An efficient fault-containing self-stabilizing algorithm for finding a maximal independent set

Ji-Cherng Lin and Tetz C. Huang

Department of Computer Engineering and Science, Yuan-Ze University,

135 Yuan-Tung Road, Chung-Li 320, Taiwan, R.O.C.

Abstract

In this paper, we design a fault-containing self-stabilizing algorithm that finds a maximal independent set for an asynchronuous distributed system. Our algorithm is an improvement on the self-stabilizing algorithm in Shukla, Rosenkrantz and Ravi [9]. In the single-fault situation, the worst-case stabilization time of Shukla's algorithm is $\Omega(n)$, where n is the number of nodes in the system, whereas the worst-case stabilization time of our algorithm is $O(\Delta)$, where Δ is the maximum node degree in the system. Compared also with the fault-containing algorithm that is induced from applying the general transformer in Ghosh et al. [7] to Shukla's algorithm, our algorithm is also seen to be faster in stabilization time, in the single-fault situation. Therefore, our algorithm can be considered to be the most efficient fault-containing self-stabilizing algorithm for the maximal-independent-set finding up to this point.

Index Terms: Central demon, maximal independent set, single transient fault, fault-containment, restrictions on guard conditions, primary variables, auxiliary secondary variables, stabilization time, contamination number.

1 Introduction

E.W. Dijkstra first introduced the notion of self-stabilization in a distributed system in his classic paper [1] in 1974. According to him, a distributed system is self-stabilizing if starting with any arbitrary initial global state, the system can automatically adjust itself to eventually converge to a legitimate state and then stay in the legitimate states thereafter unless it incurs a subsequent transient fault (cf. also [2-3]).

Although a self-stabilizing algorithm can function automatically to restore the legitimate states of the system whenever the system incurs a transient fault and slips out of the legitimate states, the time required for the system to return back to legitimate states, i.e., the stabilization time, may not be short. Even when only one processor in the system is faulty, i.e., in the single-fault situation, it may take a long time for the system to self-stabilize, due to the spread of the fault which causes the contamination of other processors in the system (cf. Figure 1 in Section 2.1.) Ghosh, Gupta, Herman and Pemmaraju introduced the idea of the *fault containment* of a self-stabilizing algorithm in 1996 and 1997 [4-8]. Their point of view is that since more often than not, the transient fault which a self-stabilizing system encounters is a single fault, modifying the algorithm in the system to improve the stabilization time in the single-fault situation is most desirable. The local variable in the to-be-modified algorithm which is directly related to solving the problem of the system (e.g., problems of center-finding, shortest path-finding, mutual exclusion, ..., etc.) is called the *primary variable*. In order to keep the fault from spreading in the system in the single-fault situation, one may add restrictions on the guard conditions in the to-be-modified algorithm to restrain "wrong" nodes from adjusting their primary variables and thus causing the spread of the fault. In order to implement these restrictions, new variables may be involved, which are called *auxiliary variables*. In this way, the central demon is helped (or compelled) by the added device to select the "right" node (which is the faulty node for most of the time) to adjust its primary variable. Consequently, the primary variables of the whole system immediately

get back to normal right after only one processor in the system adjusts its primary variable. The only one move of the primary variable, together with not too many moves of auxiliary variables, constitutes the whole process of self-stabilization of the modified fault-containing algorithm in the single-fault situation. Therefore, the stabilization time is significantly reduced in the single-fault situation. In other words, after the original algorithm of the system is added the fault-containment device, in order for a node i in the system to adjust its primary variable, i needs to satisfy not only a certain guard condition of the original algorithm but also some added restrictions. Some of these added restrictions may involve states of nodes that are not i's direct neighbors. Since the main spirit of the distributed system is that each node in the system can only access the data of its direct neighbors, in order to implement these restrictions in the modified algorithm, auxiliary variables are introduced to help a node to collect data of those non-direct neighbors.

The main work of this paper is to design a fault-containing self-stabilizing algorithm based on the algorithm in [9]. As mentioned above and we should reiterate here, the key spirit of the fault-containment is to reduce the stabilization time in the single-fault situation, and the main contribution of this paper is on this respect. In the single-fault situation, the worst-case stabilization time of the algorithm in [9] is $\Omega(n)$ and the contamination number of it is at least $n-1$, where n is the number of nodes in the system, whereas the worst-case stabilization time of our algorithm is $O(\Delta)$, where Δ is the maximum node degree in the system, and the contamination number of our algorithm is 1.

The rest of this paper is organized as follows. In Subsection 2.1, relevant information about the algorithm in [9] is presented. In Subsections 2.2, we exhibit clearly the process of designing our fault-containing algorithm. In Section 3, the correctness proofs of our algorithm are given, which include (1) no deadlock, (2) convergence and closure, and (3) fault containment. In Section 4, the stabilization time is computed. Finally in Concluding Remarks in Section 5, the general approach for the fault-containment in Ghosh et al. [5, 7] is discussed.

2 The algorithm

2.1 The algorithm in [9]

Let $G = (V, E)$ be a connected undirected graph. Let each node i maintain a variable $Ind.i$ whose value is either 0 or 1. The algorithm in [9] is as follows. Note that in the algorithm, and in the rest of the paper as well, notation $N(i)$ stands for the set of all i's neighbors.

{For any node i}

$R1$ $\quad Ind.i = 0 \wedge \forall j \in N(i), Ind.j = 0 \longrightarrow Ind.i := 1$

$R2$ $\quad Ind.i = 1 \wedge \exists j \in N(i)$ s.t. $Ind.j = 1 \longrightarrow Ind.i := 0$

Legitimate states are defined to be all those global states in which the following condition holds :

$\forall i \in V, [(Ind.i = 0 \wedge \exists j \in N(i)$ s.t. $Ind.j = 1) \vee (Ind.i = 1 \wedge \forall j \in N(i), Ind.j = 0)]$

For the sake of convenience, we shall use $L.i$ to represent the condition $(Ind.i = 0 \wedge \exists j \in N(i)$ s.t. $Ind.j = 1) \vee (Ind.i = 1 \wedge \forall j \in N(i), Ind.j = 0)$. Whenever the system reaches a legitimate state, the set $\{x \in V | Ind.x = 1\}$ can be easily seen to be a maximal independent set.

Figure 1 illustrates the execution of the above algorithm which starts with a single-fault state and ends with a legitimate state. Note that in each state, the nodes with a darkened circle are the privileged nodes. If we generalize the execution in Figure 1 in the system whose topology is a complete bipartite graph with n nodes, then it is not difficult to compute the stabilization time to be $n-1$, and the number of processors that change their Ind-values is also easily seen to be $n-1$. Therefore, the worst-case stabilization time of the above algorithm in the single-fault situation is $\Omega(n)$ and the contamination number is at least $n-1$.

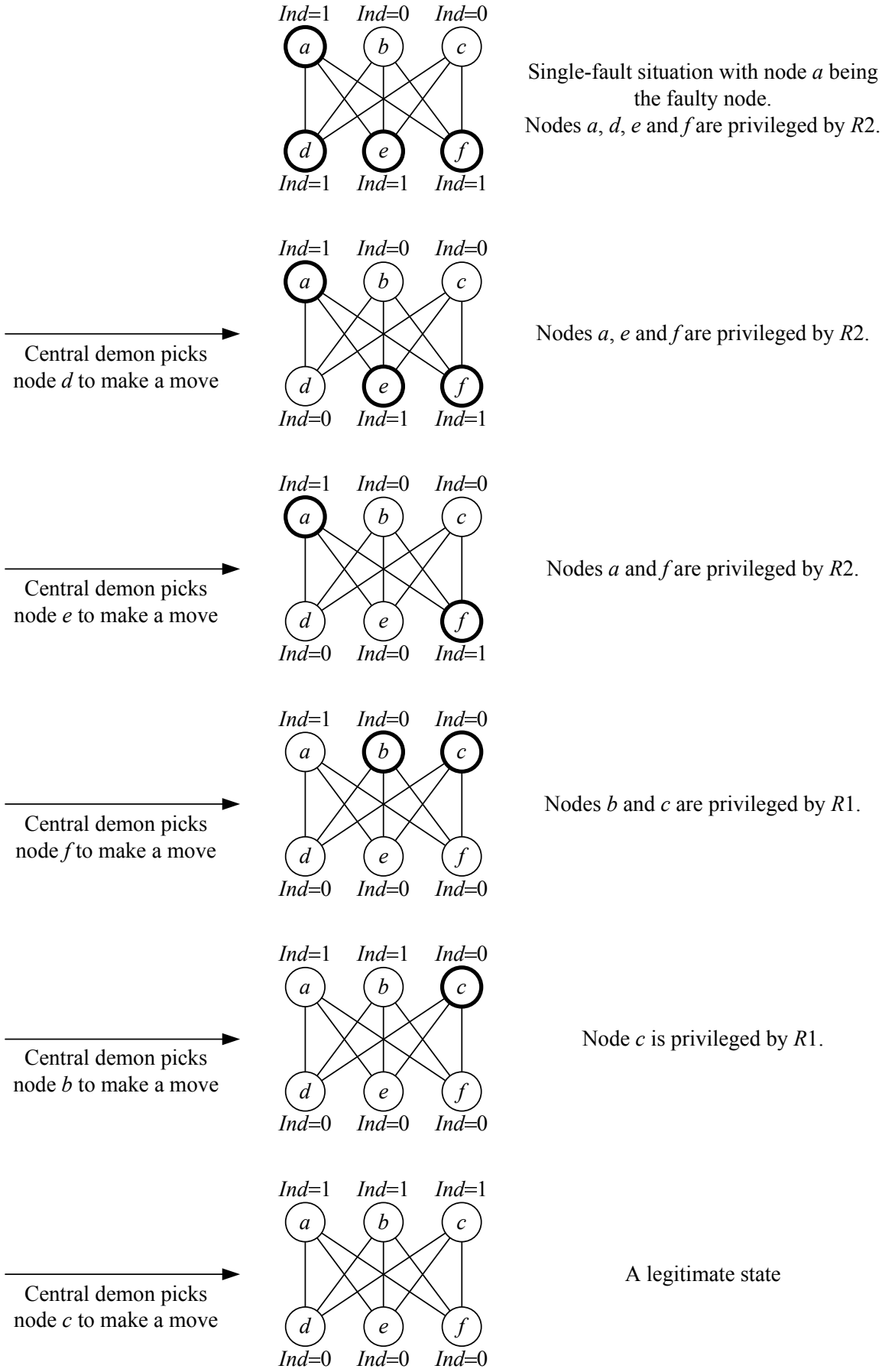

Figure 1: An execution of the algorithm in [9] which starts with a single-fault state and ends with a legitimate state.

2.2 Our fault-containing algorithm

Before we proceed to design our fault-containing self-stabilizing algorithm for finding a maximal independent set, we first introduce an additional variable p into each node of the above system in [9]. The purpose of introducing p is to help reveal information about Ind-values of each node's neighbors. To be more explicit, we introduce first a modified version for the algorithm in [9].

Algorithm 1

$R1$ $Ind.i = 0 \wedge \forall j \in N(i), Ind.j = 0 \longrightarrow Ind.i := 1$

$R2$ $Ind.i = 1 \wedge \exists j \in N(i)$ s.t. $Ind.j = 1 \longrightarrow Ind.i := 0$

$R3$ $Ind.i = 0 \wedge (\exists! j \in N(i)$ s.t. $Ind.j = 1) \wedge p.i \neq j \longrightarrow p.i := j$

$R4$ $Ind.i = 0 \wedge \exists j, k \in N(i)$ s.t. $(j \neq k \wedge Ind.j = Ind.k = 1) \wedge p.i \neq \perp \longrightarrow p.i := \perp$

$R5$ $Ind.i = 1 \wedge \forall j \in N(i), Ind.j = 0 \wedge p.i \neq \perp \longrightarrow p.i := \perp$

Note that in the above algorithm, $p.i$ takes values in $N(i) \cup \{\perp\}$, and the notation "$\exists!$" stands for "there exists a unique." We will show in a moment that the system equipped with the above algorithm is self-stabilizing (although it is still not fault-containing) with the legitimate states being all those states in which the following condition holds:

$$\begin{aligned}\forall i \in V, [&(Ind.i = 0 \wedge (\exists! j \in N(i) \text{ s.t. } Ind.j = 1) \wedge p.i = j) \vee \\ &(Ind.i = 0 \wedge \exists j, k \in N(i) \text{ s.t. } (j \neq k \wedge Ind.j = Ind.k = 1) \wedge p.i = \perp) \vee \\ &(Ind.i = 1 \wedge \forall j \in N(i), Ind.j = 0 \wedge p.i = \perp)]\end{aligned}$$

One can see that the system reaches a legitimate state if and only if the algorithm in the system stops. One can also see that in a legitimate state, not only all those nodes with Ind-value equal to 1 constitute a maximal independent set of the system but also the p-value of each node i

of the system reveals informations about Ind-values of node i's neighbors. More explicitly, when in a legitimate state, $p.i = j$ if and only if $Ind.i = 0$ and j is the only neighbor of i with the Ind-value being 1.

Theorem 1 (Self-stabilization) *Starting with any initial state, the system will eventually stop in a legitimate state.*

Proof. According to the above remarks, it suffices to show that the system will stop. Suppose not. Then there exists an infinite computation $C = (S_1, S_2, ...)$ in which for any j, the global state S_{j+1} is obtained from global state S_j after exactly one node in the system makes a move. If there are infinitely many moves in C which execute $R1$ or $R2$, let S_{i_1}, S_{i_2}, ..., where $1 < i_1 < i_2 < \cdots$, be all the states which result from these moves. Then, if we ignore all the p-values in the states, then $(S_1,\ S_{i_1},\ S_{i_2},\ ...)$ becomes an infinite computation with respect to the algorithm in [10], which is impossible. Hence, there are only finitely many moves in C which execute $R1$ or $R2$. Therefore, there exists a positive integer m such that in the suffix $C' = (S_m, S_{m+1}, ...)$ of C there is no execution of $R1$ or $R2$. So, in C', Ind-values of all nodes never change. So, in C', any node of the system can execute $R3$, $R4$ or $R5$ at most once and thus, there are only finitely many moves in C', which causes a contradiction (for C' is an infinite computation). Hence, we have shown that the system will stop and therefore, the theorem is proved. ■

Thus, we have shown that Algorithm 1 is a self-stabilizing algorithm that can find a maximal independent set for any graph. Of course, Algorithm 1 is still not fault-containing and we will modify it to get our fault-containing algorithm soon later. To enable us to do so, we should, at this stage, first investigate the single-fault situation of the system equipped with Algorithm 1. The non-trivial single-fault situation is when the faulty node has the value of its primary variable, i.e., its Ind-value, corrupted. In such a situation, there may be more than one node in the system that satisfy the negative of the condition $L.i$, or, equivalently, the condition $(Ind.i = 0 \wedge \forall j \in N(i), Ind.j = 0) \vee (Ind.i = 1 \wedge \exists j \in N(i)$ s.t. $Ind.j = 1)$. All these nodes

are privileged to make moves to change their Ind-values, according to Algorithm 1. However, if a "wrong" node is selected by the central demon first to make a move to change its Ind-value, the fault will be spread in the system. The undesirable situation like in Figure 1 may thus happen and cause the system to take a long time to self-stabilize. Therefore, more restrictions should somehow be imposed on Algorithm 1 to help (or force) the central demon to select the "right" node first to change its Ind-value. After the investigation of the non-trivial single-fault situation, we decide to classify the negative of the condition $L.i$ into following cases. Note that in the following classification, $P(i)$ denote the set $\{j \in N(i) | p.j = i\}$.

1. $Ind.i = 0 \wedge \forall x \in N(i), Ind.x = 0$

    1.1. $|P(i)| \geq 2$ (Case 1)

    1.2. $p.i = \perp$ (Case 2)

    1.3. $|P(i)| \leq 1 \wedge p.i \neq \perp$ (let $j = p.i$)

        1.3.1. $\exists u \in N(j) - \{i\}$ s.t. $Ind.u = 1$ (Case 3)

        1.3.2. $\forall u \in N(j) - \{i\}$, $Ind.u = 0$

            1.3.2.1. $\exists v \in N(j) - \{i\}$ s.t. $p.v = j$ (Case 4)

            1.3.2.2. $\forall v \in N(j) - \{i\}$, $p.v \neq j$ (Case 5)

2. $Ind.i = 1 \wedge \exists x \in N(i)$ s.t. $Ind.x = 1$

    2.1. $p.i \neq \perp$ (Case 6)

    2.2. $\exists j, k \in N(i)$ s.t. $(j \neq k \wedge Ind.j = Ind.k = 1)$ (Case 7)

    2.3. $p.i = \perp \wedge \exists! j \in N(i)$ s.t. $Ind.j = 1$

        2.3.1. $\exists u \in N(j) - \{i\}$ s.t. $Ind.u = 1$ (Case 8)

        2.3.2. $\forall u \in N(j) - \{i\}$, $Ind.u = 0$

            2.3.2.1. $\exists v \in N(j) - \{i\}$ s.t. $p.v = j$ (Case 9)

            2.3.2.2. $\forall v \in N(j) - \{i\}$, $p.v \neq j$

                2.3.2.2.1. $\exists k \in N(i) - \{j\}$ s.t. $p.k = i$ (Case 10)

2.3.2.2.2. $\forall k \in N(i) - \{j\}$, $p.k \neq i$ (Case 11)

Corresponding to above 11 cases, we also obtain the following 11 lemmas. In all these lemmas, we assume that the system starts in a legitimate state and incurs a single fault at a certain time instant t_0 and the single fault causes the change of Ind-value of the faulty node and possibly its p-value also. For the sake of presentation in the following lemmas, for any time instant t, we say that a predicate is true at t^+ if it is true right after t and that a predicate is true at t^- if it is true right before t. Also by definition, a predicate is true at t if it is true at both t^+ and t^-; and a predicate is true in a time interval I if it is true at any instant t in I. To illustrate how to use these expressions, for instance, if a processor i makes a move to change its Ind-value from 1 to 0 at a time instant t, then $Ind.i = 1$ at t^-, $Ind.i = 0$ at t^+ and $Ind.i$ at t is not well-defined. If processor i does not change its Ind-value at t, then $Ind.i$ at t^-, $Ind.i$ at t^+ and $Ind.i$ at t are all the same.

Lemma 2 *(Corresponding to Case 1) If at t_0^+, $Ind.i = 0$ and $|P(i)| \geq 2$, then node i is the faulty node.*

Lemma 3 *(Corresponding to Case 2) If at t_0^+, $Ind.i = 0$, for every $x \in N(i)$, $Ind.x = 0$, and $p.i = \perp$, then node i is the faulty node.*

Lemma 4 *(Corresponding to Case 3) If at t_0^+, $Ind.i = 0$, for every $x \in N(i)$, $Ind.x = 0$, $p.i = j$ and there exists a $u \in N(j) - \{i\}$ such that $Ind.u = 1$, then node i is the faulty node.*

Lemma 5 *(Corresponding to Case 4) If at t_0^+, $Ind.i = 0$, for every $x \in N(i)$, $Ind.x = 0$, $p.i = j$, for every $u \in N(j) - \{i\}$, $Ind.u = 0$ and there exists a $v \in N(j) - \{i\}$ such that $p.v = j$, then i is not the faulty node.*

Lemma 6 *(Corresponding to Case 5) Suppose at t_0^+, $Ind.i = 0$, for every $x \in N(i)$, $Ind.x = 0$, $p.i = j$, for every $u \in N(j) - \{i\}$, $Ind.u = 0$ and $p.u \neq j$. If i is the first node in the system*

to change the Ind-value after t_0, then, right after i changes its Ind-value, the Ind-values of the whole system get back to normal, i.e., for every node $x \in V$, $L.x$.

Proof. Let t_1 be the first instant after t_0 at which the system changes the *Ind*-value. Then node i must change its *Ind*-value at t_1. If i is the faulty node, then the lemma is obviously true. If i is not the faulty node, then $Ind.i = 0$ and $p.i = j$ at t_0^-. Since the system is in legitimate state at t_0^- and $p.i = j$ at t_0^-, we have that $Ind.j = 1$ at t_0^-. Since $Ind.j = 0$ at t_0^+, we see that j is the faulty node. Hence, for every $u \in N(j) - \{i\}$, u is not the faulty node and $Ind.u = 0$ and $p.u \neq j$ at t_0^-. Since $Ind.j = 1$ at t_0^-, we have that for every $u \in N(j) - \{i\}$, $p.u = \perp$ at t_0^- and there exists a node $y \in N(u) - \{j\}$ such that $Ind.y = 1$ at t_0. Hence, for every $u \in N(j) - \{i\}$, $L.u$ is true at t_0^+ and at t_1^+. For every $u \in N(i)$, since $Ind.u = 0$ at t_0^+, $Ind.u = 0$ at t_1^+. Since $Ind.i = 1$ at t_1^+, we have that $L.i$ is true at t_1^+ and for every $u \in N(i)$, $L.u$ is true at t_1^+. From all above, we have that for every $x \in N(i) \cup N(j)$, $L.x$ is true at t_1^+. It is obvious that for every $x \in V - (N(i) \cup N(j))$, $L.x$ is true at t_1^+. Therefore, for every $x \in V$, $L.x$ is true at t_1^+ and hence the lemma is proved. ■

Lemma 7 *(Corresponding to Case 6) If $Ind.i = 1$ and $p.i \neq \perp$ at t_0^+, then node i is the faulty node.*

Lemma 8 *(Corresponding to Case 7) If at t_0^+, $Ind.i = 1$ and there exist two distinct nodes $j, k \in N(i)$ such that $Ind.j = Ind.k = 1$, then node i is the faulty node.*

Lemma 9 *(Corresponding to Case 8) Suppose at t_0^+, $Ind.i = 1$ and there exists an unique node $j \in N(i)$ such that $Ind.j = 1$ and there exists a node $u \in N(j) - \{i\}$ such that $Ind.u = 1$. If i is the first node in the system to change the Ind-value after t_0, then, right after i changes its Ind-value, the Ind-values of the whole system cannot get back to normal, in fact, $L.j$ is false.*

Lemma 10 *(Corresponding to Case 9) If at t_0^+, $Ind.i = 1$ and there exists an unique node $j \in N(i)$ such that $Ind.j = 1$ and there exists a node $v \in N(j) - \{i\}$ such that $p.v = j$, then node i is the faulty node.*

Lemma 11 *(Corresponding to Case 10) Suppose at t_0^+, $Ind.i = 1$, there exists an unique node $j \in N(i)$ such that $Ind.j = 1$ and there exists a $k \in N(i) - \{j\}$ such that $p.k = i$. If i is the first node in the system to change the Ind-value after t_0, then, right after i changes its Ind-value, the Ind-values of the whole system cannot get back to normal, in fact, $L.k$ is false.*

Lemma 12 *(Corresponding to Case 11) Suppose at t_0^+, $Ind.i = 1$, there exists an unique node $j \in N(i)$ such that $Ind.j = 1$, $p.i = \perp$, for every $u \in N(j) - \{i\}$, $Ind.u = 0$ and $p.u \neq j$ and for every $k \in N(i) - \{j\}$, $p.k \neq i$. If i is the first node in the system to change the Ind-value after t_0, then, right after i changes its Ind-value, the Ind-values of the whole system get back to normal, i.e., for every node $x \in V$, $L.x$.*

With the help of above analysis, it is now clear how restrictions should be imposed on Algorithm 1 in order for the system to contain the fault after it incurs the single-fault situation. Explicitly, any node that satisfies the condition in Case 4, Case 8 or Case 10 should be prohibited to make a move to change its *Ind*-value. Thus, we obtain the following prototype for our fault-containing algorithm:

Prototype 1

$R1$ $Ind.i = 0 \wedge (\forall x \in N(i), Ind.x = 0) \wedge |P(i)| \geq 2 \longrightarrow Ind.i := 1$

$R2$ $Ind.i = 0 \wedge (\forall x \in N(i), Ind.x = 0) \wedge p.i = \perp \longrightarrow Ind.i := 1$

$R3$ $Ind.i = 0 \wedge (\forall x \in N(i), Ind.x = 0) \wedge |P(i)| \leq 1 \wedge$

$\exists j \in N(i)$ s.t. $(p.i = j \wedge \exists u \in N(j) - \{i\}$ s.t. $Ind.u = 1) \longrightarrow Ind.i := 1$

$R4$ $Ind.i = 0 \wedge \forall x \in N(i), Ind.x = 0 \wedge |P(i)| \leq 1 \wedge$

$\exists j \in N(i)$ s.t. $[p.i = j \wedge \forall u \in N(j) - \{i\}, (Ind.u = 0 \wedge p.u \neq j)] \longrightarrow Ind.i := 1$

$R5$ $Ind.i = 1 \wedge (\exists x \in N(i)$ s.t. $Ind.x = 1) \wedge p.i \neq \perp \longrightarrow Ind.i := 0$

$R6$ $Ind.i = 1 \wedge \exists j, k \in N(i)$ s.t. $(j \neq k \wedge Ind.j = Ind.k = 1) \longrightarrow Ind.i := 0$

$R7$ $Ind.i = 1 \wedge p.i = \perp \wedge (\exists! j \in N(i)$ s.t. $Ind.j = 1) \wedge \forall u \in N(j) - \{i\}$, $Ind.u = 0 \wedge$

$\exists v \in N(j) - \{i\}$ s.t. $p.v = j \longrightarrow Ind.i := 0$

$R8$ $Ind.i = 1 \wedge p.i = \perp \wedge (\exists! j \in N(i) \text{ s.t. } Ind.j = 1) \wedge$

$0pt$ $\quad \forall u \in N(j) - \{i\},\ (Ind.u = 0 \wedge p.u \neq j) \wedge \forall k \in N(i) - \{j\}, p.k \neq i \longrightarrow Ind.i := 0$

$R9$ $Ind.i = 0 \wedge (\exists! j \in N(i) \text{ s.t. } Ind.j = 1) \wedge p.i \neq j \longrightarrow p.i := j$

$R10$ $Ind.i = 0 \wedge \exists j, k \in N(i) \text{ s.t. } (j \neq k \wedge Ind.j = Ind.k = 1) \wedge p.i \neq \perp \longrightarrow p.i := \perp$

$R11$ $Ind.i = 1 \wedge \forall j \in N(i), Ind.j = 0 \wedge p.i \neq \perp \longrightarrow p.i := \perp$

Legitimate states are all those global states in which the following condition holds :

$\forall i \in V, [(Ind.i = 0 \wedge (\exists! j \in N(i) \text{ s.t. } Ind.j = 1) \wedge p.i = j) \vee$

$(Ind.i = 0 \wedge \exists j, k \in N(i) \text{ s.t. } j \neq k \wedge Ind.j = Ind.k = 1 \wedge p.i = \perp) \vee$

$(Ind.i = 1 \wedge \forall j \in N(i), Ind.j = 0 \wedge p.i = \perp)]$

For the sake of convenience, the preceding condition satisfied by i is denoted as $Q.i$, and the guard conditions in $R9$, $R10$ and $R11$ in above prototype are denoted as $G_1.i$, $G_2.i$ and $G_3.i$, respectively. We also denote the negative of any condition P by $\neg P$. Then, one can check that the condition $\neg Q.i$ is equivalent to $\neg L.i \vee G_1.i \vee G_2.i \vee G_3.i$. Note that we have already classified the condition $\neg L.i$ into 11 cases previously. Therefore, if i satisfies $\neg Q.i$, then it must satisfy one of the conditions in Cases 1-11, or it must satisfy one of $G_1.i$, $G_2.i$ and $G_3.i$.

We have been convinced already by above analysis that prototype 1 has the fault-containment property. Before we transform Prototype 1 into a distributed algorithm, we still need to be convinced that it has the no-deadlock property also, because the no-deadlock property is crucial for it to be self-stabilizing. The following lemma meets this need.

Lemma 13 (No deadlock) *At the prototype level, the system is never deadlocked in an illegitimate state.*

Proof. Suppose the system is in an illegitimate state. Then there exists an $i \in V$ such that i satisfies $\neg Q.i$. According to above remark, i must satisfy one of the conditions in Cases 1-11 in the previous classification or i must satisfy one of the guard conditions $G_1.i$, $G_2.i$ and $G_3.i$ in $R9$,

$R10$ and $R11$, respectively, in Prototype 1. If i satisfies the guard condition in $R9$, $R10$ or $R11$ in Prototype 1, then i can execute $R9$, $R10$ or $R11$, respectively. If i satisfies the condition in Case 1, Case 2, Case3, Case 5, Case 6, Case 7, Case 9 or Case 11 in the above classification, then it can execute $R1$, $R2$, ..., or $R8$ in Prototype 1, respectively. If i satisfies the condition in Case 4, then node j in the condition can execute $R1$. If i satisfies the condition in Case 8, then node j in the condition can execute $R6$. Finally, if i satisfies the condition in Case 10, then node j in the condition can execute $R11$. Hence, the lemma is proved. ■

Note that some rules in above prototype require a node to collect informations from neighbors at distance 2. However, in a distributed system, a node is not allowed to read informations of nodes other than its direct neighbors. Therefore, in order to break through the limitation, we use auxiliary variables q (question) and a (answer) to fulfill the job of collecting informations. (It should be mentioned here that the idea of applying auxiliary variables q and a is borrowed from Ghosh et al. [4-8].) Thus, our fault-containing algorithm is finally ready.

Algorithm 2 [Fault-containing self-stabilizing algorithm for finding a maximal independent set]

$R1$ $Ind.i = 0 \wedge (\forall x \in N(i), Ind.x = 0) \wedge |P(i)| \geq 2 \longrightarrow Ind.i := 1$

$R2$ $Ind.i = 0 \wedge (\forall x \in N(i), Ind.x = 0) \wedge p.i = \perp \longrightarrow Ind.i := 1$

$R3$ $Ind.i = 0 \wedge (\forall x \in N(i), Ind.x = 0) \wedge |P(i)| \leq 1 \wedge \exists j \in N(i)$ s.t. $(p.i = j \wedge a.j = 0) \wedge$
$q.i = 0 \longrightarrow q.i := 1$

$R4$ $Ind.i = 0 \wedge \exists j \in N(i)$ s.t. $\{(Ind.j = 0 \wedge p.j = i \wedge q.j = 1) \wedge$
$[(\exists k \in N(i) - \{j\}$ s.t. $Ind.k = 1) \vee \forall x \in N(i) - \{j\}, (Ind.x = 0 \wedge p.x \neq i)]\} \wedge a.i \neq 1$
$\longrightarrow a.i := 1$

$R5$ $Ind.i = 0 \wedge (\forall x \in N(i), Ind.x = 0) \wedge |P(i)| \leq 1 \wedge \exists j \in N(i)$ s.t. $(p.i = j \wedge a.j = 1) \wedge$
$q.i = 1 \longrightarrow Ind.i := 1$

$R6$ $Ind.i = 1 \wedge (\exists x \in N(i)$ s.t. $Ind.x = 1) \wedge p.i \neq \perp \longrightarrow Ind.i := 0$

$R7$ $Ind.i = 1 \wedge \exists j, k \in N(i)$ s.t. $(j \neq k \wedge Ind.j = Ind.k = 1) \longrightarrow Ind.i := 0$

$R8$ $Ind.i = 1 \wedge p.i = \perp \wedge (\exists! j \in N(i) \text{ s.t. } Ind.j = 1) \wedge a.j = 0 \wedge q.i = 0 \longrightarrow q.i := 1$

$R9$ $Ind.i = 1 \wedge (\exists! j \in N(i) \text{ s.t. } Ind.j = 1) \wedge p.j = \perp \wedge q.j = 1 \wedge$

$\exists k \in N(i) - \{j\} \text{ s.t. } p.k = i \wedge a.i \neq 1 \longrightarrow a.i := 1$

$R10$ $Ind.i = 1 \wedge (\exists! j \in N(i) \text{ s.t. } Ind.j = 1) \wedge p.j = \perp \wedge q.j = 1 \wedge \forall k \in N(i) - \{j\}, p.k \neq i \wedge$

$a.i \neq 2 \longrightarrow a.i := 2$

$R11$ $Ind.i = 1 \wedge p.i = \perp \wedge (\exists! j \in N(i) \text{ s.t. } Ind.j = 1) \wedge q.i = 1 \wedge$

$[a.j = 1 \vee (a.j = 2 \wedge \forall k \in N(i) - \{j\}, p.k \neq i)] \longrightarrow Ind.i := 0$

$R12$ $(Ind.i = 0 \wedge \exists j \in N(i) \text{ s.t. } Ind.j = 1) \vee (Ind.i = 1 \wedge \forall j \in N(i), Ind.j = 0) \wedge q.i = 1 \longrightarrow$

$q.i := 0$

$R13$ $[[Ind.i = 0 \wedge \neg[\exists j \in N(i) \text{ s.t. } (Ind.j = 0 \wedge p.j = i \wedge q.j = 1)]] \vee$

$[Ind.i = 1 \wedge \neg[(\exists! j \in N(i) \text{ s.t. } Ind.j = 1) \wedge p.j = \perp \wedge q.j = 1]]] \wedge a.i \neq 0 \longrightarrow a.i := 0$

$R14$ $Ind.i = 0 \wedge (\exists! j \in N(i) \text{ s.t. } Ind.j = 1) \wedge p.i \neq j \longrightarrow p.i := j$

$R15$ $Ind.i = 0 \wedge \exists j, k \in N(i) \text{ s.t. } j \neq k \wedge Ind.j = Ind.k = 1 \wedge p.i \neq \perp \longrightarrow p.i := \perp$

$R16$ $Ind.i = 1 \wedge \forall j \in N(i), Ind.j = 0 \wedge p.i \neq \perp \longrightarrow p.i := \perp$

Legitimate states are all those global states in which the following condition holds :

$\forall i \in V, q.i = 0 \wedge a.i = 0 \wedge$

$[(Ind.i = 0 \wedge (\exists! j \in N(i) \text{ s.t. } Ind.j = 1) \wedge p.i = j) \vee$

$(Ind.i = 0 \wedge \exists j, k \in N(i) \text{ s.t. } j \neq k \wedge Ind.j = Ind.k = 1 \wedge p.i = \perp) \vee$

$(Ind.i = 1 \wedge \forall j \in N(i), Ind.j = 0 \wedge p.i = \perp)]$

For later use in the correctness proof, we mention here that the system is in an illegitimate state if and only if

$\exists i \in V \text{ s.t. } [(Ind.i = 0 \wedge \forall j \in N(i), Ind.j = 0) \vee$

$(Ind.i = 1 \wedge \exists j \in N(i) \text{ s.t. } Ind.j = 1) \vee$

$q.i = 1 \vee a.i \neq 0 \vee$

$(Ind.i = 0 \wedge (\exists! j \in N(i) \text{ s.t. } Ind.j = 1) \wedge p.i \neq j) \vee$

$(Ind.i = 0 \wedge \exists j, k \in N(i)$ s.t. $(j \neq k \wedge Ind.j = Ind.k = 1) \wedge p.i \neq \perp) \vee$

$(Ind.i = 1 \wedge \forall j \in N(i), Ind.j = 0 \wedge p.i \neq \perp)]$.

The checking of this can be easily done.

To help readers to comprehend the relationship between Algorithm 2 and prototype 1, we explain it in more detail here. For instance, rules $R3$, $R4$ and $R5$ in Algorithm 2 are devised to implement rules $R3$ and $R4$ in Prototype 1. When a node i satisfies the condition $[Ind.i = 0 \wedge \forall x \in N(i), Ind.x = 0 \wedge |P(i)| \leq 1 \wedge p.i = j]$, it wants to know if node j satisfies the condition $[(\exists k \in N(j) - \{i\}$ s.t. $Ind.k = 1)] \vee [\forall x \in N(j) - \{i\}, (Ind.x = 0 \wedge p.x \neq j)]$. According to $R3$ in Algorithm 2, it sets $q.i = 1$ to ask j whether j satisfies the condition. If j satisfies the condition, then according to rule $R4$ in Algorithm 2, it sets $a.j = 1$ to let i know that its answer is affirmative. Then, according to rule $R5$ in Algorithm 2, node i can make a move to change its Ind-value. Thus, rules $R3$ and $R4$ are faithfully implemented. Note that in rule $R3$ of Algorithm 2, the condition $a.j = 0$ is placed there as a requisite. This is a little subtle. One can see that if the condition is taken away from $R3$, then node i may change its q-value to 1 in the situation when node j does not satisfy the condition and yet $a.j = 1$, due to whatever reason. Node i may then go ahead to execute $R5$ and this is not what $R3$ or $R4$ in Prototype 1 really wants.

3 Correctness proofs

In this section, we should prove the correctness of Algorithm 2 which includes (1) no deadlock, (2) self-stabilization and (3) fault containment. However, due to limitation of space, we only prove the no-deadlock property rigorously. One might ask why we need to do so again, since the fault-containment property and the no-deadlock property have just be settled at the prototype level. A simple answer to this question is that there exists a gap between a proof at the prototype level and a proof at the algorithm level, due to the participation of the auxiliary variables q

and a in the algorithm, which causes some complication. One can see this without difficulty if he or she compares the following proofs with previous proofs carefully. As a matter of fact, we have encountered, in our other research, the situation in which although a prototype has the no-deadlock property, yet, after transforming it into a distributed algorithm, the induced algorithm has a deadlock in an illegitimate state.

Theorem 14 (No deadlock) *The system is never deadlocked in an illegitimate state.*

Proof. Suppose the system is in an illegitimate state. Then, as mentioned earlier,

$\exists i \in V$ s.t. $[(Ind.i = 0 \wedge \forall j \in N(i), Ind.j = 0) \vee$

$(Ind.i = 1 \wedge \exists j \in N(i)$ s.t. $Ind.j = 1) \vee$

$q.i = 1 \vee a.i \neq 0 \vee$

$(Ind.i = 0 \wedge (\exists! j \in N(i)$ s.t. $Ind.j = 1) \wedge p.i \neq j) \vee$

$(Ind.i = 0 \wedge \exists j, k \in N(i)$ s.t. $(j \neq k \wedge Ind.j = Ind.k = 1) \wedge p.i \neq \perp) \vee$

$(Ind.i = 1 \wedge \forall j \in N(i), Ind.j = 0 \wedge p.i \neq \perp)]$.

Case 1. $\forall x \in V, [(Ind.x = 0 \wedge \exists y \in N(x)$ s.t. $Ind.y = 1) \vee$

$(Ind.x = 1 \wedge \forall y \in N(x), Ind.y = 0)]$.

Subcase 1.1. $\exists i \in V$ s.t. $q.i = 1$. Then i can execute $R12$.

Subcase 1.2. $\forall x \in V$, $q.x = 0$. Then

$\exists i \in V$ s.t. $[a.i \neq 0 \vee$

$(Ind.i = 0 \wedge (\exists! j \in N(i)$ s.t. $Ind.j = 1) \wedge p.i \neq j) \vee$

$(Ind.i = 0 \wedge (\exists j, k \in N(i)$ s.t. $j \neq k \wedge Ind.j = Ind.k = 1) \wedge p.i \neq \perp) \vee$

$(Ind.i = 1 \wedge (\forall j \in N(i), Ind.j = 0) \wedge p.i \neq \perp)]$.

If $a.i \neq 0$, then since $\forall j \in N(i), q.j = 0$, i can execute $R13$.

If $Ind.i = 0 \wedge (\exists! j \in N(i)$ s.t. $Ind.j = 1) \wedge p.i \neq j$, then i can execute $R14$.

If $Ind.i = 0 \wedge \exists j, k \in N(i)$ s.t. $(j \neq k \wedge Ind.j = Ind.k = 1) \wedge p.i \neq \perp$, then i can execute $R15$.

If $Ind.i = 1 \wedge \forall j \in N(i), Ind.j = 0 \wedge p.i \neq \perp$, then i can execute $R16$.

Case 2. $\exists i \in V$ s.t. $(Ind.i = 0 \wedge \forall x \in N(i), Ind.x = 0)$

Subcase 2.1. $|P(i)| \geq 2 \vee p.i = \perp$. Then i can execute $R1$ or $R2$.

Subcase 2.2. $|P(i)| \leq 1 \wedge p.i \neq \perp$. (Let $p.i = j$ and note that $Ind.j = 0$.)

Subcase 2.2.1. $q.i = 0 \wedge a.j = 0$. Then i can execute $R3$.

Subcase 2.2.2. $q.i = 0 \wedge a.j \neq 0$.

Subcase 2.2.2.1. $\forall u \in N(j) - \{i\}$, $\neg(Ind.u = 0 \wedge p.u = j \wedge q.u = 1)$.

Since $q.i = 0$, $\forall u \in N(j)$, $\neg(Ind.u = 0 \wedge p.u = j \wedge q.u = 1)$. Hence, j can execute $R13$.

Subcase 2.2.2.2. $\exists u \in N(j) - \{i\}$ s.t. $(Ind.u = 0 \wedge p.u = j \wedge q.u = 1)$.

Subcase 2.2.2.2.1. $\forall v \in N(j) - \{u\}, Ind.v = 0$. Then j can execute $R1$.

Subcase 2.2.2.2.2. $\exists v \in N(j) - \{u\}$ s.t. $Ind.v = 1$.

Subcase 2.2.2.2.2.1. $a.j = 2$. Then j can execute $R4$.

Subcase 2.2.2.2.2.2. $a.j = 1$.

Subcase 2.2.2.2.2.2.1. $\forall y \in N(u), Ind.y = 0$.

If $|P(u)| \geq 2$, u can execute $R1$; Otherwise, u can execute $R5$.

Subcase 2.2.2.2.2.2.2. $\exists! y \in N(u)$ s.t. $Ind.y = 1$. Then u can execute $R14$.

Subcase 2.2.2.2.2.2.3. $\exists x, y \in N(u)$ s.t. $x \neq y \wedge Ind.x = Ind.y = 1$.

Then u can execute $R15$.

Subcase 2.2.3. $q.i = 1 \wedge a.j = 1$. Then i can execute $R5$.

Subcase 2.2.4. $q.i = 1 \wedge a.j \neq 1$.

If $\exists u \in N(j) - \{i\}$ s.t. $Ind.u = 1$, then j can execute $R4$.

If $\forall u \in N(j) - \{i\}, Ind.u = 0 \wedge p.u \neq j$, then j can execute $R4$.

If $\forall u \in N(j) - \{i\}, Ind.u = 0 \wedge \exists v \in N(j) - \{i\}$ s.t. $p.v = j$,

then since $|P(j)| \geq 2$, j can execute $R1$.

Case 3. $\exists i \in V$ s.t. $(Ind.i = 1 \wedge \exists x \in N(i)$ s.t. $Ind.x = 1)$

Subcase 3.1. $Ind.i = 1 \wedge \exists j, k \in N(i)$ s.t. $j \neq k \wedge Ind.j = Ind.k = 1$. Then i can execute $R7$.

Subcase 3.2. $Ind.i = 1 \wedge \exists! j \in N(i)$ s.t. $Ind.j = 1$

Subcase 3.2.1. $\exists u \in N(j) - \{i\}$ s.t. $Ind.u = 1$. Then j can execute $R7$.

Subcase 3.2.2. $\forall u \in N(j) - \{i\}, Ind.u = 0$.

Subcase 3.2.2.1. $p.i \neq \perp$. Then i can execute $R6$.

Subcase 3.2.2.2. $p.j \neq \perp$. Then j can execute $R6$.

Subcase 3.2.2.3. $p.i = \perp \wedge p.j = \perp$.

Subcase 3.2.2.3.1. $q.i = 0 \wedge a.j = 0$. Then i can execute $R8$.

Subcase 3.2.2.3.2. $q.i = 0 \wedge a.j \neq 0$. Then j can execute $R13$.

Subcase 3.2.2.3.3. $q.i = 1 \wedge a.j = 0$. Then j can execute $R9$ or $R10$.

Subcase 3.2.2.3.4. $q.i = 1 \wedge a.j = 1$. Then i can execute $R11$.

Subcase 3.2.2.3.5. $q.i = 1 \wedge a.j = 2$.

Subcase 3.2.2.3.5.1. $\forall k \in N(i) - \{j\}, p.k \neq i$. Then i can execute $R11$.

Subcase 3.2.2.3.5.2. $\exists k \in N(i) - \{j\}$ s.t. $p.k = i$.

Subcase 3.2.2.3.5.2.1. $q.j = 0$.

If $a.i = 0$, then j can execute $R8$; Otherwise, i can execute $R13$.

Subcase 3.2.2.3.5.2.2. $q.j = 1$.

If $a.i = 1$, then j can execute $R11$; Otherwise, i can execute $R9$.

Thus, we have considered all cases of illegitimate states, and we have shown that in any case, there always exists a node in the system which is privileged to make a move. Therefore, the theorem is proved. ■

Theorem 15 (Self-stabilization) *Starting with any initial state, the system will eventually stop in a legitimate state.*

Proof. Omitted. ■

Theorem 16 (Fault containment) *From any single-fault state till reaching a legitimate state, the system changes the Ind-value at most once.*

Proof. Omitted. ■

4 The stabilization time

We have also computed the stabilization time of the algorithm in the single-fault situation in the proof of the following theorem. Again, due to the space limitation, the proof is omitted.

Theorem 17 *In the single-fault situation, the stabilization time of the algorithm is $O(\Delta)$, where Δ is the maximum node degree.*

5 Concluding Remarks

In [5, 7], Ghosh et al. take a general approach in studying the fault-containment of the self-stabilizing algorithm. In those papers, they introduce a transformer that can convert any non-reactive self-stabilizing algorithm P into a non-reactive fault-containing self-stabilizing algorithm Q. However, if we apply the transformer to Shukla's algorithm in [9], the induced fault-containing algorithm has $\Omega(n^2)$ as its stabilization time in the single-fault situation, which is far from satisfactory. From this observation, we are led to believe that up to this point, the general approach for the fault-containment problem has not yet been successful, and with the stabilization time $O(\Delta)$ in the single-fault situation, the algorithm proposed in the paper can be considered as the most efficient fault-containing self-stabilizing algorithm for finding a maximal independent set.

References

[1] E. W. Dijkstra, "Self-stabilizing system in spite of distributed control," *Commun. ACM,* vol. 17, no. 11, pp. 643-644, 1974.

[2] E. W. Dijkstra, "Self-stabilizing system in spite of distributed control (EWD391)," Reprinted in: *Selected writing on computing: a personal perspective.* Berlin Heidelberg New York: Springer, 1982, pp. 41-46.

[3] E. W. Dijkstra, "A belated proof of self-stabilization," *Distrib. Comput.*, vol. 1, no. 1, pp. 5-6, 1986.

[4] S. Ghosh and A. Gupta, "An exercise in fault-containment: self-stabilizing leader election," *Inform. Process. Lett.*, vol. 59, pp. 281-288, 1996.

[5] S. Ghosh, A. Gupta and T. Herman, "Fault-containing self-stabilizing distributed protocols," Unpublished Manuscript, 2000.

[6] S. Ghosh, A. Gupta and S. V. Pemmaraju, "A fault-containing self-stabilizing spanning tree algorithm," *Journal of Computing and Information*, vol. 2, no. 1, pp. 322-338, 1996.

[7] S. Ghosh, A. Gupta, T. Herman and S. V. Pemmaraju, "Fault-containing self-stabilizing algorithms," in *Proc. PODC*, 1996, pp. 45-54.

[8] A. Gupta, *Fault-containing in self-stabilizing distributed algorithm.* Ph.D. thesis, University of Iowa, 1997.

[9] S. K. Shukla, D. J. Rosenkrantz and S. S. Ravi, "Observations on self-stabilizing graph algorithms for anonymous networks," in *Proc. WSS*, 1995, pp. 7.1-7.15.