PAPER

Hierarchical Composition of Self-Stabilizing Protocols Preserving the Fault-Containment Property

Yukiko YAMAUCHI[†a)], ***Student Member***, **Sayaka KAMEI**[††], **Fukuhito OOSHITA**[†], **Yoshiaki KATAYAMA**[†††], ***Members***, **Hirotsugu KAKUGAWA**[†], ***Nonmember***, ***and*** **Toshimitsu MASUZAWA**[†], ***Member***

SUMMARY A desired property of large distributed systems is self adaptability against the faults that occur more frequently as the size of the distributed system grows. Self-stabilizing protocols provide autonomous recovery from finite number of transient faults. Fault-containing self-stabilizing protocols promise not only self-stabilization but also containment of faults (quick recovery and small effect) against small number of faults. However, existing composition techniques for self-stabilizing protocols (e.g. fair composition) cannot preserve the fault-containment property when composing fault-containing self-stabilizing protocols. In this paper, we present *Recovery Waiting Fault-containing Composition* (*RWFC*) framework that provides a composition of multiple fault-containing self-stabilizing protocols while preserving the fault-containment property of the source protocols.

key words: *fault-containment, self-stabilization, composition*

1. Introduction

Large scale networks that consist of a large number of processes communicating with each other have been developed in recent years. In large scale networks such as the Internet, it is desirable that the system recovers from small scale faults without the effect of faults spreading over the entire network. In dynamic networks such as sensor networks and inter-vehicle networks, it is expected that the system recovers quickly after a fault so that it can adapt to the dynamic changes. A distributed system consists of processes that communicate with each other by communication links. It is necessary to take measures against faults when we design distributed protocols because faults often occur in these networks (e.g. memory crash at processes, topology change) and the effect of faults may affect the entire network. There exist many fault-tolerant distributed protocols that provide autonomous recovery from faults or prevent the effect of faults from spreading over the network. A self-stabilizing protocol [8] converges to a legitimate configuration from any arbitrary initial configuration. This property provides autonomous adaptability against any number of transient faults. 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.

Researchers have tried to add adaptive fault-tolerance by conditioning the fault scenario, e.g. the severity of a fault. The severity of a fault is measured by the number of the processes corrupted by the fault. When the states of f processes are corrupted in a legitimate configuration, we call the obtained configuration an *f-faulty configuration*.

An f-fault-containing self-stabilizing protocol promises self-stabilization and fault-containment [15]–[17], [21]: from any f'-faulty configuration ($f' \leq f$), it reaches a legitimate configuration in the time and in the space depending on f or less. (We call it f-fault-containing protocol.) Many fault-containing protocols bound the time to recover and the number of processes affected by the fault with polynomial in f or some constant.

Executing two different self-stabilizing protocols in parallel is known as *fair composition* [9], [10]. Fair composition provides hierarchical composition of two (or more) protocols such that a protocol (called the upper protocol) utilizes as its input the output of the other (called the lower protocol), and guarantees self-stabilization of the obtained protocol. However, a fair composition does not preserve the fault-containment property when composing fault-containing protocols.

In this paper, we present a simple framework for composition of fault-containing protocols that preserves the fault-containment property of source protocols. This composition framework is important both theoretically and practically. Our strategy is to control the execution of source protocols so that the upper protocol waits until the lower protocol recovers. Our framework suggests the possibility of a uniform framework for composition of fault-containing protocols and a novel design technique for fault-containing protocols. However the proposed framework currently puts several assumptions on source protocols. So, we examine the sufficient conditions for the proposed framework.

Related work. Self-stabilization was first introduced by Dijkstra [8]. Since then, many self-stabilizing protocols have been designed for many problems e.g. spanning tree construction [6], [10], leader election [24] and token circulation [20].

Manuscript received June 30, 2008.
Manuscript revised November 6, 2008.

[†]The authors are with the Graduate School of Information Science and Technology, Osaka University, Toyonaka-shi, 560–8531 Japan.

[††]The author is with the Department of Information Engineering, Graduate School of Engineering, Hiroshima University, Higashihiroshima-shi, 739–8527 Japan.

[†††]The author is with the Graduate School of Computer Science and Engineering, Nagoya Institute of Technology, Nagoya-shi, 466–8555 Japan.

a) E-mail: y-yamaut@ist.osaka-u.ac.jp

DOI: 10.1587/transinf.E92.D.451

Copyright © 2009 The Institute of Electronics, Information and Communication Engineers

Various types of adaptive self-stabilization have been studied to improve the recovery of self-stabilizing protocols from faults: fault-containment [15], [16], time-adaptive stabilization [23], superstabilization [12], local stabilization [1], and time-to-fault adaptive stabilization [14]. The main issue is the time complexity for recovery. Their aim is to guarantee the recovery time bounded by the number of corrupted processes in an initial configuration. Fault-containing protocols were presented by Ghosh et al. [15], [16]. Many fault-containing protocols can be obtained by adding the property of fault-containment to existing self-stabilizing protocols. Ghosh et al. introduced fault-containment using priority scheduler in [18]. Priority scheduler provides a weak priority rule that makes the recovery actions of faulty processes precede the actions of correct processes. There exist such fault-containing protocols obtained by composing multiple layers of protocols where each protocol is not fault-containing by itself [2], [3].

Composition of self-stabilizing protocols is expected to ease the design of new protocols and to extend usability of existing protocols. The fair composition of self-stabilizing protocols was introduced by Dolev et al. [9], [10]. Beauquier et al. introduced a cross-over composition in [4] which uses the lower protocol as a filter to the execution of the upper protocol and improves the adaptability to scheduler. Dolev et al. proposed parallel composition in [13], that enables parallel search and accelerates the stabilization by executing multiple self-stabilizing protocols in parallel. The synthesize technique is also used in [14]. However a composition of fault-containing protocols has not been proposed and we first present such a composition.

Contribution. In this paper, we present a framework for composition of fault-containing protocols that guarantees the obtained protocol is also fault-containing. We call this composition *fault-containing composition*. Our strategy is to stop the upper protocol until the lower protocol recovers so that the upper protocol executes on the correct output of the lower protocol. This framework can be applied to a large subclass of fault-containing protocols but the constraint seems to be reasonable.

2. Preliminary

2.1 Network and Processes

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 $N_p^0 = \{p\}$, and for each $i \geq 1$, $N_p^i = \bigcup_{q \in N_p^{i-1}} N_q \setminus \{p\} \cup \bigcup_{0 \leq j \leq i-1} N_p^j$. The set of processes denoted by N_p^i is called *i-neighbor* of p. The distance between p and q ($p \neq q$) is denoted by $dist(p, q) = j$ iff $q \notin N_p^{j-1} \wedge q \in N_p^j$. The i-neighbor of p denotes the set of processes such that their distances from p are smaller than or equal to i but except p.

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 processes. The output variables are the output of the system for external observers. The inner variables are other internal working variables.

We adopt *locally shared memory model*[†] as a communication model: each process p can read the values of the local variables at $q \in N_p \cup \{p\}$. Each process changes the value of its local variables by executing a protocol. 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 values of p's local variables (except input variables). A process with a guard evaluated *true* is called *enabled*. In a computation step, a *distributed daemon* selects a nonempty subset of enabled processes and these processes execute the corresponding actions. The evaluation of guards and the execution of the corresponding action at a process 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 a maximal sequence of configurations $E = \sigma_0, \sigma_1, \sigma_2, \cdots$ that satisfies (i) σ_{i+1} is obtained by applying one computation step to σ_i or (ii) σ_i is the final configuration. Maximality means that the sequence is either infinite, or it is finite and no process is enabled in 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 becomes 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 c is *legitimate* iff c satisfies the legitimate predicate. A *non-reactive* problem is a problem such that no process changes the values of its output variables after the system reaches a legitimate configuration, e.g. spanning tree construction, leader election. A *reactive* problem is a problem such that processes change

[†]There exist *message passing model* and *link register model*. In message passing model, processes communicate with each other by sending and receiving messages. Link register model models each link as a register and each message is written to and read from the register. However, the idea presented in this paper does not depend on the communication model. Many researchers have tried to transform a protocol designed for the locally shared memory model into a protocol on other communication models [10], [11], [27].

the values of their output variables after the system reaches a legitimate configuration, e.g. token circulation. In this paper we consider non-reactive problems. We say a distributed protocol $P(T)$ has solved 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. The input variables to the protocol are not changed during the execution of the protocol.

For non-reactive problems, self-stabilizing protocols are defined as follows.

Definition 1: Non-reactive self-stabilization
A distributed protocol P is self-stabilizing iff it satisfies the following two properties:
Stabilization: starting from any arbitrary initial configuration, it 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 self-stabilizing protocol autonomously recovers from any initial configuration corrupted by any number of faults.

A configuration is *f-faulty*[†] iff the minimum number of processes such that we have to change their local states (except input variables) to make the configuration legitimate is f. So, an f-faulty configuration is the configuration just after a fault corrupts f processes. We say process p is *faulty* iff we have to change p's local state to make the configuration legitimate and otherwise *correct*.

An *f-fault-containing* protocol autonomously reaches a legitimate configuration from any f'-faulty configuration ($f' \leq f$) in a polynomial time in f, and the number of processes affected is bounded by a polynomial in f, e.g. f, f^2 (not $|V|$). We say a processes is *contaminated* iff the process changes its variables during the recovery from an f'-faulty configuration ($f' \leq f$).

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 contaminated processes and the number of rounds to reach a legitimate configuration bounded by some polynomial in f (not $|V|$).

We simply denote an f-fault-containing self-stabilizing protocol as f-fault containing protocol.

The performance of an f-fault-containing protocol is measured by stabilization time, recovery time, and contamination number.

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 any arbitrary f'-faulty configuration ($f' \leq f$).
Contamination number : the maximum (worst) number of contaminated processes from any arbitrary 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 $L((P_1 * P_2)) \equiv L(P_1) \wedge L(P_2)$. In a legitimate configuration of the composite protocol, each source protocol should be in a legitimate configuration.

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. For the corruption of input to P_1, we make the following assumption.

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

A fault can change the states of processes but cannot change their input variables. The input to P_1 can be seen as system parameters, e.g. topology, ID of each process.

In this paper, we put some assumptions on the source protocols of fault-containing compositions. We consider a subclass of fault-containing protocols Π such that each f-fault-containing protocol $P \in \Pi$ satisfies Assumption 2, 3, 4, and 5. Many existing fault-containing protocols satisfy Assumption 2, 3, 4, and 5 [15], [17], [18], [21].

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

Consider a fault-containing composition $(P_1 * P_2)$. Starting from an f'-faulty configuration ($f' \leq \min\{f_1, f_2\}$), if the output of P_1 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 though the original fault is small enough for P_2 to guarantee fault-containment. Assumption 2 promises the possibility of fault-containment of not only P_2 but also the entire protocol $(P_1 * P_2)$. Because the input to P_1 is not changed by any fault (Assumption 1), this assumption guarantees that P_1 recovers to the unique legitimate configuration and ensures the possibility of fault-containment of P_2 in the composite protocol.

[†]In general, the legitimate configuration obtained by changing the local states of f processes is not always unique. However, in this paper we assume the corresponding legitimate configuration is unique because we assume later that the legitimate configuration of the protocol is uniquely defined by the input and the input is not corrupted by faults.

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	t_1	c_1	k_1
P_2	f_2	t_2	c_2	k_2
$(P_1 * P_2)$	$f_{1,2} = \min\{f_1, f_2\}$	$t_{1,2}$	$c_{1,2}$	$k_{1,2}$

Assumption 3: Legitimate predicate
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, process p is enabled if the predicate $cons_p(P)$ is evaluated *false*.

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

Many fault-containing protocols satisfies Assumption 4: for a faulty process p and a neighboring correct process q, the predicate $cons_p(P)$ ($cons_q(P)$, respectively) involves the local variables at q (p, respectively). Because p is faulty, there can be some inconsistency between the local state of p and that of q.

Note that if p and all its neighbors are faulty, $cons_p(P) = true$ may hold at p. This is because $cons_p(P)$ involves the local variables at p and its neighbors and the values of these corrupted variables happen to seem consistent. In this case, p cannot determine whether it is faulty or not.

The *inconsistency range* of P is the maximum (worst) distance from any faulty process to the process q that evaluates $cons_q(P)$ *false* because of the faulty process during the recovery from an f'-faulty configuration ($f' \leq f$).

Assumption 5: Inconsistency range
Let k be the inconsistency range of P. Starting from any f'-faulty configuration ($f' \leq f$), for each faulty process p, in every configuration there exists at least one process $q \in N_p^k \cup \{p\}$ such that $cons_q(P)$ is evaluated *false* until the local variables at p takes the values that they take in the legitimate configuration.

The upper bound of the inconsistency range of a protocol is obtained by its contamination number or recovery time that are always larger than or equals to the inconsistency range. We can obtain a more accurate value of the inconsistency range by analyzing the behavior of the protocol. In many 1-fault-containing protocols [15], [17], [18], [21], inconsistency range is 1: in theses protocols, in a 1-faulty configuration the faulty process or its neighbors may suspect it is faulty and exchange the local information with neighbors. If a correct process finds the faulty process, the process waits until the faulty process changes its variables.

3. Fault-Containing Composition

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.

Fair composition of fault-containing protocols cannot preserve the fault-containment property. This is because the parallel execution of the source protocols allows the upper protocol P_2 to execute on an incorrect output of the lower protocol P_1. 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 adapt to the changes in the output variables of P_1. During the recovery of P_1, at most c_1 processes change their output of P_1 and the changes in the input to P_2 appear as corruptions by fault(s) in P_2. If c_1 is greater than f_2, P_2 cannot guarantee fault-containment. Additionally, even if c_1 is not greater than f_2, if these processes change their output in P_1 repeatedly, the repeated change of the output of P_1 is considered as multiple changes in input for P_2, hence it is considered as multiple faults for P_2 in the context of fault-containment. In this case, P_2 cannot guarantee fault-containment because it is necessary to provide fault-containment that no more fault occurs during the recovery from an f-faulty configuration (See Definition 2).

We can avoid these problems by executing P_2 after P_1 reaches the legitimate configuration. We call this approach *Recovery Waiting Fault-containing Composition* (*RWFC*). Our strategy is to stop the execution of P_2 until P_1 recovers. Thus, starting from an f-faulty configuration ($f \leq f_{1,2}$), when P_1 reaches its unique legitimate configuration[†], there are at most f faulty processes in P_2. Then P_2 can recover with its fault-containment property and the whole composite protocol succeeds in containing the effect of faults.

We can allow *faulty* processes to execute P_2 before P_1 reaches the legitimate configuration because in an f-faulty configuration, even if faulty processes executes P_2 before P_1 recovers, the number of faulty processes in P_2 is still no larger than f ($\leq f_{1,2}$). What is important is that no *correct* process executes P_2 before P_1 recovers. If some correct process executes P_2 before the recovery of P_1, the number of faulty processes in P_2 may exceed f_2.

A corruption at process p in P_1 can change the evaluation of guards of P_1 and P_2 only at p and its neighbors. This is because the guards of each process involve the local variables at the process itself and its neighbors. So, it is possible that $cons_s(P_2)$ is evaluated *false* at some process $s \in N_p$. If

[†]A fault cannot change the input variables of P_1 (Assumption 1) and P_1 reaches the unique legitimate configuration (Assumption 2).

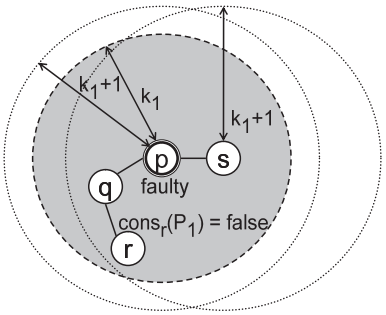

Fig. 1 Inconsistency range around a faulty process.

s executes P_2, the effect of the corruption at p spreads in P_2. To prevent this, it is necessary that each process in N_p does not execute P_2 until the variables at p takes the values that they take in the legitimate configuration. By forcing all processes in N_p to stop the execution of P_2 during the recovery of P_1, we can prevent the effect of the fault from spreading in P_2. From Assumption 5, there exists at least one process r in $N_p^{k_1}$ for p and in $N_s^{k_1+1}$ for s such that $cons_r(P_1)$ is evaluated *false* during the recovery of P_1 (See Fig. 1). We force P_2 to stop by using this property.

We force each process p to check the inconsistency of each $q \in N_p^{k_1+1}$. For simplicity, we first assume that each process can evaluate $cons_q(P_1)$ for each $q \in N_p^{k_1+1}$ with the *inconsistency detector*. The inconsistency detector guarantees that starting from any f-faulty configuration ($f \leq f_{1,2}$), it provides $\bigwedge_{q \in N_p^{k_1+1}} cons_q(P_1)$ to p in $O(|N_p^{k_1+1}|)$ rounds. We just define the specification and the interface of the inconsistency detector in Sect. 3.1, because our focus is not on the implementation of the inconsistency detector but on the fault-containing composition. We show an implementation of the inconsistency detector in Sect. 4.

3.1 Specification of the Inconsistency Detector

The inconsistency detector provides the evaluation of $\bigwedge_{q \in N_p^{k_1+1}} cons_q(P_1)$ to each process $p \in V$. Each process p has two variables, req_p and res_p: when p requests the inconsistency detector to evaluate $\bigwedge_{q \in N_p^{k_1+1}} cons_q(P_1)$, p sets $req_p = 1$, otherwise 0. The inconsistency detector stores the result in res_p that takes a value in $\{true, false, \perp\}$ and p receives the result by reading res_p. (Note that p cannot change the value of res_p.)

Assumption 6: Specification of the inconsistency detector
(i) In a legitimate configuration, $req_p = 0 \wedge res_p = \perp$ holds at each process $p \in V$.
(ii) If process $p \in V$ changes req_p from 0 to 1 when $res_p = \perp$, res_p takes *true* or *false* in α rounds with changing the state of only the processes in N_p^{β}:

- if $\bigwedge_{q \in N_p^{k_1+1}} cons_q(P_1) = false$ holds when the inconsistency detector changes res_p from $\perp$, res_p takes *false*.

Guarded actions for process p
$S1$: $G(P_1) \rightarrow A(P_1)$
$S2$: $G(P_2) \wedge req_p = 0 \wedge res_p = \perp \rightarrow req_p = 1$
$S3$: $G(P_2) \wedge req_p = 1 \wedge res_p = false \rightarrow req_p = 0$
$S4$: $G(P_2) \wedge req_p = 1 \wedge res_p = true$
$\rightarrow A(P_2); req_p = 0$

Fig. 2 *RWFC* for $(P_1 * P_2)$.

- if $\bigwedge_{q \in N_p^{k_1+1}} cons_q(P_1) = true$ holds when the inconsistency detector changes res_p from $\perp$, res_p takes *true* or *false*. Even when $res_p = false$ holds, the inconsistency detector returns $res_p = true$ in a constant number of requests.

(iii) α and β are bounded by some polynomial in k_1.
(iv) When $req_p = 0 \wedge res_p \neq \perp$ holds at process $p \in V$, the inconsistency detector sets $res_p = \perp$ in $O(1)$ rounds.

After p requests the evaluation to the inconsistency detector, if $req_p = 1 \wedge res_p = true$ holds, process p can determine that $cons_q(P_1) = true$ holds at each $q \in N_p^{k_1+1}$.

3.2 Framework for Fault-Containing Composition

RWFC framework checks the consistency of P_1 by using the inconsistency detector whenever the upper protocol needs to be executed.

Figure 2 shows *RWFC* for $(P_1 * P_2)$ at process p that provides $f_{1,2}$-fault-containing protocol. For each $i \in \{1, 2\}$, $G(P_i)$ is the disjunction of all guards of protocol P_i at p, and $A(P_i)$ indicates the corresponding action of one of the enabled guards of $G(P_i)$.

Starting from an f-faulty configuration ($f \leq f_{1,2}$), process p can execute P_1 whenever it has an enabled guard of P_1 by executing $S1$. However, when p has an enabled guard of P_2, p should check the inconsistency of P_1 among $N_p^{k_1+1}$. Process p requests the evaluation to the inconsistency detector by executing $S2$ and checks the result with $S3$ and $S4$. If there is no process $q \in N_p^{k_1+1}$ that finds inconsistency in P_1, then p executes P_2 by executing $S4$. Otherwise, p waits the recovery of P_1 by executing $S3$.

3.3 Correctness Proof

First, we show the stabilization of *RWFC*. Starting from an arbitrary initial configuration, each process can execute P_1 whenever it has an enabled guard of P_1. Thus, it is obvious that eventually P_1 reaches the legitimate configuration and the output of P_1 (the input to P_2) eventually becomes unchanged. After that, if process p requests the inconsistency detector to evaluate $\bigwedge_{q \in N_p^{k_1+1}} cons_q(P_1)$, p always receives $res_p = true$. Thus, the execution of $(P_1 * P_2)$ is that of P_2. So, it is obvious that $(P_1 * P_2)$ eventually reaches the legitimate configuration. The following lemma holds clearly.

Lemma 1: Starting from an arbitrary initial configuration,

RWFC for $(P_1 * P_2)$ eventually reaches the legitimate configuration.

Secondly, we show the fault-containment of *RWFC*. Starting from an f-faulty configuration ($f \leq f_{1,2}$), P_1 reaches the legitimate configuration in its recovery time and with its contamination number (Lemma 2). Until P_1 reaches the legitimate configuration, each correct process that is a neighbor of a faulty process cannot execute P_2 (Lemma 2). However, a faulty process may execute P_2 before P_1 reaches the legitimate configuration, e.g. if $req_p = 1 \wedge res_p = true$ holds at a faulty process p in an f-faulty configuration ($f \leq f_{1,2}$), then p can execute P_2. Though p executes P_2 before P_1 recovers, the number of faulty processes in the resulting configuration of P_2 is still no larger than f_2. Thus, after P_1 reaches the legitimate configuration, P_2 can reach the legitimate configuration with its fault-containment property.

The composite protocol $(P_1 * P_2)$ via *RWFC* preserves the fault-containment property of the source protocols (Theorem 1). The performance of the obtained protocol depends on those of P_1, P_2, and the inconsistency detector.

Starting from an f-faulty configuration ($f \leq f_{1,2}$), P_1 first reaches the legitimate configuration with its fault-containment property.

Lemma 2: Starting from any f-faulty configuration ($f \leq f_{1,2}$), P_1 reaches the legitimate configuration with its recovery time and contamination number. During the recovery of P_1, each correct process that is a neighbor of a faulty process cannot execute P_2.

Proof. Starting from an f-faulty configuration ($f \leq f_{1,2}$), P_1 reaches the legitimate configuration with its recovery time and contamination number because $S1$ is P_1 itself.

In an f-faulty configuration, $req_q = 0 \wedge res_q = \perp$ holds at each correct process q. If a correct process q is a neighbor of a faulty process and q has an enabled guard in P_2, q changes req_q from 0 to 1 by executing $S2$ and the inconsistency detector returns the evaluation of $\bigwedge_{r \in N_q^{k_1+1}} cons_r(P_1)$. From Assumption 5, if P_1 is not in the legitimate configuration, q receives $res_q = false$. So, correct processes neighboring some faulty process(es) do not execute P_2 with incorrect output from P_1. ■

Lemma 3: After P_1 reaches the legitimate configuration, P_2 reaches the legitimate configuration with the recovery time of $t_2\alpha$ and the contamination number of $c_2\Delta^\beta$, where Δ is the maximum degree in G.

Proof. From Lemma 2, there exist at most f ($\leq f_{1,2}$) faulty processes in P_2 when P_1 reaches the legitimate configuration. Thus, P_2 reaches the legitimate configuration with its fault-containment property: for the variables of P_2, the recovery time and the contamination number is still t_2 and c_2.

However, each process p should check the consistency of P_1 with the inconsistency detector whenever p has an enabled guard of P_2. From Assumption 6, this forces each $q \in N_p^\beta$ to change their states and imposes α rounds for p to obtain the result. Thus, in *RWFC*, it takes $t_2\alpha$ rounds for P_2 to reach the legitimate configuration with the number of $c_2\Delta^\beta$ processes changing their local states. ■

From Assumption 6, α and β are bounded by some polynomial in k_1.

Theorem 1: *RWFC* provides an $f_{1,2}$-fault-containing protocol $(P_1 * P_2)$ for $f_{1,2} = \min\{f_1, f_2\}$. The recovery time is $(t_1 + t_2\alpha)$ and the contamination number is $\max\{c_1, c_2\Delta^\beta\}$.

Proof. From Lemma 2 and 3, *RWFC* executes P_1 and P_2 in the coordinated order and each protocol executes its own recovery actions. So the maximum number of faults that the obtained protocol guarantees fault-containment is $f_{1,2} = \min\{f_1, f_2\}$. From Lemma 3, the recovery time is $(t_1 + t_2\alpha)$ and the contamination number is $\max\{c_1, c_2\Delta^\beta\}$. ■

4. The Inconsistency Detector

In this section, we show an implementation of the inconsistency detector.

The inconsistency detector should provide the communication between process p and each $q \in N_p^{k_1+1}$ to evaluate $\bigwedge_{q \in N_p^{k_1+1}} cons_q(P_1)$ whenever p changes req_p from 0 to 1. In the locally shared memory model, process p can read only the local variables at its direct neighbors. Thus, it is necessary to broadcast the request to each process $q \in N_p^{k_1+1}$ and each $q \in N_p^{k_1+1}$ should return the evaluation of $cons_q(P_1)$ to p.

Recall that the legitimate predicate $L(P_1) \equiv \forall p \in V : cons_p(P_1)$ is a stable predicate on configurations in P_1. Thus, starting from a target faulty configuration, once $L(P_1) = true$ holds, $L(P_1)$ remains *true* thereafter. However, the fault-containment property guarantees that only the processes in the inconsistency range of each faulty process p change their states during the recovery. So, starting from a target faulty configuration, once $cons_q(P_1)$ holds for each $q \in N_p^{k_1+1}$ for a faulty process p, $cons_q(P_1)$ remains *true* at all $q \in N_p^{k_1+1}$ thereafter. Consequently, the inconsistency detector should answer whether there is a configuration where $\bigwedge_{q \in N_p^{k_1+1}} cons_q(P_1)$ holds between the time when p requests by changing req_p from 0 to 1 and the time the inconsistency detector answers to p by changing res_p from $\perp$ to a value in $\{true, false\}$.

One simple solution for evaluating a stable predicate is to use *PIF (Propagation of Information with Feedback)* protocols that take a snapshot of global configurations by broadcasting a request to all processes and gathering feedbacks from all processes.

However, we do not need any global detection but the local detection among $N_p^{k_1+1}$ for process p. One way to involve $N_p^{k_1+1}$ into some task is to use the breadth first tree of height $(k_1 + 1)$ rooted at p. Whenever process p changes req_p to 1, p constructs the breadth first tree and by using a PIF protocol on the breadth first tree, p broadcasts the request to each $q \in N_p^{k_1+1}$ and q feedbacks the evaluation of $cons_q(P_1)$ to p. We can use the breadth first tree construc-

tion protocol in [19] by setting the height of the tree $k_1 + 1$.

However, this simple implementation cannot provide the correct evaluation of $\bigwedge_{q \in N_p^{k_1+1}} cons_q(P_1)$ to p. Because each process executes P_1 during the request and feedback of a PIF protocol, the evaluation of $cons_q(P_1)$ at $q \in N_p^{k_1+1}$ may change during the feedback: e.g. after q sends $cons_q(P_1) = true$ as a feedback, if the evaluation of $cons_q(P_1)$ changes from *true* to *false* (it may be caused by some state change of the processes in N_q), p cannot obtain the correct evaluation of $\bigwedge_{q \in N_p^{k_1+1}} cons_q(P_1)$.

Generally, to evaluate a stable predicate among processes, PIF is used twice. The first PIF propagates the request to each process and each process starts to observe the stable predicate. The second PIF gathers the result of observation at each process via the feedback of PIF. In this way, one can evaluate a stable predicate on configurations.

Cournier et al. proposed a snap-stabilizing PIF protocol for arbitrary networks [7]. Their protocol *PIF* guarantees that each process returns the feedback after all processes in V received the request. Thus, by using *PIF*, we can collect the observation of the stable predicate with a single PIF execution.

We allow each process p to execute *PIF* independently in parallel so that each process $q \in N_p^{k_1+1}$ can evaluate $\bigwedge_{r \in N_q^{k_1+1}} cons_r(P_1)$ when q changes req_q from 0 to 1. This is done, for example, by attaching the ID of q to the broadcast and feedback. This imposes additional memory of size of $O(|N_p^{k_1+1}| \log n)$ at p to manage different trees while this does not impose additional time complexity.

We modify *PIF* as follows:
(i) process p constructs the breadth first tree of height (k_1+1) rooted at p when it changes req_p from 0 to 1.
(ii) process q starts to observe $cons_q(P_1)$ when it receives the request of the *PIF* protocol. If $cons_q(P_1) = true$ holds during the observation, q records it.
(iii) q returns the result of the observation to p with the feedback of *PIF*.

The snap-stabilization property of *PIF* guarantees that starting from an arbitrary initial configuration, whenever the root process begins the broadcast, every process receives the broadcast and the root process receives feedback from every process in $O(N)$ rounds. Thus, in our implementation, the broadcast and feedback take $O(N_p^{k_1+1})$ rounds. The breadth-first tree is constructed in $O(k_1 + 1)$ rounds. Thus, p receives the feedback from all processes in $N_p^{k_1+1}$ in $O(|N_p^{k_1+1}|)$ rounds. Consequently, the value of α in Assumption 6 is a polynomial in k_1. Because only the processes in $N_p^{k_1+1}$ change their states, the value of β in Assumption 6 is $k_1 + 1$. So, the condition (iii) of Assumption 6 is satisfied.

To satisfy the condition (iv) of Assumption 6, the inconsistency detector should check req_p and res_p, and whenever $req_p = 0 \wedge res_p \neq \perp$ holds at p, it should change res_p to $\perp$.

5. Conclusion

We present *RWFC* framework that provides hierarchical composition of two fault-containing protocols with preserving the fault-containment property of source protocols. Our strategy is to stop the execution of the upper protocol until the lower protocol recovers. We can compose more than two fault-containing protocols with *RWFC* by applying *RWFC* repeatedly to the source protocols. Though the strategy is very simple, it provides significant improvement on composing fault-containing protocols. Furthermore, this framework helps designing new fault-containing protocols: we can easily built new fault-containing protocols on top of existing fault-containing protocols.

We defined and implemented the inconsistency detector that enables each process to communicate with the processes in its inconsistency range of the lower protocol. Our implementation is based on an existing snap-stabilizing PIF protocol that is executed among the processes in the inconsistency range of the lower protocol. The performance of the obtained protocol depends on the inconsistency detector. Though the PIF protocol imposes additional communication overhead and execution time, the effect is contained around faulty processes in the inconsistency range of the lower protocol. The inconsistency range of the lower protocol is small because the lower protocol is fault-containing. Thus, the overhead imposed by the PIF protocol is small and do not spread over the entire network.

To accelerate the composite protocol by *RWFC*, we can use the legitimacy of output variables of the lower protocol (called *output legitimacy*) instead of overall legitimacy. This is because the upper protocol just uses the output variables of the lower protocol as its input. However, to adopt output legitimacy, it is necessary that when the system starts from a target faulty configuration, once the lower protocol reaches the output legitimate configuration, it remains in the output legitimate configuration(s) thereafter. Note that not all the fault-containing protocols provide this property for output legitimacy.

Future work. *RWFC* puts several assumptions on the source protocols, e.g. the unique legitimate configuration, inconsistency detection. It is necessary to relax these assumptions to extend the application of fault-containing composition. So it is necessary to consider a framework for fault-containing protocols such that the recovery scenario is more complicated. There may be other keys to check the configuration of the lower protocol to control the execution of the upper protocol.

Acknowledgment

This work is supported in part by 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, Grant-in-Aid

for JSPS Fellows (20-1621), and Kayamori Foundation of Informational Science Advancement.

References

[1] Y. Afek and S. Dolev, "Local stabilizer," Proc. 5th Israeli Symposium on Theory of Computing and Systems, pp.74–84, Ramat-Gan, Israel, June 1997.
[2] A. Arora and H. Zhang, "LSRP: Local stabilization in shortest path routing," Proc. International Conference of Dependable Systems and Networks, pp.139–148, San Francisco, USA, June 2003.
[3] Y. Azar, S. Kutten, and B. Patt-Shamir, "Distributed error confinement," Proc. 22nd Annual ACM Symposium on Principles of Distributed Computing, pp.33–42, Boston, USA, July 2003.
[4] J. Beauquier, M. Gradinariu, and C. Johnen, "Cross-over composition - Enforcement of fairness under unfair adversary," Proc. 5th Workshop on Self-Stabilizing Systems, pp.19–34, Lisbon, Portugal, Oct. 2001.
[5] S.C. Bruell, S. Ghosh, M.H. Karaata, and S.V. Pemmaraju, "Self-stabilizing algorithms for finding centers and medians of trees," SIAM J. Comput., vol.29, pp.600–614, 1999.
[6] N.S. Chen, H.P. Yu, and S.T. Huang, "A self-stabilizing algorithm for constructing a spanning tree," Inf. Process. Lett., vol.39, pp.147–151, 1991.
[7] A. Cournier, A.K. Datta, F. Petit, and V. Villain, "Snap-stabilizing PIF algorithm in arbitrary networks," Proc. 22nd International Conference on Distributed Computing Systems, p.199, Vienna, Austria, July 2002.
[8] E.W. Dijkstra, "Self-stabilizing system in spite of distributed control," Commun. ACM, vol.17, no.11, pp.643–644, 1974.
[9] S. Dolev, A. Israeli, and S. Moran, "Self-stabilization of dynamic systems," Proc. MCC Workshop on Self-Stabilizing Systems, MCC Technical Report no.STP-379-89, 1989.
[10] S. Dolev, A. Israeli, and S. Moran, "Self-stabilization of dynamic systems assuming only read/write atomicity," Distributed Computing, vol.7, pp.3–16, 1993.
[11] S. Dolev, A. Israeli, and S. Moran, "Resource bounds for self-stabilizing message driven protocols," Proc. 10th Annual ACM Symposium on Principles of Distributed Computing, pp.281–293, Montreal, Canada, Aug. 1991.
[12] S. Dolev and T. Herman, "Superstabilizing protocols for dynamic distributed systems," Proc. 2nd Workshop on Self-Stabilizing Systems, pp.3.1–3.15, Las Vegas, USA, 1995.
[13] S. Dolev and T. Herman, "Parallel composition of stabilizing algorithms," Proc. 4th Workshop on Self-Stabilizing Systems, pp.25–32, Austin, USA, June 1999.
[14] S. Dolev and T. Herman, "Parallel composition for time-to-fault adaptive stabilization," Distributed Computing, vol.20, pp.29–38, 2007.
[15] S. Ghosh and A. Gupta, "An exercise in fault-containment: Self-stabilizing leader election," Inf. Process. Lett., vol.59, pp.281–288, 1996.
[16] S. Ghosh, A. Gupta, T. Herman, and S.V. Pemmaraju, "Fault-containing self-stabilizing algorithms," Proc. 15th Annual ACM Symposium on Principles of Distributed Computing, pp.45–54, Philadelphia, USA, May 1996.
[17] S. Ghosh, A. Gupta, and S.V. Pemmaraju, "Fault-containing network protocols," Proc. 12th ACM Symposium on Applied Computing, pp.431–437, San Jose, USA, Feb. 1997.
[18] S. Ghosh and X. He, "Fault-containing self-stabilization using priority scheduling," Inf. Process. Lett., vol.73, pp.145–151, 2000.
[19] S.T. Huang and N.S. Chen, "A self-stabilizing algorithm for constructing breadth-first trees," Inf. Process. Lett., vol.41, pp.109–117, 1992.
[20] S.T. Huang and N.S. Chen, "Self-stabilizing depth-first token circulation on networks," Distributed Computing, vol.7, no.1, pp.61–66, 1993.
[21] Y. Katayama and T. Masuzawa, "A fault-containing self-stabilizing protocol for constructing a minimum spanning tree," IEICE Trans. Inf. & Syst. (Japanese Edition), vol.J84-D-I, no.9, pp.1307–1317, Sept. 2001.
[22] K. Kotani, Y. Katayama, T. Masuzawa, and N. Tokura, "A self-stabilizing algorithm for constructing a minimum weight spanning tree," IEICE Technical Report, COMP 92-5, 1992.
[23] S. Kutten and B. Patt-Shamir, "Time-adaptive self stabilization," Proc. 16th Annual ACM Symposium on Principles of Distributed Computing, pp.149–158, Santa Barbara, USA, Aug. 1997.
[24] X. Lin and S. Ghosh, "Maxima finding in a ring," Proc. 28th Annual Allerton Conference on Computers, Communication and Control, pp.662–671, 1991.
[25] G. Tel, Introduction to dustributed algorithms, 2nd ed., Cambridge Univ. Press, Cambridge, U.K., 2000.
[26] 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," Proc. 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems, pp.516–532, Dallas, USA, Nov. 2006.
[27] K. Yoshida, H. Kakugawa, and T. Masuzawa, "Observation on light weight implementation of self-stabilizing node clustering algorithms in sensor networks," Proc. International Association of Science and Technology for Development International Conference on Sensor Networks, pp.1–8, Creta, Greece, Sept. 2008.

Yukiko Yamauchi received the M.E. degree in computer science from Osaka University in 2006. She is now a student of Graduate School of Information Science and Technology, Osaka University and a research fellow of JSPS. Her research interests include distributed algorithms. She is a student member of IPSJ.

Sayaka Kamei received the B.E., M.E. and D.E. degrees in electronics engineering in 2001, 2003 and 2006 respectively from Hiroshima University. She is currently an assistant professor of Dept. of Information Engineering, Graduate School of Engineering, Hiroshima University. Her research interests include distributed algorithms. She is a member of IEEE Computer Society.

Fukuhito Ooshita received the M.E. and D.I. degrees in computer science from Osaka University in 2002 and 2006. Since 2003, he has been an Assistant Professor in the Graduate School of Information Science and Technology at Osaka University. His research interests include distributed algorithms and parallel algorithms. He is a member of ACM, IEEE, and the Information Processing Society of Japan.

Yoshiaki Katayama received his B.E., M.E., and D.E. in computer science from Osaka University. He has been working at Information Technology Center, Nara Institute of Science and Technology (NAIST) from 1994 to 2003. He is now an associate professor of Graduate School of Engineering, Nagoya Institute of Technology. His research interests include distributed algorithms, network applications and ubiquitous computing. He is a member of IPSJ, ACM and IEEE Computer Society.

Hirotsugu Kakugawa received the B.E. degree in engineering in 1990 from Yamaguchi University, and the M.E. and D.E. degrees in information engineering in 1992, 1995 respectively from Hiroshima University. He is currently an associate professor of Osaka University. He is a member of the IEEE Computer Society and the Information Processing Society of Japan.

Toshimitsu Masuzawa received the B.E., M.E. and D.E. degrees in computer science from Osaka University in 1982, 1984 and 1987. He had worked at Osaka University during 1987–1994, and was an associate professor of Graduate School of Information Science, Nara Institute of Science and Technology (NAIST) during 1994–2000. He is now a professor of Graduate School of Information Science and Technology, Osaka University. He was also a visiting associate professor of Department of Computer Science, Cornell University between 1993–1994. His research interests include distributed algorithms, parallel algorithms and graph theory. He is a member of ACM, IEEE, and IPSJ.