Analyzing Self–Stabilization with Finite–State Machine Model

Su–Chu Hsu Shing–Tsaan Huang

Institute of Computer Science
National Tsing Hua University
Hsinchu, Taiwan 30043
Republic of China

Abstract

The paper presents a new approach for analyzing self–stabilization with the finite–state machine model. A finite–state machine is used to model the behavior of each node in a distributed system when the self–stabilizing algorithms are applied. The approach is useful to analyze the correctness of self–stabilizing algorithms and their time complexity. A self–stabilizing algorithm for finding mazimal matching is used as an example to describe how the finite–state machine model is applied. The results show that the approach is promising. Based on the approach, we get a simpler proof for the correctness and obtain a tighter upper bound of the time complexity than the one proved by a variant function.

1 Introduction

In distributed systems, unexpected perturbations on local variables are common; consequently, the development of distributed algorithms with fault–tolerance is desirable. The term *self–stabilization* was first introduced by Dijkstra [5]. A self–stabilizing system starting from any arbitrary initial state, possibly illegitimate, is always guaranteed to converge to a legitimate state in finite time without any outside intervention. For the self–stabilizing systems, the most attractive feature is that each node can locally detect faults and recover automatically. Such a property is very desirable for designing distributed systems with fault–tolerance. The concept of self–stabilization was regarded by Lamport [14] as Dijkstra's most brilliant contribution in distributed systems and a milestone in the area on fault–tolerance.

This work was supported by the National Science Council of the Republic of China in Taiwan under the Contract NSC 80–0408–E007–04.

In the original paper on self–stabilization [5], Dijkstra proposed three self–stabilizing protocols for the mutual exclusion problem of nodes in a ring. The system is self–stabilizing in the sense that, regardless of the number of tokens that may exist initially, the system is guaranteed to reach a state in which only one token exists fairly circulating along the ring. Following Dijkstra, some more discussions for the problem were carried out in [2], [8] and [10]. The earliest paper extending Dijkstra's work that we know of is by Kruijer in 1979 [13]. Kruijer investigated the self–stabilization of nodes connected in a tree structure. Recently, some related work addressing the self–stabilization of nodes configured in a general network for distributed problems can be found in [4], [7], [9] and [11].

Dijkstra's three self–stabilizing protocols were published in 1974, and the "belated" proof of the protocols was provided in 1986 [6]. Apparently the proof of correctness for self–stabilizing algorithms is not a trivial matter. Kessels proposed an approach for the first time by using a *variant function* to prove the correctness of self–stabilizing algorithms [12]. The basic concepts are : (i) to give a variant function whose value is bounded, and (ii) to prove that the variant function monotonically decreases (or increases) each time when the algorithm is applied. This approach was adopted and applied in different problems discussed in [4], [7], [9], [10] and [11]. In 1988, Chang was the first to use probabilistic analysis to get the average time complexity of one of Dijkstra's protocols [3]. However, in Chang's analysis, he only considered the special case in which a system starts with a fixed initial state. Herman also used probabilistic analysis to prove that his proposed algorithm can probabilistically converge [8]. However, based on these approaches, it is difficult to get a satisfiable time complexity of the algorithms.

The main contribution of this paper is to develop a

new approach with the FSM (finite–state machine) model for analyzing the correctness of the self–stabilizing algorithms and their time complexity. A nondeterministic finite–state machine is used to model the behavior of each node in the system when the self–stabilizing algorithm is applied. The basic concept of our approach is that regardless of any initial state, each node will converge to a terminal state in finite time and remains so forever. The number of nodes in the system is finite. Hence, regardless of any initial state of the system, it will stabilize in finite time. A self–stabilizing algorithm for finding maximal matching in distributed systems reported by Hsu and Huang [9] is used as an example to describe how the FSM model is applied. Consider a distributed system modeled as graph G(V,E), where V is the set of vertices which represents the set of processors and E is the set of edges which represents the set of the links of the network. The problem is to develop a self–stabilizing algorithm (a fault–tolerant graph algorithm) for finding maximal matching in G. The algorithm had been proved and analyzed by using a variant function in [9]. The upper bound of $O(|V|^3)$ was found. Based on the FSM model, we get a simpler proof for the correctness of the algorithm. A tighter upper bound $O(|E|)$ is obtained. The results show that our approach is promising.

The rest of the paper is organized as follows. In Section 2, the FSM model is introduced. In Section 3, an example is given to illustrate how the FSM model is applied. In Section 4, we first review the approach which uses a variant function for analyzing the algorithm in [9], and then compare it with our new approach. Finally, some remarks are drawn in Section 5.

2 The Finite–State Machine Model

In the paper, we propose a new approach that uses the finite–state machine model for analyzing self–stabilization. In the approach, we make the following assumptions. A distributed system can be viewed to consist of a set of loosely connected systems which do not share a global memory but can share information by exchanging messages only. Each node is allowed to have only a partial view of the global state which depends on the connectivity of the system as well as the propagation delay of different messages. Each node maintains its local variables. Due to any unexpected perturbation, each variable may vary but its value is always within its domain. The domain is a set covers all possible values of the variable. Thus, the initial state of the system can be arbitrary, possibly in an illegitimate state. The objective in a distributed system is to arrive at a global final state (legitimate state) regardless of any initial state.

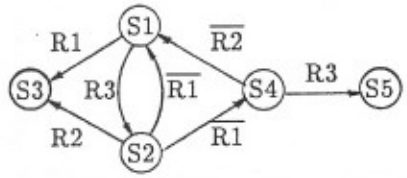
A self–stabilizing algorithm is expressed by a set of rules. The rules are expressed as : "*condition* $\Longrightarrow$ *a corresponding move*". The *condition* is defined to be boolean functions of the node's own variables and the variables of its neighbors. When the condition a node is true, the node may make the corresponding *move*. Any node for which the condition is true is said to have *privilege*. In some instances, many nodes may have the privileges at the same time. However, it is required that only one node makes its move at one time, and the next move depends on the result from the previous move. In the present paper, we assume that there exists a central demon as introduced in [5] which is used to activate a single node each time randomly. The activated node executes an atomic step which is composed of three substeps: (i) reading the variables of all its neighbors, (ii) checking whether it has privilege, and (iii) making the corresponding move (modifying its variables) if it has privilege. During the execution of the atomic step no action for other node is allowed. Note that there may have different moving sequences because of the random selection by the central demon.

In our analysis, each node has a *state*. The state is defined by its variables and the variables of its neighbors. Note that the state so defined is different from that of [4], [5], [7], [8], [10] and [11]. In those previous works, the state of a node is defined only by its local variables. As will be seen, the state of the paper is used to analyze the self–stabilizing algorithms rather than to express the rules. Let I denote all the possible initial states of each node in the system. Also let F represent all the possible terminal states of each node. Assume that once a node reaches a terminal state, it remains so forever. Hence, when all nodes are in the states of F, we say the system stabilizes.

We use a nondeterministic finite–state machine FSM=(I,R,T,F) to model the behavior of each node, where I and F are the set of initial and terminal states respectively, R is the set of rule symbols, and T is the set of all possible state transitions of the node. There are two kinds of rule symbols, R_j and $\overline{R_j}$. Each directed edge of T represents a state transition of the node due to an application of the rule by itself or by another node. We use $currentstate \xrightarrow{R_j} neztstate$ to denote the state transition of the node from the current state to the next state due to an application of rule (R_j) by itself. Whereas, $currentstate \xrightarrow{\overline{R_j}}$

nextstate is used to denote the state transition of the node from the current state to the next state due to an application of rule (R_j) by another node.

Figure 1 is an example showing the state–transition diagram of an FSM. There are five states for each node and three rules are used. A node with state S2 changes its state to S3 if itself applies (R2); a node with state S4 changes its state to S1 if another node applies (R2), etc.



FSM=(I,R,T,F)

I={S1,S2,S3,S4,S5} : the set of initial states;

R={R1,R2,R3,$\overline{R1}$,$\overline{R2}$} : the set of rule symbols;

F={S3,S5} : the set of terminal states;

T={T1,T2,T3,T4,T5,T6,T7} : the set of all possible state transitions, where

T1: S1 $\xrightarrow{R1}$ S3 T5: S2 $\xrightarrow{\overline{R1}}$ S4

T2: S1 $\xrightarrow{R3}$ S2 T6: S4 $\xrightarrow{\overline{R2}}$ S1

T3: S2 $\xrightarrow{R2}$ S3 T7: S4 $\xrightarrow{R3}$ S5.

T4: S2 $\xrightarrow{\overline{R1}}$ S1

Figure 1 An example for FSM

The main difficulty of this approach is how to derive the set of state transitions, T. An example in Section 3 will show how to derive T in details.

Proving the correctness of a self–stabilizing algorithm needs to verify that the algorithm satisfies the following requirements :

(i) If the system reaches a legitimate state, no node can make further moves.

(ii) If the system is in any illegitimate state, there exists at least one node which can make a move.

(iii) Regardless of any initial state and regardless of any moving sequence selected by the central demon, the system is guaranteed to stabilize after a finite number of moves.

The purpose for verifying requirement (iii) is to prove the convergence of the algorithm. The basic concept in proving that the algorithm meets requirement (iii) is to show that regardless of any initial state of each node and regardless of which node is selected to make a move by the central demon, each node will converge to a terminal state in a finite number of moves and remains so thereafter. If there is no cycles in the state–transition diagram of the FSM, the proof is obvious. If there exist cycles, the proof will be done as long as we can prove that the number of times that each node goes through the cycles is finite. In our analysis, we particularly focus on the edges which result in cycles in the diagram. An example in the following section will describe it in details. Because the number of nodes in the system is finite and each node converges to a terminal state in a finite number of moves, the system stabilizes in a finite number of moves.

Finally, the basic steps involved in applying the FSM model can be summarized as follows :

(i) Design the rules of self–stabilizing algorithm.

(ii) Define the states of each node.

(iii) Derive the state–transition diagram in terms of the rules and the states.

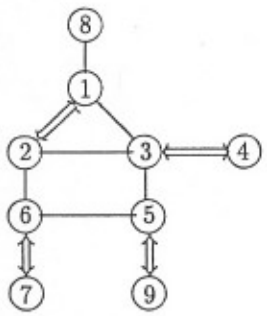
(iv) Prove the correctness of the algorithm.

(v) Analyze the time complexity of the algorithm.

3 An Example

In this section, we first describe the problem for finding maximal matching in a distributed network and the algorithm proposed in [9]. Then we show how the FSM model is applied.

Consider an undirected graph G(V,E), where $|V| \doteq n$. A *matching* of G(V,E) is a set of edges M, M $\subseteq$ E, in which no two edges connect to a common node. A matching M is maximal if it is not properly contained in any other matching. For example, in Figure 2, G(V,E) with V={1,2, ..., 9} is given. M={(1,2), (3,4), (6,7), (5,9)} is a maximal matching of G. {(1,2), (1,8), ... } is not a matching because edges (1,2) and (1,8) connect to a common node. {(1,2), (3,4), (6,7)} is a matching but not a maximal matching because it is properly contained in {(1,2), (3,4), (6,7), (5,9)}.



M={(1,2),(3,4),(6,7),(5,9)} is a maximal matching

Figure 2 An example for a maximal matching

Figure (3) is used to illustrate some of the above state–transitions which may not be obvious. Figure 3(a) describes case (2). In the case, besides i, initially there is a neighbor k of j selecting j. After j applies (R1) and selects k, S.i changes its state from *waiting* to *chaining*. Figure 3(b) describes case (5). In the case, initially S.i and S.j both are *chaining*. After j applies (R3) and lets $j \longrightarrow null$, S.i will change its state from *chaining* to *waiting*. Figure 3(c) and 3(d) describe case (8). Suppose i has a neighbor set $\{j_1, j_2, \ldots, j_m, j\}$. Initially, except j, all the neighbors of i have gotten matched. In Figure 3(c), S.j is *free* and there is a neighbor k of j selecting j. After j applies (R1) and gets matched, S.i changes its state from *free* to *dead*. In Figure 3(d), j selects k and S.j is *waiting*. After k applies (R1) and j gets matched, S.i changes its state from *free* to *dead*.

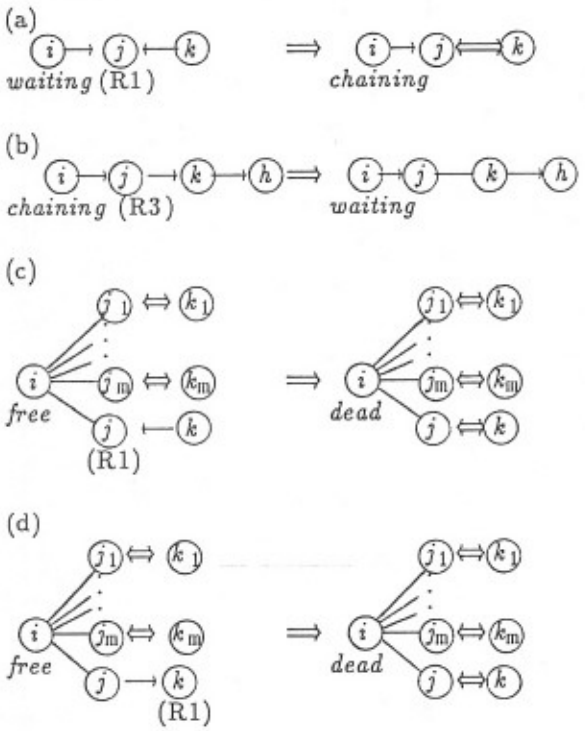


Figure 3 Illustration for state–transitions.

Note that for each state except the terminal states, the above observations considered have been covered all possible cases of i's neighbors. From the above discussions, we have all possible state transitions as below.

T1: *waiting* $\xrightarrow{\overline{R1}}$ *matched*

T2: *waiting* $\xrightarrow{\overline{R1}}$ *chaining*

T3: *chaining* $\xrightarrow{R3}$ *free*

T4: *chaining* $\xrightarrow{R3}$ *dead*

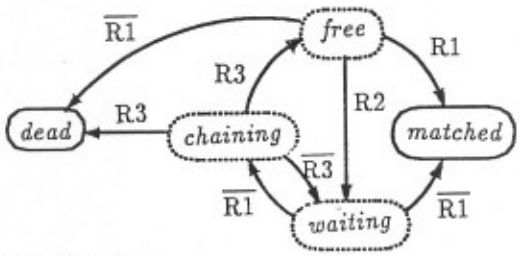
T5: *chaining* $\xrightarrow{\overline{R3}}$ *waiting*

T6: *free* $\xrightarrow{R1}$ *matched*

T7: *free* $\xrightarrow{R2}$ *waiting*

T8: *free* $\xrightarrow{\overline{R1}}$ *dead*

From the discussion of Section 3.1, we know the initial state of node i may be *matched*, *dead*, *waiting*, *free*, or *chaining*, i.e., I={*matched*,*dead*,*waiting*,*free*, *chaining*}. The terminal state of node i must be either *matched* or *dead* when the system reaches a legitimate state. Based on the above state transitions, we get R={R1,R2,R3,$\overline{R1}$,$\overline{R3}$} and T={T1,T2,T3,T4,T5,T6, T7,T8}. Therefore, the state–transition diagram of the FSM can be obtained as below.



FSM=(I,R,T,F)
I={*matched*,*dead*,*waiting*,*free*,*chaining*};
R={R1,R2,R3,$\overline{R1}$,$\overline{R3}$};
T={T1,T2,T3,T4,T5,T6,T7,T8};
F={*matched*,*dead*}.

Figure 4 The FSM of each node for the maximal matching problem

Regardless of which initial state of a node, the state transitions of the node can be derived (accepted) from the finite–state machine of Figure 4. By the finite–state machine of Figure 4, we can easily get the following Lemma 1 and Lemma 2. These two lemmas will be used in proving Lemma 4.

Lemma 1 If we remove the transition T2: *waiting* $\xrightarrow{\overline{R1}}$ *chaining*, the diagram becomes acyclic.

Proof: In the state–transition diagram of Figure 4, note that there are two cycles: *waiting*$\longrightarrow$*chaining*$\longrightarrow$ *free*$\longrightarrow$*waiting* and *waiting*$\longrightarrow$*chaining*$\longrightarrow$*waiting*. It is easy to see that if we remove the transition T2, the diagram becomes acyclic. □

Assume that each node i in G knows $N(i)$, the set of its adjacent nodes (neighbors). We let node i maintain a pointer variable. The pointer points to one of i's neighbors and is used to indicate which neighbor i selects to match. If i's pointer points to null, then it means i does not select anyone. The notation $i \longrightarrow j$ is used to denote that the pointer of i points to j, and $i \longrightarrow null$ is used to mean that the pointer of i points to null. The notation $i \Longleftrightarrow j$ is used to represent $i \longrightarrow j$ and $j \longrightarrow i$; i.e., i and j mutually select each other and get matched. Due to unexpected perturbations, the pointer of each node may be affected and vary but it is always within its domain, $\{null\} \cup N(i)$.

3.1 Design the Rules of the Algorithm

The following set of rules is the algorithm reported in [9]. It is identically stored and executed in each node i. The conditions of the rules of each node i are defined to be boolean functions of its own pointer and the pointers of its neighbors. Node i enjoying the selected privilege will then make its move by modifying its pointer.

The Self–Stabilizing Algorithm for Maximal Matching :

(R1) $(i \longrightarrow null) \wedge (\exists j : j \in N(i) : j \longrightarrow i)$
$\Longrightarrow$ Let $i \longrightarrow j$

(R2) $(i \longrightarrow null) \wedge (\forall k : k \in N(i) : \neg(k \longrightarrow i)) \wedge$
$(\exists j : j \in N(i) : j \longrightarrow null)$
$\Longrightarrow$ Let $i \longrightarrow j$

(R3) $(i \longrightarrow j) \wedge (j \longrightarrow k) \wedge (k \neq i)$
$\Longrightarrow$ Let $i \longrightarrow null$

Rule (R1) indicates that if node i does not select anyone and a neighbor j selects i, then i selects j. Rule (R2) describes the situation that involves three conditions : (i) node i selects no one, (ii) none of i's neighbors selects i, and (iii) there exists a neighbor j selecting no one. If all three conditions hold, then i selects j. Rule (R3) means that if node i has selected j and j has selected another node, then i must give up selecting j. Note that if i has a neighbor which has selected i, then i could not apply (R2). This fact will be used in the proof of Lemma 2 in Section 4. Note that the self–stabilizing algorithm is a fault–tolerant graph algorithm rather than the token based distributed algorithms.

3.2 Define the States

In our discussion, S.i is used to denote the state of node i. As mentioned earlier, the states so defined are not used in the design of the rules but used for the analysis of the algorithm. If $i \longrightarrow j$, then there are three possible states. As mentioned before, the states are defined by i's pointer and the pointers of its neighbors.

(1) S.i=*waiting* if $(i \longrightarrow j) \wedge (j \longrightarrow null)$;
(2) S.i=*matched* if $i \longrightarrow j \wedge j \longrightarrow i$ (i.e., $i \Longleftrightarrow j$);
(3) S.i=*chaining* if $(i \longrightarrow j) \wedge (j \longrightarrow k) \wedge (k \neq i)$.

S.i=*waiting* means i has selected j and waits for j to select it. S.i=*matched* means i has gotten matched. And S.i=*chaining* means i has selected j but j has selected another node.

If $i \longrightarrow null$, then there are two possible states.
(4) S.i=*dead* if $(i \longrightarrow null) \wedge (\forall j : j \in N(i) : S.j = matched)$;
(5) S.i=*free* if $(i \longrightarrow null) \wedge (\exists j : j \in N(i) : S.j \neq matched)$.

S.i=*dead* means i has no chance to get matched because all its neighbors have gotten matched. And S.i=*free* means that although i does not select anyone, it still has chances to get matched.

Therefore, the state of each node may be *waiting*, *matched*, *chaining*, *dead*, or *free*. A global state of the system is defined as the collection of all states of the nodes in the system. By the definition of maximal matching, when the system reaches maximal matching, each node must be either *matched* or *dead*. Thus, the system is said to be in a legitimate state (i.e., stabilized) if each node is either *matched* or *dead*. When the following GMM is true, the system has reached a legitimate state.

GMM $\equiv$ ($\forall i ::$ S.i=*matched* $\vee$ S.i=*dead*)

3.3 Derive the State–Transition Diagram

In this subsection, we derive the state–transition diagram from the rules of the algorithm and the states of the nodes. For each kind of states, we discuss how the state changes when rules are applied. By the definitions of the states of the nodes and the applications of the rules, we can have the following observations of the state–transitions.

(i) It is easy to see that no matter which rule is applied, once S.i=*matched* or *dead*, it remains unchanged.

(ii) If S.i=*waiting* (i.e., $i \longrightarrow j$, $j \longrightarrow null$), then S.i will change
(1) from *waiting* to *matched* if j applies (R1) and lets $j \longrightarrow i$, or
(2) from *waiting* to *chaining* if j applies (R1) and lets $j \longrightarrow k$, $k \neq i$.

(iii) If S.i=*chaining* (i.e., $i \longrightarrow j$, $j \longrightarrow k$, $k \neq i$), then S.i will change
(3) from *chaining* to *free* if i applies (R3), or
(4) from *chaining* to *dead* if i applies (R3), or
(5) from *chaining* to *waiting* if j applies (R3).

(iv) If S.i=*free* (i.e., $i \longrightarrow null$ and $(\exists j : j \in N(i) :$ S.$j \neq matched$)), then S.i will change
(6) from *free* to *matched* if i applies (R1), or
(7) from *free* to *waiting* if i applies (R2), or
(8) from *free* to *dead* if j changes its state to *matched*, because j or j's neighbor applies (R1).

Lemma 2 Once a node's state changes from *waiting* to *chaining*, a neighbor of the node must change its state from *free* to *matched* at the same time.
Proof: Suppose S.i is *waiting* (i.e., $i \longrightarrow j$ and $j \longrightarrow null$). That S.i changes from *waiting* to *chaining* only occurs when node j applies (R1) letting $j \longrightarrow k$, $k \neq i$. As mentioned earlier, j could not apply (R2), because i has selected j. Thus, by applying (R1), node j will get matched with some node k. □

3.4 Prove the Correctness of the Algorithm

According to the definition of GMM and the rules of the algorithm, it is obvious that when GMM is true, no rules can be applied. Thus, it is easy to see that the algorithm meets requirement (i). Lemma 3 will prove that the algorithm satisfies requirement (ii). By means of the diagram of Figure 4, Lemma 4 and Theorem 1 show that the algorithm meets requirement (iii) and an upper bound for the algorithm can be obtained.

Lemma 3 If GMM is false, there exists at least one node which can make a move; i.e., there exists some rule to be applied.
Proof: GMM is false $\Longrightarrow$ ($\exists i$: : $\neg$(S.i=*matched* $\vee$ S.i=*dead*)). In other words, there exists some node i whose state is *waiting*, *chaining* or *free*.
(1)S.i=*waiting* (i.e., $i \longrightarrow j$, $j \longrightarrow null$): j can make a move by applying (R1).
(2)S.i=*chaining* (i.e., $i \longrightarrow j$, $j \longrightarrow k$, $k \neq i$): i can apply (R3).
(3)S.i=*free* (i.e., $i \longrightarrow null$ and ($\exists j$:$j \in$N(i): S.$j \neq$ *matched*)): This case indicates that S.j may be *dead*, *waiting*, *chaining* or *free*.
(i) S.j=*dead*: By the definition of *dead*, S.j=*dead* is impossible, because S.i=*free*.
(ii) S.j=*waiting* ($j \longrightarrow k$): By (1), k can apply (R1).
(iii) S.j=*chaining* ($j \longrightarrow k$, $k \longrightarrow h$, $h \neq j$): By (2), j can apply (R3).
(iv) S.j=*free* : By contradiction, we assume that no node can apply a rule. That means that there is no node whose state is *waiting* or *chaining* according to (1) and (2); i.e., S.i=S.j=*free* and all other nodes are *free*, *dead* or *matched*. In this case, by the rules it is obvious that both i and j can apply (R2). It is contrary to the assumption. □

Lemma 4 Node i can make at most O(|N(i)|) transitions before it goes to the state *matched* or *dead*.
Proof: By Lemma 1, if we remove the transition *waiting*$\longrightarrow$*chaining*, then there is no cycle in the diagram of Figure 4. In the acyclic diagram, it is clear that the state of each node i will converge to *matched* or *dead* within three transitions. If we can establish that the number of times that node i goes through the transition *waiting*$\longrightarrow$*chaining* is within O(|N(i)|), then the proof is obtained. By Lemma 2, if node i's state changes from *waiting* to *chaining*, then there must exist a neighbor of i whose state changes from *free* to *matched* at the same time. Once a node's state is *matched*, it will remain so thereafter. Furthermore, the number of the neighbors of node i is |N(i)|. It follows that the number of times that node i can go through the transition *waiting*$\longrightarrow$*chaining* is O(|N(i)|). □

3.5 Analyze the Time Complexity of the Algorithm

Because the number of the nodes in the system is finite and each node converges to a terminal state in a finite number of moves, the time complexity of the algorithm can be analyzed as below.

Theorem 1 Regardless of any initial state, the system converges to a legitimate state within O(|E|) moves.
Proof: By Lemma 4, the state of each node i will converge to *matched* or *dead* within O(|N(i)|) transitions. In the worst case, the maximal total number of transitions for the system reaching a legitimate state is $\Sigma_{i \in V}$ O(|N(i)|)= O(|E|). It is possible that several state transitions of different nodes may occur at the same time when a node makes a move. Thus, the maximal total number of moves is less than the maximal total number of transitions. Therefore, regardless of any initial state, the system converges to a legitimate state within O(|E|) moves. □

4 Comparisons

In order to clearly compare the basic concepts, advantages and disadvantages of our approach with the previous one of [9], we first review the relevant parts of the previous approach, and then compare it with our approach.

In order to prove the algorithm meets requirement (iii), a verification method based on a variant function was reported in [9]. The basic concepts of the method are : (1) to give a variant function whose value is bounded; (2) to prove the variant function monotonically decreases (or increases) when nodes make moves.

Let m, d, w, f and c denote the total number of nodes whose state are *matched*, *dead*, *waiting*, *free* and *chaining* respectively. A variant function F was defined as: $F \equiv (m+d, w, f, c)$.

The comparison of the values of F is by lexicographical order. Each global state of the system corresponds to one value of F.

By the definition of GMM, the value of F corresponding to the legitimate state is $(n,0,0,0)$. Clearly, it is the upper bound of F. Thus, if F can be proved to monotonically increase for each move, the algorithm will meet requirement (iii).

Lemma 5 F monotonically increases each time when rule (R1), (R2) or (R3) is applied.

Proof: There are three cases to be discussed.

(1)If (R1) is applied, then there will be a pair of nodes which can get *matched*. In other words,

$(S.i=free) \wedge (S.j=waiting \wedge j \longrightarrow i)$

(after i applies (R1))

$\Longrightarrow (S.i=matched) \wedge (S.j=matched)$.

Because the states of i and j are changed to *matched*, the states of some neighbors of i and/or j may be changed from *free* to *dead*. Furthermore, it should be clear that no node can have its state changed from *matched* or *dead* to any other state no matter which rule is applied. Therefore, after (R1) is applied, no matter how states of nodes will be affected, at least we know that m increases by 2 and d may increase. It follows that F increases.

(2)If (R2) is applied by node i, then we have $(i \longrightarrow null) \wedge (\forall k : k \in N(i) : \neg(k \longrightarrow i)) \wedge (\exists j : j \in N(i) : j \longrightarrow null)$ before the application of (R2). It can be easily verified that after i applies (R2), only $S.i$ changes from *free* to *waiting*. In other words,

$(S.i=free)$

(after i applies (R2))

$\Longrightarrow (S.i=waiting)$.

Thus, after (R2) is applied, m, d and c are unchanged, f decreases by 1 and w increases by 1. It follows that F increases.

(3)If (R3) is applied by node i, then $i \longrightarrow null$ after the application of (R3). Only the following two cases are possible.

(i) $(S.i=chaining) \wedge (\forall k : k \in N(i) : \neg(k \longrightarrow i))$

(after i applies (R3))

$\Longrightarrow (S.i=free \vee S.i=dead)$

(ii) $(S.i=chaining) \wedge (k \longrightarrow i)$

/* Note that $S.k=chaining$. */

(after i applies (R3))

$\Longrightarrow (S.i=free) \wedge (S.k=waiting)$

In case (i), it is clear that although c decreases by 1, either f or d should increase by 1. Hence, F increases. In case (ii), although c decreases at least by 2, f should increase by 1 and w should increase at least by 1. Thus, F increases.

By (1), (2) and (3), we have that F monotonically increases each time when rule (R1), (R2) or (R3) is applied. □

The following Lemma 6 is given to support Theorem 2. The proof of Lemma 6 can be found in [1].

Lemma 6 The number of the non—negative integer solutions (x_1,x_2,x_3,x_4) for the equation $x_1+x_2+x_3+x_4=n$ is $\binom{n+3}{3}=\frac{(n+1)*(n+2)*(n+3)}{6}$.

Theorem 2 Regardless of any initial state, the system will converge to a legitimate state within $O(|V|^3)$ moves.

Proof: Regardless of which state of the system, the value of $m+d+w+f+c$ is always equal to $n(=|V|)$ and the values of $m+d$, w, f and c are always between 0 and n individually. In the worst case, the initial value of F is $(0,0,0,n)$. One would like to know how many moves it may need to transfer F from $(0,0,0,n)$ to $(n,0,0,0)$ in the worst case . Such a problem can be reduced to the problem described in Lemma 6 with $x_1=m+d$, $x_2=w$, $x_3=f$ and $x_4=c$. Thus, the maximal total number of moves for the system to converge to a legitimate state is $\frac{(n+1)*(n+2)*(n+3)}{6} - 1 = O(n^3) = O(|V|^3)$. □

As shown above, in [9] a variant function was used to analyze the correctness of the algorithm and its time complexity. Compared with the current approach using FSM, it can be seen that the previous approach using variant function is more complex. Besides that, the variant function only maps the system to a value which can not describe the behavior of the system in details. Thus, the obtained upper bound $O(|V|^3)$ is not tight enough.

Sometimes, it is difficult to analyze the behavior of the whole system when rules are applied. We can first analyze the behavior of each node, and then by collecting the behaviors of the nodes, the behavior of the whole system can be obtained. The currently proposed approach with the finite—state machine model adopts this idea. In the approach, we derive all the state—transitions for each node from the states of the nodes and the rules of the algorithm. The state is defined not only based on the information of a node but also based on the information of its neighbors. When we analyze the behavior of each node, we actually have gotten some global information of the system. Such a property is very worthwhile to note.

The most important part of the new approach is how to analyze and prove that each node eventually converges to a terminal state even though there are cycles in the state–transition diagram. In our example, we particularly focus on the directed edges which form cycles in the state–transition diagram. We prove that the number of times that each node goes through those edges is finite. This makes the analysis much simpler. In our analysis, we get a simpler proof for the correctness of the algorithm and obtain a tighter upper bound $O(|E|)$.

5 Conclusions

In this paper, we propose a new approach with the finite–state machine model to analyze the self–stabilizing algorithms. A self–stabilizing algorithm for finding maximal matching given in [9] is used to as an example illustrate how the approach is applied. The results show that the proposed approach is promising. We expect that it will have broader applications.

Because of the randomness of the system topology, the arbitrary initial state of the system, and the nondeterministic behavior of the central demon, the self–stabilizing algorithms are more complex and difficult to be analyzed than the conventional distributed algorithms. How to analyze self–stabilizing algorithms is a challenge research topic. Besides the variant function and the FSM model, other approaches for analyzing the algorithms deserve further studies.

References

[1] R.C. Bose and B. Manvel, *Introduction to Combinatorial Theory*, John Wiley & Sons, Inc., pp. 48, 1984.

[2] J.E. Burns and J. Pachl "Uniform Self–Stabilizing Rings." ACM Trans. on Programming Language and Systems, Vol. 11, No. 2 , pp. 330–344, 1989.

[3] E.J.H. Chang, G.H. Gonnet and D. Rotem "On the Cost of Self–Stabilization", Inf. Process. Lett., Vol. 24 , pp. 311–316, 1987.

[4] N.S. Chen, F.P. Yu and S.T. Huang "A Self–Stabilizing Algorithm for Constructing Spanning Trees." Inf. Process. Lett., Vol. 39, pp. 147–151, 1991.

[5] E.W. Dijkstra "Self–Stabilizing Systems in Spite of Distributed Control." Commun. ACM, Vol. 17, No. 11, pp. 643–644, 1974.

[6] E.W. Dijkstra "A Belated Proof of Self–Stabilization." Distributed Comput., Vol. 1, No. 1, pp. 5–6, 1986.

[7] M.G. Gouda and T. Herman "Stabilizing Unison", Inf. Process. Lett., Vol. 35, pp. 171–175, 1990.

[8] T. Herman "Probabilistic Self–Stabilization." Inf. Process. Lett., Vol. 35, pp. 63–67, 1990.

[9] S.C. Hsu and S.T. Huang "A Self–Stabilizing for Maximal Matching." Tech. Report.

[10] S.T. Huang "Leader Election in Uniform Rings." submitted to ACM Trans. on Programming Language and Systems, under revision.

[11] S.T. Huang and N.S. Chen "A Self–Stabilizing Algorithm for Constructing Breadth–First Trees." to appear in Inf. Process. Lett.

[12] J.L.W. Kessels "An Exercise in Proving Self–Stabilization with a variant Function." Inf. Process. Lett., Vol. 29, pp. 39–42, 1988.

[13] H.S.M. Kruijer "Self–Stabilization (in Spite of Distributed Control) in Tree–Structured Systems." Inf. Process. Lett., Vol. 8, pp. 91–95, 1979.

[14] L. Lamport "Solved Problems, Unsolved Problems and Non–Problems in Concurrency." Proc. 3rd ACM Symposium on Principles of Distributed Computing, pp. 1–11, 1984.