Towards Scalable Model Checking of Self-Stabilizing Programs[☆,☆☆]

Jingshu Chen[a], Fuad Abujarad[b], Sandeep Kulkarni[a]

[a]*Michigan State University, 3115 Engineering Building, East Lansing, MI, 48824, US*
[b]*Yale University,464 Congress Ave, Suite 260, New Haven, CT, 06519, US*

Abstract

Existing approaches for verifying self-stabilization with a symbolic model checker have relied on the use of weak fairness. We point out that this approach has limited scalability. To overcome this limitation, first, we show that if self-stabilization is possible without fairness then cost of verifying self-stabilization is substantially lower. In fact, we observe from several case studies that cost of verification under weak fairness is more than 1000 times that of the cost without fairness.

For the case where weak fairness is essential for self-stabilization, we demonstrate the feasibility of two approaches for improving scalability: (1) decomposition and (2) utilizing the weaker version of self-stabilization, namely *weak stabilization*. In the first approach, designer partitions the program into components where each component satisfies its property without fairness.We show that the first approach enables us to verify Huang's mutual exclusion program for uniform rings with 31 processes (state space 10^{138}) whereas without this approach, it was not possible to verify the same program with 5 processes (state space 10^{10}). In the second approach, a weaker version of self-stabilization is verified. For Hoepman's ring-orientation program on odd-length ring, we show that it is possible to verify weak stabilization for 301 processes (state space 10^{181}) whereas self-stabilization could not be verified for 9 processes (state space 10^5) under weak fairness. Furthermore, one can utilize transformation algorithms to convert weak stabilizing

☆This work was partially sponsored by AFOSR FA9550-10-1-0178, NSF CNS Grant 0914913, NSF CAREER CCR-0092724 and ONR Grant N00014-01-1-0744.

☆☆A preliminary version of this paper appeared in Principles of Distributed Systems - 14th International Conference, OPODIS 2010.

Email addresses: chenji15@cse.msu.edu (Jingshu Chen), fuad.abujarad@yale.edu (Fuad Abujarad), sandeep@cse.msu.edu (Sandeep Kulkarni)

programs to probabilistically stabilizing programs. Hence, for the case where it is not possible to verify deterministic self-stabilization, one can obtain the assurance provided by probabilistic self-stabilization at a significantly reduced cost. Finally, we also present 5 case studies to illustrate the scalability of stabilization with techniques suggested in this paper.

Keywords: Self-stabilization, Fairness, Fault-tolerance, Verification, Model checking

1. Introduction

Self-stabilization[2], an ability to converge to a legitimate state from an arbitrary initial state, enables a program to automatically recover from the occurrence of (transient) faults. In particular, if a self-stabilizing program is perturbed by a transient fault then the program is guaranteed to recover to a legitimate state after faults stop. This property is especially useful in a large, distributed network where predicting the exact dynamic situation is difficult or impossible. Hence, several algorithms such as routing, leader election, mutual exclusion [5, 6, 2] are designed to be self-stabilizing.

Contrary to traditional verification where we only consider the set of reachable states starting from some initial state(s), verification of self-stabilizing programs requires us to consider all possible states that could be substantially larger. Hence, verification of self-stabilization programs is a challenging task [1, 10, 11]. Especially, in contexts where self-stabilization is used to provide recovery from unexpected transient faults, it is crucial that the program eventually recovers to a legitimate state. Hence, verification of convergence is important for self-stabilizing programs. Moreover, due to complex recovery algorithms used in self-stabilizing programs, it is desirable to automate the verification of the self-stabilization property.

One approach for automated verification of self-stabilization is to utilize model checking, a technique to automatically verify whether a given model meets a given property. Unlike theorem proving approaches (e.g. [11]), the model checking approach does not require the designer to have considerable experience in logic reasoning and hence, it is widely used in verifying the distributed algorithms. Moreover, if the program does not meet the given property, the process of model checking typically produces a counterexample. Thus, model checking can be used as a tool by the designer while developing self-stabilizing protocols. However, it is subject to the state explosion problem [12]. Moreover, since self-stabilization permits an arbitrary initial state, the state explosion problem is expected to be more severe in the context of self-stabilizing programs.

In previous work, Tsuchiya et al [1] have proposed an approach for model checking self-stabilizing programs. In this work, the problem of state space explosion is reduced with the help of symbolic techniques. In particular, authors use Ordered Binary Decision Diagrams (OBDDs) [9] to represent programs. They utilize SMV [8] for verification.

Although the work in [1] demonstrates feasibility of applying model checking for verifying self-stabilizing programs, it also shows that verification is feasible only for programs with a small number of processes. To overcome this limitation, in this paper, we focus on the bottlenecks involved in verification of self-stabilizing programs. In particular, we focus on the issue of fairness and its effect on verification performance for self-stabilizing programs.

Existing model checkers have focused on weak fairness in their representation of fairness. We show that if self-stabilization is possible under unfair computation, verification cost can be significantly lower. In fact, we observe from several case studies where that the cost of verification under weak fairness is more than 1000 times that of the cost under no fairness. The practical meaning of this observation is if the extra effort required to verify self-stabilization under weak fairness is not necessary, the state space reached by model checking of self-stabilizing programs could be larger.

One can deal with failure of model checking caused by space/time limitations by two approaches: (1) manually assist the model checker to make it more effective, or (2) verify a slightly different property (or model) that still provides *good* assurance. Hence, for the case where weak fairness is essential for self-stabilization and model checking cannot be applied due to space/time limitations, we propose two approaches. The first approach requires manual effort by the designer in decomposing the given self-stabilizing program. The second approach focuses on a weaker version of self-stabilization, namely *weak stabilization* [24]. We show that both these approaches improve the scalability by orders of magnitude.

Our analysis with weak stabilization partially validates and partially repudiates the *suggestion* in [24] that weak stabilization is significantly easier to prove than self-stabilization. Specifically, we point out that the suggestion is valid for the case where weak fairness is needed for self-stabilization. However, the suggestion is inaccurate for the case where self-stabilization is achievable without fairness. In particular, we find an unexpected result that for this case, the cost of verifying weak stabilization is virtually identical to that of verifying stabilization. We note that this is especially surprising given that weak stabilization requires that from every state there is *some* path to reach a legitimate state whereas stabilization requires that from every state *any* path reaches a legitimate state.

Organization of the paper. The rest of the paper is organized as follows. In Section 2, we provide formal definitions of the programs, computations, fairness constraints and self-stabilization. In Section 3 and Section 4, we consider different programs and compare the time for verifying them under different levels of fairness. Section 5 and 6 discuss two approaches for designers to verify the programs at hand that require weak fairness to ensure self-stabilization. Section 7 discusses the related work. Finally, Section 8 makes concluding remarks and discusses future work.

2. Preliminaries

In this section, we present the formal definition of the program, state space, computations and fairness constraints. These definitions are based on previous work in [10, 13, 14].

Definition 1. (Program) A program, p, is described using a finite set of variables $V_p = \{v_0, v_1, \ldots, v_n\},\ n \geq 0$, and a finite set of program actions $A_p = \{a_0, a_1, \ldots, a_m\}$ $m \geq 0$. Each variable, $v_i \in V_p$, is associated with a finite domain of values, D_i. Each action, $a_i \in A_p$, is defined as follows: $a_i :: g_i \longrightarrow st_i$; where g_i is a Boolean formula involving program variables and st_i is a statement that updates a subset of program variables.

For such a program, we define the notion of state, state space and state predicate.

Definition 2. (State) A state, s, of program p is identified by assigning each variable in V_p a value from its respective domain. ∎

Definition 3. (State space) The state space, S_p, of p is the set of all possible states of p. ∎

Definition 4. (State predicate) A state predicate of p is a Boolean expression defined over the program variables V_p. Thus, a state predicate C of p identifies the subset, $S_C \subseteq S_p$, where C is *true* in a state s iff $s \in S_C$. ∎

Definition 5. (Enabled) The action a_i:: $g_i \rightarrow st_i$, is enabled in a state s iff g_i is $true$ in s. ∎

Observe that action in a program corresponds to a set of transitions $(s_0,\ s_1)$ where s_0 is the initial state and s_1 is the next state that is obtained by executing the statement of the action that is enabled in s_0. Thus, program transitions are defined as follows:

Definition 6. (Transitions) Transitions of p are defined by the following set: $\{(s_0, s_1) \mid s_0,\ s_1\ \in S_p, \wedge(\exists a_i \in A_p :: g_i$ is true in s_0 and s_1 is obtained by executing st_i from s_0) $\}$.

Definition 7. (Computation) A sequence of states, $\sigma = \langle s_0, s_1, ...\rangle$ is a computation of p iff:

1. $\forall j : 0 < j < length(\sigma) : (s_{j-1}, s_j)$, is a transition of p, and
2. if σ is finite and terminates in s_l then all the guards of the program actions are false in s_l. ∎

Intuitively a fair computation allows for a fair resolution of non-determinism. Next, we introduce the definition of weakly-fair computation. Intuitively, in a weakly-fair computation, if a guard of an action is continuously true then that action must be executed. Thus weakly-fair computation is defined as follows.

Definition 8. (Weakly-fair computation) $\sigma = \langle s_0, s_1, ...\rangle$ is weakly-fair computation of p iff:

1. σ is a computation of p, and
2. if any action, say a_i, of p is enabled in all states $s_j, s_{j+1}, s_{j+2} \ldots$ then $\exists k : k \geq j : s_{k+1}$ is obtained by executing st_i from state s_k. ∎

Remark. Note that Definition 7 does not consider fairness. Hence, as needed, we use the term *unfair computation* to distinguish the computation without fairness from the one with fairness. The term *unfair computation* has the same meaning as *computation* in Definition 7.

Definition 9. (Stabilization) Let p be a program and let I be a state predicate of p. We say that p is self-stabilizing for I iff:

1. closure: if (s_0, s_1) is a transition of p and $s_0 \in I$, then $s_1 \in I$;
2. convergence: every computation of p reaches I, i.e., $\forall \sigma : \sigma$ is of the form $\langle s_0, s_1, s_2, \ldots\rangle$ and σ is computation of $p : (\exists j :: s_j \in I)$. ∎

Note that the above definition can be instantiated with unfair computations or with weakly-fair computations. In the former case, we say that the program p is self-stabilizing for I without fairness. And, in the latter case, we say that program p is self-stabilizing for I under weak fairness. Finally, whenever I or fairness level is clear from the context, we omit it.

3. Model Checking Self-stabilizing Program under Weak Fairness

In this section, we describe modeling of self-stabilizing program with weak fairness assumption in symbolic model checker (SMV[8]) proposed in [1]. Our case studies include three classic examples in the literature of self-stabilization: Dijkstra's K-state program [2], Ghosh's mutual exclusion program[3] and Hoepman's ring-orientation program[4]. We chose these three studies for comparing verification time with [1]. Our case studies show that scalability of verifying self-stabilization is unlikely to change significantly with improved hardware. It also provides baseline –that utilizes current state of art– for considering optimizations considered later in this paper

This section is organized as follows: First, in Section 3.1, we recall the approach proposed in [1] for modeling self-stabilizing programs in SMV. This section also presents the three case studies. Next, in Section 3.2, we analyze the results. This analysis leads us to our first approach for improving scalability of verification for self-stabilizing programs.

3.1. Modeling Self-stabilizing Program under Weak Fairness

Modeling Approach. In a SMV program, we use a *module* to model the behavior of each process. Within each *module*, we use an ASSIGN declaration to model each action of the corresponding process. For example, given a program action $g_1 \longrightarrow st_1$, we model it as follows:

$ASSIGN$
    init(v) := initial values;
    next(v) := case g_1: $\mathcal{F}_{st_1}(v)$;
                1:v;
                esac;

where v denotes variable changed in st_1 and $\mathcal{F}_{st_1}$ denotes the assignment function used in st_1.

To guarantee a fair scheduling of processes, SMT provides a fairness constraint, which has a structure of "FAIRNESS ρ". Model checker only explores paths in which ρ happens infinitely often. Hence, we can force each process to be selected to run infinitely often by adding the declaration FAIRNESS *running*, where *running* is an internal variable in SMV for each process which is set *true* when a transition from that process executes. Thus the FAIRNESS clause guarantees that each process gets a chance to execute its enabled actions infinitely often. According to Definition 8, the fair scheduler in the modeling approach is a weakly-fair one because it can guarantee a process to be executed if that process is continuously enabled.

3.1.1. Case Study 1: K-State Token Ring Program

The K-state program consists of $N+1$ processes, numbered from 0 to N. The program topology is a unidirectional ring. Each process $p.i$, $0 \leq i \leq N$, has one variable $x.i$ that denotes the current state value. Each variable has the domain $[0, \ \ldots, \ K-1]$.

The program consists of two types of actions. The first type is for process 0. This action is enabled when $x.0$ equals $x.N$. When $p.0$ executes its action, it increments $x.0$ by 1 in modulo K arithmetic. The second type of action is for process $p.i$, $i \neq 0$. This action is enabled when $x.i$ is not equal to $x.(i-1)$. When $p.i$ executes its action, it copies $x.(i-1)$. Thus, the actions are as follows:

$$
\begin{array}{llll}
K_0:: & x.0 = x.N & \longrightarrow & x.0 = (x.0+1)\ mod\ K; \\
K_i:: & x.i \neq x.(i-1) & \longrightarrow & x.i = x.(i-1);
\end{array}
$$

Remark 3.1. *This program is known to be self-stabilizing if $K > N$. In subsequent discussion, we let $K = N+1$.*

Legitimate states. The state where x values of all processes is 0 is a legitimate state. In this state, only process 0 is enabled. After process 0 executes, $x.0$ changes to 1 and all other x values are still 0. In this state, only process 1 is enabled. Hence, it can execute and change $x.1$ to 1. Continuing this further, eventually, we reach a state where all x values are 1 where process 0 is the only enabled process and process 0 will increment $x.0$ to 2. The legitimate states of the K-state program are equal to all the states reached in such subsequent execution.

Modeling K-state program in SMV under weakly-fair computation. SMV provides a simple approach for modeling weak fairness. In particular, the behavior of each process can be instantiated from a specific module. As the program requires, there are two types of actions and hence we use two modules, one for K_0 and one for K_i ($i \neq 0$). The module for K_0 specifies variable $x.0$ and takes one parameter, the x value of its predecessor. In SMV, the transition $(s.0, s.1)$ for action $K.0$ is specified by the keyword ASSIGN. Within ASSIGN, '$init(x.0) := 0, 1, 2;$' specifies the value of the variable in the source state, i.e., $s.0$. Moreover, '$next(x.0) := case\ x.0 = x.N : (x.0+1)\ mod\ 3;\ 1 : x.0;\ esac;$' specifies the value in the target state, i.e., $s.1$. If the guard $(x.0 = x.N)$ is $true$

	Execution time(s)							
Numbers of Processes	3	4	5	6	7	8	9	10
Results from our experiments	0	0.03	0.63	5.33	34.30	139.10	1276.08	N/A
Results reported in [1]	0.1	0.4	4.6	43.5	285.2	1836.0	N/A	N/A
Approximate state space	10^1	10^2	10^3	10^4	10^5	10^7	10^8	10^10

Table 1: Verification Results for the K-state program

and $s.1$ is obtained by executing $x.0 = x.0 + 1\ mod\ 3$ from state $s.0$, otherwise the value of $x.0$ remains unchanged. Thus, module for action $K.0$ can be written as follows:

MODULE $type\_K_0(x.N)$
VAR $x.0 : 0, 1, 2;$
ASSIGN $init(x.0) := 0, 1, 2;$
$next(x.0) := case\ (x.0 = x.N) : (x.0 + 1)\ mod\ 3\ ;\ 1\ :\ x.0;\ esac;$
FAIRNESS $running$

Thus action K_0 can be instantiated from module $type\_K_0$ as follows: $K_0\ :\ process\ type\_K_0(x.N);$.

The module for K_i is similar with the one for K_0. It specifies variable $x.i$ and takes $x.(i-1)$, as parameter. The transition $(s.0, s.1)$ for action $K.i$ is specified by the keyword ASSIGN. Within ASSIGN statement, $init(x.i) := 0, 1, 2;$, specifies the value of the variable in the source state. And $next(x.i) := case\ !(x.i = x.j) : x.i = x.(i-1);\ 1 : x.i;\ esac;$, specifies the value in the target state. Hence Action K_i is instantiated from module $type\_K_i$ as follows: $K_i\ : process\ type\_K_i(x.(i-1));$. Finally, each process has the declaration FAIRNESS RUNNING to ensure that SMV only considers computation paths where each process executes infinitely often.

Verification results of the K-state program. We verified the K-state program for $3 \leq K \leq 10$. Table 1 gives the verification time for model checking the K-state program for different values of K. N/A in this table means the result was not available within an admissible amount of time (1 hour).

3.1.2. Case Study 2: Ghosh's Binary Mutual Exclusion Protocol

In this section, we present our second case study, namely, Ghosh's binary mutual exclusion protocol [3]. Ghosh's binary mutual exclusion protocol considers a network system of $2m - 1 (m \geq 2)$ nodes, numbered from 0 to $2m - 1$. The neighbor relation is defined as follows:

	Execution time(s)						
Numbers of Processes	8	10	12	14	16	18	20
Results from our experiments	0.4	2.93	22.43	138.05	693.27	2819.05	N/A
Results reported in [1]	3.1	22.9	182.0	1161.5	N/A	N/A	N/A
Approximate state space	10^2	10^3	10^3	10^4	10^4	10^5	-

Table 2: Verification Results for Ghosh's mutual exclusion program

- n_0 has one neighbor n_1;
- $n_{2i-1}(1 \leq i \leq m-1)$ has three neighbors $n_{2i-2},\ n_{2i}$, and n_{2i+1};
- $n_{2i}(1 \leq i \leq m-1)$ has three neighbors $n_{2i-2},\ n_{2i-1}$, and n_{2i+1};
- n_{2m-1} has one neighbor n_{2m-2}.

The state s_i of each node n_i can be either 0 or 1. Each node can read its own state and the state of its neighbor nodes. The protocol defines the four types of actions as follows:

for n_0:
$s_0 \neq s_1 \longrightarrow s_0 = 1 - s_0;$
for n_{2m-1}:
$s_{2m-1} = s_{2m-2} \longrightarrow s_{2m-1} = 1 - s_{2m-1};$
for $n_{2i-1}(1 \leq i \leq m-1)$:
$s_{2i-2} = s_{2i-1} = s_{2i} \ \wedge \ s_{2i-1} \neq s_{2i+1} \longrightarrow s_{2i-1} = 1 - s_{2i-1};$
for $n_{2i}(1 \leq i \leq m-1)$:
$s_{2i-2} = s_{2i-1} = s_{2i+1} \ \wedge \ s_{2i} \neq s_{2i+1} \longrightarrow s_{2i} = 1 - s_{2i};$

We model the program and check the self-stabilization property of the protocol using the same approach as mentioned in Section 3.1.1. The verification results for this case are shown in Table 2.

3.1.3. Case Study 3: Hoepman's Uniform Ring-orientation Program

In this section, we present our third case study, namely, Hoepman's uniform ring-orientation program [4]. Hoepman's uniform deterministic ring-orientation program considers a system of n nodes, numbered from 0 to $n-1$, which are organized as a uniform ring of odd length. Each node n_i has a $color$, $Color_i$, with domain $\{0, 1\}$. To impose a direction, each node stores a $phase$, $Phase_i$, with domain $\{0,1\}$. A global legitimate state is one where all the nodes are oriented in the same direction. A ring orientation program is self-stabilizing iff it reaches a legitimate state from any initial state. To achieve self-stabilization, this program defines the following four types of actions for each node n_i:

$$
\begin{aligned}
& Color_{neighbor1} = Color_{neighbor2} \\
& \qquad\qquad \longrightarrow \; Color_i = 1 - Color_{neightbor1}, \\
& \qquad\qquad \; Phase_i = 1;
\end{aligned}
$$

$$
\begin{aligned}
& Color_{neighbor1} = Color_i = 1 - Color_{neighbor2} \\
& \wedge\, Phase_i = Phase_{neighbor2} = 1 \\
& \wedge\, Phase_{neighbor1} = 0 \\
& \qquad\qquad \longrightarrow \; Color_i = 1 - Color_i, \\
& \qquad\qquad \; (Phase_i) = 0, \\
& \qquad\qquad \; direction_i = n_{neighbor1} \hookrightarrow n_i \hookrightarrow n_{neightbor2};
\end{aligned}
$$

$$
\begin{aligned}
& Color_{neighbor2} = Color_i = 1 - Color_{neighbor1} \\
& \wedge Phase_i = Phase_{neighbor1} = 1 \\
& \wedge Phase_{neighbor2} = 1 \\
& \qquad\qquad \longrightarrow \; Color_i = 1 - Color_i, \\
& \qquad\qquad \; Phase_i = 0, \\
& \qquad\qquad \; direction_i{=}n_{neighbor1} \hookleftarrow n_i \hookleftarrow n_{neightbor2};
\end{aligned}
$$

$$
\begin{aligned}
& (Color_{neighbor1} = Color_i = 1 - Color_{neighbor2} \\
& \wedge Phase_{neighbor1} = Phase_i) \\
& |(Color_{neighbor2} = Color_i = 1 - Color_{neighbor1} \\
& \wedge Phase_i = Phase_{neighbor2}) \\
& \qquad\qquad \longrightarrow \; Phase_i = 1 - Phase_i;
\end{aligned}
$$

In the above actions, Action 1 requires that if a node has the same color as both its neighbors, then it inverts its color. This action creates patterns such as 001 and 110 around the ring since it is of odd length. Actions 2 and 3 require that if one node has the same color and the opposite phase as one of its neighbors, then the direction is from the node with phase 0 to the node with phase 1. Action 4 requires that if one node has the same color and the same phase as one of its neighbors, then it inverts its phase.

We model the program and check the self-stabilization property of the protocol using the same approach as mentioned in Section 3.1.1. The verification results for this case are shown in Table 3.

	Execution time(s)			
Numbers of Processes	3	5	7	9
Results from our experiments	0.17	18.23	1113.77	N/A
Results reported in [1]	1.3	128.1	N/A	N/A
Approximate state space	10^1	10^3	10^4	-

Table 3: Verification Results for Hoepman's ring-orientation program on Odd-length ring

3.2. Analysis

From these case studies, we observe that the approach proposed in [1] has limited scalability. In the first case study, in [1], authors have shown the feasibility of verification of k-state program for up-to $K = 8$. In particular, the time reported in [1] for $K = 8$ is 1836.0s whereas the time for the corresponding verification is 139.1s. Since the underlying tool as well as the program remains the same, this change is due to improved hardware over last few years. However, what this result does show is that in spite of the improved hardware, the ability to verify under weak fairness remains essentially the same. Specifically, if we assume a reasonable time constraint permissible (e.g., one hour) for verification then the change in hardware made it possible to achieve verification for $K = 9$ as opposed to $K = 8$.

As discussed in [26], one of the reasons for this is that to model fairness, one needs to ensure that each process can execute infinitely often. Achieving this increases the OBDD size for reachable states quadratically. Our first approach utilizes this observation to reduce the cost of verification of self-stabilization.

4. Approach 1: Model Checking Self-stabilizing Program Under Unfair Computation

In this section, we propose an approach of modeling self-stabilizing programs under unfair computation in SMV. We illustrate this approach by evaluating the three case studies mentioned in the previous section. To compare the verification performance of two modeling approaches, we perform the experiments in the same hardware setting. The verification results show that this approach is significantly more scalable.

4.1. Modeling Self-stabilizing Program under Unfair Computation

Now, we describe how to model self-stabilizing program under unfair computation in SMV. Recall that a self-stabilizing program $\mathcal{P}$ consists of a set of guarded commands of the following form:

$$
\begin{array}{ll}
action_1: & g_1 \longrightarrow st_1; \\
action_2: & g_2 \longrightarrow st_2; \\
\ldots & \\
action_i: & g_i \longrightarrow st_i; \\
\ldots & \\
action_n: & g_n \longrightarrow st_n;
\end{array}
$$

To model $\mathcal{P}$ under unfair computation in SMV, we use the TRANS keyword and model $action_i$ as follows:

$$
\begin{aligned}
conjunction_i: \ & g_i \wedge (\bigwedge_{st_i \text{ updates } v_j} next(v_j) = \mathcal{F}_{st_i}(v_j)) \\
& \wedge (\bigwedge_{st_i \text{ does not update } v_j} next(v_j) = v_j)
\end{aligned}
$$

In the above modeling, v_i ($i = 0, .., num - 1$, where num is the numbers of variables in the program) denotes the variables assigned by statement st_i. $\mathcal{F}_{st_i}(v_i)$ denotes the assignment function used in st_i. $conjunction_i$ models $action_i$. The above formula requires that g_i must be true in the initial state. Moreover, if v_j updated by $action_i$ then $next(v_j)$ corresponds to the value given by st_i. Otherwise, v_j remains unchanged. Since the use of TRANS in SMV requires the user to explicitly ensure that the transition relation is total. The transition relation is total if every state has a successor state. Hence, we add an additional action "$\neg(\bigvee_{i=1,\ldots,n} g_i) \longrightarrow skip$;" to the program. Using the approach for modeling actions, this action is modeled as follows:

$$
conjunction_{addtional}: \ \neg(\bigvee_{i=1,\ldots,n} g_i) \wedge (\bigwedge next(v_j) = v_j);
$$

Thus, in the modeling approach under unfair computation, the whole program is modeled as one transition relation in SMV.

4.1.1. Case Study 1: K-State Token Ring Program (Cont'd)

In this section, we continue with the verification of K-state program. We first discuss how we model the K-state program in SMV under unfair computation. Then, we provide the verification results under unfair computation. We compare these results with the corresponding verification results under weak computation.

The comparison results show that for K-state program, the verification performance is substantially improved under unfair computation.

Modeling K-state program in SMV under unfair computation.

We illustrate how we model the K-state program in SMV under unfair computation. Figure 1 gives a SMV program of K-state program with k=3. As shown in Figure 1, $x0$, $x1$ and $x2$ denotes the states of the three processes. Lines 1-6 define $x0$, $x1$, $x2$ and initialize them. Lines 7-9 define $x0priv$, $x1priv$ and $x2priv$, that are used to describe privilege condition for each process. Lines 10-13 define $condition1$ and $condtion2$ to describe the self-stabilization. Line 15 specifies the program action, which is a disjunction of three possible actions, one for each process. The extra action is used to ensure the totalness of the program. These three cases include: 1)process 0 is privileged, $x0$ is assigned new value and states of other two processes $x1$ and $x2$ remains unchanged; 2) process 1 is privileged, $x1$ is assigned new value and states of other two processes $x0$ and $x2$ remains unchanged; and, 3) process 2 is privileged, $x2$ is assigned new value and states of other two processes $x1$ and $x0$ remains unchanged. Note that if multiple processes are privileged then one of them is non-deterministically chosen for execution. The extra action, where none of the processes is enabled is not needed, as in any state, at least one of the processes can execute. As expected, adding this action does not affect the performance.

Verification results of the K-state program. We verified the K-state program for $3 \leq K \leq 9 \; or \; K = 50$. Table 4 gives the verification time for model checking the K-state program for different values of K. N/A in this table means the result was not available within an admissible amount of time (1 hour).

	Execution time(s)									
Numbers of Processes	3	4	5	6	7	8	9	10	50	51
Unfair	0	0	0	0	0.02	0.03	0.05	0.08	3466.30	N/A
Weakly-fair	0	0.03	0.63	5.33	34.30	139.10	1276.08	N/A	N/A	N/A
Approximate state space	10^1	10^2	10^3	10^4	10^5	10^7	10^8	10^{10}	10^{84}	-
Size of BDD (unfair)	292	786	2423	4373	8346	10067	11475	14870	1842498	-
Size of BDD (weakly fair)	680	3435	11251	17880	42131	108723	564794	N/A	N/A	-

Table 4: Verification Results for the K-state program

4.1.2. Case Study 2: Ghosh's Binary Mutual Exclusion Protocol (Cont'd)

In this section, we consider Ghosh's mutual protocol under unfair computation. We first describe how we model Ghosh's mutual protocol in SMV under

MODULE main

VAR

$x0 : \{0, 1, 2\};$ (1)

$x1 : \{0, 1, 2\};$ (2)

$x2 : \{0, 1, 2\};$ (3)

INIT

$x0 = \{0, 1, 2\}\&$ (4)

$x1 = \{0, 1, 2\}\&$ (5)

$x2 = \{0, 1, 2\}$ (6)

DEFINE

$x0priv := (x0 = x2);$ (7)

$x1priv :=!(x1 = x0);$ (8)

$x2priv :=!(x2 = x1);$ (9)

$$\begin{aligned} condition1 := ((&x0priv\&!x1priv\&!x2priv)| \\ &(!x0priv\&x1priv\&!x2priv)| \\ &(!x0priv\&!x1priv\&x2priv)); \end{aligned} \tag{10}$$

$$condition2 := AFx0priv\&AFx1priv\&AFx2priv; \tag{11}$$

$$legitimate := condition1\&condition2; \tag{12}$$

SPEC AF legitimate (13)

TRANS (14)

$$\begin{aligned} &((x0 = x2)\&next(x0) = (x0 + 1)mod3\&next(x1) = x1\&next(x2) = x2) \\ &\quad |(!(x1 = x0)\&next(x1) = x0\&next(x2) = x2\&next(x0) = x0) \\ &\quad |(!(x2 = x1)\&next(x2) = x1\&next(x0) = x0\&next(x1) = x1) \\ &\quad |(next(x2) = x2\&next(x0) = x0\&next(x1) = x1) \end{aligned} \tag{15}$$

Figure 1: Modeling K-state program in SMV under unfair computation.

unfair computation. Then, we provide the verification results under unfair computations and compare these results with the corresponding results under weak fair computations. The comparison results show the scalability of modeling self-stabilizing program under unfair computation for Ghosh's mutual protocol.

Modeling Ghosh's mutual protocol in SMV under unfair computation. We model Ghosh's mutual protocol in SMV under unfair computation in the similar approach presented in Section 4.1. For the SMV program of Ghosh's mutual exclusion protocol with $n = 8$ modeled under unfair computation, we use x_i ($i = 0 \ldots 7$) denotes the states of the eight processes. We define $xipriv(i = 0 \ldots 7)$, that are used to describe privilege condition for each process. The program action is a disjunction of eight possible cases, including: 1) process 0 is privileged, x_0 is assigned new value and states of other processes remain unchanged; 2) process 1 is privileged, x_1 is assigned new value and states of other processes remains unchanged; and so on. Once again, the modeling captures non-deterministic execution one of the privileged process.

Verification results of Ghosh's mutual protocol. We verified the Ghosh's mutual protocol for $n = 2i$ where $4 \leq i \leq 10$ or $i = 25, 50$. The verification results are shown in Table 5.

Execution time(s)									
Numbers of Processes	8	10	12	14	16	18	20	50	100
Unfair	0	0	0	0	0.02	0.02	0.03	0.35	4.77
Weakly-fair	0.4	2.93	22.43	138.05	693.27	2819.05	N/A	N/A	N/A
Approximate state space	10^2	10^3	10^3	10^4	10^4	10^5	10^6	10^{16}	10^{31}
Size of BDD (unfair)	1099	1811	2831	4153	5813	7847	10000	10134	10288
Size of BDD (weakly fair)	10082	10483	11693	20786	43294	99088	N/A	N/A	N/A

Table 5: Verification Results for Ghosh's Mutual Exclusion Program

4.1.3. Case Study 3: Hoepman's Uniform Ring-orientation Program (Cont'd)

In this section, we consider modeling Hoepman's uniform ring program under unfair computation in SMV.

Verification results of Hoepman's uniform ring program. We verified the Hoepman's uniform ring program for $n = 2i + 1$ where $1 \leq i \leq 4$ and $i = 50, 100, 150$ and 200. The verification results for this case are shown in Table 6. We compare the results with the corresponding results of the modeling approach under weak computation. The comparison results show the scalability of modeling self-stabilizing program under unfair computation for Hoepman's uniform ring program.

Execution time(s)									
Numbers of Processes	3	5	7	9	51	101	201	301	303
Unfair	0	0.03	0.08	0.12	11.95	95.65	875.23	3420.98	N/A
Weakly-fair	0.17	18.23	1113.77	N/A	N/A	N/A	N/A	N/A	N/A
Approximate state space	10^1	10^3	10^4	10^5	10^{31}	10^{60}	10^{121}	10^{181}	-
Size of BDD (under unfair)	4090	10047	11680	10543	63468	125809	324420	726620	-
Size of BDD (weakly fair)	10425	61570	776284	N/A	N/A	N/A	N/A	N/A	-

Table 6: Verification Results for Hoepman's Ring-orientation Program on Odd-length Ring

4.2. Analysis

Based on the results in Tables 4, 5 and 6, verification of significantly large system is possible if we consider unfair computations. As an illustration, consider

K-state program: It was possible to achieve verification for $K = 50$ in less than 1 hour. And, in this case, the corresponding state space is 10^{85}. By contrast, verification with weak fairness could not complete when state space was 10^{11}. This difference in performance can be explained by observing the size of OBDDs representing state sets in the forward search.

Compared with the approach in [1], the approach presented here has the OBDD size for the reached states running linearly. The overall time complexity for this approach is $O(n^3)$. As introduced in [26], this performance dues to three factors: a linear increase in the transition relation OBDD, a linear increase in the state set OBDD, and a linear increase in the number of iterations required for successful verification.

5. Approach 2: Decomposition

The results in Section 3 show that verification of self-stabilization under unfair computation is substantially faster than that under weakly-fair computation. Thus, the natural question is what can a designer do if the program at hand requires weak fairness to provide self-stabilization, i.e., the program is not self-stabilizing under unfair computations. Examples of such programs include [15, 7, 16].

Generally, there are two approaches that can be used when model checking fails due to lack of sufficient time or space. The first approach is to require the designer to perform extra work (e.g., abstraction, decomposition, identifying partial order reductions, etc.) that will reduce the cost of verification. The second approach is to verify a variation of the model/property such that the variation will still provide a reasonable assurance about the goal at hand.

In this section, we focus on the first such approach where we show that decomposition of a self-stabilizing program can provide substantial benefit in reducing the cost of verification. We note that while decomposition is one of the approaches in reducing cost of verification, the effect of this approach in model checking of self-stabilizing programs is not addressed. To address this limitation, in this approach, designer needs to partition the program into components such that each component satisfies its property without fairness. Subsequently, we can use existing composition results to show that their composition is correct under fair execution. There are several such approaches to show that the composed program is self-stabilizing based on the properties of individual components. Since the focus of this paper is not to identify new strategies of interference freedom, we only consider some of the simple and commonly used approaches and describe them, next. We note, however, that the subsequent discussion also applies to other approaches [14, 10] for proving self-stabilization of a program that consists of several components.

Let $C1$ and $C2$ be two components (programs) such that variables of $C1$ and $C2$ are disjoint. Let p be the program obtained by combining the actions of $C1$ and $C2$. Let legitimate states of $C1$ and $C2$ be $I1$ and $I2$ respectively. Let $C1[]C2$ denote *composition* of $C1$ and $C2$ where composition of two (or more) components is the program obtained by taking union of actions of components. Then, the well-known and simple theorem about the composition is as follows:

Theorem 1. *If*

- $C1$ *is weakly-fair stabilizing for* $I1$
- $C2$ *is weakly-fair stabilizing for* $I2$

Then

- $C1[]C2$ *is weakly-fair stabilizing for* $I1 \wedge I2$*.*

Although straightforward, this theorem can assist in reducing verification time if the fairness requirement is needed essentially to ensure that both components get a chance to execute. In other words, if the components themselves are self-stabilizing under unfair computations then the designer can verify the preconditions of this theorem easily under unfair computation. Self-stabilization under unfair computations implies self-stabilization under weak fairness. Hence, preconditions of the theorem can be proven easily. Moreover, the conclusion of the theorem allows us to ensure that the composed program is self-stabilizing under weak fairness.

There are several such theorems that provide the ability to conclude self-stabilization property of the composed program by self-stabilization property of the components. Another well-known theorem relates to superposition where program p consists of two components $C1$ and $C2$, where $C1$ is superposed on $C2$. In other words, $C1$ can only read the variables of $C2$ and $C2$ can neither read nor write variables of $C1$. Then, the well-known theorem about superposition is as follows:

Theorem 2. *If*

- $C1$ *is weakly-fair stabilizing for* $I1$

- *$C2$ is weakly-fair stabilizing for $I2$*
- *After a state in $I2$ is reached, no action in $C2$ is enabled*

Then

- *$C1[]C2$ is weakly-fair stabilizing for $I1 \wedge I2$, where $C1[]C2$ is the program obtained by taking union of actions of $C1$ and $C2$.*

Again, similar to the approach above, we may be able to verify each component without fairness assumption. However, fairness is required for the composed program to ensure that each component gets a chance to execute. Again, this will allow us to conclude correctness of the composed program under weak fairness by expediting the verification time for individual components.

There are several instances where such superposition or variations thereof are used. In particular, one variation is that it suffices if $C1$ ensure convergence to $I1$ by assuming that $I2$ holds already. Also, the third condition (termination of $C2$) can also be replaced by other non-interference conditions that are less restrictive. Next, we discuss some of these examples.

5.1. Case Study 4: Huang's Mutual Exclusion in Uniform Rings

In [16], authors propose a self-stabilizing mutual exclusion program that consists of two components: (1) leader election component and (2) token circulation component. The first component consists of a leader election program on an oriented uniform ring where the number of processes is prime. The second component consists of a token circulation component that requires a unique process (such as process 0 in Case Study 1.) Since verification of the second component is similar to that in Section 4.1.1, we only focus on the leader election component.

The leader election component maintains a variable $v.j$ at every process j. The actions of the processes are as follows ($left$ and $right$ denote the left and right neighbor of process j in the ring):

$$
\begin{array}{lll}
K_1:: & v.left = v.j = v.right & \longrightarrow v.j = (v.j + 1)\ mod\ n; \\
K_2:: & gap^1(left, j) < gap(j, right) & \longrightarrow v.j = (v.j + 1)\ mod\ n;
\end{array}
$$

1

$$
gab(a, b) = \begin{cases} n & \text{if } a = b \\ (b - a)\ mod\ n & \text{otherwise} \end{cases}
$$

Execution time(s)							
Number of Processes	3	5	7	11	23	29	31
$Unfair_{leader\ election}$	0	0	0.05	0.48	47.12	271.05	704.48
$Unfair_{token\ circulation}$	0	0	0.02	0.15	14.57	70.18	103.8
$Unfair_{total\ with\ decomposition}$	0	0	0.07	0.63	61.69	341.23	808.28
$Weakly-fair_{leader\ election}$	0	4.15	N/A	N/A	N/A	N/A	N/A
$Weakly-fair_{token\ circulation}$	0	0.63	-	-	-	-	-
$Weakly-fair_{total\ with\ decomposition}$	0	4.79	N/A	N/A	N/A	N/A	N/A
$Weakly-fair_{total\ without\ decomposition}$	not Self-Stabilization under this model						
$Weakly-fair_{total\ without\ decomposition}$	0.17	N/A	N/A	N/A	N/A	N/A	N/A
Approximate state space	10^4	10^{10}	10^{17}	10^{34}	10^{93}	10^{127}	10^{138}

Table 7: Verification Results for Huang's Mutual Exclusion Program

The above actions require that a process increments its value if either (1) its value equals that of its left and its right neighbor or (2) gap with the left process is less than the gap with the right process.

This program requires fairness for correctness. Without fairness, leader election component may not be able to execute. However, each component can be verified separately without fairness. Finally, based on Theorem 2, we can conclude that the overall program is self-stabilizing under weak fairness. Table 7 gives the verification performance by utilizing symbolic model checking procedure for verification of leader election component. From this table, we can see that verification of self-stabilization is significantly more scalable with decomposition and the use of unfair computation for verifying self-stabilization of each component. Moreover, the significant benefit in reduction in time is based on the use of unfair scheduler as opposed to the use of decomposition. (In Table 7, '-' denotes that the experiment was not performed for token circulation since the corresponding experiment for leader election could not be completed in the permissible time.)

5.2. Case Study 5: Self-stabilizing Program based on Raymond's Tree algorithm

This second example is the self-stabilizing program based on Raymond's tree algorithm for mutual exclusion [23]. In this program, the processes are arranged in a *fixed* [2] tree, called the parent tree. On this fixed tree, a dynamic *holder* tree is superposed such that the holder of a process is one of its tree neighbors (including itself). A process j has a token iff its holder $(h.j)$ equals j. There is one action that allows a process to send the token to its neighbors ($K_{passing}$). In this action, if process k has a token then it can pass it to its neighbor j by changing the holder

[2]By fixed, we mean that $p.j$ is fixed and hard coded in the actions themselves and, hence, cannot be corrupted.

Execution time(s)				
Number of Processes	7	15	31	63
$Unfair_{convergence}$	0.02	0.10	1.95	N/A
$Unfair_{passing}$	0	0.10	39.97	N/A
$Weakly-fair_{convergence}$	0.1	17.67	N/A	N/A
Approximate State Space	10^5	10^{17}	10^{46}	10^{113}

Table 8: Verification Results for Self-stabilizing Mutual Exclusion based on [23]

relation of j and k. Additionally, there are three convergence actions. The first action ensures that the holder of a process is a tree neighbor. The second action ensures that on any edge between j and $(p.j)$, either holder of j is same as $p.j$ or the holder of $p.j$ is j. And, the third action ensures that holder relation does not have cycles. Thus, the actions of the self-stabilizing tree program are as shown next:

$$
\begin{array}{lll}
K_{passing}:: & h.k = k \wedge h.j = k & \longrightarrow h.j = j, h.k = j; \\
K_{convergence}:: & h.j \neq NBR.j \cup j & \longrightarrow h.j = p.j; \\
& j \neq p.j \wedge h.j \neq p.j \wedge h.(p.j) \neq j & \longrightarrow h.j = p.j; \\
& j \neq p.j \wedge h.j = p.j \wedge h.(p.j) = j & \longrightarrow h.(p.j) = p.j;
\end{array}
$$

This program requires fairness for stabilization; without fairness, processes could simply execute the token passing action ($K_{passing}$) thereby preventing stabilization. However, correctness of the convergence actions and the correctness of closure actions can be independently verified without fairness. Furthermore, the results from [18] can be used to show that these two components do not interfere and, hence, the overall program is self-stabilizing. Table 8 gives the verification time for each component under unfair scheduler. It also gives verification time for convergence under weakly-fair scheduler. Since the token passing component changes the variables from two different processes, we were not able to implement it under weakly-fair scheduler. However, it is straightforward to observe that the time for verification of the composed program (with token passing and convergence actions) will be more than the time for verification with convergence actions alone. Hence, the benefit of decomposition and use of unfair scheduler in reducing the cost of verification follows from the results in Table 8.

5.3. Other Examples and Approaches for Identifying Components

Another example is that of distributed reset [7] where the program consists of a tree layer and a wave layer. The tree layer constructs a tree from the processes that are still up. Subsequently, the wave layer utilizes this tree to achieve distributed reset. Again, weak fairness ensures that each component can always execute although the component itself can be verified without fairness.

For the case where decomposition is not straightforward the proof of stabilization can assist in identifying the desired decomposition. Specifically, one common way to prove self-stabilization is to use the approach of Gouda and Multari [17]. Specifically, in this approach, the state space itself is partitioned into concentric circles, R_0, R_1, $\ldots$, R_n, where R_0 corresponds to the entire state space, R_n corresponds to the set of legitimate states and $R_i \supset R_j$ if $0 < i < j < n$. It is required that if the program starts in any state in R_i, $0 \leq i < n$ then (1) it always stays in states in R_i, and (2) it eventually reaches a state in R_{i+1}. Again, fairness can assist in this approach in ensuring that the overall program is self-stabilizing although one or more convergence requirements can be verified without fairness thereby reducing the time for verification.

As an example, each recovery action in the case study 5 is responsible for fixing certain constraints in the program. In particular, this program forms a set of concentric circles as follows: (1) $R_0 = true$, (2) $R_1 = h.j \in \{j, p.j\} \cup children.j$, (3) $R_2 = R_1 \wedge (j \neq p.j \Rightarrow (h.j = p.j \vee h.(p.j) = j)$, and (4) $R_3 = R_2 \wedge (j \neq p.j \Rightarrow \neg(h.j = p.j \wedge h.(p.j) = j)$. We note that the time for convergence of each of these steps under unfair computation is orders of magnitude less than the corresponding cost of verifying the program consisting of all three actions under weak fairness.

6. Approach 3: Utilizing Weak Stabilization

In this section, we focus on the second approach for improving scalability for verification of self-stabilizing programs. Specifically, if the program at hand requires weakly-fair computations to provide self-stabilization and the time for such verification is prohibitive, the designer can focus on a variation of self-stabilization, namely weak self-stabilization [24]. In [24], Gouda has shown that weak stabilization is a 'good approximation' of stabilization. Furthermore, in [25], Devismes et al have shown how to transform a weak-stabilizing program into a probabilistic stabilizing program. Thus, if the assurance of self-stabilization is not possible, the designer can obtain a slightly lower assurance provided by weak stabilization. Moreover, the designer can utilize the transformation in [25] to obtain probabilistic assurance regarding self-stabilization. Next, we recall the definition of weak stabilization and a relevant theorem about it from [24].

Definition 10. (Weak stabilization) Let p be a program and let I be a state predicate of p. We say that p is weakly stabilizing for I iff:

1. closure: if (s_0, s_1) is a transition of p and $s_0 \in I$, then $s_1 \in I$;
2. weak convergence: for every state s, there is a computation of p that starts at s and reaches I. ∎

Definition 11. Strongly-fair computation $\sigma = \langle s_0, s_1, ...\rangle$ is strongly-fair computation iff:

1. σ is an unfair computation of p, and
2. If any state s is included infinitely often in σ and (s, s') is a transition of p then the subsequence $\langle s, s'\rangle$ must be included infinitely often in σ. ∎

Theorem 3. *A weakly-stabilizing system is also a self-stabilizing system if:*

1. *The system has a finite number of states, and*
2. *Every computation is under strong fairness.*

We can utilize the above result as follows: If we cannot verify that p is self-stabilizing due to time/space limitations, we can verify that p is weakly stabilizing. By Definition 10, this does not require one to model fairness explicitly. This will allow us to obtain some assurance (although somewhat weaker) about p. Additionally, the designer can utilize the transformation from [25] to obtain program p' that is probabilistically stabilizing. The result from [25] argues that all the deterministic weak stabilizing programs can automatically be turned into probabilistic ones if we assume the scheduling is probabilistic. The assumption about scheduler is indeed the case for practical purposes. This result hints at more practical use of weak stabilization since the transformation approach removes the burden of designing and proving probabilistic stabilization for designers, leaving them with easier task of designing and verifying weak stabilizing programs.

Next, we revisit case studies 1-5 to evaluate the cost of verifying weak stabilization. In verification of weak stabilization, we use a specification "AG EF *legitimate*" to denote *weak convergence* where *legitimate* denotes the convergence constraint, EF p denotes p is eventually true in some computation path and AG q denotes q is infinitely true in every computation path. Because weak stabilization only requires that convergence holds in some computation path not in all the computation paths, it is reasonable that verification of weak stabilization is faster than that of stabilization where the latter requires convergence holds in all the computation paths. Table 9 compares the cost of verifying self-stabilization under weak fairness with that of verifying weak stabilization. As we can see, the cost of verifying weak stabilization is substantially less (and is very close to the cost of verifying self-stabilization without fairness). Also, verification of weak stabilization is significantly more scalable than that of self-stabilization. For example, in Dijkstra's K-state program, it was possible to verify self-stabilization for only 9 processes (state space 10^8) whereas it was possible to verify weak stabilization for 50 processes (state space 10^{84}).

7. Related work

Formal verification of self-stabilization has been studied mainly in two disjoint directions. One is mechanism theorem proving and the other one is model checking. While mechanism theorem proving is very powerful (especially in the infinite system) [19, 11, 20], it's very hard for those who do not have considerable experience in logic reasoning. Model checking is the easiest way but limited to the state explosion problem. However, with the advancement of model checking techniques, researchers try to use OBDD to represent the state, that is, symbolic model checking, to overcome this limitation. Benefits from this, the work in [1] show verification based on symbolic model checking is feasible for the systems with a small number of processes. To solve this bottleneck, our work shows that verification (based on symbolic model checking) for unfair computation is significantly faster than that for weak-fair computation if weak fair computation is not required for the correctness of self stabilization. Thus, it is possible to verify much larger systems if we consider unfair computation.

One may wonder whether the same results would apply if one used a model checker such as SPIN [22] that uses explicit state space. We did not pursue this question in detail in this paper. However, we note that in [1] authors have shown that SPIN is unable to verify self-stabilization except for a small set of processes. We note that their observation is valid with or without fairness.

Recently the research of SAT(Boolean satisfiability) based model checker has emerged as a viable alternative to BDDs based model checker. Due to recent advances in tools that provide SAT based model checking, verification of self-stabilization based on SAT should be meaningful. In our future work, we plan to study cases modeled in SAT based model checker.

(a) Verification results for the K-state program

Execution time(s)									
Number of Processes	3	4	5	6	7	8	9	10	50
Weak stabilization	0	0	0	0	0.02	0.03	0.05	0.08	3485.27
Stabilization under weak fairness	0	0.03	0.63	5.33	34.30	139.10	1276.08	N/A	N/A
Approximate state space	10^1	10^2	10^3	10^4	10^5	10^7	10^8	10^{10}	10^{84}

(b) Verification results for Ghosh's mutual exclusion program

Execution time(s)									
Number of Processes	8	10	12	14	16	18	20	50	100
Weak stabilization	0	0	0	0	0	0.02	0.03	0.35	4.9
Stabilization under weak fairness	0.4	2.93	22.43	138.05	693.27	2819.05	N/A	N/A	N/A
Approximate state space	10^2	10^3	10^3	10^4	10^4	10^5	10^6	10^{16}	10^{31}

(c) Verification results for Hoepman's ring-orientation program on odd-length ring

Execution time(s)									
Number of Processes	3	5	7	9	51	101	201	301	401
Weak stabilization	0	0.037	0.08	0.13	11.88	95.9	881.5	3442.18	N/A
Stabilization under weak fairness	0.17	18.23	1113.77	N/A	N/A	N/A	N/A	N/A	N/A
Approximate state space	10^1	10^3	10^4	10^5	10^{31}	10^{60}	10^{121}	10^{181}	10^{241}

(d) Verification results for Huang's mutual exclusion program for uniform rings

Execution time(s)				
Number of Processes	3	5	7	11
Weak stabilization	0	0.07	0.63	N/A
Stabilization under weak fairness	0.17	N/A	N/A	N/A
Approximate state space	10^4	10^{10}	10^{17}	10^{34}

(e) Verification results for self-stabilizing mutual exclusion based on Raymond tree[23]

Execution time(s)				
Number of Processes	7	15	31	63
Weak stabilization	0	0.12	2.15	N/A
$Weakly - fair_{convergence}$	0.1	17.67	N/A	N/A
Approximate state space	10^{11}	10^{35}	10^{92}	10^{226}

Table 9: Cost of verifying weak stabilization vs. Cost of verifying self-stabilization under weak fairness

8. Conclusion

In this paper, we focused on scalable model checking of self-stabilizing algorithms. While a significant percentage of the literature on self-stabilization routinely assumes weak fairness, where if an action is continuously enabled, it is guaranteed to be executed, we argued that verification under such weak fairness is not scalable. Our observation was that in many cases, the assumption of weak fairness is superfluous. And, in these cases, scalable verification of self-stabilization is possible under unfair computation model. We illustrated this in the context of three case studies, Dijkstra's K-state program, Ghosh's mutual exclusion program and Hoepman's ring-orientation program. In particular, we showed that the time for verification with unfair computations is approximately $0.001\% - 0.1\%$ of that for weakly-fair computations.

For the case where program cannot preserve self-stabilization property under unfair computations, we argued that the designers can utilize two approaches. The first approach is to decompose the program into parts where each part can be verified under unfair computation. Subsequently, composition of these parts can be proved to be correct using existing theorems in the literature. While the idea of such decomposition has been considered to be useful, its effect on model checking of self-stabilizing programs has not been considered. Moreover, as we showed in this paper, the benefit obtained in this approach relies on both *the ability to decompose* and *ability to utilize unfair computations*. An approach that relies only on one of them provides limited benefit.

We showed how this approach can be used in the context of two case studies, Huang's mutual exclusion program and self-stabilizing mutual exclusion program based on Raymond's tree algorithm. In both case studies, scalability of verification increased substantially (e.g., from 10^4 states to 10^{138} states for Huang's mutual exclusion algorithm.) Also, the approach in [17] can be used to identify layers that assist in self-stabilization. These layers, in turn, can form the components that one can verify independently. Since most of reduction in time is obtained by the use of unfair scheduler, one can obtain the savings in this manner even if the components themselves are not substantially smaller than the original program.

The second approach is to utilize weak stabilization that has been proved to be a reasonable implementation of stabilization [24]. We also showed that verification of weak stabilization is substantially more scalable. This validates the *suggestion* in [24] that weak stabilization is easier to verify than self-stabilization. Specifically, when weak fairness is essential to verify correctness of a stabilizing protocol, the time for verification of weak stabilization is orders of magnitude less than that of verifying stabilization. Furthermore, a weak stabilizing program can be transformed into a probabilistically stabilizing program thereby providing additional assurance to designer. Hence, this approach can be used when verification of stabilization cannot be achieved due to prohibitive cost.

Our work also repudiates the suggestion in [24] that verification of weak stabilization is easier than that of stabilization. Specifically, for the case where we compare the cost of verification under unfair computation, the cost of verifying weak stabilization is comparable to that of stabilization. This result was especially surprising since weak stabilization guarantees that from every state, there is a path to a legitimate state. By contrast, stabilization ensures that every path reaches a legitimate state.

To our knowledge, this is the first paper that has shown feasibility of verifying case studies 4 and 5. Also, it is the first paper that has shown feasibility of verifying the first three case studies with large number of processes. Thus, the results in this paper provide several avenues to designers of self-stabilizing systems to verify correctness of their programs or to identify bugs. Also, while techniques such as decomposition are well-known techniques for reducing the cost of verification, their effect on automatic verification of self-stabilizing programs has not been considered in the literature. This paper shows the feasibility of this approach.

In future, we intend to provide a simplified tool that will allow designers to specify programs in guarded commands and utilize verification under different levels of fairness. Also, in [21], authors propose new techniques for verification under different levels of fairness. This work is based on SPIN [22] and utilizes explicit state space. They show that with their approach, the verification cost is approximately 12% of that for original time. While this benefit is substantially less than one where one utilizes unfair computations, one future work is to apply this in the context of symbolic verification of self-stabilizing programs for verification under weakly-fair computations.

[1] Tsuchiya, Tatsuhiro and Nagano, Shin'ichi and Paidi, Rohayu Bt and Kikuno, Tohru: Symbolic Model Checking for Self-Stabilizing Algorithms. IEEE Trans. Parallel Distrib. Syst. 12, 81–95 (2001)

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

[3] Ghosh, S.: Binary Self-stabilization in Distributed Systems. Information Processing Letter, vol.40, no.3, 153-159, 1991.

[4] Hoepman, J.H.: Uniform Deterministic Self-Stabilizing Ring-Orientation on Odd-Length Rings. Proc. 8th Int'l workshop on Distributed Algorithms, pp.265-279, 1994.

[5] Dolev, S.: Self-Stabilizing Routing and Related Protocols.Journal of Parallel and Distributed Computing, Volume 42, Number 2, April 1997 , pp. 122-127(6).

[6] Ghosh, Sukumar and Gupta, arobinda: An exercise in fault-containment: Self-stabilizing leader election. Information Processing Letters, pp. 281-288, 1996.

[7] Arora,A. and Gouda, M.: Distributed Reset, IEEE Transactions on Computers, pp. 1026-1038, September, 1994

[8] McMillan, K. L.: Symbolic Model Checking. Kluwer Academic, 1993.

[9] Bryant, Randal E., Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers, v.35 n.8, p.677-691, Aug. 1986

[10] Dolev, Shlomi: Self-Stabilization, MIT Press, ISBN 0-262-04178-2, 2000.

[11] Qadeer, S. and Shankar, N.: Verifying a self-stabilizing mutual exclusion algorithm. In David Gries and Willem-Paulde Roever, editors, IFIP International Conference on Programming Concepts and Methods (PROCOMET' 98), pages 424-443, Shlter Island, NY, June 1998. Chapman & Hall.

[12] Clarke, E. M. and Grumberg, O.: Avoiding the state explosion problem in temporal logic model checking. Proceedings of the sixth annual ACM Symposium on Principles of distributed computing, Pages: 294-303, 1987.

[13] Arora, Anish and Gouda, Mohamed: Closure and Convergence: A Foundation of Fault-Tolerant Computing. IEEE Transactions on Software Engineering, v.19, issure 11, p. 1015-1027, 1993.

[14] Ghosh, Sukumar: Distributed Systems: An Algorithmic Approach, 2006 CRC Press (ISBN 1584885645).

[15] Arora, A. and Kulkarni, S. S.:Designing masking fault-tolerance via nonmasking fault-tolerance, IEEE Transactions on Software Engineering, vol 24(6), pp 435-450, June 1998.

[16] Huang, Shing-Tssan: Leader election in uniform rings, ACM Trans. Program. Lang. Syst., v.15, 1993.

[17] Gouda, Mohamed G. and Multari, Nicholas J.: Stabilizing Communication Protocols, IEEE Trans. Computers, v.40, 1991.

[18] Abujarad, Fuad and Kulkarni, Sandeep: Constraint Based Automated Synthesis of Nonmasking and Stabilizing Fault-Tolerance, Proceedings of the 2009 28th IEEE International symposium on Reliable Distributed Systems, Pages:119-128,2009.

[19] Prasetya, I.S.W.B.: Mechanically Verified Self-Stabilizing Hierarchical Algorithms, Proceedings of the Third International Workshop on Tools and Algorithms for Construction and Analysis of Systems, Pages: 399-415, 1997.

[20] Kulkarni, S.S. and Rushby, John M. and Shankar Natarajan: A case-study in component-based mechanical verification of fault-tolerant programs, Workshop on Self-stabilizing System, Pages: 33-40, 1999.

[21] Sun, Jun and Liu, Yang and Dong, Jinsong and Pang, Jun: PAT: Towards Flexible Verification under Fairness, Proceedings of the 21st International Conference on Computer Aided Verification, Grenoble, France, Pages: 709-714, June 26 - July 2, 2009.

[22] Holzmann, G.J.: The model checker SPIN, IEEE Trans. Software Eng., vol. 23, no.5, pp.279-295, May 1997.

[23] Raymond, Kerry: A tree-based algorithm for distributed mutual exclusion, ACM Transactions on Computer Systems (TOCS), vol.7, pages 61-77, 1989.

[24] Gouda, Mohamed G.: The Theory of Weak Stabilization, WSS ’01: Proceedings of the 5th International Workshop on Self-Stabilizing Systems, 2001.

[25] Devismes, Stéphane and Tixeuil, Sébastien and Yamashita, Masafumi: Weak vs. Self vs. Probabilistic Stabilization, ICDCS ’08: Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems, pages 681–688, 2008.

[26] K.L. McMillan: The SMV system for SMV version 2.5.4, Novemeber 6, 2000.

[27] E. W. Dijkstra, A Discipline of Programming, Prentice-Hall, 1990, New Jersey, USA.