends at v_0.

Since only the process $v_{j \bmod n}$ is enabled at σ'_j $(0 \le j \le n)$, both the step complexity and the round complexity for the token circulation (i.e., the partial execution E') are clearly n.

Both of the above scenarios have the same round complexity $O(n)$, but have different step complexities: one is $n(n-1)/2$ and the other is n. This implies that the round complexity cannot be easily derived from the step complexity, even when the step complexity can be evaluated.

Notice that E' can be also regarded as a partial execution starting from σ'_0 by a partial schedule $\mathcal{Q} = \{v_0\}\{v_1\}\{v_2\}\ldots\{v_{n-1}\}$. In such observation, the round complexity is 1 because $\mathcal{Q}$ includes all processes exactly once. This shows that an execution may have different round complexities depending on the schedules.

IV. Linear-State-Transition Protocol

This section defines a class of protocols called linear-state-transition protocols, which are focused on in the following sections.

A protocol is called a *linear-state-transition protocol* (*LST protocol*) if it satisfies the following conditions **C1** and **C2**.

C1 (Linear-state-transition) For any configuration σ, each process has the same local history in all executions starting from σ. (This condition implies that the local history of each process is uniquely determined by the initial configuration σ.)

C2 (Non-interference)[7] Let σ be any configuration and v be any process. If v is enabled at σ, then v remains enabled until v executes an action. (This condition implies that an enabled process never becomes disabled by actions of other processes.)

The condition **C1** implies that the synchronous execution of an LST protocol is uniquely determined from the initial configuration. The condition **C1** also implies that each process behaves the same actions in all the execution, while an LST protocol may have several executions even from the same initial configuration because of different asynchronous schedules. The condition **C2** implies that an LST protocol is somewhat insensitive to the order in which process are activated.

Examples of LST protocols include the stabilizing alternator,[3] the stabilizing agent traversal protocol,[4] the stabilizing pipelined PIF protocol[5] and the synchronizers.[6,9]

As an example of LST protocols, we present a self-stabilizing α synchronizer.[2] The protocol realizes synchronization between neighboring processes. The action of α synchronizer is quite simple and is presented in Fig. 4.

Now we show that the self-stabilizing α synchronizer is an LST protocol. It is clear that the condition **C1** (linear-state-transition) is satisfied because the only possible action of process v is $phase_v := phase_v + 1$. Once $phase_v \le phase_w$ holds for a neighbor w of v, $phase_v \le phase_w$ remains holding until v increments the value of $phase_v$. Thus, if a process v is enabled, v remains enabled until it executes its action. Therefore the condition **C2** (Non-interference) holds.

```
$\forall w \in N_v[phase_v \le phase_w]$
        $\rightarrow phase_v := phase_v + 1$
```

Fig. 4 Self-stabilizing α synchronizer: an action of process v.

V. Efficiency Evaluation of LST Protocols

This section presents key properties of LST protocols and proposes a method based on the properties for analyzing efficiency of LST protocols.

A. Key Properties of LST Protocols

The following theorem shows a key property of LST protocols. The key property forms a basis of the efficiency evaluation method proposed in Section B.

Theorem 1. *Let σ_0 be any initial configuration. For an LST protocol, any action that is executed at round t in the synchronous execution starting from σ_0 is executed at round t or earlier in any execution starting from σ_0.* $\Box$

Theorem 1 implies that the synchronous execution of an LST protocol is the slowest execution with respect to the number of actions executed at each process. Thus, the number of rounds required until some specific actions are executed in the synchronous execution can be the maximum number of rounds in all executions. Since the synchronous execution of a deterministic protocol is uniquely determined from an initial configuration, analysis of the round complexity of an LST protocol is much simplified by considering only the synchronous executions.

In the rest of this subsection, we prove Theorem 1. Theorem 1 is proved with the help of a special type of asynchronous sequential executions called *pseudo synchronous* executions. A pseudo synchronous execution simulates the synchronous execution in the sense that all configurations appearing in the synchronous execution also appear in the same order in the pseudo synchronous execution starting from the same initial configuration (Lemma 1).

Definition 1. *Let σ_0 be any configuration. A* pseudo synchronous execution *starting from σ_0 is an execution by a sequential schedule $\mathcal{Q} = \mathcal{Q}^1\mathcal{Q}^2\cdots$ with each term $\mathcal{Q}^i$ $(i \geq 1)$ being a partial sequential schedule such that:*
- *each process of P appears exactly once in $\mathcal{Q}^i$*
- *Let σ^{i-1} be the last configuration of the partial (pseudo synchronous) execution by the partial sequential schedule $\mathcal{Q} = \mathcal{Q}^1\mathcal{Q}^2\cdots\mathcal{Q}^{i-1}$. (For convenience, let $\sigma^0 = \sigma_0$ if $i = 1$.) Let Enabled(σ^{i-1}) be the set of enabled processes at σ^{i-1} and Disabled(σ^{i-1}) be the set of disabled processes at σ^{i-1} (i.e, Disabled$(\sigma^{i-1}) = P -$ Enabled(σ^{i-1})). Then all processes in Disabled(σ^{i-1}) appear before any process in Enabled(σ^{i-1}) in $\mathcal{Q}^i$.*

The schedule $\mathcal{Q} = \mathcal{Q}^1\mathcal{Q}^2\cdots$ is called a pseudo synchronous schedule. $\Box$

To simulate the synchronous execution, only the processes enabled at the beginning of each round should execute actions in the round. Processes disabled at the beginning of a round may become enabled in the round because of other processes' actions. To prevent such processes from executing actions in the round, processes disabled at the beginning of the round are activated first.

In Definition 1, σ^0 is the initial configuration, and σ^i $(i \geq 1)$ is the last configuration of the i^{th} round. A pseudo synchronous execution is not uniquely determined from the initial configuration σ_0, but the configurations σ^i $(i \geq 1)$ are uniquely determined for an LST protocol. That is, all pseudo synchronous executions starting from the same initial configuration have the same configuration at the end of each round. Concerning the configurations $\sigma^0, \sigma^1, \ldots,$ the following lemma holds.

Lemma 1. *For an LST protocol, the subsequence of configurations $\sigma_0(= \sigma^0), \sigma^1, \sigma^2, \ldots$ appearing in a pseudo synchronous execution coincides with the synchronous execution.*

Proof. Let $\sigma_0, \sigma_1, \sigma_2, \ldots$ be the synchronous execution. We will show by induction on i that $\sigma^i = \sigma_i$ holds for each i $(i \geq 0)$.

(Induction basis) It follows from the definition that $\sigma^0 = \sigma_0$ holds.

(Inductive step) With assumption of $\sigma^i = \sigma_i$, we prove $\sigma^{i+1} = \sigma_{i+1}$. The definition of the synchronous execution implies that every enabled process at σ^i executes a single action and no disabled process at σ^i executes an action between σ^i and σ^{i+1}.

Let $\mathcal{Q}^{i+1} = Q_1, Q_2, \ldots, Q_n$ be a partial pseudo synchronous schedule that leads from σ^i to σ^{i+1}. Recall from the definition of the pseudo synchronous schedule that $|Q_i| = 1$ holds for each i $(1 \leq i \leq n)$, and thus, let $Q_i = \{v^i\}$. In the partial pseudo synchronous schedule, there exists x $(0 \leq x < n)$ such that $Disabled(\sigma^i) = \{v^1, v^2, \ldots, v^x\}$ and

Enabled$(\sigma^i) = \{v^{x+1}, v^{x+2}, \ldots, v^n\}$ hold. Let σ be a configuration reached from σ^i by applying the partial schedule $Q_1, Q_2, \ldots, Q_x$. It follows from *Disabled*$(\sigma^i) = \{v^1, v^2, \ldots v^x\}$ that no process executes its action between σ^i and σ. Thus, $\sigma = \sigma^i$ holds and all the processes in *Enabled*(σ^i) remains enabled at σ. The condition **C2** guarantees that every process in *Enabled*(σ^i) executes a single action by applying the schedule $Q_{x+1}, \ldots, Q_n$ to σ. For the resultant configuration σ^{i+1}, $\sigma^{i+1} = \sigma_{i+1}$ holds from the condition **C1**. □

Based on the condition **C1**, we introduce the precedence relation $<$ on the states of each process v.

Definition 2. *Let σ_0 be the initial configuration and s_i^0 be the initial state of process v_i. From the condition* **C1**, *the state transition of v_i is uniquely determined from σ_0 and let $s_i^0, s_i^1, s_i^2, \ldots$ be the local history of v_i (common to every execution starting from σ_0). For convenience, we assume without loss of generality that the states $s_i^0, s_i^1, s_i^2, \ldots$ are mutually distinct*[§]*. Then, the precedence relation $<$ on the states of v_i is defined as follows: $s_i^j < s_i^k \iff j < k$. The precedence relation $\leq$ on the process states is defined as follows: $s_i^j \leq s_i^k \iff j \leq k$.* □

We define the precedence relation on configurations from the precedence relation on process states.

Definition 3. *For two configurations $\sigma = (s_0, s_1, \ldots, s_{n-1})$ and $\sigma' = (s'_0, s'_1, \ldots, s'_{n-1})$, the precedence relation $<$ on the configurations is defined as follows.*

$$\sigma < \sigma' \iff \forall i\ (1 \leq i \leq n)\ [s_i \leq s'_i] \wedge \exists j\ (1 \leq j \leq n)\ [s_j < s'_j]$$

The precedence relation $\leq$ on the configurations is defined as follows.

$$\sigma \leq \sigma' \iff \forall i\ (1 \leq i \leq n)\ [s_i \leq s'_i]$$ □

Lemma 2. *For an LST protocol, let E_{ps} be a pseudo synchronous execution starting from any configuration σ_0 and let σ be any configuration that appears in E_{ps}. Let E be any execution starting from σ_0 and let δ be any configuration that appears in E. If $\sigma < \delta$ holds, there exists a partial execution $E_{\sigma,\delta}$ that starts from σ and reaches δ.*

Proof. We say that configuration α is *β-reachable* when there exists a partial execution that starts from α and reaches a configuration β. We say that configuration α is *β-unreachable* when α is not β-reachable. For contradiction, we assume σ is δ-unreachable and $\sigma < \delta$ holds.

From the definition, the initial configuration σ_0 is δ-reachable. Since configuration σ appearing in a pseudo synchronous execution E_{ps} is δ-unreachable, E_{ps} has consecutive configurations γ and γ' ($\gamma < \gamma'$) such that γ is δ-reachable and γ' is δ-unreachable (Fig. 5). Let v be the process that executes an action between γ and γ'. Since γ is δ-reachable, there exists a partial execution, say $E_\gamma = \gamma, \omega_1, \omega_2, \ldots, \omega_{m-1}, \omega_m (= \delta)$ from γ to δ. Let $\mathcal{Q}_\gamma = \{v_1\}, \{v_2\}, \ldots, \{v_m\}$ ($v_i \in P$) be the partial sequential schedule for E_γ.

Obviously ω_1 is δ-reachable and γ' is δ-unreachable. So ω_1 is different from γ' and thus $v \neq v_1$. From the condition **C2**, process v is enabled at ω_1. Let ω'_1 be the configuration that is reached when process v executes its action at ω_1. Similarly, process v_1 is enabled at γ'. Let ω''_1 be the configuration that is reached when process v_1 executes its action at γ'. From the condition **C1**, $\omega'_1 = \omega''_1$ holds (Fig. 5).

Since ω'_1 is reached from ω_1 and process v_2 is enabled at ω_1, v_2 is enabled at ω'_1 from the condition **C2**. By repeating a similar argument, we can construct a partial sequential execution $E'_\gamma = \gamma', \omega'_1, \omega'_2, \ldots, \omega'_m$ starting from γ' by $\mathcal{Q}_\gamma$. We can see that action of process v changes configuration from ω_i to ω'_i for each i $(1 \leq i \leq m)$.

Since γ' is δ-unreachable, ω'_i is δ-unreachable. It follows for $\mathcal{Q}_\gamma = \{v_1\}, \{v_2\}, \ldots, \{v_m\}$ that $v_i \neq v$ $(1 \leq i \leq m)$ holds. Then, for $\sigma = (s_0, s_1, \ldots, s_{n-1})$ and $\delta = (s'_0, s'_1, \ldots, s'_{n-1})$, $s_v > s'_v$ holds. This contradicts the assumption $\sigma < \delta$. □

The following lemma implies that any pseudo synchronous execution is the slowest execution of an LST protocol in the sense that the number of actions executed by each process is minimum at the end of each round.

[§] The assumption can be validated by assigning a monotonously increasing value (e.g., a counter value) to each state.

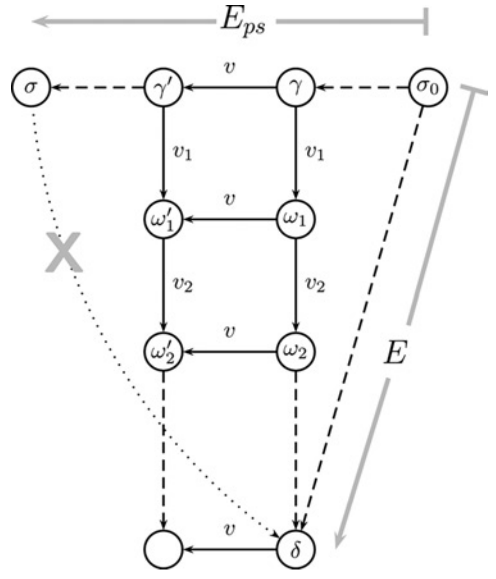

Fig. 5 Proof of Lemma 2.

Lemma 3. *Let σ_0 be any initial configuration. For an LST protocol, any action that is executed at round t in a pseudo synchronous execution starting from σ_0 is executed at round t or earlier in any execution starting from σ_0.*

Proof. We prove the theorem by induction. For $t = 1$, the theorem obviously holds from the condition **C2**. Let $\sigma^t = (s_0, s_1, \ldots, s_{n-1})$ and $\sigma^{t+1} = (s'_0, s'_1, \ldots, s'_{n-1})$ be the configurations at the ends of rounds t and $t+1$ in a pseudo synchronous execution respectively. For any execution E, let $\delta^t = (r_0, r_1, \ldots, r_{n-1})$ and $\delta^{t+1} = (r'_0, r'_1, \ldots, r'_{n-1})$ be the configurations at the ends of rounds t and $t+1$ respectively.

For induction, we show $\sigma^{t+1} \leq \delta^{t+1}$ under the assumption that $\sigma^t \leq \delta^t$ (i.e., $s_i \leq r_i$ for each i $(0 \leq i \leq n-1)$). We consider the state of any process v_i. If v_i is disabled at configuration σ^t (i.e., $s'_i = s_i$), $s'_i \leq r'_i$ obviously holds. In the case that $s_i < r_i$ holds, $s'_i \leq r'_i$ obviously holds since v_i executes at most one action between σ^t and σ^{t+1}. Thus, it is sufficient to show that process v_i is enabled at δ^t when v_i is enabled at σ^t and $s_i = r_i$ holds. From Lemma 2, σ^t is δ^t-reachable since we assume $\sigma^t \leq \delta^t$. Thus, from the condition **C2**, v_i is enabled at δ^t since v_i is enabled at σ^t and $s_i = r_i$. Consequently, $s'_i \leq r'_i$ holds. □

From Lemma 1 and Lemma 3, Theorem 1 immediately follows.

B. A Method for Evaluating Efficiency of LST Protocols

Generally a distributed problem Π specifies the possible initial configurations and the expected executions starting from each of the initial configurations. A protocol for solving Π is required that all of its possible executions are the expected ones Π specifies.

In the following, we introduce two theorems for evaluating time complexity of LST protocols, the *counting theorem* and the *reach theorem*. In these theorems, we focus on some set of specified actions or some set of specified configurations. These specified actions and configurations are determined according to the context of the problem. Typical examples of such specified actions include the token passing actions in token circulation protocols and the *decide* actions in the repetition of a wave algorithm.[1] Typical examples of specified configurations include the configurations with solutions in non-reactive problems (e.g., leader election and tree construction problems) and the safe configurations[2] in self-stabilizing protocols.

Theorem 2. ***(Counting theorem)*** *For an LST protocol, if some specified actions are executed k times by the end of round t in the synchronous execution, then the actions are executed k times or more by the end of round t in any execution starting from the same configuration.* □

The counting theorem implies that an LST protocol makes the least progress in the synchronous execution. This helps us to evaluate the round complexity until a specified action is executed: the round complexity of the synchronous executions gives the worst-case round complexity.

The reach theorem is used to evaluate the round complexity of an LST protocol until it reaches one of the specified configurations. We consider only the specified configurations satisfying the *closure property*: once the protocol reaches one of these specified configurations, then all of the subsequent configurations are also among the specified configurations. For such specified configurations, Theorem 1 implies the following: if an LST protocol reaches a specified configuration by the end of round t in the synchronous execution, then it reaches a specified configuration by the end of round t or earlier in any execution from the same initial configuration. Thus the worst-case reach time of an LST protocol in asynchronous executions can be directly derived from the reach time of the synchronous execution from the same initial configuration.

Theorem 3. ***(Reach theorem)*** *Consider some specified configurations satisfying the closure property. For an LST protocol, if the synchronous execution starting from any configuration reaches a specified configuration at the end of round t, then any execution starting from any configuration reaches a specified configuration at the end of round t.* □

It is valuable to mention what impact the reach theorem makes especially on efficiency analysis of self-stabilizing protocols. Self-stabilizing protocols are required to reach a safe configuration from any initial configuration. The safe configurations satisfy the closure property and are defined depending on the problem definition and the self-stabilizing protocol. Thus, the worst-case convergence time (i.e., the worst-case reach time to the safe configurations) in asynchronous executions can be directly derived from the convergence time of the synchronous execution.

VI. Application for Stabilizing Alternator

In this section, we apply the method proposed in the previous section to analyze efficiency of an LST protocol. We evaluate the performance of *the stabilizing alternator* proposed by Gouda and Haddix,[3,8] which is an example of the LST protocols. Notice that Gouda and Haddix[3,8] did not show the performance of the alternator and we first evaluate its performance using the properties of LST protocols.

A. Stabilizing Alternator

In this subsection, we briefly introduce the stabilizing alternator proposed by Gouda and Haddix.[3]

The alternator is an underlying protocol for protocol transformation: it transforms a protocol that works correctly under any sequential schedule to a protocol for the same task that works under any schedule. The transformation is important for the following reasons. Protocol design is much easier under the sequential schedules. In fact, many self-stabilizing protocols have been proposed under the sequential schedules. But the sequential schedule is not practical since real distributed systems allow several processes to execute actions simultaneously. Thus it is highly desired to transform protocols for sequential schedules so that they can work under any schedules.

In the shared-state model, the action of each process is determined from the states of the process and its neighbors. This implies that protocols designed under the sequential schedules correctly work under any schedule if no neighboring processes execute actions simultaneously (i.e., local mutual exclusion). The alternator realizes the local mutual exclusion.

The alternator is a self-stabilizing protocol: starting from any initial configuration, the system reaches a configuration after which no neighboring processes execute specific actions (defined below) simultaneously.

Figure 6 shows a guarded action of process v. In the protocol, q_v is a variable of process v that stores an integer in $\{0, 1, \dots, 2d-1\}$, N_v is a constant denoting the set of neighbors of v, and d is a constant denoting the length of the longest simple cycle (if a distributed system has no cycle, then let $d = 2$).

```
$\forall w \in N_v[(q_v \neq q_w \vee v < w) \wedge (q_v \neq ((q_w - 1) \mod 2d))]$
        $\rightarrow q_v := (q_v + 1) \mod 2d$
```

Fig. 6 Stabilizing alternator: an action of process v

The following lemma holds for the alternator.

Lemma 4. [3] *In any execution starting from an arbitrary initial configuration, the followings hold.*

1. *Each process v becomes enabled infinitely often (i.e., v updates q_v infinitely often).*
2. *There exists a suffix of the execution that satisfies $q_v \neq 2d-1$ or $q_w \neq 2d-1$ at every configuration for any neighboring processes v and w.* □

The stabilizing alternator is utilized in protocol transformation. Let $\mathcal{A}$ be a self-stabilizing protocol that works correctly under any sequential schedule. Transformation from $\mathcal{A}$ to a self-stabilizing protocol $\mathcal{A}'$ for the same task that works under any schedule is realized as follows: each process v executes the action of the self-stabilizing alternator and executes the actions of $\mathcal{A}$ concurrently only when variable q_v is updated from $2d-1$ to 0.

Lemma 4 guarantees that any execution of protocol $\mathcal{A}'$ has an infinite suffix where no neighboring processes execute actions of $\mathcal{A}$ simultaneously. Since protocol $\mathcal{A}$ is a self-stabilizing protocol, $\mathcal{A}'$ converges to its intended behavior eventually.

B. Satisfaction of LST Conditions

In this subsection, we first show that the alternator is an LST protocol. To show that the alternator is an LST protocol, we consider conditions **C1** and **C2**. It is clear that **C1** (linear-state-transition) is satisfied because the only possible action of process v is $q_v := (q_v + 1) \mod 2d$. Thus, it is sufficient to show that the alternator satisfies **C2** (non-interference).

Lemma 5. *In the alternator, once a process becomes enabled, then the process remains enabled until it executes an action.*

Proof. Let v be a process and σ be a configuration such that v is enabled at σ. Without loss of generality, we assume that $q_v = 0$ at σ. Since process v is enabled at σ, each neighbor w of v satisfies $(2 \leq q_w \leq 2d-1) \vee (q_w = 0 \wedge v < w)$. For contradiction, assume that v becomes disabled before it executes its action. Let w be the process whose execution of its action changes v to disabled. Two cases to consider are:

1. Case that $q_w = 0 \wedge v < w$ holds at σ: Since w cannot execute its action until v executes its action, w never changes v to disabled.
2. Case that $2 \leq q_w \leq 2d-1$ holds at σ: Process w can execute actions. But from the protocol, w satisfies $2 \leq q_w \leq 2d-1$ after execution of its actions (w is disable when $q_w = 2d-1$ because $q_v = 0$). Thus w never changes v to disabled.

In each case, v remains enabled and it is contradiction. □

C. Performance Analysis

In this subsection, we analyze the performance of the alternator. Recall that the performance is measured by the number of the specified actions executed in some number of rounds. We count the number of executions of the statement $q_v := (q_v + 1) \bmod 2d$.

We consider the synchronous execution of the alternator and count the number of executions of the statement. First we show that the alternator reaches an *ideal* configuration (defined below) in $2n$ rounds from any configuration. Once the synchronous execution reaches the ideal configuration, each process makes an action at every round in the synchronous execution. Thus, we can conclude that each process makes at least m actions by the end of the $(m + 2n)$-th round.

We define an *ideal* configuration as follows: A configuration is ideal iff each process satisfies the following condition:

$$\begin{aligned} \forall w \in N_v[\quad & q_v \neq q_w \\ \wedge \quad & q_v \neq (q_w + 1) \mod 2d] \end{aligned}$$

From the protocol in Fig. 6, we can see that all processes are enabled at any ideal configuration. In the synchronous execution, each process increments its variable at any ideal configuration. So the resultant configuration in the synchronous execution is also ideal. Therefore the following lemma holds:

Lemma 6. *In the synchronous execution, each process of the alternator executes its action at every round once the alternator reaches an ideal configuration.* □

To evaluate the number of rounds required to reach an ideal configuration, we define a directed graph $G.\sigma$ for a configuration σ. The graph $G.\sigma$ was originally defined in[3] for the correctness proof.

Definition 4. *For a distributed system $\mathcal{S} = (P, L)$ and a configuration σ, a directed graph $G.\sigma = (P, L')$ is defined as follows:*

$$\begin{aligned} L' = \{(v, w) \in L \mid \quad & (q_v = q_w \ \wedge \ v > w) \\ \vee \quad & (q_v = (q_w - 1) \mod 2d) \text{ holds at configuration } \sigma\} \end{aligned}$$

(The vertex set of $G.\sigma$ is P. We do not distinguish the vertices from the processes. So we simply call a vertex in $G.\sigma$ a process.) □

From the definition of $G.\sigma$ and the protocol of the alternator, process v is enabled at configuration σ iff v has no outgoing edge in $G.\sigma$. The following lemma guarantees that there exists an enabled process at any configuration.

Lemma 7. [3] *For any configuration σ, $G.\sigma$ has no directed cycle.* □

For $G.\sigma$, we define a *head process* as follows:

Definition 5. *For a directed graph $G.\sigma$, a* head process *is a process that has no outgoing edge but has at least one incoming edge.* □

Lemma 7 guarantees that there is at least one head process at any configuration such that $L' \neq \emptyset$. From the definition, if v is a head process in $G.\sigma$, v is enabled at σ. Concerning each of the two conditions $q_v = q_w \ \wedge \ v > w$ and $q_v = (q_w - 1) \mod 2d$ for edges of $G.\sigma$, we introduce the following two observations.

Observation 1. *Let σ_{i-2}, σ_{i-1} and σ_i be consecutive configurations in the synchronous execution of the alternator and v be a head process at σ_i. If v is also a head process at σ_{i-1}, the followings hold:*
- *There exits a neighbor u of v such that $(q_u = q_v) \wedge (u > v)$ holds at σ_{i-1}. So u has an outgoing edge (u, v) at σ_{i-1} and σ_i.*
- *v is not a head process at σ_{i-2}.* □

Observation 2. *Let σ_{i-1} and σ_i be consecutive configurations in the synchronous execution of the alternator and v be a head process at σ_i. If v is not a head process at σ_{i-1}, v has an outgoing edge (v, w) such that $q_v = (q_w - 1) \mod 2d$ holds and w is a head process at σ_{i-1}.* □

We define an *inheritance list* for the following discussion. Intuitively, an inheritance list consists of a history of head processes. Note that an inheritance list is defined for each head process at each configuration.

Definition 6. *Let $\sigma_0, \sigma_1, \sigma_2, \ldots$ be the synchronous execution starting from σ_0. We define* an inheritance list *for each head process v in $G.\sigma_i$ as follows.*
- *The inheritance list of v at the initial configuration σ_0 is a list $\langle v \rangle$.*
- *If v is a head process in $G.\sigma_{i-1} (i \geq 1)$, the inheritance list of v at σ_i is that of v at σ_{i-1}.*
- *If v has an outgoing edge in $G.\sigma_{i-1}$, let w be any process such that edge (v, w) exists in $G.\sigma_{i-1}$. The inheritance list of v at σ_i is obtained by appending v to the inheritance list of w in $G.\sigma_{i-1}$.*

We define the length *of an inheritance list to be the number of processes contained in the list.* □

From the definition of the inheritance list and Observation 1 and 2, the following lemma holds.

Lemma 8. *Let $\sigma_0, \sigma_1, \sigma_2, \ldots$ be a synchronous execution of the alternator and ℓ_i be the minimum length of the inheritance lists existing at σ_i. Then, the followings hold.*

- *$\ell_0, \ell_1, \ell_2, \ldots$ is monotonically non-decreasing.*
- *$\ell_i < \ell_{i+2}$ holds for each i ($i \geq 0$).* □

Now we analyze how many steps are needed to reach an ideal configuration.

Lemma 9. *In the synchronous execution from any initial configuration, the alternator reaches an ideal configuration within $2n$ rounds.*

Proof. First, we show that any process appears in any inheritance list at most once.

For contradiction, we assume that a process v appears twice in an inheritance list. Let $\langle v = v_0, v_1, v_2, \ldots, v_{m-1}, v_m = v\rangle$ ($m \leq d$) be a part of the inheritance list, and without loss of generality, assume $v_i \neq v_j$ ($0 \leq i < j \leq m-1$). Notice that the partial list forms a simple cycle. This implies that the inheritance list is inherited along the cycle.

Let σ^i be the configuration where process v_i becomes a head process inheriting the inheritance list from v_{i-1}. From Observation 1 and 2, there exists the edge (v_{i+1}, v_i) in $G.\sigma^v$. Thus the following holds at σ^i:

$$(q_{v_i} = q_{v_{i+1}} \;\wedge\; v_i < v_{i+1}) \tag{1}$$

$$\vee(q_{v_i} = (q_{v_{i+1}} + 1) \mod 2d) \tag{2}$$

We observe the execution from σ^i to σ^{i+1} in the two cases: one satisfying (1) and the other satisfying (2). For the both cases, assume that $q_{v_i} = \lambda$ holds at σ^i.

1. Case that (1) is satisfied: During the execution from σ^i to σ^{i+1}, exactly two rounds are spent. At σ^{i+1}, $q_{v_{i+1}} = \lambda$ holds.
2. Case that (2) is satisfied: For the execution from σ^i to σ^{i+1}, exactly one round is spent. At σ^{i+1}, $q_{v_{i+1}} = \lambda - 1$ holds.

Now we derive contradiction. Without loss of generality, we can assume $q_{v_0} = 0$ holds at σ^0.

Let x be the number of process pairs (v_i, v_{i+1}) among $\{(v_i, v_{i+1}) \mid 0 \leq i \leq m-2\}$ that satisfy (1) at σ^i. Then, the number of process pairs (v_i, v_{i+1}) among $\{(v_i, v_{i+1}) \mid 0 \leq i \leq m-2\}$ that satisfy (2) at σ^i is $m - x - 1$.

From the above observations on the number of rounds spent from σ^i to σ^{i+1}, $2x + (m - x - 1) = x + m - 1$ rounds are spent from σ^0 to σ^{m-1} and, thus, $q_v \leq x + m - 1$ holds at σ^{m-1}. This is because $q_v = 0$ holds at σ^0 and v can increment q_v by at most one at each round.

On the other hand, from the above observation on the states of v_i and v_{i+1}, $q_{v_{m-1}} = 2d - (m - x - 1)$ holds at σ^{m-1}. Because of $m \leq d$, $(q_{v_{m-1}} - q_v) \mod 2d \geq 2d - 2m + 2 \geq 2$ holds at σ^{m-1}, and thus edge (v, v_{m-1}) does not exist in $G.\sigma^{v_{m-1}}$ (Neither formula (1) nor (2) are satisfied). Therefore v cannot inherit the inheritance list from v_{m-1}. This contradicts to the fact that $v_m = v$.

Since no process appears in any inheritance list twice or more, the length of any inheritance list is at most n. From Lemma 8, the synchronous execution starting from any configuration reaches a configuration within $2n$ rounds where there is no inheritance list. In that configuration, there is no head process and the configuration is ideal. □

Lemma 6 and 9 lead the following theorem.

Theorem 4. *Let m be a positive integer. In any synchronous execution, each process of the alternator executes more than m actions within $2n + m$ rounds.* □

Since the alternator is an LST protocol, we obtain the following theorem from Theorem 1 and Theorem 4.

Theorem 5. *Let m be a positive integer. In any execution, each process of the alternator executes more than m actions within $2n + m$ rounds.* □

VII. Conclusion

We presented a new method to analyze efficiency of asynchronous protocols by observing their synchronous executions. The method is not universal and LST protocols are defined as a class of protocols to which the proposed method can be applied. It is worthwhile to say that several existing self-stabilizing protocols belong to the class. To show the effectiveness of the method, we applied the method to analyze efficiency of the alternator, a well-known self-stabilizing protocol for synchronization.

This paper proves the possibility of a new approach to efficiency analysis of asynchronous protocols. One of our future works is to extend the protocol class to which the proposed method can be applied and to extend the method so that it can be applied to a wider class of protocols. We believe that the proposed method or its variation can be applied to a wider class of protocols. The intuition behind the claim is that processes are least activated in the synchronous execution. Thus, the proposed method can be applied to any protocol with the property such that more activation guarantees more progress. For such protocols, the round complexity in the synchronous executions give us the worst-case round complexity in all executions.

Acknowledgments

This work was supported in part by JSPS, Grants-in-Aid for Scientific Research ((B)15300017), MIC, Strategic Information and Communications R&D Promotion Programme, and "The 21st Century Center of Excellence Program" of the Ministry of Education, Culture, Sports, Science and Technology, Japan.

References

[1]Tel, G., "Introduction to Distributed Algorithms," 2nd ed., Cambridge University Press, Cambridge, 2000.

[2]Dolev, S., "Self-stabilization," The MIT Press, Cambridge, 2000.

[3]Gouda, M. G., and Haddix, F., "The Alternator," Proceedings of the Workshop on Self-Stabilizing System, IEEE Computer Society, Washington, 1999, pp. 48–53.

[4]Herman, T., and Masuzawa, T., "Self-Stabilizing Agent Traversal," Proceedings of the Workshop on Self-Stabilizing Systems, Springer, New York, 2001, pp. 152–166.

[5]Kondou, D., Masuda, H., and Masuzawa, T., "A Self-Stabilizing Protocol for Pipelined PIF," Proceedings of the International Conference on Distributed Computing Systems, IEEE Computer Society, Washington, 2002, pp. 181–190.

[6]Johnen, C., Alima, L. O., Datta, A. K., and Tixeuil, S., "Self-Stabilizing Neighborhood Synchronizer in Tree Networks," Proceedings of the International Conference on Distributed Computing Systems, IEEE Computer Society, Washington, 1999, pp. 487–494.

[7]Brown, G. M., Gouda, M. G., and Wu, C. L., "A Self-Stabilizing Token System," Proceedings of the Hawaii International Conference on System Sciences, IEEE Computer Society, Washington, 1987, pp. 218–223.

[8]Gouda, M. G., and Haddix, F., "The Linear Alternator," Proceedings of the Workshop on Self-Stabilizing System, Carleston University Press, Montreal, 1997, pp. 31–47.

[9]Awerbuch, B., "Complexity of Network Synchronization," *Journal of the ACM*, Vol. 32, No. 4, 1985, pp. 804–823.

[10]Dijkstra, E. W., "Self-Stabilizing Systems in spite of Distributed Control," *Communications of the ACM*, Vol. 17, No. 11, 1974, pp. 643–644.

Shlomi Dolev
Associate Editor