故障封じ込め自己安定プロトコルに対する タイマーを利用した合成手法

山内 由紀子 *, 亀井 清華 †, 大下 福仁 *,
片山 喜章 ‡, 角川 裕次 *, 増澤 利光 *

* 大阪大学 大学院情報科学研究科 コンピュータサイエンス専攻
† 広島大学 大学院工学研究科 情報工学専攻
‡ 名古屋工業大学 大学院工学研究科 情報工学専攻

自己安定プロトコルは任意の数の一時故障からのシステムの自律的な復帰を保証する．故障封じ込め自己安定プロトコルは自己安定性とともに，小規模故障に対して，故障の影響の拡大を回避しながら，システムが迅速に復帰することを保証する．本稿では，故障封じ込めの性質を保存したまま，故障封じ込め自己安定プロトコルを合成する手法を提案する．提案手法は [1] で既に提案されている合成手法の適用制限を緩和し，より多くの故障封じ込め自己安定プロトコルに適用することができる．

Timer-based composition technique for self-stabilizing protocols preserving the fault-containment property

Yukiko Yamauchi*, Sayaka Kamei†, Fukuhito Ooshita*,
Yoshiaki Katayama‡, Hirotsugu Kakugawa* , Toshimitsu Masuzawa*

* Graduate School of Information Science and Technology, Osaka University
† Department of Information Engineering, Hiroshima University
‡ Graduate School of Computer Science and Engineering, Nagoya Institute of Technology

Self-stabilizing protocols provide autonomous recovery from finite number of transient faults. Fault-containing self-stabilizing protocols promise not only self-stabilization but also quick recovery and small effect from small scale of faults. In this paper, we introduce a timer-based composition of fault-containing self-stabilizing protocols that preserves the fault-containment property of source protocols. Our framework can be applied to a larger subclass of fault-containing self-stabilizing protocols than existing compositions [1].

1 Introduction

Large scale networks that consist of a large number of processes communicating with each other have been developed in these years. It is necessary to take measures against faults (e.g. memory crash at processes, topology change, etc.) when we design distributed protocols for large scale networks.

A self-stabilizing protocol converges to a legitimate configuration from any arbitrary initial configuration. Self-stabilization was first introduced by Dijkstra [2]. Since then, many self-stabilizing protocols have been designed for many problems [3, 4, 5]. The stabilization property provides autonomous adaptability against any number of transient faults that corrupt memory contents at processes. In practice, the adaptability to small scale faults is important because catastrophic faults rarely occur. However, self-stabilization does not promise efficient recovery from small scale faults and sometimes the effect of a fault spreads over the entire network.

When a fault corrupts f processes by changing their memory contents arbitrarily in a legitimate configuration, the obtained configuration is called an *f-faulty configuration*. An *f-fault-containing self-stabilizing protocol* promises self-stabilization and fault-containment [6, 7, 8]: starting from an f'-faulty configuration ($f' \leq f$), it reaches a legitimate configuration in the time and with the number of processes affected proportional to f or less.

Executing two different self-stabilizing protocols in parallel is well known as *fair composition* [5]. Fair composition provides hierarchical composition of two (or more) self-stabilizing protocols such that the output of one protocol (called the lower protocol) is used as the input to the other (called the upper protocol), and guarantees self-stabilization of the obtained protocol. However, fair composition cannot preserve the fault-containment property of source protocols when composing fault-containing self-stabilizing protocols.

Related work. *Global neighborhood synchronizers* are often used as a fundamental component in the context of fault-containment. Global synchronization is used for each process to measure time to correct some informations or to keep its state unchanged for some period of time. Ghosh et al. proposed a technique to transform a non-reactive self-stabilizing protocol to a corresponding 1-fault-containing protocol [7]. Their transformer utilizes a global neighborhood synchronizer that provides synchronization from 1-faulty configuration. An obtained 1-fault-containing protocol guarantees that the *output* of the protocol recovers quickly. However, the effect of a fault spreads over the entire network in the global neighborhood synchronizer. This is because the global neighborhood synchronizer involves all processes in the network into the synchronization. However, the expected property for fault-containment is temporal and spatial containment of the effect of faults: the recovery actions are taken when and where it is necessary.

Contributions. Yamauchi et al.[1] defined composition of fault-containing self-stabilizing protocols, which they call *fault-containing composition* and proposed the first composition technique for fault-containing composition. Recovery Waiting Fault-containing Composition (*RWFC*) strategy is to prevent the execution of the upper protocol until the lower protocol recovers. In [1], *RWFC* strategy is implemented as follows: each process evaluates a *local predicate* to check local consistency of the current configuration of the lower protocol whenever the process wants to execute the upper protocol. If the process finds the lower protocol locally consistent, then the process executes the upper protocol. Otherwise, the process cannot execute the upper protocol. However, each process has to communicate with distant processes to evaluate the local predicate. Moreover, they put many restrictions on source protocols and it regulates the application of the composition framework.

In this paper, we present a novel timer-based technique for fault-containing composition. Though we adopt *RWFC* strategy, the proposed composition utilizes *recovery time* of fault-containing protocols. Recovery time is the maximum time for the system to recover from a target faulty configuration. We force the upper protocol to stop during the recovery time of the lower protocol. After that, the upper protocol can execute on the correct input from the lower protocol. Thus, the upper protocol can recover with its fault-containment property and the composite protocol promises fault-containment as a whole.

Our framework does not need communications among distant processes and relaxes the restrictions on source protocols: in [1] it is necessary that each process has to keep detecting the inconsistency of the lower protocol during the recovery of the lower protocol by communicating with distant processes while in this paper each process has to detect the inconsistency of the lower protocol in the initial configuration by communicating direct neighbors.

We use *local timers* at processes to measure the recovery times of the source protocols. Global neighborhood synchronizers are often used to implement local timers. However, a fault-containing protocol bounds the effect of faults with *contamination radius*: the maximum (worst) distance from any faulty process to any process affected by the faulty process is smaller than or equals to the contamination radius. We introduce a *local neighborhood synchronizer* that emulates M synchronized rounds among the k-neighbors of the initiator that initiates the synchronization.

2 Preliminary

A system is a network which is represented by an undirected graph $G = (V, E)$ where the vertex set V is a set of processes and the edge set E is a set of bidirectional communication links. Each process has a unique identity. Process p is a neighbor of process q iff there exists a communication link $(p, q) \in E$. A set of neighbors of p is denoted by N_p. Let $N_p^0 = \{p\}$, $N_p^1 = N_p$ and for each $i \geq 2$, $N_p^i = \bigcup_{q \in N_p^{i-1}} N_q \setminus \{p\}$. The set of processes denoted by N_p^i is called *i-neighbor* of p. The distance between p and q ($q \neq p$) is denoted by $dist(p, q)$ and $dist(p, q) = j$ iff $q \notin N_p^{j-1} \wedge q \in N_p^j$.

Each process p maintains local variables and the values of all local variables at p define the local state of p. Local variables are classified into three classes: input, output, and inner. The input variables indicate the input to the system and they are not changed by the system. The output variables are the output of the system for external observers. The inner variables are internal working variables used to compute output variables.

We adopt *locally shared memory model* as a communication model: each process p can read the value of the local variables at $q \in N_p \cup \{p\}$. A protocol at each process p consists of a finite number of *guarded actions* in the form of $\langle guard \rangle \rightarrow \langle action \rangle$. A $\langle guard \rangle$ is a boolean expression involving the local variables of p and N_p, and an $\langle action \rangle$ is a statement that changes the value of p's local variables (except input variables). A process with a guard evaluated *true* is called *enabled*. We adopt *distributed daemon* as a scheduler: in a computation step, distributed daemon selects a nonempty set of enabled processes and these processes execute the corresponding actions. The evaluation of guards and the execution of the corresponding action is *atomic*: these computations are done without any interruption. A configuration of a system is represented by a tuple of local states of all processes. An *execution* is an infinite sequence of configurations $E = \sigma_0, \sigma_1, \sigma_2, \cdots$ such that σ_{i+1} is obtained by applying one computation step to σ_i or σ_{i+1} is the final configuration.

Distributed daemon allows *asynchronous* executions. In an asynchronous execution, the time is measured by computation steps or *rounds*. Let $E = \sigma_0, \sigma_1, \sigma_2, \cdots$ be an asynchronous execution. The first round $\sigma_0, \sigma_1, \sigma_2, \cdots, \sigma_i$ is the minimum prefix of E such that for each process $p \in V$ if p is enabled in σ_0, either p's guard is disabled or p executes at least one step in $\sigma_0, \sigma_1, \sigma_2, \cdots \sigma_i$. The second and latter rounds are defined recursively by applying the definition of the first round to the remaining suffix of the execution $E' = \sigma_{i+1}, \sigma_{i+2}, \cdots$.

A *problem* (task) T is defined by a legitimate predicate on configurations. A configuration σ is *legitimate* iff σ satisfies the legitimate predicate. In this paper we treat *non-reactive* problems: no process changes its state after the system reaches a legitimate configuration, e.g. spanning tree construction, leader election, etc. We say a distributed protocol $P(T)$ solves T in a configuration iff the configuration satisfies the legitimate predicate $L(P(T))$. The input (output) of $P(T)$ is represented by the conjunction of input (output, respectively) variables at each process. We omit T if T is clear.

Definition 1 ***Self-stabilization***
Protocol P is self-stabilizing iff it satisfies the following two properties:
Stabilization : *starting from any arbitrary initial configuration, it eventually reaches a legitimate configuration.*
Closure : *once it reaches a legitimate configuration, it remains in legitimate configurations thereafter.*

A *transient fault* corrupts some processes by changing the values of their local variables arbitrarily. A configuration is *f-faulty* iff the minimum number of processes such that we have to change their local states (except inner variables) to make the configuration legitimate is f. We say process p is *faulty* iff we have to change p's local state to make the configuration legitimate and otherwise *correct*.

Definition 2 ***f-fault-containment***
A self-stabilizing protocol is f-fault-containing *iff it reaches a legitimate configuration from any f'-faulty configuration ($f' \leq f$) with the number of processes that change their states according to the fault and the time to reach a legitimate configuration depending on f (not $|V|$).*

We denote an f-fault-containing self-stabilizing protocol as an f-fault-containing protocol. The performance of an f-fault-containing protocol is measured by stabilization time, recovery time, and contamination radius:
Stabilization time : the maximum (worst) number of rounds to reach a legitimate configuration from an arbitrary initial configuration.
Recovery time : the maximum (worst) number of rounds to reach a legitimate configuration from an f'-faulty configuration ($f' \leq f$).
Contamination radius : the maximum distance from any faulty process to the process that changes its local state according to the faulty process during the recovery from an f'-faulty configuration ($f' \leq f$).

A hierarchical composition of two protocols P_1 and P_2 is denoted by $(P_1 * P_2)$ where the variables of P_1 and those of P_2 are disjoint except that the input to P_2 is the output of P_1. We define the output variables of $(P_1 * P_2)$ is the output variables of P_2. A legitimate configuration of $(P_1 * P_2)$ is defined by $L((P_1 * P_2))$ where $L(P_1 * P_2) = L(P_1) \wedge L(P_2)$.

Definition 3 ***Fault-containing composition***
Let P_1 be an f_1-fault-containing protocol and P_2 be an f_2-fault-containing protocol. A hierarchical composition $(P_1 * P_2)$ is a fault-containing composition of P_1 and P_2 iff $(P_1 * P_2)$ is an $f_{1,2}$-fault-containing protocol for some $f_{1,2}$ such that $0 < f_{1,2} \leq \min\{f_1, f_2\}$.

In a hierarchical composition, the input to P_2 can be corrupted by a fault when the fault corrupts the output variables of P_1. However, the input to P_1 can be seen as the system parameters, e.g. topology, ID of each process, etc.

Assumption 1 *For any hierarchical composition $(P_1 * P_2)$, the input to P_1 is not corrupted by any fault.*

We consider a subclass of fault-containing protocols Π such that each f-fault-containing protocol $P \in \Pi$ satisfies Assumption 2, 3, and 4. Many existing fault-containing protocols [6, 8] satisfy Assumption 2, 3, and 4.

Assumption 2 *The legitimate configuration of P is uniquely defined by the input variables.*

Consider a composition $(P_1 * P_2)$ of an f_1-fault-containing protocol P_1 and an f_2-fault-containing protocol P_2. Starting from an f'-faulty configuration ($f' \leq \min\{f_1, f_2\}$), if the output of P_1 after P_1 reaches a legitimate configuration is different from what it was before the fault, then the input to P_2 changes and the output of P_2 may change drastically to adopt it. Then, P_2 cannot guarantee fault-containment. Because the input to P_1 is not changed by any fault (Assumption 1), Assumption 2 guarantees that P_1 recovers to the unique legitimate configuration and ensures the possibility of fault-containment of P_2 in the composite protocol.

Assumption 3 *The legitimate predicate $L(P)$ for P is represented in the form $L(P) \equiv \forall p \in V : cons_p(P)$. The predicate $cons_p(P)$ involves the local variables at p and its neighbors, and it is defined over the values of output, inner, and input variables.*

We say process p is *inconsistent* iff $cons_p(P)$ is *false*, otherwise *consistent*. Because we work on non-reactive problems, the predicate $cons_p(P)$ is evaluated *false* when process p is enabled.

Assumption 4 *In an f'-faulty configuration ($f' \leq f$), if a faulty process p is a neighbor of correct process(es), at least one correct process q neighboring to p or p itself evaluates $cons_q(P)$ (or $cons_p(P)$) false.*

For a faulty process p and a neighboring correct process q, $cons_p(P)$ ($cons_q(P)$, respectively) involves the local variables at q and p. Because p is faulty, there can be some inconsistency between the local state of p and that of q.

3 The Composition Framework

Let P_1 be an f_1-fault-containing protocol and P_2 be an f_2-fault-containing protocol. Our goal is to produce $f_{1,2}$-fault-containing protocol $(P_1 * P_2)$ for $f_{1,2} = \min\{f_1, f_2\}$. In the rest of the paper, we use the notations shown in Table 1.

Table 1: Notations for the source protocols and the composite protocol

protocol	number of maximum faults	recovery time	contamination number	inconsistency range
P_1	f_1	r_1	c_1	k_1
P_2	f_2	r_2	c_2	k_2
$(P_1 * P_2)$	$f_{1,2} = \min\{f_1, f_2\}$	$r_{1,2}$	$c_{1,2}$	$k_{1,2}$

Fair composition of fault-containing protocols cannot preserve the fault-containment property of source protocols. When a fault corrupts the output variables of P_1 at f processes ($f \leq f_{1,2}$), during the recovery of P_1, P_2 can be executed in parallel to adopt the changes in the output variables of P_1. The number of contaminated processes in P_1 may become larger than f_2 and this causes the number of processes that change their local states in P_2 becomes larger than f_2. These processes can change its state repeatedly until P_1 recovers. If more than f_2 processes change their states repeatedly in P_2, then P_2 cannot guarantee fault-containment even though f (the number of the processes that the original fault corrupts) is smaller than f_2.

We implement *RWFC* strategy with *local timers* at processes. We implement timers at processes with a *local neighborhood synchronizer* that synchronizes the processes in $\max\{c_1, c_2\}$-neighbors for each faulty process for $(r_1 + r_2)$ rounds. We first define the specification of the local neighborhood synchronizer in Section 3.1 and show our composition framework in Section 3.2. Finally, we present an implementation of the local neighborhood synchronizer in Section 3.3.

3.1 Specification of the Local Neighborhood Synchronizer

In this section we define a specification of our local neighborhood synchronizer for fault-containing composition $(P_1 * P_2)$.

Specification 1 *Each process $p \in V$ maintains a counter variable t_p that takes an integer in $[0..(r_1+r_2)]$. The local neighborhood synchronizer is self-stabilizing and in a legitimate configuration, $t_p = 0$ holds at $\forall p \in V$.*

The local neighborhood synchronizer should be implemented with a typical technique of synchronizers [7]. We say a process is *s-consistent* iff its counter variable differs at most one with those at all its neighbors involved in the synchronization. Synchronization is realized by making each counter variable s-consistent and then decrementing it with preserving the s-consistency.

Synchronization radius is the maximum distance from any faulty process and a process involved in the synchronization caused by the faulty process. From Assumption 4, the distance between a process that finds inconsistency in the source protocols and any contaminated process is at most $k_{1,2} = \max\{c_1, c_2\} + \max\{f_1, f_2\}+1$. It is necessary to involve all $k_{1,2}$-neighbors for each faulty process into the synchronization so that all $\max\{c_1, c_2\}$-neighbors of each faulty process are involved in the synchronization.

A *counter sequence* of process p is the sequence of the value of t_p from an initial configuration.

Specification 2 *Starting from an f-faulty configuration ($f \leq f_{1,2}$), the local neighborhood synchronizer should provide the following five properties:*
Containment: *synchronization radius is $O(k_{1,2})$.*
Synchronization: *each processes involved in the synchronization decrements its counter variable with keeping s-consistency.*
Correct sequence: *a counter sequence $v_p^0, v_p^1, \cdots$ of any correct process p involved in the synchronization has a prefix $v_p^0, v_p^1, \cdots, v_p^{i-1}, v_p^i$ for some i such that $v_p^0 = v_p^1 = \cdots = v_p^{i-1} = 0$ and $v_p^i = r_1 + r_2$.*
Faulty sequence: *a counter sequence $v_q^0, v_q^1, \cdots$ of any faulty process q has a suffix $v_q^i, v_q^{i+1}, \cdots, v_q^j, \cdots$ for some i and j such that $v_q^k - v_q^{k+1} \leq 1$ for $i \leq k \leq j$ and $v_q^j = v_q^{j+1} = \cdots = 0$.*
Termination: *the local neighborhood synchronizer reaches a legitimate configuration in $(r_1 + r_2 + O(1))$ rounds.*

We do not assume that faulty processes decrement their counter variables from (r_1+r_2). From Assumption 4, when a faulty process is surrounded by other faulty processes, it cannot determine whether it is correct or not.

Specification 3 *The following APIs are available at each process $p \in V$ for the application of the local neighborhood synchronizer:*
call_start_synch_NS: *when this function call is executed at process p, it starts the synchronization involving $k_{1,2}$-neighbors of p. These processes decrements their counter variables from $(r_1 + r_2)$ to 0 with keeping s-consistency and the system reaches the legitimate configuration in $O(r_1 + r_2)$ rounds.*
call_exec_NS: *when this function call is executed at process p, if p is enabled in the local neighborhood synchronizer, then it executes one of the corresponding actions and if p decrements t_p, this function call returns true, otherwise false. If p is not enabled, then p does nothing and this function call returns $\perp$.*

3.2 The Framework *FC-LNS*

Our composition framework *FC-LNS* (*Fault-containing Composition with the Local Neighborhood Synchronizer*) is shown in Figure 1. Process p executes the guarded actions of the local neighborhood synchronizer by executing call_exec_NS, and whenever it decrements t_p, p executes the source protocols by executing the procedure $A(t_p)$ that selects which source protocol is executed at p. If p finds inconsistency in P_1 when $0 < t_p \leq r_2$ or in P_1 or P_2 when $t_p = 0$, then it initiates the synchronization of the local neighborhood synchronizer by executing call_start_synch_NS. Thus, p and its $k_{1,2}$-neighbors execute P_1 untill P_1 reaches the legitimate configuration. After that, they executes P_2 on the correct output from P_1 and P_2 reaches the legitimate configuration with its fault-containment property.

Procedure $A(t_p)$ **for process** p
  if $(r_2 \leq t_p < r_1 + r_2)$ **then execute** P_1
  else execute P_2;

Actions for any process p
  true $\longrightarrow$
  if call_exec_NS $=$ *true* **then** $A(t_p)$;
  if $\{(0 < t_p \leq r_2) \wedge \neg cons_p(P_1)\} \vee \{(t_p = 0) \wedge (\neg cons_p(P_1) \vee \neg cons_p(P_2))\}$
    then call_start_synch_NS

Figure 1: *FC-LNS*

Theorem 1 *FC-LNS provides a* $\min\{f_1, f_2\}$*-fault-containing protocol* $(P_1 * P_2)$ *for an* f_1*-fault- containing protocol* P_1 *and* f_2*-fault-containing protocol* P_2*. The contamination radius of the obtained protocol is* $O(\max\{c_1, c_2\} + \max\{f_1, f_2\})$ *and the recovery time is* $O(r_1 + r_2)$*.*

Proof. For each faulty process p, each process $q \in N_p^{\max\{c_1,c_2\}}$ counts down t_q from $(r_1 + r_2)$ to 0. Thus, P_1 first reaches the legitimate configuration with its fault-containment property and then P_2 reaches the legitimate configuration with its fault-containment property. Each process involved in the synchronization can execute P_2 on the correct input from P_1 in $O(r_1 + r_2)$ rounds and the recovery time of the obtained protocol is $O(r_1 + r_2)$.

Starting from an f-faulty configuration ($f \leq f_{1,2}$), call_start_synch_NS is executed at faulty processes and some correct processes neighboring a faulty process. Thus, contamination radius of the obtained protocols is $O(\max\{c_1, c_2\} + \max\{f_1, f_2\})$. $\square$

3.3 Local Neighborhood Synchronizer

In this section we present an implementation of the local neighborhood synchronizer *LNS* that meets the specifications in Section 3.1.

For any given M and k, *LNS* provides the synchronization of M rounds among k-neighbors of the *initiator*. The synchronization consists of three phases. In the first phase, an initiator arises and the shortest path tree rooted at the initiator is constructed to involve all the k-neighbors of the initiator into the synchronization. Then, in the second phase, the synchronized count-down of counter variables takes place among k-neighbors of the initiator. In the third phase, the shortest path tree is released from the root to the leaves.

Each process p has two variables, t_p and d_p: t_p is the counter variable and d_p is the depth variable which is used to construct the shortest path tree. In a legitimate configuration, $t_p = 0 \wedge d_p = 0$ holds at $\forall p \in V$.

Let p be an initiator. Each process $q \in N_p^k$ constructs the shortest path tree by setting $d_q = k - dist(p, q)$ where $dist(p, q)$ denotes the distance between p and q. The parent(s) of q is any neighbor $r \in N_q$ where $d_r = d_q + 1$. A process $s \in N_q$ is a child of q iff $d_s = d_q - 1$.

Predicates

$safe\_d_p \equiv \{d_p = k\} \vee \{d_p = 0\} \vee \{(0 < d_p < k) \wedge (\exists q \in N_p : d_q - d_p = 1)\}$
$OK\_d_p \equiv safe\_d_p \wedge (\forall q \in N_p : |d_p - d_q| \leq 1)$
$safe\_t_p \equiv \{t_p = 0\} \vee \{t_p = M\} \vee \{(\exists q \in N_p : |t_p - t_q| \leq 1) \wedge$
$\quad (\forall q \in N_p : (t_q = 0 \wedge (t_p = M \vee d_p = 0)) \vee (|t_p - t_q| \leq 1))\}$
$OK\_t_p \equiv safe\_t_p \wedge [\{t_p = 0\} \vee \{(d_p > 0) \wedge (\forall q \in N_p : |t_p - t_q| \leq 1)\} \vee$
$\quad \{(d_p = 0) \wedge (\exists q \in N_p : d_q = 1 \wedge |t_p - t_q| \leq 1)\}]$
$init_p \equiv I_p(1) \vee I_p(2) \vee I_p(3)$
$\quad I_p(1) \equiv \{(t_p > 0) \vee (d_p > 1)\} \wedge \neg(t_p = M \wedge d_p = k) \wedge \{\forall q \in N_p : t_q = 0 \wedge d_q = 0\}$
$\quad I_p(2) \equiv \{(0 < d_p < k) \wedge (\forall q \in N_p : d_p \geq d_q) \wedge (\exists q \in N_p : d_q > 0)\}$
$\quad I_p(3) \equiv \{\neg safe\_t_p \wedge (t_p \neq M \vee d_p \neq k) \wedge (\forall q \in N_p : t_p \geq t_q)\}$
$raise_p \equiv R_p(1) \wedge R_p(2)$
$\quad R_p(1) \equiv (t_p \neq M)$
$\quad R_p(2) \equiv \{\exists q \in N_p : (t_q = M) \wedge (d_q > 0) \wedge \neg((t_p = M - 1) \wedge (d_p = d_q - 1)) \wedge (\forall r \in N_p : d_r < d_p \rightarrow t_r = M)\}$
$maxd_p \equiv M_p(1) \wedge M_p(2)$
$\quad M_p(1) \equiv (t_p > 0) \wedge (d_p \neq k)$
$\quad M_p(2) \equiv (\max_{q \in N_p}\{d_q\} \neq 0) \wedge (\max_{q \in N_p}\{d_q\} - 1 > d_p)$
$dec_p \equiv OK\_d_p \wedge OK\_t_p \wedge D_p(1) \wedge D_p(2)$
$\quad D_p(1) \equiv (t_p > 0) \wedge (\forall q \in N_p : t_p \geq t_q)$
$\quad D_p(2) \equiv (\forall q \in N_p : t_p = t_q \rightarrow d_p \geq d_q)$
$clrd_p \equiv C_p(1) \wedge C_p(2)$
$\quad C_p(1) \equiv (t_p = 0) \wedge \{\forall q \in N_p : (t_q = 0)\}$
$\quad C_p(2) \equiv (d_p > 0) \wedge \{\forall q \in N_p : (d_p \geq d_q) \vee (d_q = 0)\}$

Actions for any process p

$S_1 \quad init_p \vee Predicate_p^{init} \longrightarrow t_p = M;\ d_p = k$
$S_2 \quad raise_p \longrightarrow t_p = M;$ **if** $(\neg((\max_{q \in N_p}\{d_q\} = k - 1) \wedge (d_p \neq k)))$ **then** $d_p = \max_{q \in N_p}\{d_q\} - 1$
$S_3 \quad maxd_p \longrightarrow d_p = \max_{q \in N_p}\{d_q\} - 1$
$S_4 \quad dec_p \longrightarrow t_p = t_p - 1;\ Action_p^{dec}$
$S_5 \quad clrd_p \longrightarrow d_p = 0$

Figure 2: *LNS* $(Predicate_p^{init}, Action_p^{dec})$

The protocol *LNS* is shown in Figure 2. Parameter $Predicate_p^{init}$ is a predicate that involves local variables at p and all its neighbors and parameter $Action_p^{dec}$ is a set of actions that changes the value of local variables at p except t_p and d_p.

To distinguish process p's state, we introduce the four predicates: $safe\_d_p$, OK_d_p, $safe\_t_p$, and OK_t_p. The predicate $safe\_d_p$ is evaluated *true* when p has at least one parent iff p is an internal process ($0 < d_p < k$). The predicate OK_d_p is evaluated *true* when p is an internal process and it has at least one parent and other neighbors are its children or when p is not an internal process and d_p differs at most one with all its neighbors. The predicate $safe\_t_p$ is evaluated *true* iff p has at least one neighbor that $|t_p - t_q| \leq 1$ and other neighbors wait to join the shortest path tree. The predicate OK_t_p is evaluated *true* when p is an initiator or an internal process ($d_p > 0$) and $\forall q \in N_p : |t_p - t_q| \leq 1$ holds or when p is not a leaf process ($d_p = 0$) and it has at least one parent q where $|t_p - t_q| \leq 1$ holds. OK_t_p represents the consistency of t_p and OK_d_p represents the consistency of d_p. If OK_t_p and OK_d_p hold at process p, p attended the shortest path tree correctly and t_p is synchronized with all its neighbors.

The first phase starts when some process, called *initiator*, executes S_1. Process p that satisfies one of the following conditions executes S_1 and sets $t_p = M$ and $d_p = k$: (a) it finds its variables corrupted and other neighbors are correct ($I_p(1) = ture$), (b) it was involved in a shortest path tree but there is no correct parent ($I_p(2) = ture$), (c) it finds counter variables at itself and at neighbors not s-consistent and the value of t_p is larger than those at all neighbors ($I_p(3) = ture$). Note that in a 1-faulty configuration, a faulty process p cannot find its corruption with $I_p(1)$ if $t_p = 0 \wedge d_p = 1$ holds. This is because, when the shortest path tree is released after the synchronized count-down, $t_p = 0 \wedge d_p = 1$ holds just before p sets d_p to 0.

After p executes S_1, each process $q \in N_p^k$ executes S_2 (and S_3 if necessary) and q is involved in the shortest path tree by setting $t_q = M$ and $d_q = k - dist(p, q)$. When $t_q \neq M$ ($R_q(1) = true$), if process q finds that $t_r = M$ holds at some neighbor r that is not its parent ($R_q(2) = true$), then q executes S_2 and becomes a child of r by setting $t_q = M$ and $d_q = d_r - 1$. However, d_q does not always takes the value $k - dist(p, 1)$ after it executes S_2. Then, q updates d_q by executing S_3 whenever it finds a neighbor s where $d_s > d_q + 1$ ($M_q(2) = true$). After $t_q = M \wedge d_q = k - dist(p, q)$ holds at q and all its neighbors get involved in the shortest path tree, q goes into the second phase.

In the second phase, q decrements t_q by executing S_4. The synchronization is realized by decrementing t_q with keeping the s-consistency ($D_q(2) = true$). To keep the s-consistency among counter variables at all the neighbors, we force q to decrement its counter variable iff the value of t_q is locally maximum ($D_q(1) = true$). Thus, after q decrements t_q, $|t_q - t_r| \leq 1$ holds for $\forall r \in N_q$. Process q decrements t_q after each $s \in N_q$ where $d_s > d_q$ decremented its counter variable. Thus, the execution of S_4 starts from the initiator and each process $q \in N_p^k \cup \{p\}$ counts down t_q from M to 0. The second phase finishes when t_q reaches 0.

In the third phase, after all the neighbors finish the count-down ($C_q(1) = true$), q executes S_5 and sets $d_q = 0$. However, q waits its parent to execute S_5 ($C_q(2) = true$). So, the execution of S_5 also starts from the initiator to the leaf and the shortest path tree is released. Eventually, the third phase ends and $t_q = 0 \wedge d_q = 0$ holds at $\forall q \in V$.

APIs of *LNS* defined in Specification 3 is given as its parameters. We should set $Predicate_p^{init}$ and $Action_p^{dec}$ as follows:

$$\begin{aligned} Predicate_p^{init} =& \{(0 < t_p \leq r_2) \wedge \neg cons_p(P_1)\} \vee \{(t_p = 0) \wedge \neg(cons_p(P_1) \wedge cons_p(P_2))\} \\ Action_p^{dec} =& A(t_p) \end{aligned}$$

The following theorem holds for *LNS* .

Theorem 2 *Protocol LNS is self-stabilizing.*

Lemma 1, 2, 3, 4, and 5 holds for *LNS* and *LNS* provides the five specification in Specification 1 and 2 with $M = r_1 + r_2$ and $k = k_{1,2}$. (Due to space limitation, we omit proofs for these lemmas.)

Lemma 1 ***(Containment)***
Starting from an f-faulty configuration ($f \leq f_{1,2}$), LNS provides the containment property.

Lemma 2 ***(Synchronization)***
Starting from an f-faulty configuration ($f \leq f_{1,2}$), LNS provides the synchronization property.

Lemma 3 ***(Correct sequence)***
Starting from an f-faulty configuration ($f \leq f_{1,2}$), LNS provides the correct sequence property.

Lemma 4 ***(Faulty sequence)***
Starting from an f-faulty configuration ($f \leq f_{1,2}$), LNS provides the faulty sequence property.

Lemma 5 ***(Termination)***
Starting from an f-faulty configuration ($f \leq f_{1,2}$), LNS provides the termination property.

4 Conclusion

We proposed a novel timer-based fault-containing composition. To implement timers, we designed a local neighborhood synchronizer protocol. Local neighbor synchronizers are very useful in the field of fault-containment, e.g. adding fault-containment property to self-stabilizing protocols by using a local neighborhood synchronizer. Some specific implementation of local neighborhood synchronizers should be developed for each application.

Our next goal is to establish a composition framework for various types of source protocols preserving their fault-tolerance.

Acknowledgement. This work is supported in part by JSPS Research Fellowships for Young Scientists, Global COE (Centers of Excellence) Program of MEXT, Grant-in-Aid for Scientific Research ((B)19300017, (B)17300020, (B)20300012, and (C)19500027)) of JSPS, Grand-in-Aid for Young Scientists ((B)18700059 and (B)19700075) of JSPS, and Kayamori Foundation of Informational Science Advancement.

References

[1] Y. Yamauchi, S. Kamei, F. Ooshita, Y. Katayama, H. Kakugawa, and T. Masuzawa. Composition of fault-containing protocols based on recovery waiting fault-containing composition framework. In *Proceedings of SSS2006*, pages 516–532, 2006.

[2] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of ACM*, 17(11):643–644, 1974.

[3] N. S. Chen, H. P. Yu, and S. T. Huang. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters*, 39:147–151, 1991.

[4] S. T. Huang and N. S. Chen. Self-stabilizing depth-first token circulation on networks. *Distributed Computing*, 7(1):61–66, 1993.

[5] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems. In *Proceedings of WSS 1989*, 1989.

[6] S. Ghosh and A. Gupta. An exercise in fault-containment: self-stabilizing leader election. *Information Processing Letters*, 59(5):281–288, 1996.

[7] S. Ghosh, A. Gupta, T. Herman, and S. V. Pemmaraju. Fault-containing self-stabilizing algorithms. In *Proceedings of PODC 1996*, pages 45–54, 1996.

[8] S. Ghosh and X. He. Fault-containing self-stabilization using priority scheduling. *Information Processing Letters*, 73:145–151, 2000.