On the Analysis of Self-Stabilizing Algorithms Using Model Checking

January 2011

Masahiro KIMOTO

On the Analysis of Self-Stabilizing Algorithms Using Model Checking

Submitted to
Graduate School of Information Science and Technology
Osaka University

January 2011

Masahiro KIMOTO

Abstract

A self-stabilizing system is a distributed system that has two properties: *convergence* and *closure*. *Convergence* specifies that the system can reach *legitimate (safe) configurations* from any configuration. *Closure* specifies that once the system reaches a legitimate configuration, it continues to be within the set of legitimate configurations. Because of these properties, self-stabilizing systems need not be initialized and can automatically recover from erroneous configurations. A self-stabilizing algorithm is an algorithm that enables a system to be self-stabilizing. These algorithms have been proposed to deal with various problems.

The *time complexity* of a self-stabilizing algorithm is the maximal number of steps required to reach a legitimate configuration from an illegitimate one. The improvement of time complexity is an important performance issue, because illegitimate configurations can lead to malfunctions. To improve time complexity, it is also important to be able to compute it for a given self-stabilizing algorithm.

We address these issues in these dissertation. The contribution comprises of three parts: the first, providing a new lower bound on the time complexity of Dijkstra's three-state self-stabilizing mutual exclusion algorithm; the second, providing the exact time complexity of a self-stabilizing maximal matching algorithm

proposed by Hsu and Huang; the third, devising a new self-stabilizing maximal matching algorithm, which outperforms the Hsu–Huang algorithm in terms of time complexity.

In this line of research, we use model checking as an analysis tool. Model checking is a formal verification method based on state exploration. Although model checking can only be used for examining small-sized self-stabilizing algorithms, it allows us to fully analyze their behaviors and compute their time complexity. We describe how NuSMV, a major model checker, can be used for analyzing the mutual exclusion algorithm and the maximal matching algorithm.

List of Major Publications

[1] Masahiro Kimoto, Tatsuhiro Tsuchiya, and Tohru Kikuno, "Extraction of Fault-Prone Modules Based on Fault Tracking Data from Open Source Software Repository," In 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN2007), Supplemental Proceedings, pages 366–367, June 2007.

[2] Masahiro Kimoto, Tatsuhiro Tsuchiya, and Tohru Kikuno, "The Lower Bound on the Stabilization Time of Dijkstra's Three State Mutual Exclusion Algorithm," IEICE Technical Report, volume 108, number 11, pages 41 – 47, April 2008. (In Japanese)

[3] Masahiro Kimoto, Tatsuhiro Tsuchiya, and Tohru Kikuno, "On the Time Complexity of Dijkstra's Three-State Mutual Exclusion Algorithm," IEICE Transactions on Information Systems, volume E92-D, number 8, pages 1570–1573, August 2009.

[4] Masahiro Kimoto, Tatsuhiro Tsuchiya, and Tohru Kikuno, "The Time Complexity of Hsu and Huang's Self-Stabilizing Maximal Matching Algorithm," IEICE Transactions on Information Systems, volume E93-D, number 10, pages 2850–2853, October 2010.

[5] Masahiro Kimoto, Tatsuhiro Tsuchiya, and Tohru Kikuno, "Computing the Stabilization Time of Self-Stabilizing Algorithms Using Symbolic Model Checking," In 4th Symposium on Science and Technology for System Verification, Proceedings, pages 151 – 160, November 2007. (in Japanese)

Acknowledgements

During the course of this work, I am fortunate to have received assistance from many individuals. In particular, I would like to thank my supervisor Professor Tohru Kikuno for his continuous support, encouragement, and guidance for this work.

I am also very grateful to the members of my dissertation review committee, Professor Takao Onoye and Professor Toshimitsu Masuzawa for their invaluable comments and helpful criticism of this dissertation.

I would like to express my special thanks to Associate Professor Tatsuhiro Tsuchiya for his continuous assistance and helpful advice.

Finally, I wish to thank many friends in the Graduate School of Information Science and Technology at Osaka University for their valued help.

Contents

Chapter 1

Introduction

1.1 Background

A self-stabilizing system is a distributed system that has two properties: *convergence* and *closure*. *Convergence* specifies that the system can reach *legitimate (safe) configurations* from any configuration. *Closure* specifies that once the system reaches a legitimate configuration, it continues to be within the set of legitimate configurations.

Because of these properties, self-stabilizing systems need not be initialized and can automatically recover from erroneous configurations. Thus, self-stabilizing systems are tolerant of transient faults, such as the loss of memory contents and message omission. A self-stabilizing algorithm is an algorithm that enables a system to be self-stabilizing.

The notion of self-stabilization of a distributed system was introduced to computer science by Dijkstra in 1974 [13]. Originally, mutual exclusion was the only application of self-stabilizing algorithms. However, self-stabilizing algorithms for

various problems have been proposed recently [26, 18, 17, 29, 28, 24, 7]. Practical applications include, for example, Internet servers [33] and FDDI media access control protocols [12].

The *time complexity* of a self-stabilizing algorithm is the maximal number of steps required to reach a legitimate configuration from an illegitimate one. The improvement of time complexity and the reduction of required memory are the main performance issues in the field of self-stabilizing systems [1, 2, 4, 22]. The improvement of time complexity is particularly important because being in illegitimate configurations can lead to malfunctions. To improve time complexity, it is important to be able to compute it for any given self-stabilizing algorithm.

However, there are self-stabilizing algorithms whose time complexity is difficult to analyze [3]. For example, the upper bound for the algorithm proposed in [13] was presented in [8]. This upper bound was much greater than the time complexity computed in [37, 30].

In this dissertation, we propose a method to automatically compute time complexity on the basis of symbolic model checking. On addition, we analyze time complexity based on the behavior of the worst case, which is obtained by the symbolic model checking tool NuSMV.

1.2 Main Results

1.2.1 On Time Complexity of Dijkstra's Self-Stabilizing Three-State Mutual Exclusion Algorithm

As for the first contribution, we propose a method to compute the time complexity of Dijkstra's self-stabilizing three-state mutual exclusion algorithm, and provide a very tight lower bound for time complexity.

Dijkstra's three-state mutual exclusion algorithm is one of the first self-stabilizing algorithms [13]. Although more than 30 years have passed since its invention, the exact worst-case time complexity of this algorithm is still unknown. The best-known lower bound on the worst-case time complexity was given by Chernoy, Shalon, and Zaks [9]. They proved a lower bound of $1\frac{5}{6}n^2 - O(n)$ by showing that there is an execution of length $1\frac{5}{6}n^2 - 10\frac{1}{6}n + 14$ when $n = 3k$, where k is a natural number.

In this dissertation we give a lower bound on the worst-case time complexity, which matches the known best bound $1\frac{5}{6}n^2 - O(n)$ [9] but is more accurate. This bound is given by showing a very long computation derived by analyzing the behavior of the worst case behavior of the algorithm with some processes. The behavior is obtained by model checking. On addtion, our result applies when $n = 3k + 1$ and $n = 3k + 2$. For the reason explained in chapter 3, we conjecture that the new bound is the exact worst-case time complexity.

1.2.2 On Time Complexity of Hsu and Huang's Self-Stabilizing Maximal Matching Algorithm

As for the second contribution, we propose a method to compute the time complexity of Hsu and Huang's self-stabilizing maximal matching algorithm, and give the exact time complexity.

This algorithm is the first self-stabilizing maximal matching algorithm and has been regularly cited in the literature. Because of its technical importance, the time complexity of this particular algorithm has been well studied. In [25], Hsu and Huang show that it is bounded by $O(n^3)$, where n is the number of nodes. In [34], Tel provides an almost tight upper bound, which is $\frac{1}{2}n^2 + 2n + 1$ if n is even and $\frac{1}{2}n^2+n-\frac{1}{2}$ if n is odd. In [35], Tel gives a more concise proof for the $O(n^2)$ bound than [34]. In [23], Hedetniemi, Jacobs, and Srimani provide an upper bound of $2|E| + n$, where $|E|$ is the number of edges. This gives a better bound than the one in [34] when $|E| = O(n)$.

In this dissertation, we provide the exact time complexity of the Hsu–Huang algorithm. The fact that the known upper bound is very similar to the time complexity computed by model checking is helpful for us to find the exact time complexity.

1.2.3 New Fast Self-Stabilizing Maximal Matching Algorithm

As for the third contribution, we propose a new self-stabilizing maximal matching algorithm. The proposed algorithm assumes the same model as the Hsu–Huang algorithm and runs faster. In particular, the new algorithm reduces the worst-case time complexity by approximately half. Through the design of a new algo-

rithm, model checking is very useful for verifying whether a new algorithm is self-stabilizing.

1.3 Overview of Dissertation

The remainder of this dissertation is organized as follows. In chapter 2, we describe self-stabilizing algorithms and model checking. We describe the first contribution in chapter 3 , entitled "On Time Complexity of Dijkstra's Self-Stabilizing Three-State Mutual Exclusion Algorithm." In this chapter, we prove a lower bound of the time complexity of the algorithm by showing that there is a very long execution . We describe the second contribution in chapter 4, entitled "On Time Complexity of Hsu and Huang's Self-Stabilizing Maximal Matching Algorithm." In this chapter, we prove the exact time complexity of the algorithm. We describe the third contribution in chapter 5, entitled "New Fast Self-Stabilizing Maximal Matching Algorithm." In this chapter, we propose a new self-stabilizing maximal matching algorithm that is faster than the existing algorithm. In chapter 6, we summarize this dissertation.

Chapter 2

Preliminaries

2.1 Self-Stabilizing Algorithms

We consider a distributed system that consists of n processes, $p_0, p_1, \cdots, p_{n-1}$. The topology of the system is modeled by an undirected graph each of whose vertices correspond to a process.

A process is a finite state machine. A process p_i is defined as a three-tuple $M_i = (Q_i, \Sigma_i, \delta_i)$:

- Q_i is a finite set of states.
- Σ_i is a finite state set of p_i's neighbors.
- $\delta_i : Q_i \times \Sigma_i \rightarrow Q_i$ is a state-transition function.

We say that process p_i and process p_j are neighbors if p_i is adjacent to p_j on the graph. A process can communicate with its neighbors, and Σ_i is defined as the Cartesian product of the states of its neighbors. For example, when process p_j

and process p_k are neighbors of process p_i, Σ_i is :

$$\Sigma_i = Q_j \times Q_k \ .$$

A state transition function δ_i is described in the guarded command language [14]. In the language, δ_i is described as a list of actions:

$$\delta_i = \text{if} < action > [] \cdots [] < action > \text{fi} \ .$$

The symbol "[]" separates the different actions. Each action is described as follows:

$$< action > ::= < guard > \Rightarrow < statement > \ .$$

The guard is a Boolean expression over the states of process p_i and its neighbors. When the guard is satisfied, process p_i updates the state according to the statement. When more than one guard is satisfied, process p_i updates the state by the statement, which is non-deterministically selected .

A distributed algorithm specifies δ_i for each process p_i. In each step of the execution of process p_i, p_i reads the states of its neighbors, and updates the state by δ_i.

The global state (or configuration) of a system is the vector of the states of all of its processes. Therefore, the set of configurations G is given as follow:

$$G = Q_0 \times Q_1 \times \cdots \times Q_{n-1} \ .$$

We say that an action is enabled at a configuration if and only if the guard holds

at that configuration. A process is enabled if and only if at least one action of the process is enabled. We assume that, in each step, exactly one enabled process is selected and it updates the state.

We denote by $g \rightarrow g'$ $(g, g' \in G)$ the fact that there is a process that is enabled at g and its execution yields g'. A sequence of configurations $g_0 g_1 g_2 \cdots g_k$ is a computation if and only if for every $i \geq 0$ $g_i \rightarrow g_{i+1}$ holds.

Let P be a predicate that identifies the desirable configurations of the system. We assume that P is a Boolean expression over the states of all the processes of the system. We say that a configuration is legitimate if and only if the configuration satisfies P. Let L denote the set of the legitimate states.

A distributed algorithm is a self-stabilizing algorithm if it satisfies the following two properties:

1. Convergence — For any configuration $g_0 \in G$, and any computation $g_0 g_1 \cdots g_k$ that starts with g_0, there is an integer k (≥ 0) such that $g_k \in L$.

2. Closure — For any configuration $g \in L$, $g \rightarrow g'$ implies $g' \in L$.

The convergence time ct of a computation $g_0 g_1 \cdots g_k$ is the number of steps required for reaching a legitimate state from g_0. If $g_0 g_1 \cdots g_k$ is a computation of a self-stabilizing algorithm, ct is defined as follows:

$$ct = \begin{cases} 0 & g_0 \in L \\ i \text{ such that } g_{i-1} \notin L \wedge g_i \in L & g_0 \notin L\,. \end{cases} \tag{2.1}$$

The time complexity r of a self-stabilizing algorithm is the convergence time in the worst case:

$$r = \max_{\forall computation\ C} \{ct\ for\ C\}\ . \tag{2.2}$$

2.2 Symbolic Model Checking

Model checking is an automatic technique for verifying finite-state concurrent systems [11]. In model checking, the system is modeled as a Kripke structure.

Let AP be a set of atomic propositions. A Kripke structure M over AP is a three-tuple $M = (S, R, L)$ where

1. S is a finite set of states.
2. $R \subseteq S \times S$ is a transition relation that must be total, that is, for every state $s \in S$, there is a state $s' \in S$ such that $(s, s') \in R$.
3. $L : S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.

We say that an infinite sequence of states $\pi = s_0 s_1 \cdots$ is a path in the structure M from a state s if $s_0 = s$ and $(s_i, s_{i+1}) \in R$ holds for all $i \geq 0$.

In symbolic model checking, a Kripke structure is represented by Ordered Binary Decision Diagrams (OBDDs). With this data structure, the memory and time required to verify the system can be dramatically reduced.

2.2.1 Computational Tree Logic

To verify a system described as a Kripke structure, we need to specify the properties that should be satisfied on the structure. To describe the properties, we usually use the computational tree logic (CTL).

The syntax of a CTL formula is given by the following rules:

- If $f \in AP$, then f is a CTL formula.
- If f and g are CTL formulas, then $\neg f$, $f \wedge g$ and $f \vee g$ are CTL formulas.
- If f is a CTL formula, then **AX** f is a CTL formula.
- If f is a CTL formula, then **EX** f is a CTL formula.
- If f is a CTL formula, then **AF** f is a CTL formula.
- If f is a CTL formula, then **EF** f is a CTL formula.
- If f is a CTL formula, then **AG** f is a CTL formula.
- If f is a CTL formula, then **EG** f is a CTL formula.
- If f and g are CTL formulae, then **A**(f **U** g) is a CTL formula.
- If f and g are CTL formulae, then **E**(f **U** g) is a CTL formula.
- If f and g are CTL formulae, then **A**(f **R** g) is a CTL formula.
- If f and g are CTL formulae, then **E**(f **R** g) is a CTL formula.

Here, **AX**, **EX**, **AF**, **EF**, **AG**, **EG**, **AU**, **EU**, **AR**, and **ER** are composed of two components: path quantifiers and temporal operators. The path quantifiers are **A** and **E**. Quantifier **A** means "for all paths," and quantifier **E** means "for some paths." These quantifiers are used to specify that all of the paths or some of the paths starting from a particular state have some property. The temporal operators **X**, **F**, **G**, **U**, and **R** describe the properties of a path.

- **X** ("neXt time") requires that a property holds in the second state on the path.
- **F** ("in the Future") requires that a property will hold at some state of the path.
- **G** ("Globally") requires that a property holds at every state of the path.
- **U** ("Until") requires that if there is a state on the path where the second property holds, then on every preceding state of the path, the first property holds.
- **R** ("Release") is the logical dual of the U.

The semantics of CTL with respect to a Kripke structure M is defined as follows. Let π^i be the suffix of π starting at s_i. If f is a CTL formula, $M, s \models f$ means that f holds at state s in the Kripke structure M, and $M, \pi \models f$ means that f holds at the first state of π in the structure M. Similarly, $M \models f$ means that f holds at every state in the Kripke structure M. When the Kripke structure M is clear from the context, we will usually omit it. The relation $\models$ is defined inductively as follows (assuming that ap is an atomic proposition and f, g are CTL formulae) :

1. $M, s \models ap \Leftrightarrow ap \in L(s)$.
2. $M, s \models \neg f \Leftrightarrow M, s \not\models f$.
3. $M, s \models f \wedge g \Leftrightarrow M, s \models f$ and $M, s \models g$.
4. $M, s \models f \vee g \Leftrightarrow M, s \models f$ or $M, s \models g$.

5. $M, \pi \models f \Leftrightarrow s$ is the first state of π and $M, s \models f$.

6. $M, s \models \mathbf{EX}\ f \Leftrightarrow$ there is a path π from s such that $M, \pi^1 \models f$.

7. $M, s \models \mathbf{AX}\ f \Leftrightarrow$ for every path π starting from s, $M, \pi^1 \models f$.

8. $M, s \models \mathbf{EF}\ f \Leftrightarrow$ there is a path π from s and a $k \geq 0$ such that $M, \pi^k \models f$.

9. $M, s \models \mathbf{AF}\ f \Leftrightarrow$ for every path π starting from s, there is a $k \geq 0$ such that $M, \pi^k \models f$.

10. $M, s \models \mathbf{EG}\ f \Leftrightarrow$ there is a path π from s such that for all $i \geq 0$, $M, \pi^i \models f$.

11. $M, s \models \mathbf{AG}\ f \Leftrightarrow$ for every path π starting from s and for all $i \geq 0$, $M, \pi^i \models f$.

12. $M, s \models \mathbf{E}(f\ \mathbf{U}\ g) \Leftrightarrow$ there is a path π from s and there exists a $k \geq 0$ such that $M, \pi^k \models g$ and for all $0 \leq j < k$, $M, \pi^j \models f$.

13. $M, s \models \mathbf{A}(f\ \mathbf{U}\ g) \Leftrightarrow$ for every path π starting from s and there exists a $k \geq 0$ such that $M, \pi^k \models g$ and for all $0 \leq j < k$, $M, \pi^j \models f$.

14. $M, s \models \mathbf{E}(f\ \mathbf{R}\ g) \Leftrightarrow$ there is a path π from s such that for all $j \geq 0$, if for every $i < j$ $M, \pi^i \not\models f$ then $M, \pi^j \models g$.

15. $M, s \models \mathbf{A}(f\ \mathbf{R}\ g) \Leftrightarrow$ for every path π starting from s, for all $j \geq 0$, if for every $i < j$ $M, \pi^i \not\models f$ then $M, \pi^j \models g$.

2.2.2 Real-Time CTL

With CTL, we can describe a property. For example, the property that an atomic proposition ap will eventually hold for any path can be stated as $AFap$. However,

we occasionally need to describe a property such as that an atomic proposition ap will hold in 50 steps. Such properties are needed, for example, to verify network communication protocols or embedded real-time control systems.

To describe such properties, E. A. Emerson et al. augmented CTL to *Real-Time CTL* (RTCTL)[16]. In RTCTL, we can describe a property that holds in a bounded number k of steps. To specify a bound k, we use a notation such as $\mathbf{AF}^{\leq k}$.

Here, some RTCTL operators are simply abbreviations of other RTCTL operators.

- $\mathbf{AF}^{\leq k}\, f \equiv \mathbf{A}(true\, \mathbf{U}^{\leq k}\, f)$.
- $\mathbf{EF}^{\leq k}\, f \equiv \mathbf{E}(true\, \mathbf{U}^{\leq k}\, f)$.
- $\mathbf{AG}^{\leq k}\, f \equiv \neg\mathbf{EF}^{\leq k}\neg f$.
- $\mathbf{EG}^{\leq k}\, f \equiv \neg\mathbf{AF}^{\leq k}\neg f$.
- $\mathbf{A}(f\, \mathbf{R}^{\leq k}\, g) \equiv \neg\mathbf{E}(\neg f\, \mathbf{U}^{\leq k}\neg g)$.
- $\mathbf{E}(f\, \mathbf{R}^{\leq k}\, g) \equiv \neg\mathbf{F}(\neg f\, \mathbf{U}^{\leq k}\neg g)$.

AX and **EX** already specify the exact number of steps when the property should be hold (which is 1), so we need not define their RTCTL versions. As a result, we only need to define the semantics of $\mathbf{A}(f\, \mathbf{U}^{\leq k}\, g)$ and $\mathbf{E}(f\, \mathbf{U}^{\leq k}\, g)$.

- $M, s \models \mathbf{E}(f\, \mathbf{U}^{\leq k}\, g) \Leftrightarrow$ there is a path π from s and there exists a $0 \leq i \leq k$ such that $M, \pi^i \models g$ and for all $0 \leq j \leq i$, $M, \pi^j \models f$.
- $M, s \models \mathbf{A}(f\, U^{\leq k} g) \Leftrightarrow$ for every path π starting from s and there exists a $0 \leq i \leq k$ such that $M, \pi^i \models g$ and for all $0 \leq j \leq i$, $M, \pi^j \models f$.

2.2.3 NuSMV

NuSMV is a software tool for symbolic model checking that can support RTCTL. In NuSMV, a verified system or algorithm is described in a special language, the NuSMV language.

The description of a system in the NuSMV language is called a NuSMV program. A NuSMV program is composed of one or more modules, each of which specifies a finite state machine , and there must be one module with the name main.

Each module contains variable declarations that determine its state space, the initial state and the state transition function of the machine.

Variable declarations start with the keyword VAR, and are composed of variables and their types. The type of a variable can be Boolean, an enumerated type, or a user-defined module. An example of variable declarations is as follows:

```
VAR
  flag : boolean;
  enum : { a, b, c };
  user : A;

MODULE A
...
```

Initial states and the state transition function are described as a collection of parallel assignments to a variable. The execution of an assignment updates the value of a variable. Assignments start with the keyword ASSIGN.

Initial states are assigned by specifying the initial values of the variables by using expression $init(x)$, where x is a variable. The expression $next(x)$ is used to specify a value assigned to the variable in the next state. For conditional assignments, a case expression is used.

An example of assignments is as follows.

```
init(x) := 0;
next(x) := x + 1;
next(y) := case
  condition1 : expression1;
  condition2 : expression2;
  condition3 : expression3;
  ...
  1          : y;
esac;
```

Noted that the case expression is evaluated as the first right-hand side expression whose corresponding left-hand side condition holds.

Chapter 3

On Time Complexity of Dijkstra's Three-State Mutual Exclusion Algorithm

3.1 Introduction

In this chapter, we prove a lower bound of the time complexity of Dijkstra's self-stabilizing three-state mutual exclusion algorithm by showing that there is a very long computation.

This algorithm is one of the first self-stabilizing algorithms [13]. Although more than 30 years have passed since its invention, the exact worst-case time complexity of this algorithm is still unknown. The best-known lower bound on the worst-case time complexity was given by Chernoy, Shalon, and Zaks [9]. They proved the lower bound of $1\frac{5}{6}n^2 - O(n)$ by showing that there is an computation of length $1\frac{5}{6}n^2 - 10\frac{1}{6}n + 14$ when $n = 3k$, where k is a natural number.

In this dissertation we give a lower bound on the worst-case time complexity, which matches the best-known bound $1\frac{5}{6}n^2 - O(n)$ [9] but is more accurate. On addition, our result applies when $n = 3k + 1$ and $n = 3k + 2$.

The remainder of this chapter is organized as follows. In Section 3.2, we describe the system and the algorithm. In Section 3.3, we prove a lower bound of time complexity. In Section 3.4, we show a method to compute the time complexity using the model checking tool NuSMV. In Section 3.5, we summarize this chapter.

3.2 Algorithm

We consider a system consisting of n processes $p_0, p_1, \cdots, p_{n-1}$ that are arranged in a ring. Process $p_i, (0 \leq i \leq n-1)$ is adjacent to $p_{(i-1) \bmod n}$ and $p_{(i+1) \bmod n}$. Process p_i has a local state $x_i \in \{0, 1, 2\}$ and can read the state of its adjacent processes. A **configuration** is an n-tuple of process states $(x_0, x_1, \cdots, x_{n-1})$ $(\in \{0, 1, 2\}^n)$. Dijkstra's three-state mutual exclusion algorithm is described as follows (addition and subtraction are modulo 3):

Process p_0 (called bottom):

if $x_0 + 1 = x_1 \Rightarrow x_0 := x_0 + 2$ **fi**

Process $p_i, 1 \leq i \leq n-2$ (called other):

if

$x_i + 1 = x_{i-1} \Rightarrow x_i := x_i + 1$ []

$x_i + 1 = x_{i+1} \Rightarrow x_i := x_i + 1$

fi

Process p_{n-1} (called top):

Table 3.1: The algorithm in a tabular form (g' is the next configuration to g, i.e., $C \rightarrow C'$)

Type	Process	g	g'
0	p_0	$x_0 < x_1$	$x_0 > x_1$
1	p_i	$x_{i-1} > x_i = x_{i+1}$	$x_{i-1} = x_i > x_{i+1}$
2	p_i	$x_{i-1} = x_i < x_{i+1}$	$x_{i-1} < x_i = x_{i+1}$
3	p_i	$x_{i-1} > x_i < x_{i+1}$	$x_{i-1} = x_i = x_{i+1}$
4	p_i	$x_{i-1} > x_i > x_{i+1}$	$x_{i-1} = x_i < x_{i+1}$
5	p_i	$x_{i-1} < x_i < x_{i+1}$	$x_{i-1} > x_i = x_{i+1}$
6	p_{n-1}	$x_{n-2} > x_{n-1} < x_0$	$x_{n-2} < x_{n-1} > x_0$
7	p_{n-1}	$x_{n-2} = x_{n-1} = x_0$	$x_{n-2} < x_{n-1} > x_0$

if

$x_{n-2} = x_{n-1} = x_0 \Rightarrow x_{n-1} := x_{n-2} + 1$ []

$x_{n-2} = x_{n-1} + 1 = x_0 \Rightarrow x_{n-1} := x_{n-2} + 1$

fi

A process is **enabled** if the **if** condition is true. As described in chapter 2, the algorithm runs in steps. In each step, exactly one enabled process executes the statement of the algorithm, resulting in a new configuration. We write $g \in G \rightsquigarrow g' \in G$ if there is a computation that starts with g and leads to g'. Given a computation $g_0 g_1 \cdots g_l$, a **schedule** is a sequence of processes $p_1 p_2 \cdots p_l$ such that for any $i, 1 \leq i \leq l$, p_i is enabled in g_{i-1} and the execution of the statement by p_i in g_{i-1} yields g_i.

Since this algorithm is intended to ensure mutual exclusion, a configuration is **legitimate** if exactly one process is enabled [13]. A configuration is **illegitimate** if it is not legitimate.

Proposition 1. [15] Dijkstra's three-state mutual exclusion algorithm is self-stabilizing,

that is, the following two conditions hold:

- A legitimate configuration occurs in any computation starting with any configuration.
- If a configuration g is legitimate, then any configuration g' such that $g \rightarrow g'$ is legitimate.

The worst-case time complexity (or **stabilization time** in some literature) of the algorithm is the maximum number of steps executed until a legitimate state is reached. Formally, the worst-case time complexity is the length of the longest computation $g_0 g_1 \cdots g_l$ such that g_i is illegitimate for any $i, 0 \leq i < l$ and g_l is legitimate. Let $T(n)$ denote the worst-case time complexity of the algorithm. When n is fixed, a number $LB(n)$ is a lower bound on the worst-case time complexity if $LB(n) \leq T(n)$.

3.3 Lower Bound

Our proof of a lower bound is relatively direct; We show some extremely lengthy computations where only the very last configuration is legitimate. Then we obtain the length of these computations. By definition, the worst-case time complexity is greater than or at least equal to that length, thus, the length of these computations is a lower bound on the worst-case time complexity.

Our results apply when $n \geq 9$. There are three cases to consider: (1) $n = 3k$; (2) $n = 3k + 1$; and (3) $n = 3k + 2$, where k is a natural number. For each of these cases, we provide a long computation that comprises three parts. First we show the results for Case (1) and then proceed to the other two cases.

To concise the proofs concise, we use the same notations as [9]. Notation $x_{i-1} < x_i$ means $x_i = (x_{i-1} + 1) \bmod 3$, while $x_{i-1} > x_i$ means $x_i = (x_{i-1} - 1) \bmod 3$. For example, configuration $(1, 1, 0, 1, 2, 2, 0)$ is represented as $1 = 1 > 0 < 1 < 2 = 2 < 0$. With these notations, the algorithm is represented as a collection of eight types of moves (types 0 to 7), as shown in Table 3.1. Regular expressions over $\{<, >, =\}$ are used to denote configurations. For example, $[=><^2=<]$ is a possible notation for $(1, 1, 0, 1, 2, 2, 0)$.

Lemma 1. When $n = 3k, n \geq 6$, there is an computation of length $n + 3$ from $[<><^{n-3}]$ to $[==<^{n-3}]$.

Proof. We show the existence of schedule $\underbrace{p_3}_{type\ 5}\underbrace{p_4p_5\cdots p_{n-2}}_{2}\underbrace{p_{n-1}}_{7}\underbrace{p_0}_{0}\underbrace{p_1}_{4}\underbrace{p_1}_{2}\underbrace{p_0}_{0}\underbrace{p_1}_{1}\underbrace{p_2}_{4}$.

$[<><^{n-3}]$,

$[<>>=<^{n-5}]$, after 1 step of type 5:

$[<>><^{n-5}=]$, after $n-5$ steps of type 2:

$[<>><^{n-4}]$, after 1 step of type 7

(Note that $x_{n-2} = x_{n-1} = x_0$ in the previous configuration because $n = 3k$):

$[>>><^{n-4}]$, after 1 step of type 0:

$[=<><^{n-4}]$, after 1 step of type 4:

$[<=><^{n-4}]$, after 1 step of type 2:

$[>=><^{n-4}]$, after 1 step of type 0:

$[=>><^{n-4}]$, after 1 step of type 1:

$[==<^{n-3}]$, after 1 step of type 4. □

Lemma 2. When $n \geq 9$, $2 \leq k \leq n - 6$, and $(n - k - 1) \bmod 3 = 0$, there is a computation of length $n + 9k + 10$ from $[=^k<^{n-k-1}]$ to $[=^{k+3}<^{n-k-4}]$.

Proof. We show the existence of schedule $\underbrace{p_k p_{k+1} \cdots p_{n-2}}_{\text{type 2}}\ \underbrace{p_{n-1}}_{7}\ \underbrace{p_{k-1}p_{k-2}\cdots p_1}_{2}$

$\underbrace{p_k p_{k-1}\cdots p_2}_{2}\ \underbrace{p_{k+1}p_k\cdots p_3}_{2}\ \underbrace{p_0}_{0}\ \underbrace{p_2}_{5}\ \underbrace{p_2p_3\cdots p_{k+1}}_{1}\ \underbrace{p_1p_2\cdots p_k}_{1}\ \underbrace{p_{k+1}}_{4}\ \underbrace{p_{k+1}p_k\cdots p_1}_{2}$

$\underbrace{p_{k+2}p_{k+1}\cdots p_2}_{2}\ \underbrace{p_{k+3}p_{k+2}\cdots p_3}_{2}\ \underbrace{p_0}_{0}\ \underbrace{p_2}_{5}\ \underbrace{p_2p_3\cdots p_{k+3}}_{1}\ \underbrace{p_1p_2\cdots p_{k+2}}_{1}\ \underbrace{p_{k+3}}_{4}$.

$[=^k<^{n-k-1}]$,

$[=^{k-1}<^{n-k-1}=]$, after $n-k-1$ steps of type 2:

$[=^{k-1}<^{n-k}]$, after 1 step of type 7.

(Because that $x_{n-2} = x_{n-1} = x_0$ in the previous configuration

since $(n-k-1) \bmod 3 = 0$):

$[<=^{k-1}<^{n-k-1}]$, after $k-1$ steps of type 2:

$[<<=^{k-1}<^{n-k-2}]$, after $k-1$ steps of type 2:

$[<<<=^{k-1}<^{n-k-3}]$, after $k-1$ steps of type 2:

$[><<=^{k-1}<^{n-k-3}]$, after 1 step of type 0:

$[>>=^{k}<^{n-k-3}]$, after 1 step of type 5:

$[>=^{k}><^{n-k-3}]$, after k steps of type 1:

$[=^{k}>><^{n-k-3}]$, after k steps of type 1:

$[=^{k+1}<^{n-k-2}]$, after 1 step of type 4:

$[<=^{k+1}<^{n-k-3}]$, after $k+1$ steps of type 2:

$[<<=^{k+1}<^{n-k-4}]$, after $k+1$ steps of type 2:

$[<<<=^{k+1}<^{n-k-5}]$, after $k+1$ steps of type 2:

$[><<=^{k+1}<^{n-k-5}]$, after 1 step of type 0:

$[>>=^{k+2}<^{n-k-5}]$, after 1 step of type 5:

$[>=^{k+2}><^{n-k-5}]$, after $k+2$ steps of type 1:

$[=^{k+2}>><^{n-k-5}]$, after $k+2$ steps of type 1:

$[=^{k+3}<^{n-k-4}]$, after 1 step of type 4. □

Lemma 3. When $n \geq 6$, there is a computation of length $10n-30$ from $[=^{n-4}<<<]$ to $[=^{n-4}>=>]$.

Proof. We show the existence of schedule $\underbrace{p_{n-4}p_{n-3}p_{n-2}}_{\text{type 2}}\ \underbrace{p_{n-1}}_{7}\ \underbrace{p_{n-5}p_{n-6}\cdots p_1}_{2}$

$\underbrace{p_{n-4}p_{n-5}\cdots p_2}_{2}\ \underbrace{p_{n-3}p_{n-4}\cdots p_3}_{2}\ \underbrace{p_0}_{0}\ \underbrace{p_2}_{5}\ \underbrace{p_2p_3\cdots p_{n-3}}_{1}\ \underbrace{p_1p_3\cdots p_{n-4}}_{1}\ \underbrace{p_{n-3}}_{4}\ \underbrace{p_{n-3}p_{n-4}\cdots p_1}_{2}$

$\underbrace{p_{n-4}p_{n-5}\cdots p_2}_{2}\ \underbrace{p_0}_{0}\ \underbrace{p_{n-1}}_{7}\ \underbrace{p_{n-2}p_{n-3}\cdots p_3}_{2}\ \underbrace{p_2}_{5}$

$\underbrace{p_2p_3\cdots p_{n-3}}_{1}\ \underbrace{p_1p_2\cdots p_{n-4}}_{1}\ \underbrace{p_{n-2}}_{1}$.

$[=^{n-4}<<<]$,

$[=^{n-5}<<<=]$, after 3 steps of type 2:

$[=^{n-5}<<<<]$, after 1 step of type 7:

$[<=^{n-5}<<<]$, after $n-5$ steps of type 2:

$[<<=^{n-5}<<]$, after $n-5$ steps of type 2:

$[<<<=^{n-5}<]$, after $n-5$ steps of type 2:

$[><<=^{n-5}<]$, after 1 step of type 0:

$[>>=^{n-4}<]$, after 1 step of type 5:

$[>=^{n-4}><]$, after $n-4$ steps of type 1:

$[=^{n-4}>><]$, after $n-4$ steps of type 1:

$[=^{n-3}<<]$, after 1 step of type 4:

$[<=^{n-3}<]$, after $n-3$ step of type 2:

$[<<=^{n-3}]$, after $n-3$ step of type 2:

$[><=^{n-3}]$, after 1 step of type 0:

$[><=^{n-4}<]$, after 1 step of type 7:

$[><<=^{n-4}]$, after $n-4$ steps of type 2:

$[>>=^{n-3}]$, after 1 step of type 5:

$[>=^{n-4}>=]$, after $n-4$ steps of type 1:

$[=^{n-4}>>=]$, after $n-4$ steps of type 1:

$[=^{n-4}>=>]$, after one step of type 1. □

Theorem 1. When $n = 3k \geq 9$, we have:

$$1\frac{5}{6}n^2 - 4\frac{1}{6}n - 2 \leq T(n) .$$

Proof. By Lemmas 1, 2, and 3, there is a computation such that

(i) it is represented as: $[<><^{n-3}] \rightsquigarrow [=^2<^{n-3}] \rightsquigarrow [=^5<^{n-6}] \rightsquigarrow \cdots \rightsquigarrow [=^{n-4}<^3] \rightsquigarrow [=^{n-4}>>=] \rightarrow [=^{n-4}>=>]$, and

(ii) the length is

$$\begin{aligned} & n + 3 + \sum_{i=1}^{m-2}(n + 9(3i-1) + 10) + 10n - 30 \\ &= 1\frac{5}{6}n^2 - 4\frac{1}{6}n - 2 \end{aligned} .$$

The final configuration of the computation, that is, $[=^{n-4}>=>]$, is legitimate, because only p_{n-3} is enabled. Now, consider the immediate predecessor configuration to the final configuration, that is, the $(1\frac{5}{6}n^2 - 4\frac{1}{6}n - 3)$−th configuration. This configuration, represented as $[=^{n-4}>>=]$, is not a legitimate configuration because p_{n-3} and p_{n-2} are both enabled.

From (ii) of Proposition 1, if a legitimate configuration occurs in a computation, then all successor configurations in the computation must be legitimate. Because the $(1\frac{5}{6}n^2 - 4\frac{1}{6}n - 3)$−th configuration is illegitimate, every configuration in the computation, except for the final configuration, is illegitimate. Therefore the worst-case time complexity is greater than or at least equal to the length of the computation. □

For the case $n = 3k + 1$ (Case (2)) and the case $n = 3k + 2$ (Case (3)), a bound is obtained in almost the same manner, except that Lemma 1 is replaced with Lemma 4 and Lemma 5, respectively.

Lemma 4. When $n = 3k + 1 \geq 7$, there is a computation of length $n + 10$ from $[<>>><^{n-5}]$ to $[===<^{n-4}]$.

Proof. One such computation is $[<>>><^{n-5}] \rightarrow [<>=<^{n-4}] \rightsquigarrow [<><^{n-4}=]$
$\rightarrow [<><^{n-3}] \rightarrow$
$[>><^{n-3}] \rightarrow [=<^{n-2}] \rightarrow [<=<^{n-3}] \rightarrow [>=<^{n-3}] \rightsquigarrow [><<=<^{n-5}] \rightarrow [>>==<^{n-5}] \rightsquigarrow$
$[>==><^{n-5}] \rightsquigarrow [==>><^{n-5}] \rightarrow [===<^{n-4}]$. The corresponding schedule is

$$\underbrace{p_3}_{type\ 4}\ \underbrace{p_3p_4\cdots p_{n-2}}_{2}\ \underbrace{p_{n-1}}_{7}\ \underbrace{p_0}_{0}\ \underbrace{p_1}_{4}\ \underbrace{p_1}_{2}\ \underbrace{p_0}_{0}\ \underbrace{p_2p_3}_{2}\ \underbrace{p_2}_{5}\ \underbrace{p_2p_3}_{1}\ \underbrace{p_1p_2}_{1}\ \underbrace{p_3}_{4}.$$ □

Lemma 5. When $n = 3k + 2 > 8$, there is a computation of length $2n + 11$ from $[<>><^{n-4}]$ to $[====<^{n-5}]$.

Proof. One such computation is $[<>><^{n-4}] \rightarrow$
$[<>>>=<^{n-6}] \rightsquigarrow [<>>><^{n-6}=] \rightarrow$
$[<>>><^{n-5}] \rightarrow [>>><^{n-5}] \rightarrow [=<>><^{n-5}] \rightarrow [<=>><^{n-5}] \rightarrow [>=>><^{n-5}] \rightarrow$
$[>=>>>=<^{n-7}] \rightsquigarrow [>=>>><^{n-7}=] \rightarrow$
$[>=>>><^{n-6}] \rightarrow [=>>>><^{n-6}] \rightarrow$
$[==<>><^{n-6}] \rightsquigarrow [<==>><^{n-6}] \rightarrow$
$[>==>><^{n-6}] \rightsquigarrow [==>>><^{n-6}] \rightarrow$
$[===<><^{n-6}] \rightsquigarrow [<===><^{n-6}] \rightarrow$
$[>===><^{n-6}] \rightsquigarrow [===>><^{n-6}] \rightarrow [====<^{n-5}]$. The corresponding schedule is $\underbrace{p_4}_{type\ 5}\ \underbrace{p_5p_6p_7\cdots p_{n-2}}_{2}\ \underbrace{p_{n-1}}_{7}\ \underbrace{p_0}_{0}\ \underbrace{p_1}_{4}\ \underbrace{p_1}_{2}\ \underbrace{p_0}_{0}\ \underbrace{p_5}_{5}\ \underbrace{p_6p_7p_8\cdots p_{n-2}}_{2}\ \underbrace{p_{n-1}}_{7}$
$\underbrace{p_1}_{1}\ \underbrace{p_2}_{4}\ \underbrace{p_2p_1}_{2}\ \underbrace{p_0}_{0}\ \underbrace{p_1p_2}_{1}\ \underbrace{p_3}_{4}\ \underbrace{p_3p_2p_1}_{2}\ \underbrace{p_0}_{0}\ \underbrace{p_1p_2p_3}_{1}\ \underbrace{p_4}_{4}.$ □

Theorem 2. When $n = 3k + 1, n \geq 10$, we have:

$$1\frac{5}{6}n^2 - 4\frac{1}{2}n - 1\frac{1}{3} \leq T(n) .$$

Proof. Consider a computation $[<>>><^{n-5}] \rightsquigarrow [=^3<^{n-4}] \rightsquigarrow [=^6<^{n-7}] \rightsquigarrow \cdots \rightsquigarrow [=^{n-4}<^3] \rightsquigarrow [=^{n-4}>>=] \rightarrow [=^{n-4}>=>]$. By Lemmas 4, 2 and 3, this computation indeed exists and its length is

$$\begin{aligned} & n + 10 + \sum_{i=1}^{m-2}(n + 9 \cdot 3i + 10) + 10n - 30 \\ &= 1\frac{5}{6}n^2 - 4\frac{1}{2}n - 1\frac{1}{3} \end{aligned} .$$

The final configuration $[=^{n-4}>=>]$ is legitimate because only p_{n-3} is enabled. On the other hand, its immediate predecessor configuration $[=^{n-4}>>=]$ is illegitimate, because p_{n-3} and p_{n-2} are both enabled. Hence, from (ii) of Proposition 1, every configuration in the computation, except for the final configuration, is illegitimate. Therefore, the worst-case time complexity is greater than or at least equal to the length of the computation. □

Theorem 3. When $n = 3k + 2, n \geq 11$, we have

$$1\frac{5}{6}n^2 - 3\frac{5}{6}n - 9\frac{2}{3} \leq T(n) .$$

Proof. Consider a computation $[<>><^{n-4}] \rightsquigarrow$
$[=^4<^{n-5}] \rightsquigarrow [=^7<^{n-8}] \rightsquigarrow \cdots \rightsquigarrow [=^{n-4}<^3] \rightsquigarrow [=^{n-4}>>=] \rightarrow [=^{n-4}>=>]$. By

Lemmas 5, 2, and 3, this computation indeed exists and its length is

$$2n + 11 + \sum_{i=1}^{m-2}(n + 9(3i + 1) + 10) + 10n - 30$$
$$= 1\frac{5}{6}n^2 - 3\frac{5}{6}n - 9\frac{2}{3}$$

The final configuration $[=^{n-4}>=>]$ is legitimate, while its immediate predecessor configuration $[=^{n-4}>>=]$ is illegitimate. By the same argument as the proof of Theorems 1 and 2, every configuration in the computation, except for the final configuration, is illegitimate. Thus, the worst-case time complexity is greater than or at least equal to the length of the computation. □

3.4 Computing of Time Complexity Using NuSMV

The use of model checking for analyzing self-stabilizing algorithms was studied in [36, 37]. We use these studies with some modifications to derive the worst-case computations .

Here we explain how to translate a distributed algorithm written in the guarded command language into the NuSMV program, and how to calculate the time complexity. Figures 3.1, 3.2, 3.3, and 3.4 show the NuSMV program that represents the three-state algorithm where $n = 4$.

3.4.1 Communication among processes

The communication among processes is described in the main module (Figure 3.1). The main module declares four processes: p_0, p_1, p_2, and p_3. The behavior of p_0,

p_i $(0 < i < n-1)$, and p_{n-1} are specified by the modules *bottom*, *other*, and *top*, respectively.

A process can access other processes via variables specified as parameters of a module type. For example, in Figure 3.1, process p_0 can access the state of process p_1 via the variable `state` of p_1.

3.4.2 Processes

We assume that a distributed algorithm is described in the guarded command language. The behavior of a process is expressed as a module in the NuSMV program. Figures 3.2, 3.3, and Figure 3.4 describe the behavior of process p_0, process p_i $(0 < i < n-1)$, and process p_{n-1}, respectively.

In each module, the variable `state` denotes the state of the corresponding process p_i, while L and R aliases the state of its left neighbor p_{i-1}, and its right neighbor p_{i+1}, respectively.

Here, the keyword DEFINE is used to associate a symbol with a commonly used expression. Each definition of `priv` denotes that the process is enabled.

As mentioned before, we assume that in each step of computation, exactly one process is selected from the enabled processes and it updates its state. In a NuSMV program, however, all processes declared in the main module are executed in a synchronous manner. The variable `run` denotes that a process is selected. A process updates the state only if the value of `run` is true. The keyword INVAR is used to specify a Boolean expression that is true for any reachable states. Thus, by adding the declaration

```
INVAR
```

```
MODULE main
VAR
  p0 : bottom(p1.state);
  p1 : other(p0.state, p2.state);
  p2 : other(p1.state, p3.state);
  p3 : top(p2.state, p0.state);
DEFINE
  legitimate := (p0.priv + p1.priv + p2.priv + p3.priv = 1);

INVAR
  p0.run + p1.run + p2.run + p3.run = 1
```

Figure 3.1: NuSMV program of the three-state algorithm (main module).

```
run -> priv
```

to each process, we can guarantee that a process is always selected from the enabled processes. On addition, by adding the declaration

```
INVAR
  p0.run + p1.run + p2.run + p3.run = 1
```

to the main module, we can specify that the number of selected processes is exactly one.

The statement is divided into one or more assignments each of which updates a variable. The value of a case expression is determined by the first expression on the right-hand side of a ":" such that the condition on the left hand side is true. Thus, the right-hand side expression corresponds to the statement, and the condition is a conjunction of `run` and the guard, which means that when a process is selected, the value is updated according to the action whose guard is true.

```
MODULE bottom(R)
VAR
  state : { 0, 1, 2 };
  run   : boolean;
DEFINE
  priv := (state + 1) mod 3 = R;
INVAR
  run -> priv
ASSIGN
  next(state) := case
    run : (state + 2) mod 3;
    1   : state;
  esac;
```

Figure 3.2: NuSMV program of the three-state algorithm (bottom module).

```
MODULE other(L, R)
VAR
  state : { 0, 1, 2 };
  run   : boolean;
DEFINE
  priv := ((state + 1) mod 3 = L) |
          ((state + 1) mod 3 = R);
INVAR
  run -> priv
ASSIGN
  next(state) := case
    run : (state + 1) mod 3;
    1   : state;
  esac;
```

Figure 3.3: NuSMV program of the three-state algorithm (other module).

```
MODULE top(L, R)
VAR
  state : { 0, 1, 2 };
  run   : boolean;
DEFINE
  priv := (L = R) & (state != ((L + 1) mod 3));
INVAR
  run -> priv
ASSIGN
  next(state) := case
    run : (L + 1) mod 3;
    1   : state;
  esac;
```

Figure 3.4: NuSMV program of the three-state algorithm (top module).

3.4.3 Computing of the Time Complexity

In the main module, legitimate configurations are defined by the symbol `legitimate`.

The convergence time ct from configuration $g \in G$ is the least number of steps such that $g \models AF^{\leq ct} legitimate$ holds. Thus, the time complexity r of a self-stabilization algorithm is calculated as follows:

$$r = \max_{\forall g \in G} \{\min \{k \mid g \models AF^{\leq k} P\}\} .$$

In a NuSMV program, we can calculate the stabilization time by a COMPUTE MAX statement:

```
COMPUTE MAX [ 1, legitimate ]
```

This statement takes two Boolean expressions. The general form of this statement is `COMPUTE MAX [ A, B ]`, where A is the expression that evaluates to

true in the initial configurations. Then, the statement enforces NuSMV to compute the following value:

$$r = \max_{\forall g \in G'} \left\{ \min \left\{ k \mid g \models AF^{\leq k} P \right\} \right\} ,$$

where $G' \subseteq G$ and the Boolean expression `A` holds for any $g \in G'$.

The Boolean expression 1 means tautology. This reflects the fact that a stabilizing algorithm can start from any configuration.

3.4.4 Extracting Worst-Case Execution

Using the computed time complexity, the worst-case computation can be extracted by checking the following RTCTL specification:

```
SPEC ABF 0 .. (r - 1) legitimate          ,
```

where r is the computed time complexity. This specification corresponds to the expression $AF^{\leq r-1} legitimate$. NuSMV checks that any computation starting from any state reaches a legitimate configuration in $r - 1$ steps. This specification does not hold because there is a computation whose length is r. Thus, NuSMV provides one of the worst-case computations as a counterexample.

A counterexample is a sequence of the values of each variable as follows:

```
--> State 1.1 <--
p0.state = 0
p1.state = 1
p2.state = 2
p3.state = 3
```

```
--> State 1.2 <--
p0.state = 1
...
```

We can obtain the sequence used in our proof by using the program shown in Appendix 3.6.2.

3.4.5 Analysis of Counterexamples

Here, we can obtain only one counterexample. We can get another counterexample by adding INIT constraints as follows:

```
INIT
  !(
    (p0.state = 0) &
    (p1.state = 1) &
    (p2.state = 2) &
    (p3.state = 0)
  )
```

INIT constraints are used to specify a Boolean expression that must hold on initial configurations. By removing an initial configuration of a counterexample using INIT, we can obtain another counterexample for the RTCTL specification.

The RTCTL specification eventually holds when we iterate to add an INIT constraint and get a counterexample because the number of initial configurations is finite. Then, we found that only a few configurations can be the initial configuration of a worst-case computation. The result leads us to the proof described in this chapter.

Table 3.2: Exact worst-case time complexity. It coincides perfectly with our lower bound for $9 \leq n \leq 20$.

n	worst-case time complexity $T(n)$
9	109
10	137
11	170
12	212
13	250
14	296
15	348
16	396
17	455
18	517
19	575
20	647

3.4.6 Results

Using NuSMV, we also mechanically computed the exact worst-case time complexity for $9 \leq n \leq 20$. Interestingly, the complexity exactly matches our lower bound. Table 3.2 shows the concrete figures for this range of n. Based on this finding, we conjecture that our lower bound is the exact worst-case time complexity when $n \geq 9$. If our conjecture is true, then it is also true under a distributed scheduler, because any single step under a distributed scheduler can be simulated by a sequence of steps under a centralized scheduler [5, 8].

3.5 Summary

In this chapter, we proved a lower bound of the time complexity of Dijkstra's self-stabilizing three-state mutual exclusion algorithm, and showed a method for

computing time complexity by using model checking.

The best-known lower bound on the worst-case time complexity was given by Chernoy, Shalon, and Zaks [9]. They proved a lower bound of $1\frac{5}{6}n^2 - O(n)$ by showing that there is a schedule of length $1\frac{5}{6}n^2 - 10\frac{1}{6}n + 14$ when $n = 3k$. Although our bound matches $1\frac{5}{6}n^2 - O(n)$, ours is tighter than $1\frac{5}{6}n^2 - 10\frac{1}{6}n + 14$ when $n = 3k$. When $n = 3k \geq 9$, we have:

$$\begin{aligned} &\left(1\frac{5}{6}n^2 - 4\frac{1}{6}n - 2\right) - \left(1\frac{5}{6}n^2 - 10\frac{1}{6}n + 14\right) \\ &= 6n - 16 > 0 \end{aligned}$$

On addition, our result applies when $n = 3k + 1$ and $n = 3k + 2$.

We have thus assumed that exactly one enabled process executes the statement of the algorithm in each step. This model is often referred to as the centralized scheduler model. A different model could be that any subset of enabled processes can be selected in each step, which is called the distributed scheduler model. The three-state algorithm is correct in the latter model [5]. Clearly, the proposed lower bound holds under the distributed scheduler, because any computation in the centralized scheduler model is also possible in the distributed scheduler model.

We obtained the computations used in our proofs by analyzing the algorithm's behavior with the NuSMV model checking tool [10]. Model checking is a state exploration-based verification technique. The use of model checking for analyzing self-stabilizing algorithms was studied in [36, 37]. We used these studies with some modifications to derive the computations used.

On the following appendix, we show the program for generating a NuSMV program (3.6.1) and the script for formatting a counterexample (3.6.2).

3.6 Appendix

3.6.1 Script for Generating NuSMV Program

The script `gen` is written in Perl. When you enter a natural number $n > 2$, this script generates the NuSMV program of Dijkstra's self-stabilizing three-state mutual exclusion algorithm with n processes.

```
#! /usr/bin/env perl
# File name: gen
use strict;
use warnings;

sub usage;

@ARGV || usage;

my $N = (shift) - 0;

$N > 2 || usage;

print <<EOT;
MODULE bottom(R)
VAR
  state : { 0, 1, 2 };
  run   : boolean;
```

```
DEFINE
  priv := (state + 1) mod 3 = R;
INVAR
  run -> priv
ASSIGN
  next(state) := case
    run : (state + 2) mod 3;
    1   : state;
  esac;

MODULE other(L, R)
VAR
  state : { 0, 1, 2 };
  run   : boolean;
DEFINE
  priv := ((state + 1) mod 3 = L) |
          ((state + 1) mod 3 = R);
INVAR
  run -> priv
ASSIGN
  next(state) := case
    run : (state + 1) mod 3;
    1   : state;
  esac;
```

```
MODULE top(L, R)
VAR
  state : { 0, 1, 2 };
  run   : boolean;
DEFINE
  priv := (L = R) &
          (state != ((L + 1) mod 3));
INVAR
  run -> priv
ASSIGN
  next(state) := case
    run : (L + 1) mod 3;
    1   : state;
  esac;

MODULE main
VAR
  p0 : bottom(p1.state);
EOT

for(my $i = 1; $i < $N - 1; ++$i){
    print "  p$i : other(p", $i - 1, ".state, p",
      $i + 1, ".state);\n";
```

```
}
print "  p", $N - 1, " : top(p", $N - 2,
  ".state, p0.state);\n";

print "DEFINE\n";
print "  legitimate := (",
    join(" + ",
      map { "p$_.priv" } (0 .. $N - 1)),
    " = 1);\n";
print "INVAR\n";
print "  ", join(' + ',
    map { "p$_.run" } (0 .. $N - 1)
  ), " = 1\n";

sub usage {
    print STDERR <<EOT;
Usage : gen n

    n    The number of processes (n > 2)
EOT
    exit 1;
}
```

3.6.2 Script for Formatting a Result

The script `format` converts a counterexample to the sequence of $\{<, =, >\}$, which is used in our proof. This script is used as follows:

```
$ NuSMV input.smv | ./format
```

A NuSMV program "input.smv" must contain the RTCTL specification.

```
#! /usr/bin/env perl
# File name: format
use strict;
use warnings;

my @states = ();

while(<>){
    if(/State: 1\.\d+/){
        prettify(@states) if(@states);
    }elsif(/p(\d+)\.state = (\d)/){
        $states[$1 - 0] = $2 - 0;
    }
}
prettify(@states);

sub operator {
```

```
    $_[0] == $_[1] ?            "=" :
    ($_[0] + 1) % 3 == $_[1] ? "<" :
                                ">" ;
}

sub prettify {
    my @src = @_;

    print join(' ',
        map {
            operator $src[$_], $src[$_ + 1]
        } (0 .. $#src - 1),
    ), "\n";
}
```

Chapter 4

On Time Complexity of Hsu and Huang's Maximal Matching

4.1 Introduction

In this chapter, we discuss the time complexity of the self-stabilizing algorithm proposed by Hsu and Huang in [25], which finds a maximal matching in a network. This algorithm is the first self-stabilizing maximal matching algorithm and has been regularly cited in the literature. Based on this algorithm, many self-stabilizing algorithms were thereafter developed for the maximal matching problem and its variants [6, 20, 19, 21, 32, 27].

Because of its technical importance, the time complexity of this particular algorithm has been well studied. In [25], Hsu and Huang show that it is bounded by $O(n^3)$, where n is the number of processes. In [34], Tel provides an almost tight upper bound, which is $\frac{1}{2}n^2 + 2n + 1$ if n is even and $\frac{1}{2}n^2 + n - \frac{1}{2}$ if n is odd. In [35], Tel gives a more concise proof for the $O(n^2)$ bound than [34]. In [23],

Hedetniemi, Jacobs, and Srimani provide an upper bound of $2|E| + n$, where $|E|$ is the number of edges. This gives a better bound than the one in [34] but only if the network is sparse. In this chapter, we provide the exact time complexity of the Hsu–Huan algorithm.

The remainder of this chapter is organized as follows. In Section 4.2, we describe the algorithm. In Section 4.3, we prove the upper bound of the time complexity. In Section 4.4, we prove that the upper bound is the exact time complexity by showing a computation whose length matches the upper bound. In Section 4.5, we summarize this chapter.

4.2 The Hsu–Huang Algorithm

We consider a distributed system consisting of n (≥ 2) processes. The topology of the system is modeled as an undirected graph. Let $N(p)$ denote the set of a process p's adjacent processes (neighbors).

Given an undirected graph $G = (V, E)$ where V is a set of nodes and E is a set of edges, a *matching* M is a subset of edges where no two edges share a node. If no matching M' is a superset of a matching M, then M is a *maximal matching*. We consider the problem of finding a maximal matching of the graph.

Each process p has a pointer. The pointer either points to one of p's neighbors that p selects to match or has a null value. The notation $p \rightarrow q$ denotes that p's pointer points to $q \in N(p)$, the notation $p \rightarrow null$ denotes that p's pointer has a null value, and the notation $p \Leftrightarrow q$ denotes that $p \rightarrow q \wedge q \rightarrow p$.

Each process p is in one of the following five states:

1. If $\exists q \in N(p) : (p \rightarrow q) \wedge (q \rightarrow null)$, then p is *waiting*.

2. If $\exists q \in N(p) : p \Leftrightarrow q$, then p is *matched.*

3. If $\exists q \in N(p), \exists r \in N(q) : (p \rightarrow q) \wedge (q \rightarrow r) \wedge (r \neq p)$, then p is *chaining.*

4. If $(p \rightarrow null) \wedge (\forall q \in N(p) :$ q is matched), then p is *dead.*

5. If $(p \rightarrow null) \wedge (\exists q \in N(p) :$ q is not matched), then p is *free.*

A maximal matching is found iff every process is either matched or dead.

The Hsu–Huang algorithm at each process p is given by the following three rules.

if

$$(p \rightarrow null) \wedge (\exists q \in N(p) : q \rightarrow p) \Rightarrow p \rightarrow q \; [] \qquad (R_1)$$

$$(p \rightarrow null) \wedge (\forall r \in N(p) : \neg(r \rightarrow p)) \wedge (\exists q \in N(p) : q \rightarrow null) \Rightarrow p \rightarrow q \; [] \qquad (R_2)$$

$$(p \rightarrow q) \wedge (q \rightarrow r) \wedge (r \neq p) \Rightarrow p \rightarrow null \qquad (R_3)$$

fi ,

where each rule is of the form $guard \Rightarrow action$. Each rule is executed atomically and no two processes can execute a rule at a time.

A **configuration** of the system is a collection of the pointers of all the processes. In [25], it is proven that any computation of the algorithm is finite, and every process is either matched or dead in the last configuration of any maximal computation, meaning that the system always converges to a configuration where a maximal matching is obtained. The time complexity of the algorithm is the maximum number of **steps** (that is, rule executions) required to find a maximal

matching. Thus, we have

$$(\text{time complexity}) = (\text{the length of the longest computation}) - 1 \ .$$

4.3 Upper Bound on Run Lengths

Our derivation of the upper bound on the time complexity follows the basic line of [34]. In [34], similar to many self-stabilization literatures, time complexity is analyzed using the *variant function* technique. A *variant function* is a function over configurations, whose value is monotonically decreases (in our context) when processes execute a rule of the algorithm.

Our variant function is a tuple (X, Y), where X and Y are functions that map a configuration to a non-negative integer as follows:

$$\begin{aligned} X &\equiv \begin{cases} \lceil \frac{c+f+w}{2} \rceil & (\text{even } n) \\ \\ \lfloor \frac{c+f+w}{2} \rfloor & (\text{odd } n) \end{cases} \\ Y &\equiv 2c + f \end{aligned} \quad ,$$

where c, f, and w are the number of chaining, free and waiting processes, respectively. For odd n, the variant function is identical to that of [34]. The modification made is that a different expression of X is used for even n. As stated later, this subtle modification is critical to obtain the exact time complexity.

The variable function is evaluated in lexicographical order; i.e., X is evaluated first and then Y. Below we show that this function indeed decreases monotoni-

cally when a rule is executed. As in [34], an important observation is that $c+f+w$ never increases because by the design of the rules, matched or dead processes remain matched or dead. Thus, it suffices to see that either X or Y is decreased by the execution of a rule. In the following description, p, q, and r refer to p, q, and r in the rule definition described in Section 2.

Execution of Rule R_1 Rule R_1 is enabled only when p is free and q is waiting. When it is executed, p and q become matched. Dead or matched processes do not change their state. Hence, the rule execution decreases $c + f + w$ by at least 2, thus decreasing X by at least 1.

Execution of Rule R_2 Rule R_2 is enabled only when p is free and causes p to become waiting. Because no process is waiting for p ($\forall r \in N(p) : \neg(r \rightarrow p)$), the waiting process becomes neither chaining nor free. Except for p, all free processes remain free. Hence, the execution of the rule decreases Y by 1.

Execution of Rule R_3 This rule is enabled only when p is chaining and causes p to be free or dead by setting p's pointer to $null$. No process becomes chaining. On addition, no waiting process becomes free because p is the only process that makes its pointer $null$. Hence, the execution of R_3 decreases Y by at least 1.

In summary, the execution of any of the three rules either: (1) decreases X, or (2) decreases Y, but does not increase X. This leads to the following observation:

Observation 1. *Any computation $g_1 g_2 \cdots g_l$ is a concatenation of computations $\sigma_1, \cdots, \sigma_m$ such that: (1) all configurations in σ_i have the same F value; (2) if configurations g, g' occur in σ_i and σ_{i+1} respectively, then $X(g) > X(g')$, and (3)*

if configurations g, g' consecutively occur in σ_i, then $Y(g) > Y(g')$.

This is schematically represented as follows:

$$\begin{array}{cccccccc}
 & & \overbrace{\qquad\qquad\qquad\qquad}^{\sigma_i} & & \overbrace{\qquad\qquad\qquad\qquad}^{\sigma_{i+1}} & & & \\
 & \cdots & g_{j-2} & g_{j-1} & g_j & g_{j+1} & \cdots & \\
\cdots & \cdots = X(g_{j-2}) = X(g_{j-1}) & > & & X(g_j) = X(g_{j+1}) = \cdots & & \cdots & \\
 & \cdots > Y(g_{j-2}) > Y(g_{j-1}) & & & Y(g_j) > Y(g_{j+1}) > \cdots & & & .
\end{array}$$

Another observation used in obtaining the upper bound is as follows:

Observation 2. *Because a waiting process waits for a free process, $w \geq 1$ implies $f \geq 1$. Hence $G \geq 1$ if $c + f + w \geq 1$; $G = 0$ if $c + f + w = 0$.*

Our derivation of the upper bound refines the one by Tel in [34] in three ways.

- The analysis of the configurations where $X = \lfloor \frac{n}{2} \rfloor$ is refined (Lemma 6). This reduces the upper bound by 4 if n is even and by 2 if n is odd.
- The new variable function allows the reduction of the upper bound by $n - 2$ for the case of even n (Lemma 7).
- The analysis of the configurations where $X = 0$ is refined (Lemma 8). This reduces the upper bound by 1 for the case of even n.

As a result, the new bound is smaller than that of [34] by $n + 3$ if n is even, and by 2 if n is odd.

Lemma 6. *If a computation $g_1 g_2 \cdots g_l$ satisfies $X(g_1) = \cdots = X(g_l) = \lfloor \frac{n}{2} \rfloor$, then the length l of the computation is at most $2n - 2$.*

Proof. From Observation 2, $Y(g_i) \geq 1$ for any g_i. If $Y(g_1) \leq 2n-2$, then the lemma follows trivially. Thus, in the following proof, we assume that $2n-1 \leq Y(g_1) \leq 2n$ and proceed as follows. We first show that under this assumption, there is always a "cycle" of chaining processes in g_1. Then, we show that $l \leq 2n-2$ holds in two cases: (1) none of the processes consisting of the cycle execute a rule in the computation, and (2) some process in the cycle executes a rule.

Because of the assumption of $2n-1 \leq Y(g_1) \leq 2n$, either $c = n-1 \wedge f = 1 \wedge w = 0$ or $c = n \wedge f = w = 0$ holds in g_1. Hence, in g_1, every chaining process has a pointer pointing to another chaining process. (Note that even if $f = 1$, a chaining process cannot point to that free process, because if a process p has a pointer to a free process, then, by definition, p is a waiting process.)

In g_1, therefore, there is at least one cycle of chaining processes, that is, there is a sequence of processes $p_1, p_2, \cdots, p_{len}$ such that $p_i \rightarrow p_{i+1}$ for all $i, 1 \leq i \leq len-1$ and $p_{len} \rightarrow p_1$. By the definition of a chaining process, the cycle contains at least three processes, that is, $len \geq 3$.

If none of the processes consisting of a cycle execute a rule in the computation, $Y(g_l) \geq 6$, because $len \geq 3$ implies $c \geq 3$. In that case, because $2n \geq Y(g_1) > Y(g_2) > \cdots > Y(g_l) \geq 6$, the computation length l is at most $2n-5$.

Now, consider the case where some process in the cycle executes a rule in this computation. Note that only R_3 can be executed by this process. Let p be the first process that executes the rule in the cycle, and let g_i be the configuration in which this rule execution occurs. Then, $Y(g_i) - 3 \geq Y(g_{i+1})$, because p becomes free, and the process that points to p in the cycle becomes waiting, resulting in a decrease in c by at least 2 and an increase in f by 1. Because $2n \geq Y(g_1) > Y(g_2) > \cdots > Y(g_l) \geq 1$, the computation length l is at most $2n-2$.

□

Lemmas 7, 8, and 9 apply to the case of even n.

Lemma 7. *When n is even, if a computation $g_1 g_2 \cdots g_l$ satisfies $X(g_1) = \cdots = X(g_l) = x > 0$, then the length l of the computation is at most $4x$.*

Proof. $Y(g_1) \leq 4x$, because c is at most $2x$. From Observations 1 and 2, $4x \geq Y(g_1) > Y(g_2) > \cdots > Y(g_l) \geq 1$, thus, the lemma follows. □

Lemma 8. *When n is even, if a computation $g_1 g_2 \cdots g_l$ satisfies $X(g_1) = \cdots = X(g_l) = 0$, then the length l of the computation is 1.*

Proof. When n is even, if $X = 0$, then $c + f + w = 0$ and $2f + w = 0$. Hence, the computation contains exactly one configuration in which every process is matched or dead. □

Lemma 9. *When n is even, the length of any computation is at most:*

$$\frac{1}{2}n^2 + n - 1$$

Proof. Because there are n processes, $0 \leq X \leq \frac{n}{2}$ and $0 < \frac{n}{2}$. From Observation 1 and Lemmas 6, 7, and 8, the upper bound on the computation length is derived as follows:

$$(2n - 2) + \sum_{x=1}^{\frac{n}{2}-1} 4x + 1$$
$$= \frac{1}{2}n^2 + n - 1$$

□

Lemmas 10, 11, and 12 apply to the case of odd n.

Lemma 10. *When n is odd, if a computation $g_1 g_2 \cdots g_l$ satisfies $X(g_1) = \cdots = X(g_l) = x > 0$, then the length l of the computation is at most $4x + 2$.*

Proof. $Y(g_1) \leq 4x + 2$, because c is at most $2x + 1$. From Observations 1 and 2, $4x + 2 \geq Y(g_1) > Y(g_2) > \cdots > Y(g_l) \geq 1$, thus,the lemma follows. □

Lemma 11. *When n is odd, if a computation $g_1 g_2 \cdots g_l$ satisfies $X(g_1) = \cdots = X(g_l) = 0$, then the length l of the computation is at most 2.*

Proof. At each configuration g_i in the computation, either $c + f + w = 1$ or $c + f + w = 0$, because $X(g_i) = 0$.

If $c + f + w = 1$, then $2c + f = 2$, because neither $c = w = 0 \wedge f = 1$ nor $c = f = 0 \wedge w = 1$ is possible, This is because a process can be free or waiting only if at least one of its neighbors is neither matched nor dead. If $c + f + w = 0$, then $2c + f = 0$. As a result, $Y(g_i) = 2$ (if $c + f + w = 1$) or $Y(g_i) = 0$ (if $c + f + w = 0$) for any g_i in the computation, thus, the computation length is at most 2. □

Lemma 12. *When n is odd, the length of any computation is at most:*

$$\frac{1}{2}n^2 + n - \frac{3}{2}$$

Proof. Because there are n processes, $0 \leq X \leq \lfloor \frac{n}{2} \rfloor$ and $0 < \lfloor \frac{n}{2} \rfloor$. From Observation 1 and Lemmas 6, 10, and 12, the upper bound on the computation length is

derived as follows:

$$(2n-2)+\sum_{x=1}^{\lfloor \frac{n}{2} \rfloor -1}(4x+2)+2$$
$$=\frac{1}{2}n^2+n-\frac{3}{2}$$

□

4.4 Exact Time Complexity

In this section, we provide the exact time complexity, by showing an algorithm execution whose computation length exactly matches the upper bound obtained in the previous section. This example of execution is identical to that shown by Tel in [34]; however, the computation length is analyzed only for the case of even n. Here, we provide the exact computation length for the case of odd n, generalizing his result to any n. Before presenting our result, we first describe this execution to clearly show how the result is derived.

Suppose that the system consists of $n \geq 3$ processes $p_1, p_2, \cdots, p_n$ and that the topology of the system is a complete graph. On addition, suppose that initially $p_1 \rightarrow p_2, p_2 \rightarrow p_3, \cdots, p_{n-1} \rightarrow p_n, p_n \rightarrow p_1$.

1. R_3 is executed by $n-1$ processes $p_1, p_2, \cdots, p_{n-1}$. As a result, all of the $n-1$ processes become free, and p_n becomes waiting for p_1.

2. R_2 is executed by $n-2$ processes $p_2, p_3, \cdots, p_{n-1}$ to point p_1.

3. R_1 is executed by p_1 to match p_n. As a result, p_1 and p_n become matched, and the other $n-2$ processes become chaining.

Phases 1–3 result in $n-2$ chaining processes and two matched processes.

Phases 4–6 start with $n-i$ matched processes and i (≥ 2) chaining processes pointing to a matched process. Initially, i is $n-2$.

4. R_3 is executed by the i chaining processes. As a result, all of the i processes become free.

5. Let p be any one of the i free processes. The free processes other than p execute R_2 to point p. The $i-1$ steps cause these free processes to become waiting.

6. R_1 is executed by one of the waiting process, say q, to match p. As a result, p and q become matched, and the other $i-2$ processes become chaining. Phases 4–6 are repeated with i replaced with $i-2$ until at most one chaining process remains.

As a result, all the processes become matched if n is even, whereas a single chaining process remains if n is odd. In the latter case, Phase 7 is performed.

7. R_3 is executed by the chaining process, causing the process to become dead.

The number of steps of the above execution is expressed as follows: For even n:

$$\begin{aligned} & 2n-2+\sum_{x=1}^{\frac{n}{2}-1}(2x+(2x-1)+1) \\ & =\frac{1}{2}n^2+n-2 \end{aligned}$$

For odd n:

$$2n - 2 + \sum_{x=1}^{\lfloor \frac{n}{2} \rfloor - 1} (2(x+1) + 2x + 1) + 1$$
$$= \frac{1}{2}n^2 + n - \frac{5}{2} \quad .$$

For the case $n = 2$, we can consider the following scenario. Starting with two free processes, the execution of R_2 by each of the processes leads to a final configuration where both are matched. The number of steps involved in this example is two, which coincides with the above expression.

Theorem 4. *The exact time complexity of the algorithm is expressed as follows:*

$$\begin{cases} \frac{1}{2}n^2 + n - 2 & (\textit{even } n) \\ \frac{1}{2}n^2 + n - \frac{5}{2} & (\textit{odd } n) \end{cases} .$$

Proof. By Lemma 9 and Lemma 12, these expressions represent the upper bound on the time complexity. The above examples of algorithm execution show that these expressions also represent the lower bound. □

4.5 Summary

In this chapter, we analyzed the time complexity of Hsu and Huang's self-stabilizing maximal matching algorithm [25]. Refining the result by Tel [34], we provided the exact time complexity.

The fact that the upper bound by Tel is very similar to the time complexity computed using model checking (see Table 4.1) leads us to find our proof. On the

# of Processes	Upper Bound	Computed
3	7	5
4	18	10
5	17	15
6	32	22
7	31	29

Table 4.1: Upper bound and computed time complexity of Hsu–Huang algorithm

following appendix, we explain a NuSMV program for computing the time complexity of the Hsu–Huang algorithm, and show a program to generate a NuSMV program for n processes.

4.6 Appendix

4.6.1 Example of a NuSMV Program

Here we show an example of a NuSMV program to compute the time complexity of Hsu and Huang's maximal matching algorithm with three processes. Figures 4.1, 4.2, 4.3, and Figure 4.4 represent the program. Note that this program assumes that the topology of the system is a complete graph to avoid the state-explosion problem.

To compute the time complexity of the algorithm on any topology, a NuSMV program must contain flags irrespective of whether processes p and q are neighbors. This makes the state space of the program larger. Fortunately, it is sufficient to compute the time complexity of this algorithm on a complete graph to compute the worst-case time complexity.

```
MODULE main
VAR
  p1 : { 0, 2, 3 };
  p2 : { 0, 1, 3 };
  p3 : { 0, 1, 2 };
  r1 : boolean;
  r2 : boolean;
  r3 : boolean;
```

Figure 4.1: NuSMV program for maximal matching (main)

Here, we consider a computation $g_1, \cdots, g_l$ to be the longest computation on a graph G that is not a complete graph. When the number of dead processes is 1 or 0, the length of the longest computation on a graph G' that is generated by adding an edge to G is larger than or equal to that on G. When the number of dead processes is larger than or equal to 2, there is a longer computation on G' that is generated by adding an edge between the two dead processes. The processes can execute a rule to be matched each other from g_l. Thus, the longest computation on a complete graph is the worst-case computation.

Figure 4.1 represents the system. The variable **pi** ($i \in \{1, 2, 3\}$) is the pointer of a process. If pi $= 0$ holds, then the pointer does not point to any neighbors. If pi $\neq 0$ holds then the pointer points to the process pi. Because we use 0 as null value, the index of the processes starts with 1. The variable **ri** ($i \in \{1, 2, 3\}$) is used to represent that the process is selected to execute a rule.

In Figure 4.2, **mi** and **di** ($1 \leq i \leq 3$) represents expressions that hold when a process is matched or dead, and **legitimate** represents an expression that holds when all processes cannot execute a rule, that is, when a configuration is legitimate.

```
DEFINE
  m1 := ((p1 = 2) & (p2 = 1)) | ((p1 = 3) & (p3 = 1));
  m2 := ((p2 = 1) & (p1 = 2)) | ((p2 = 3) & (p3 = 2));
  m3 := ((p3 = 1) & (p1 = 3)) | ((p3 = 2) & (p2 = 3));
  d1 := (p1 = 0) & m2 & m3;
  d2 := (p2 = 0) & m1 & m3;
  d3 := (p3 = 0) & m1 & m2;
  legitimate := r1 + r2 + r3 = 0;
```

Figure 4.2: NuSMV program for maximal matching (matched and dead)

Figure 4.3 represents rules that are executed by process 1. We omit the rules for other processes because they are redundant for explanatory purposes.

For R_1 and R_2, we need to specify all of the possible combinations of conditions about their neighbors because a case expression evaluates to the first right-hand side value whose left-hand side condition holds. If we used the following expressions to describe R_1:

```
r1 & (p1=0) & (p2=1)           : {2   };
r1 & (p1=0)          & (p3=1) : {   3};
```

then p1 would always become 2 when $r1\&(p1 = 0)\&(p2 = 1)\&(p3 = 1)$ holds.

The value of **ri** ($1 \leq i \leq n$) is determined by a set of invariants. Figure 4.4 shows two of them. The first invariant specifies that a process — process 1 in this case — must be enabled for some rule when it is selected for computation. This invariant is needed for every process. The second invariant specifies that all **ri** becomes false only when all processes are matched or dead.

```
ASSIGN
  next(p1) := case
    -- R1
    r1 & (p1=0) & (p2=1) & (p3=1) : {2, 3};
    r1 & (p1=0) & (p2=1)          : {2   };
    r1 & (p1=0)          & (p3=1) : {   3};
    -- R2
    r1 & (p1=0) & (p2=0) & (p3=0) : {2, 3};
    r1 & (p1=0) & (p2=0)          : {2   };
    r1 & (p1=0)          & (p3=0) : {   3};
    -- R3
    r1 & (p1=2) & (p2=3)          : 0;
    r1 & (p1=3)          & (p3=2) : 0;
    1                             : p1;
  esac;
```

Figure 4.3: NuSMV program for maximal matching (assignment for p_1)

```
INVAR
  r1 -> (
    -- R1 or R2
    ((p1 = 0) & ((p2 = 1) | (p2 = 0) | (p3 = 1) | (p3 = 0))) |
    -- R3
    ((p1 = 2) & (p2 != 0) & (p2 != 1)) |
    ((p1 = 3) & (p3 != 0) & (p3 != 1))
  )

INVAR
  legitimate -> ((m1 | d1) & (m2 | d2) & (m3 | d3))
```

Figure 4.4: NuSMV program for maximal matching (a part of invariants)

4.6.2 Script for Generating a NuSMV Program

This script generates a NuSMV program of the Hsu–Huang algorithm with n processes. To compute the time complexity, you need to add the following expression:

```
COMPUTE MAX [ 1, legitimate ]
```

```
#! /usr/bin/env perl
# Filename: gen
use strict;
use warnings;

sub usage;
sub powerset;
sub R1;
sub R2;
sub R3;

@ARGV || usage;

my $N = (shift) - 0;

$N > 1 || usage;

my @PROCESSES = 1 .. $N;
my %NEIGHBORS = map {
```

```
  my $p = $_;
  $p => [grep { $_ != $p } @PROCESSES]
} @PROCESSES;
my %POWERS = map {
  $_ => powerset(@{$NEIGHBORS{$_}})
} @PROCESSES;

print "MODULE main\n";
print "VAR\n";
for my $i (@PROCESSES){
  print "  p$i : { 0, ",
    join(', ', @{$NEIGHBORS{$i}}),
    " };\n";
}
for my $i (@PROCESSES){
  print "  run$i : boolean;\n";
}

print "DEFINE\n";
for my $i (@PROCESSES){
  print "  m$i := (\n";

  print join(" |\n",
    map {
```

```
        ' ' x 4 . "((p$i = $_) & (p$_ = $i))"
      } @{$NEIGHBORS{$i}}
    ), "\n";

    print "  );\n";
}

for my $i (@PROCESSES){
    print "  d$i := (\n";
    print ' ' x 4, "(p$i = 0) &\n";
    print join(" &\n", map {
      ' ' x 4 . "m$_"
    } @{$NEIGHBORS{$i}}), "\n";
    print "  );\n";
}

print "  legitimate := ",
    join(" + ", map { "run$_" } @PROCESSES),
    " = 0;\n";

print "ASSIGN\n";
for my $i (@PROCESSES){
    print "  next(p$i) := case\n";
    R1 $i;
```

```
  R2 $i;
  R3 $i;
  print ' ' x 4, "1 : p$i;\n";
  print "  esac;\n\n";
}

for my $i (@PROCESSES){
  print "INVAR\n";
  print "  run$i -> (\n";

  # R1 or R2
  print ' ' x 4, "((p$i = 0) & (",
    join(" | ", map {
      "(p$_ = $i) | (p$_ = 0)"
    } @{$NEIGHBORS{$i}}), ")) |\n";

  print join(" |\n", map {
    ' ' x 4 . "((p$i = $_) & (p$_ != 0) & (p$_ != $i))"
  } @{$NEIGHBORS{$i}}), "\n";

  print "  )\n\n";
}

print "INVAR\n";
```

```
print "  legitimate -> (",
  join(" & ", map { "(m$_ | d$_)" } @PROCESSES),
  ")\n\n";

print "INVAR\n";
print "  ", join(" + ", map { "run$_" } @PROCESSES),
  " < 2\n";

sub usage {
  print STDERR <<EOT;
Usage : gen n

    n    The number of processes
EOT
}

sub powerset {
  my @result = ();
  my $mask = (1 << ($#_ + 1)) - 1;

  for(; $mask > 0; --$mask){
    my $tmp = [];

    for(my $i = 0; $i < @_; ++$i){
```

```
      if($mask & (1 << $i)){
        push(@$tmp, $_[$i]);
      }
    }

    push(@result, $tmp);
  }

  return [sort { $#{$b} <=> $#{$a} } @result];
}

sub R1 {
  my $p = shift;

  print ' ' x 4, "-- R1\n";

  for my $aref (@{$POWERS{$p}}){
    print ' ' x 4,
      join(" & ", "run$p", "(p$p = 0)",
        map {
          "(p$_ = $p)"
        } @$aref),
      " : { ", join(", ", @$aref), " };\n";
  }
```

```
}

sub R2 {
  my $p = shift;
  print ' ' x 4, "-- R2\n";

  for my $aref (@{$POWERS{$p}}){
    print ' ' x 4,
      join(" & ", "run$p", "(p$p = 0)",
        map {
          "(p$_ = 0)"
        } @$aref),
      " : { ", join(", ", @$aref), " };\n";
  }
}

sub R3 {
  my $p = shift;
  print ' ' x 4, "-- R3\n";

  for my $q (@{$NEIGHBORS{$p}}){
    print ' ' x 4,
      join(" & ", "run$p", "(p$p = $q)",
      "(p$q != 0)", "(p$q != $p)"), " : 0;\n";
```

```
    }
}
```

Chapter 5

New Fast Self-Stabilizing Maximal Matching Algorithm

5.1 Introduction

The worst-case time complexity of the Hsu–Huang algorithm in terms of the number of nodes n is $\frac{1}{2}n^2+n-2$ if n is even, and $\frac{1}{2}n^2+n-\frac{5}{2}$ if n is odd [31]. In [23], $2|E|+n-5$ and $2|E|+n$ are shown to be the lower and upper bounds on the worst-case time complexity, where $|E|$ is the number of edges.

Table 5.1: Worst-case time complexity of self-stabilizing maximal matching algorithms ($|E|$ and n denote the number of edges and nodes in a network graph).

Algorithm	w.r.t n	w.r.t $\|E\|$ and n
Hsu and Huang [25]	$T(n)=\frac{1}{2}n^2+n-2$ (n is even) $T(n)=\frac{1}{2}n^2+n-\frac{5}{2}$ (n is odd)	$2\|E\|+n-5 \leq T(n,\|E\|) \leq 2\|E\|+n$
Proposed Algorithm	$T(n)=\frac{1}{4}n^2+n-1$ (n is even) $T(n)=\frac{n^2+4n-5}{4}$ (n is odd)	$T(n,\|E\|) \leq \|E\|+n$

In this chapter, we propose a new self-stabilizing algorithm for computing a maximal matching. The proposed algorithm assumes the same model as the Hsu–Huang algorithm and runs faster. In particular, the new algorithm reduces the worst-case time complexity by approximately half. Table 5.1 compares the time complexity of these two algorithms.

The remainder of this chapter is organized as follows. Section 5.2 proposes our new algorithm. Section 5.3 describes the correctness proof and the derivation of the exact time complexity in terms of the number of nodes. Section 5.4 provides an upper bound on the time complexity in terms of the number of edges. For sparse graphs, this bound gives a better estimate for execution time than the one in terms of n. Section 5.5 summarizes this chapter.

5.2 The New Algorithm

A problem with the Hsu–Huang algorithm is that a node may repeatedly execute rule R_3. On other words, a node may withdraw its proposal for matching many times, resulting in a long computation. The proposed algorithm overcomes this problem by modifying this rule. Our algorithm consists of five rules $R_1, \cdots, R_5$. We use the same notation as that used in Chapter 4. We also use the notation $\not\rightarrow p$. This notation denotes that no neighbors points to p. R_1 and R_2 are identical to R_1 and R_2 of the Hsu–Huang algorithm, respectively. On the other hand, R_3 of

Hsu=-Huang algorithm is replaced with the following three rules:

$$(p \to q) \wedge (q \to r) \wedge (r \neq p) \wedge (\exists o \in N(p) : o \to p) \Rightarrow p \to o \; [] \qquad (R_3)$$

$$(p \to q) \wedge (q \to r) \wedge (r \neq p) \wedge \not\to p \wedge (\exists o \in N(p) : o \to null) \Rightarrow p \to o \; [] \qquad (R_4)$$

$$(p \to q) \wedge (q \to r) \wedge (r \neq p) \wedge \not\to p \wedge (\forall o \in N(p) : o \not\to null) \Rightarrow p \to null \qquad (R_5)$$

Note that the guards of the five rules are mutually exclusive. Hence, if a node is enabled, then exactly one rule is enabled at the node.

R_3 combines R_3 and R_1, that is, if a node executes R_3, then the resulting configuration is the same as the one that will occur if the node executes R_3 and then R_1 consecutively. Similarly, R_4 combines R_3 and R_2. This modification enforces a node p to make a new proposal for matching whenever p withdraws its proposal to a neighbor and some other neighbor is waiting for p (R_3) or free (R_4). As specified by R_5, a node can set its pointer to *null* only when there exists no neighbor to point at that it can point to.

The construction of the algorithm ensures that it always runs "faster" than the Hsu–Huang algorithm, in the sense that for any computation of the new algorithm, the latter algorithm has a computation of at least the same length.

More importantly, the new algorithm ensures that a node sets its pointer to *null* at most once, as formally shown by Lemma 16. This results in a reduction of the time complexity by approximately half. In the following two sections, we present the analysis of the time complexity.

5.3 Exact Time Complexity

5.3.1 Variant Function

To show the correctness and exact time complexity of our algorithm, we use the *variant function* technique similar to many self-stabilization literatures. A variant function is a function over the set of configurations, whose value is bounded and decreases monotonically when nodes execute a rule of the algorithm.

Our variant function is the same as used in Chapter 4:

$$\begin{aligned} X &\equiv \begin{cases} \lceil \frac{c+f+w}{2} \rceil & (\text{even } n) \\ \lfloor \frac{c+f+w}{2} \rfloor & (\text{odd } n) \end{cases} \\ Y &\equiv 2c + f \end{aligned} ,$$

where c, f, and w are the numbers of chaining, free, and waiting nodes, respectively. This function is evaluated in lexicographical order, i.e., X is evaluated first and then Y. Below we show that this function indeed decreases monotonically when a rule is executed. In particular, we show that for each of the five rules, its execution either decreases X or does not change X the same but decreases Y. In the following description, p, q, r, and o refer to those in the rule definition from R_1 to R_5.

Execution of rule R_1 Rule R_1 is enabled only when p is free and q is waiting. When it is executed, p and q become matched. Dead or matched nodes do not

change their state. Hence, the rule execution decreases $c + f + w$ by at least 2, thus decreasing X by at least 1.

Execution of rule R_2 Rule R_2 is enabled only when p is free and causes p to become waiting. Because no node is waiting for p (i.e., $\not\rightarrow p$), the waiting node becomes neither chaining nor free. Except for p, all free nodes remain free. Hence, the execution of the rule decreases Y by 1. X does not change, because no node becomes matched or dead.

Execution of rule R_3 Rule R_3 is enabled when p is chaining and o points to p. When it is executed, p and o become matched. Dead or matched nodes do not change their state. Hence, the rule execution decreases $c + f + w$ by at least 2, thus decreasing X by at least 1.

Execution of rule R_4 Rule R_4 is enabled only when p is chaining and there is a neighbor whose pointer has a null value. When it is executed, p becomes waiting. Because no node is waiting for p (i.e., $\not\rightarrow p$), the waiting node becomes neither chaining nor free. All free nodes remain free. Hence, the execution of the rule decreases Y by 2. X does not change because no node becomes matched or dead.

Execution of rule R_5 Rule R_5 is enabled only when p is chaining and all of its neighbors point to nodes other than p. When it is executed, p becomes free or dead and no other nodes change their state. Hence, the execution of R_5 decreases Y by at least 1. If p becomes free, then Y is decreased by 1. In this case, X does not change because no node becomes matched, or dead. If p becomes dead, then Y is decreased by 2 and $c + f + w$ by 1. Hence, in this case, X is either decreased

by 1 or left unchanged.

5.3.2 Correctness

To show the correctness of our algorithm, we show that it converges to a legitimate configuration from any configuration.

Lemma 13. *If $c + f + w > 0$, then at least one node is enabled to execute a rule.*

Proof. When $w > 0$, there is a waiting node p and the node points to a free node q, thus q is enabled to execute R_1.

When $w = 0 \wedge c > 0$, there is a chaining node p and the node is enabled to execute either R_3, R_4, or R_5.

When $w = 0 \wedge c = 0 \wedge f > 0$, a free node p has at least one neighbor q that is not matched. The node q is not dead either, because every neighbor of a dead node must be matched. Because $w = 0 \wedge c = 0$, q is free. Hence p and q are enabled to execute R_2. □

Theorem 5. *The proposed algorithm always converges to a legitimate configuration.*

Proof. Any execution of a rule decreases the variable function (F, G). The set of possible values of the variable function is finite, thus, any computation is finite. From Lemma 13 there is always an enabled node if $c + f + w > 0$. Hence, in the last configuration of any computation, every node is either matched or dead. □

5.3.3 Time Complexity in Terms of n

Here, we derive the exact time complexity in terms of n. We begin by proving three lemmas that are used in the derivation.

Lemma 14. *A dead or matched node will never be enabled.*

Proof. There is no rule that is enabled at a node if it is dead or matched. Suppose that p and q are neighboring nodes. Once $p \Leftrightarrow q$ holds, the two matched nodes will remain matched forever, because no rule execution at other nodes can change their state. A dead node also remains dead because all of its neighbors are matched, and thus, will never execute a rule. □

Lemma 15. *If $p \rightarrow q \wedge q \rightarrow null$, that is, if p is waiting for q, then p will never become enabled unless q executes a rule. Besides, the only rule that q can or will be able to execute is R_1.*

Proof. The first part of the lemma follows from the fact that no rule is enabled at p if $p \rightarrow q \wedge q \rightarrow null$. The existence of such p guarantees that there is at least one node that is waiting for q. Because of this and the fact that $q \rightarrow null$, only R_1 is enabled at q. If q executes R_1, then it will become matched and thus will no longer execute any rule (Lemma 14). □

Lemma 16. *Once a node p executes R_5, the only rule that p may be able to execute thereafter is R_1.*

Proof. The execution of R_5 causes the condition $p \rightarrow null$ to hold. Under the condition, R_3, R_4, and R_5 are not enabled at p. Below, we show that R_2 will never be enabled either while $p \rightarrow null$ holds.

Suppose that p becomes enabled to execute R_2 after its execution of R_5. In that situation, there must be its neighbor q such that $q \rightarrow null$. Note that q pointed to another node than p when p executed R_5 (otherwise, R_5 would not have been enabled at p). Hence, q must have executed R_5 to satisfy $q \rightarrow null$. However, this is impossible because the guard of R_5 cannot be true while $p \rightarrow null$ holds.

If p executes R_1, then it will become matched, meaning that p will never become enabled. Therefore, once p executes R_5, p cannot execute any rules other than R_1, and if it executes R_1, then it will never be enabled. □

Theorem 5 ensures that any computation is finite. This and the property of the variant function lead to the following observation:

Observation 3. *Any computation $g_1 g_2 \cdots g_l$ is a concatenation of computations $\sigma_1, \cdots, \sigma_m$ such that: (1) all configurations in σ_i have the same F value; (2) if configurations s, s' occur in σ_i and σ_{i+1} respectively, then $X(s) > X(s')$, and (3) if configurations s, s' consecutively occur in σ_i, then $Y(s) > Y(s')$.*

This observation is schematically represented as follows:

$$
\begin{array}{cccccccc}
 & & \overbrace{\qquad\qquad\qquad\qquad}^{\sigma_i} & & & \overbrace{\qquad\qquad\qquad\qquad}^{\sigma_{i+1}} & & \\
 & \cdots & g_{j-2} & g_{j-1} & g_j & g_{j+1} & \cdots & \\
\cdots & \cdots = X(g_{j-2}) = X(g_{j-1}) & > & & X(g_j) = X(g_{j+1}) = \cdots & & \cdots & \cdots \\
 & \cdots > Y(g_{j-2}) > Y(g_{j-1}) & & & Y(g_j) > Y(g_{j+1}) > \cdots & & & .
\end{array}
$$

In the rest of this section, we let $\rho = g_1 g_2 \cdots g_{|\rho|}$ denote any computation such that $X(g_i)$ is the same for every configuration s in the computation, where the length of ρ is denoted by $|\rho|$. We derive the upper bounds on $|\rho|$. Obviously, these

bounds also apply to $|\sigma_i|$ for σ_i in Observation 3, and thus, allow us to derive the upper bound on the computation length.

In the following analysis, we say that a node p executes a rule R in ρ if there are two consecutive configurations $g_i, g_{i+1} (1 \leq i < |\rho|)$ such that p is enabled for R at g_i and g_{i+1} is the next configuration caused by the execution of R by p at g_i. Note that for any ρ and for any g_i, g_{i+1}, such a node-rule pair (p, R) is uniquely determined because at most one rule is enabled at a node simultaneously and a rule execution can only change the pointer of the node that executes it. Note that no node executes R_1 or R_3 in ρ, because their execution always decreases the F value.

Lemma 17. *Any node executes a rule at most once in ρ.*

Proof. If $|\rho| = 1$, then the lemma follows trivially, because no node executes a rule in ρ. Hence, we assume that $\rho \geq 2$ below. As mentioned above, no node executes R_1 or R_3 in ρ.

If a node p executes R_2 or R_4 at g_i in ρ, then $p \rightarrow q \wedge q \rightarrow null$ holds for some $q \in N(p)$ at g_{i+1}. In this situation, from Lemma 15, p will not be enabled unless q executes rule R_1, thus, p executes no rule after g_{i+1} in ρ.

If p executes R_5 in ρ, then R_1 is the only rule that p may be enabled for thereafter from Lemma 16, ,thus p executes no rule in ρ after its execution of R_5. □

Lemma 18. *Let i be the total number of chaining, free, and waiting nodes at the first configuration g_1 in ρ, that is, $i \equiv c + f + w$ at g_1. Then, $|\rho| \leq i + 1$.*

Proof. From Lemma 14, a dead or matched node never executes a rule. From Lemma 17, a node executes a rule at most once in ρ. Hence, the lemma follows. □

This bound on $|\rho|$ can be improved if every node is neither dead nor matched at the beginning of ρ.

Lemma 19. *If $c+f+w=n$ holds at the first configuration g_1 of ρ, then $|\rho| \leq n$.*

Proof. Throughout the proof, we assume that $c+f+w=n$ holds at g_1. At every configuration in ρ, F has the same value, thus, $n-1 \leq c+w+f \leq n$ always holds in ρ. However, $c+w+f=n-1$ is impossible for any configuration, because the existence of a matched node requires another matched neighbor node to match and a dead node requires a matched neighbor node. Hence, $c+w+f=n$ holds for every configuration in ρ. Below, we consider three cases regarding g_1: $w>0$, $w=0 \wedge c>0$, and $w=c=0$.

If a node p is waiting for q at g_1, then from Lemma 15, it will never be enabled unless q executes R_1, which means that p executes no rule in ρ. From this and Lemma 17, if $w>0$ at g_1, then we have $|\rho| \leq (n-1)+1=n$.

If $w=0 \wedge c>0$ at g_1, then there is a cycle consisting of at least three chaining nodes, that is, $p_1 \rightarrow p_2, p_2 \rightarrow p_3, \cdots, p_l \rightarrow p_1$, because a chaining node cannot point to a free node. (Note that if a node points to a free node, then the former node is waiting.) At any node in the cycle, the only enabled rule is R_3. Hence, at least three nodes do not execute a rule in ρ. From this and Lemma 17, we have $|\rho| \leq (n-3)+1=n-2 \leq n$.

Finally, we consider the case where $w=c=0 \wedge f=n$ holds at g_1. When $c+w+f=n$, $G=0$ holds only if $c=f=0 \wedge w=n$. However, this condition is

never met, because the existence of a waiting node requires the existence of a free node. Therefore, $Y(g_i) > 0$ for any configuration g_i in ρ. From Observation 3 and the fact that $Y(g_1) = n$, we have $n \geq Y(g_0) > Y(g_1) > \cdots > Y(g_{|\rho|}) \geq 1$, thus, $|\rho| \leq n$. □

Lemma 20. *If n is even, then the time complexity of the algorithm is bounded by $\frac{1}{4}n^2 + n - 1$. If n is odd, then the complexity is bounded by $(n^2 + 4n - 5)/4$.*

Proof. Any computation can be considered to be a concatenation of computations $\sigma_1\sigma_2\cdots\sigma_m$ that satisfy the three conditions in Observation 3. Let k be a natural number such that $n = 2k$ when n is even, and $n = 2k + 1$ if n is odd. Let F_i denote the F value of σ_i. Then, $0 \leq F_i \leq k$.

At the first configuration of σ_i, $2F_i - 1 \leq c + f + w \leq 2F_i$ holds if n is even, and $2F_i \leq c + f + w \leq 2F_i + 1$ holds if n is odd. From Lemma 18, $|\sigma_i| \leq 2F_i + 1$ if n is even and $|\sigma_i| \leq 2F_i + 1 + 1$ if n is odd.

If $F_i = k$, then $c + f + w = n$ holds at the first configuration of σ_i. (As mentioned in the proof of Lemma 19, $c + f + w = n - 1$ never holds.) From Lemma 19, $|\sigma_i| \leq n$. Hence, we have an upper bound on the length of any computation as follows:

$$\sum_{\forall\sigma_i} |\sigma_i| \leq n + \sum_{i=0}^{k-1}(2i + 1) \quad \text{if } n = 2k$$

$$\sum_{\forall\sigma_i} |\sigma_i| \leq n + \sum_{i=0}^{k-1}(2i + 1 + 1) \quad \text{if } n = 2k + 1 \qquad .$$

Because the number of steps taken in a computation is smaller by 1 than the computation length, an upper bound on the time complexity is obtained by sub-

tracting 1 from these formulae as follows:

$$
\begin{aligned}
& n + \sum_{i=0}^{k-1}(2i+1) - 1 \\
& \qquad = n + k(k-1) + k - 1 \\
& \qquad = \left(\frac{n}{2}\right)^2 + n - 1 \\
& \qquad = \frac{1}{4}n^2 + n + 1 \quad \text{if } n = 2k
\end{aligned} \tag{5.1}
$$

$$
\begin{aligned}
& n + \sum_{i=0}^{k-1}(2i+1+1) - 1 \\
& \qquad = n + k(k-1) + 2k - 1 \\
& \qquad = n + \left(\frac{n-1}{2}\right)^2 + \frac{n-1}{2} - 1 \\
& \qquad = \frac{n^2 + 4n - 5}{4} \quad \text{if } n = 2k+1
\end{aligned} \tag{5.2}
$$

□

Theorem 6. *The above upper bound on the time complexity is the exact time complexity.*

Proof. We show a scenario for a general n that shows an execution in which the number of steps taken is exactly the same as the upper bound.

Consider a network graph whose topology is a complete graph. Assume that for every node its pointer has a null value.

1. All nodes except for one node set their pointer to that node by executing R_2, thus becoming waiting. This takes $n - 1$ steps.

2. The node to which all waiting nodes are pointing, say p, executes R_1, resulting in a new pair of matched nodes. All other nodes that are pointing to p, if any, become chaining. This takes 1 step.

3. If there is at least one chaining node, one such node executes R_5. The node becomes free in which case there are still other chaining nodes, or dead, in which case all other nodes are matched. This takes 1 step.

4. If there are chaining nodes, all of these chaining nodes execute R_4 to point to the free node, becoming waiting. The steps taken in this phase decreases by 2 for each iteration of these phases, from $n-3$ to 1 if $n = 2k$ or 2 if $n = 2k+1$.

5. Go to Phase 2.

If n is $2k$, then the scenario terminates with Phase 2, resulting in n matched nodes. The total number of steps is obtained as follows:

$$\begin{aligned}
&(n-1) + \sum_{i=1}^{k-1}(1+1+(n-2i-1)) + 1 \\
&\quad = n + (n+1)(k-1) - k(k-1) \\
&\quad = n + \left(n+1-\frac{n}{2}\right)\left(\frac{n}{2}-1\right) \\
&\quad = n + \left(\frac{n}{2}+1\right)\left(\frac{n}{2}-1\right) \\
&\quad = \frac{n^2}{4} + n - 1 \qquad . \qquad (5.3)
\end{aligned}$$

If n is $2k+1$, then the scenario terminates with Phase 3, resulting in $n-1$ matched nodes and one dead node. The total number of steps is obtained as

follows:

$$\begin{aligned}
&(n-1)+\sum_{i=1}^{k-1}(1+1+(n-2i-1))+1+1 \\
&\quad=(n+1)+(n+1)(k-1)-k(k-1) \\
&=(n+1)+\left(\frac{n-1}{2}-1\right)\left(n+1-\frac{n-1}{2}\right) \\
&\quad=(n+1)+\left(\frac{n-3}{2}\right)\left(\frac{n+3}{2}\right) \\
&\quad=\frac{n^2+4n-5}{4}
\end{aligned} \tag{5.4}$$

□

5.4 Time Complexity in Terms of $|E|$ and n

In this section, we provide an upper bound on the time complexity in terms of n and $|E|$, where $|E|$ is the number of edges of the network graph. In particular, we show that our algorithm always converges within $|E|+n$ steps. When the network graph is sparse, this bound serves as a better estimate for execution time than the one in terms of n.

In the analysis in this section, we count for each edge $\{p, q\}$ the number of times that p sets its pointer to q by executing R_1, R_2, R_3, or R_4 and the number of times that p nullifies its pointer pointing to q by executing R_5.

Lemma 21. *A node p sets its pointer to its neighbor $q \in N(p)$ at most once.*

Proof. If p is enabled for a rule that sets its pointer to q, then $q \rightarrow p$ or $q \rightarrow null$ holds. If $q \rightarrow p$, then the execution of that rule (in this case R_1 or R_3) will make

both p and q matched; thus, by Lemma 14, p will no longer be enabled.

Now, suppose that p executes a rule (R_2 or R_4) to point to q when $q \rightarrow null$. In this case, p will never be enabled again unless q executes R_1 from Lemma 15. If q executes R_1, then q will become matched (not necessarily to p). Hence, p will never set its point to q again. □

Lemma 22. *If a node p sets its pointer to a node q in g_i, and then later q sets its pointer to p, then p and q will be matched to each other.*

Proof. Consider a computation $g_1 g_2 \cdots$ and suppose that p sets its pointer to a node q in g_i. For p to set its pointer to q in g_i, either $q \rightarrow p$ or $q \rightarrow null$ must hold in g_i. If $q \rightarrow p$ holds in g_i, then p and q will be matched in g_{i+1}, and thus, q will no longer be enabled. Hence, we assume that $q \rightarrow null$ in g_i in the rest of the proof. Under this assumption, from Lemma 15, p will never be enabled after g_{i+1} unless q executes a rule, and the only rule that q can execute is R_1. Now, it suffices to consider only the case where q executes R_1, because if q does not execute R_1, q will never set its pointer. Suppose that q executes R_1 in g_j $(j > i)$. If q sets its pointer to a node other than p in g_j, then q will become matched in g_{j+1} and never set its pointer to p thereafter. If q sets its pointer to p in g_j, then p and q will be matched to each other in g_{j+1}. □

Lemma 23. *If p and q are matched to each other, then it is never the case that both p and q executed R_5 before they are matched.*

Proof. Let p and q be any neighboring nodes. Consider a computation $g_1 g_2 \cdots$ such that (1) p executes R_5 at g_i; (2) q does not execute R_5 at from g_1 to g_{i-1}; and

(3) p and q are not matched in $g_1, g_2, \cdots, g_j$ $(j > i)$ and matched to each other in $g_{j+1}, g_{j+2}, \cdots$. Below we show that q does not execute R_5 in this computation.

From Lemma 16, p is only enabled for R_1 at g_{i+1} or later configurations. If p executes R_1, p will become matched and will never be enabled. On the other hand, if p executes no more rules after g_{i+1}, then $p \rightarrow null$ will always hold thereafter. Therefore, for p and q to become matched to each other in g_{j+1}, p must set its pointer to q by executing R_1 in g_j. For p to execute R_1, $q \rightarrow p$ must hold at g_j.

In g_i, $q \rightarrow r$ holds for some $r \neq p$, because otherwise p is not enabled for R_5. Hence, q executes R_2 or R_4 to set its pointer to p somewhere from g_{i+1} to g_{j-1}, and thereafter leaves the pointer unchanged. From Lemma 16, if a node executes R_5, then the node will be enabled only for R_5. Therefore, p does not execute R_5 in the computation. □

Theorem 7. *The time complexity of the proposed algorithm is bounded by* $|E|+n$.

Proof. Let a denote the size of the maximal matching obtained, that is, the number of edges $\{p, q\}$ such that $p \Leftrightarrow q$ holds in the last configuration of a computation. From Lemmas 21 and 22, the total number of times that a node sets its pointer is at most $|E| + a$, because for each edge $\{p, q\}$, p sets its pointer to q at most once (Lemma 21), and both p and q set their pointer to each other only if the edge is in the obtained maximal matching (Lemma 22). Hence, $|E|+a$ is an upper bound on the total number of times that R_1, R_2, R_3, and R_4 are executed. From Lemma 16 each node executes R_5 at most once. From Lemma 23, if $\{p, q\}$ is in the maximal matching, then at most one of them executes R_5. Hence $n - a$ is an upper bound on the total number of times that R_5 is executed. Therefore, the number of rule executions is at most $(|E| + a) + (n - a) = |E| + n$. □

5.5 Summary

In this chapter, we proposed a self-stabilizing algorithm for computing a maximal matching for the state-reading model under the central daemon. The proposed algorithm runs faster than the seminal algorithm proposed by Hsu and Huang [25], which was the fastest known algorithm that assumes this system model. We showed that the proposed algorithm reduces the worst-case time complexity approximately by half, both in terms of n (the number of nodes) and n and $|E|$ (the number of edges).

The time complexity in terms of n and $|E|$ is greater than the one in terms of n when $|E| = n(n-1)/2$. Although our proof assumed that each edge is used once, many edges are not actually used . Therefore, the evaluation of the time complexity in terms of n and $|E|$ becomes greater. However, it gives a better bound when $|E| = O(n)$.

Through the design of this algorithm, we used model checking to verify whether the new algorithm can reach a legitimate state from any configuration. By this process, we could easily find errors in some ideas.

Chapter 6

Conclusion

6.1 Achievements

In this dissertation, we analyzed the time complexity of Dijkstra's self-stabilizing three-state mutual exclusion algorithm and Hsu and Huang's self-stabilizing maximal matching algorithm.

For Dijkstra's algorithm, we provided a new lower bound on the time complexity. We found that this bound equals the exact time complexity when $9 \leq n \leq 20$, where n is the number of processes.

For Hsu and Huang's algorithm, we derived the exact time complexity. The careful analysis of the Hsu–Huang algorithm also led use to devise a new self-stabilizing maximal matching algorithm. The time complexity of the new algorithm is approximately half of that of the Hsu–Huang algorithm.

In this line of research, we used model checking as an analysis tool. In particular, we used NuSMV, a symbolic model checker, to compute time complexity and dervive the worst-case execution for systems with small n.

Bibliography

[1] Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, and George Varghese. Time optimal self-stabilizing synchronization. In **STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing**, pages 652–661, New York, NY, USA, 1993. ACM.

[2] Baruch Awerbuch and Rafail Ostrovsky. Memory-efficient and self-stabilizing network reset (extended abstract). In **PODC '94: Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing**, pages 254–263, New York, NY, USA, 1994. ACM.

[3] Joffroy Beauquier and Oliver Debas. An optimal self-stabilizing algorithm for mutual exclusion on bidirectional non uniform rings. In **the Second Workshop on Self-Stabilizing Systems**, pages 17.1–17.13, 1995.

[4] Joffroy Beauquier, Maria Gradinariu, and Colette Johnen. Memory space requirements for self-stabilizing leader election protocols. In **PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing**, pages 199–207, New York, NY, USA, 1999. ACM.

[5] James E. Burns, Mohamed G. Gouda, and Raymond E. Miller. On relaxing interleaving assumptions. In **In Proceedings of the MCC Workshop Self-Stabilizing Systems, MCC Technical Report No. STP-379-89**, 1989.

[6] Subhendu Chattopadhyay, Lisa Higham, and Karen Seyffarth. Dynamic and self-stabilizing distributed matching. In **Proceedings of the twenty-first annual symposium on Principles of distributed computing**, pages 290–297. ACM, 2002.

[7] Yu Chen and Jennifer L. Welch. Self-stabilizing mutual exclusion using tokens in mobile ad hoc networks. In **DIALM '02: Proceedings of the 6th international workshop on Discrete algorithms and methods for mobile computing and communications**, pages 34–42, New York, NY, USA, 2002. ACM.

[8] Viacheslav Chernoy, Mordechai Shalom, and Shmuel Zaks. On the performance of Dijkstra's third self-stabilizing algorithm for mutual exclusion. In **9th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), Paris**, volume 4838 of **Lecture Notes in Computer Science**, pages 114–123. Springer, November 2007.

[9] Viacheslav Chernoy, Mordechai Shalom, and Shmuel Zaks. A self-stabilizing algorithm with tight bounds for mutual exclusion on a ring. In **Proc. 22nd Int'l Symp. on Distributed Computing (DISC)**, volume 5218 of **Lecture Notes in Computer Science**, pages 63–77. Springer, September 2008.

[10] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A new symbolic model checker. **Software Tools for Technology Transfer**, 2(4):410–425, 2000.

[11] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. **Model Checking**. MIT Press, 1999.

[12] Adam M. Costello and George Varghese. The fddi mac meets self-stabilization. In **ICDCS '99: Workshop on Self-stabilizing Systems**, pages 1–9, Washington, DC, USA, 1999. IEEE Computer Society.

[13] Edsger Wybe Dijkstra. Self-stabilizing systems in spite of distributed control. **Communications of the ACM**, 17(11):643–644, November 1974.

[14] Edsger Wybe Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. **Communications ACM**, 18(8):453–457, 1975.

[15] Edsger Wybe Dijkstra. A belated proof of self-stabilization. **Distributed Computing**, 1(1):5–6, January 1986.

[16] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. **Real-Time Systems**, 4(4):331–352, 1992.

[17] Sukumar Ghosh, Arobinda Gupta, and Sriram V. Pemmaraju. A self-stabilizing algorithm for the maximum flow problem. **Distributed Computing**, 10(4):167–180, 1997.

[18] Sukumar Ghosh and Mehmet Hakan Karaata. A self-stabilizing algorithm for coloring planar graphs. **Distributed Computing**, 7(1):55–59, 11 1993.

[19] Wayne Goddard, Stephen T. Hedetniemi, David P. Jacobs, and Pradip K. Srimani. A robust distributed generalized matching protocol that stabilizes in linear time. In **Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops**, pages 461–465, Washington, DC, USA, 2003. IEEE Computer Society.

[20] Wayne Goddard, Stephen T. Hedetniemi, David P. Jacobs, and Pradip K. Srimani. Self-stabilizing protocols for maximal matching and maximal independent sets for ad hoc networks. In **Proceedings of the 17th International Symposium on Parallel and Distributed Processing**. IEEE Computer Society, 2003.

[21] Wayne Goddard, Stephen T. Hedetniemi, and Zhengnan Shi. An anonymous selfstabilizing algorithm for 1-maximal matching in trees. **Information Processing Letters**, 91:797–803, 2006.

[22] Rachid Hadid. Space and time efficient self-stabilizing l-exclusion in tree networks. **ipdps**, 00:529, 2000.

[23] Stephen T. Hedetniemi, D.P. Jacobs, and P.K. Srimani. Maximal matching stabilizes in time $o(m)$. **Information Processing Letters**, 80(5):221 – 223, 2002.

[24] Ted Herman. Superstabilizing mutual exclusion. **Distributed Computing**, 13(1):1–17, 2000.

[25] Su-Chu Hsu and Shing-Tsaan Huang. A self-stabilizing algorithm for maximal matching. **Information Processing Letters**, 43(2):77–81, 1992.

[26] Shing-Tsaan Huang. Leader election in uniform rings. **ACM Transactions on Programming Languages and Systems**, 15(3):563–573, 1993.

[27] Mehmet H. Karaata and Kassem A. Saleh. A distributed self-stabilizing algorithm for finding maximum matching. **Computer Systems Science and Engineering**, 15(3):175–180, 2000.

[28] Yoshiaki Katayama, Toshiyuki Hasegawa, and Naohisa Takahashi. A super-stabilizing spanning tree protocol for a link failure. **Systems and Computers in Japan**, 38(14):41–51, 2007.

[29] Yoshiaki Katayama, Eiichiro Ueda, Hideo Fujiwara, and Toshimitsu Masuzawa. A latency optimal superstabilizing mutual exclusion protocol in unidirectional rings. **Journal of Parallel and Distributed Computing**, 62(5):865–884, 2002.

[30] Masahiro Kimoto, Tatsuhiro Tsuchiya, and Tohru Kikuno. Computing stabilization time of self-stabilizing algorithms with symbolic model checking. In **Proceedings of the 4th Symposium on Science Technology for System Verification**, pages 151–160, November 2007.

[31] Masahiro Kimoto, Tatsuhiro Tsuchiya, and Tohru Kikuno. The time complexity of hsu and huang's self- stabilizing maximal matching algorithm. **IEICE Transactions on Information and Systems**, E93-D(10):2850–2853, 10 2010.

[32] Fredrik Manne, Morten Mjelde, Laurence Pilard, and Sébastien Tixeuil. A new self-stabilizing maximal matching algorithm. **Theoretical Computer Science**, 410(14):1336 – 1345, 2009.

[33] A. Singhai and Swee-Boon Lim. The sunscalr framework for internet servers. In **FTCS '98: Proceedings of the Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing**, page 108, Washington, DC, USA, 1998. IEEE Computer Society.

[34] Gerard Tel. Maximal matching stabilizes in quadratic time. **Information Processing Letters**, 49(6):271–272, 1994.

[35] Gerard Tel. **Introduction to Distributed Algorithms**. Cambridge University Press, New York, NY, USA, 2001.

[36] Tatsuhiro Tsuchiya, Shin'ichi Nagano, Rohayu Bt Paidi, and Tohru Kikuno. Symbolic model checking for self-stabilizing algorithms. **IEEE Transactions on Parallel & Distributed Systems**, 12(1):81–95, January 2001.

[37] Tatsuhiro Tsuchiya, Yusuke Tokuda, and Tohru Kikuno. Computing the stabilization times of self-stabilizing systems. **IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences**, E83-A(11):2245–2252, November 2000.