Active Stabilization*

Borzoo Bonakdarpour[1] and Sandeep S. Kulkarni[2]

[1] School of Computer Science
University of Waterloo
Waterloo N2L 3G1, Canada
Email: borzoo@cs.uwaterloo.ca

[2] Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, USA
Email: sandeep@cse.msu.edu

Abstract. We propose the notion of *active stabilization* for computing systems. Unlike typical stabilizing programs (called passive stabilizing in this paper) that require that the faults are absent for a long enough time for the system to recover to legitimate states, active stabilizing programs ensure recovery in spite of constant perturbation during the recovery process by an adversary. We identify the relation between active and passive stabilization in terms of their behavior and by comparing their cost of verification. We propose a method for designing active stabilizing programs by a collection of passive stabilizing programs. Finally, we compare active stabilization with fault-contained stabilization and stabilization in the presence of Byzantine faults.

Keywords: Self-stabilization, reactive systems, adversary, formal methods

1 Introduction

A *self-stabilizing* system [6] ensures that it will recover to a legitimate state even if it starts executing from an arbitrary state. For this reason, self-stabilization is often utilized to provide recovery from unexpected transient errors. A typical self-stabilizing protocol in the literature considers the case where *faults* perturb the system to an arbitrary state. Subsequently, the goal of the protocol is to ensure that the system will recover to a legitimate state with the assumption that no additional faults will occur. Moreover, by the nature of self-stabilization, even if faults occur during recovery, the system will still recover to a legitimate state as long as faults do not occur forever (or they stop for a long enough time).

* This work is partially sponsored by Canada NSERC DG 357121-2008, ORF RE03-045, ORE RE-04-036, and ISOP IS09-06-037 grants, and, by USA AFOSR FA9550-10-1-0178 and NSF CNS 0914913 grants.

We call such self-stabilization as *passive* stabilization since faults play a dormant role during the recovery process.

In this paper, we introduce the concept of *active stabilization*. To illustrate the motivation for active stabilization, we begin with the problem of pursuer-evader games [4]. The intuitive description of one instance of this problem in the context of sensor networks is as follows: The system consists of a set of computing nodes with sensors (called just sensors from in the subsequent discussion). Additionally, the system contains one (or more) pursuers and one (or more) evaders. The sensors' task is to organize themselves in a structure that will facilitate the capture of the evader. For example, one approach to achieve this is to have the sensors form a tree among themselves that is rooted at the location of the evader. The goal of the pursuer is to utilize this structure to capture the evader.

In such a system, there can be several faults. For example, the state of sensors could be corrupted due to false positive and/or false negative readings. Moreover, communication errors, errors in initialization etc. may perturb the sensor network to an arbitrary state. It is anticipated that such faults are rare and, hence, we can utilize passive self-stabilization for dealing with such faults; i.e., we can assume that faults stop for a long enough time for the sensor network to stabilize to a legitimate state. However, the network could also be perturbed by the evader itself. In particular, if the goal of the sensor network is to have a tree rooted at the evader, then the evader movement is tantamount to perturbation of the network. Moreover, it may be unreasonable to assume that these faults eventually stop or that they stop for a long enough time since the evader is actively trying to perturb the system so that it does not stabilize.

We can view three different contributing factors in such a scenario: (1) the system, (2) the faults, and (3) the *adversary*. In particular, the system actions are responsible for ensuring recovery to legitimate states. The faults are events that perturb the system randomly and rarely. It is anticipated that the faults could perturb the system to an arbitrary state, thereby requiring self-stabilization. However, because these events are rare, one can assume that they stop for a sufficiently long enough time to allow the system to recover. The adversary is *actively* attempting to prevent self-stabilization. However, unlike faults, the adversary may not be able to perturb the system to an arbitrary state. For example, in the above scenario, the evader would be able to move within the vicinity of its original location; i.e., it would not be able to move to a random location from its initial location. Also, unlike faults, the adversary actions may never stop. In particular, it would be unreasonable to assume that the adversary actions stop for a long enough time for the system to stabilize.

Although it is unreasonable to assume that adversary actions would stop for a long enough time, it is necessary to assume some fairness for the system actions. In particular, in the pursuer-evader example, it is anticipated that the evader movement is limited by laws of physics and, hence, the system can take a certain number (≥ 1) of steps between two steps of the evader.

The main contributions of the paper are as follows:

– We formally define different variations of active stabilization and relate it to passive stabilization. In particular, we consider the cases where the adversary (1) cannot lead the system to illegitimate states (called active stabilization), (2) can perturb the program outside the legitimate states (called *fragile* active stabilization), and (3) can perturb the program outside the legitimate states from where the program can recover before the adversary takes another step (called *contained* active stabilization)
– We study the relation between different types of active stabilization.
– We compare the cost of automated verification of active and passive stabilization.
– The problem in designing an active stabilizing program lies in the fact that an adversary can disrupt the progress made by the program towards recovering to the invariant. Thus, we propose an approach for designing active stabilizing programs based on the convergence stair [10].
– Finally, we argue that active stabilization is a powerful and expressive concept by presenting comparison to fault-contained stabilization [8] and Byzantine self-stabilization [14]. In particular, we show that if a program is contained active stabilizing, then it is fault-contained stabilizing. We also show that a special type of active stabilization in the presence of Byzantine processes is Byzantine self-stabilization.

Organization of the paper. In Section 2, we introduce the notion of active stabilization. We compare different types of passive and active stabilization in Section 3. The complexity of automated formal methods for active stabilization is analyzed in Section 4. Section 5 discusses design methodology for active stabilizing programs. We compare active stabilization with fault-contained stabilization and Byzantine self-stabilization in Section 6. Finally, we discuss related work in Section 7 and conclude in Section 8.

2 The Concept of Active Stabilization

A traditional modeling of programs in the literature on self-stabilization includes a finite set of variables with (finite or infinite) domain. Additionally, it includes guarded commands [7] that update those program variables. Since these internals of the program are not relevant in our definition of active stabilization, in our work, we define program p in terms of its state space S_p and its transitions $\delta_p \subseteq S_p \times S_p$. Intuitively, the state space can be obtained by assigning each variable in p a value from its domain.

Definition 1 (Program). *A* program *p is of the form $\langle S_p, \delta_p \rangle$ where S_p is the state space of program p and $\delta_p \subseteq S_p \times S_p$.*

Assumption 1 *For simplicity of definitions, we assume that program p has at least one outgoing transition from every state in S_p. If such a transition does not exist for some state, say s, then we consider the program where transition (s, s) is added. While this assumption simplifies subsequent definitions since we* *do not need to consider* terminating *behavior of a program explicitly, it is not restrictive in any way.*

Definition 2 (State Predicate). *A* state predicate *of p is any subset of S_p.*

Definition 3 (Closure). *A state predicate S of $p = \langle S_p, \delta_p \rangle$ is* closed *in p iff $\forall s_0, s_1 \in S_p :: (s_0 \in S \wedge (s_0, s_1) \in \delta_p) \;\Rightarrow\; (s_1 \in S)$.*

Definition 4 (Faults). *We define* faults *for program $p = \langle S_p, \delta_p \rangle$ to be $S_p \times S_p$; i.e., the faults can perturb the program to any arbitrary state.*

The *adversary* for program, say *adv*, is defined in terms of its transitions, say $a_p \subseteq S_p \times S_p$. Note that, based on the discussion in the introduction, this allows us to model the limited set of actions the adversary may be allowed to execute.

Definition 5 (Adversary). *We define an* adversary *for program $p = \langle S_p, \delta_p \rangle$ to be a subset of $S_p \times S_p$.*

Next, we define a computation of the program, say p, in the presence of adversary, say *adv*.

Definition 6 ($\langle p, adv, k \rangle$-computation). *Let p be a program with state space S_p and transitions δ_p. Let adv be an adversary for program p. And, let k be an integer greater than 1. We say that a sequence $\langle s_0, s_1, s_2, ... \rangle$ is a $\langle p, adv, k \rangle$-*computation *iff*

- $\forall j \geq 0 :: s_j \in S_p$, *and*
- $\forall j \geq 0 :: (s_j, s_{j+1}) \in \delta_p \cup adv$, *and*
- $\forall j \geq 0 :: ((s_j, s_{j+1}) \notin \delta_p) \quad \Rightarrow \quad (\forall l \mid j < l < j + k :: (s_l, s_{l+1}) \in \delta_p)$

Observe that a $\langle p, adv, k \rangle$-computation involves only the transitions of program p or its adversary *adv*. Moreover, the adversary is required to execute with fairness to the program; i.e., the program can take at least $k - 1$ steps between two adversary steps. This ensures that the adversary cannot simply block the program from executing, thereby make it impossible to provide recovery. However, the adversary is not required to execute and the program can execute forever.

Remark 1 (Fairness among program transitions 1). Since the focus of this paper is on the defining active stabilization based on the interaction between the program and the adversary, we omit the issue of fairness among program transitions themselves. Specifically, in some instances, we can consider the program to consist of multiple processes and require that each process executes with some fairness. In this instance, the above definition can be modified to add an additional constraint that identifies fairness conditions. For reasons of space, this issue is outside the scope of this paper.

Remark 2 (Round-based computations). The definition of $\langle p, adv, k\rangle$-computation is based on the number of *steps* that a program takes between two adversary steps. In scenarios where a program consists of multiple processes, a *round* [15] based notion is sometimes used. Intuitively, in one round, every process is given at least one chance to execute. (However, the process may not actually be able to execute a transition if it was given that chance when none of its transitions could be executed.) The definition of active stabilization can also be extended to handle such a case by using *rounds* instead of *steps*. This issue is also outside the scope of this paper.

Definition 7 (Active stabilization). *Let p be a program with state space S_p and transitions δ_p. Let adv be an adversary for program p. And, let k be an integer greater than 1. We say that program p is* strong k-active stabilizing *with adversary adv for* invariant S *iff*

- *S is closed in p*
- *S is closed in adv*
- *For any sequence σ $(=\langle s_0, s_1, s_2, ...\rangle$ $)$ if σ is a $\langle p, adv, k\rangle$-computation then there exists l such that $s_l \in S$.*

The definition of active stabilization requires that the invariant S be closed in the execution by the adversary; i.e., when the invariant S is reached, the adversary does not perturb the program outside S. In other words, only a fault can do so. This requirement can be difficult to satisfy in many programs. For such programs, we introduce the notion of *fragile active stabilization* where the adversary can perturb the program outside the invariant.

Definition 8 (Fragile Active Stabilization).

Let p be a program with state space S_p and transitions δ_p. Let adv be an adversary for program p. And, let k be an integer greater than 1. We say that program p is fragile k-active stabilizing *with adversary adv for invariant S iff*

- *S is closed in p*
- *For any sequence σ $(=\langle s_0, s_1, s_2, ...\rangle$ $)$ if σ is a $\langle p, adv, k\rangle$-computation then there exists l such that $s_l \in S$.*

Observe that if p is fragile k-active stabilizing with adversary *adv* for invariant S, then starting from an arbitrary state, p will reach a state in S even if adversary *adv* tries to disrupt it. Moreover, after the program reaches S, the adversary can still perturb it outside S. However, in subsequent computation, p is still guaranteed to reach a state in S again. Thus, a fragile active stabilizing program will reach the invariant infinitely often.

One issue with fragile active stabilization is that after the adversary perturbs the program from a state in S to a state outside S, there is no bound on how long it will take to return to S. Our notion of *contained active stabilizing programs* addresses this issue by requiring the program to recover to S quickly; i.e., before the adversary can perturb it again.

Definition 9 (Contained Active Stabilization). *Let p be a program with state space S_p and transitions δ_p. Let adv be an adversary for program p. And, let k be an integer greater than 1. We say that program p is* contained k-active stabilizing *with adversary adv for invariant S iff*

- *S is closed in p*
- *For any sequence σ ($=\langle s_0, s_1, s_2, ...\rangle$) if σ is a $\langle p, adv, k\rangle$-computation then there exists l such that $s_l \in S$.*
- *For any finite sequence α ($=\langle s_0, s_1, s_2, ...s_k\rangle$) if $s_0 \in S$, $(s_0, s_1) \in adv$ and $(\forall j : 0 < j < k : (s_j, s_{j+1}) \in \delta_p$ then $s_k \in S$.*

Finally, we also define the traditional notion of stabilization. Towards this end, we define pure-computations of p; i.e., computations where only p is allowed to execute. Then, we define the standard definition of stabilization, which in this paper we will call as *passive* stabilization.

Definition 10 (pure-computation). *Let p be a program with state space S_p and transitions δ_p. We say that a sequence $\langle s_0, s_1, s_2, ...\rangle$ is a* pure-computation *iff*

- $\forall j \geq 0 :: (s_j, s_{j+1}) \in \delta_p$

Remark 3 (Fairness among program transitions 2). Similar to Remark 1, the above definition can include fairness requirements.

Definition 11 (Passive stabilization). *Let p be a program with state space S_p and transitions δ_p. We say that program p is* (passive) stabilizing *for invariant S iff*

- *S is closed in p*
- *For any sequence σ ($=\langle s_0, s_1, s_2, ...\rangle$) if σ is a pure-computation then there exists l such that $s_l \in S$.*

3 Relation between Different Types of Active and Passive Stabilization

In this section, we study the conceptual relation between different types of stabilization presented in Section 2. In Subsection 3.1, we compare active stabilization with passive stabilization. Then, in Subsection 3.2, we compare different types of active stabilization.

3.1 Relation between Active and Passive Stabilization

In this section, we evaluate the relation between active stabilization and passive stabilization. In particular, we show that any active stabilizing program is also passive stabilizing. Moreover, we present necessary conditions under which passive stabilizing programs can be transformed into fragile and contained active stabilizing programs.

Theorem 1. *If there exists k and adv such that program p is k-active stabilizing with adversary adv for invariant S, then p is passive stabilizing for invariant S.*

Proof. Since p is k-active stabilizing with adversary adv for invariant S, every $\langle p, adv, k\rangle$-computation reaches a state in S. And, by definition, a pure-computation of p is also a $\langle p, adv, k\rangle$-computation. Thus, every pure-computation of p reaches a state in S (recall that a program can execute adversary transitions as well). Furthermore, by definition of active stabilization, S is closed in p. It follows that p is passive stabilizing for invariant S. $\square$

Theorem 2. *If program $p = \langle S_p, \delta_p\rangle$ is passive stabilizing for invariant S and S_p is finite, then there exists k such that for any adversary adv, p is fragile k-active stabilizing with adversary adv for invariant S.*

Proof. First, by definition of passive stabilization, S is closed in p. Hence, to prove this theorem, we only need to prove the second constraint in the definition of fragile active stabilization. To this end, we let $k = |S_p|$. Consider any $\langle p, adv, |S_p|\rangle$-computation. This computation includes a subsequence of size $|S_p| - 1$, say α that only includes transitions of p. If α includes any state, say s such that $s \notin S$ and s occurs twice in α then there is a pure-computation of p that starts from s and never reaches S. Thus, α does not include any state outside S more than once. Hence, by the pigeon hole principle, α contains at least one state in S. $\square$

Note that the above theorem does not hold if the state space of p is infinite. We can illustrate this using a the following simple example. Let the state space of p be the set $S_p = \mathbb{Z}_{\geq 0}$ of non-negative integers and the transitions of p be $\delta_p = \{(x+1, x) \mid x \in \mathbb{Z}_{\geq 0} \cup \{(0,0)\}$. Also, let the invariant of p be $S = \{0\}$. Clearly, p is passive stabilizing for invariant S. However, for adversary $N \times N$, the above theorem is not valid.

Also, note that the above theorem will be incorrect if we remove 'fragile' from the statement of the theorem. This is due to the fact that S may not be closed in the transitions of an arbitrary adversary.

Corollary 1. *If program $p = \langle S_p, \delta_p\rangle$ is passive stabilizing for invariant S and S_p is finite, then there exists k such that for any adversary adv, p is contained k-active stabilizing with adversary adv for invariant S.*

Finally, if a program is active stabilizing with some adversary, then the program is also active stabilizing with a *slower* adversary. Thus, we have the following theorem.

Theorem 3. *If program p is k-active stabilizing with adversary adv for invariant S and $l \geq k$, then p is l-active stabilizing with adversary adv for invariant S.*

3.2 Relation between Active, Fragile Active, and Contained Active Stabilization

Since ensuring the closure of invariant in the transitions of an adversary can be unrealistic, we introduced the notion of fragile and contained active stabilization. In this section, we show that the definition of contained active stabilization and active stabilization are exchangeable; i.e., given a program p that is contained k-active stabilizing, we can find a corresponding program p' that is k-active stabilizing and vice versa. Thus, these results show that instead of showing a program to be active stabilizing, one can show that the program is contained active stabilizing.

Definition 12 **($R^j_{(p,adv)}(S)$).** *Let S be a state predicate of program p and adv be an adversary for program p. We define $R^j_{(p,adv)}(S)$, where $j \geq 0$, as follows:*

- *if $j = 0$, then*
  $R^j_{(p,adv)}(S) = S \cup \{s_1 \mid \exists s_0 \in S : (s_0, s_1) \in adv\}$.
- *if $j > 0$, then*
  $R^j_{(p,adv)}(S) = \{s_1 \mid \exists s_0 \in R^{j-1}_{(p,adv)}(S) : (s_0, s_1) \in \delta_p\}$.

Definition 13. *Let p be a program with state space S_p and transitions δ_p. We define $count_p$ to be the program where:*

- *State space $S_{count_p} = S_p \times \mathbb{Z}_{\geq 0}$, and*
- *Transitions $\delta_{count_p} = \{(\langle s_0, j\rangle, \langle s_1, j+1\rangle) \mid (s_0, s_1) \in \delta_p\}$.*

Definition 14. *Let p be a program with state space S_p and transitions δ_p. Let adv be an adversary for program p. We define $count_{adv_k}$ to be the adversary for $count_p$ where:*

- *Transitions $= \{(\langle s_0, j\rangle, \langle s_1, 0\rangle) \mid ((s_0, s_1) \in adv) \ \wedge \ (j \geq k)\}$.*

Theorem 4. *If program p is contained k-active stabilizing with adversary adv for invariant S, then $count_p$ is k-active stabilizing with adversary $count_{adv_k}$ for invariant S' where:*

$$S' = \bigcup_{l=0}^{\infty} \{\langle s, l\rangle \mid s \in R^l_{(p,adv)}(S)\}$$

Proof. To prove the above theorem, we need to prove the three conditions in the definition of active stabilization:

1. S' is closed in $count_p$.
   Towards this end, we need to show that if $count_p$ executes in any state in S', then the resulting state would also be in S'. Let $\langle s_0, l\rangle$ be a state in S'. Hence, s_0 is included in $R^l_{(p,adv)}(S)$. By Definition 13, transition of $count_p$ is of the form $(\langle s_0, l\rangle, \langle s_1, l+1\rangle)$. Furthermore, if $(\langle s_0, l\rangle, \langle s_1, l+1\rangle)$ is a transition of $count_p$, then (s_0, s_1) is a transition of p as well. Hence, by the Definition 12, s_1 is in the set $R^{l+1}_{(p,adv)}(S)$. Finally, from Definition of S', $\langle s_1, l+1\rangle$ is in the set S'. Thus, S' is closed in $count_p$.

2. S' is closed in $count_{adv_k}$.
   Let $\langle s, l\rangle$ be a state in S'. If $count_{adv_k}$ can execute in state $\langle s, l\rangle$, then $l \geq k$. Hence, by Definition 12, there exists a sequence, $\langle s_0, s_1, ..., s_l\rangle$ such that (1) the first state in the sequence is in S (i.e., $s_0 \in S$), (2) the first transition in the sequence is executed by the adversary (i.e., $(s_0, s_1) \in adv$), and (3) subsequent transitions are executed by program (i.e., $\forall x | 0 < x < l ::$ $(s_x, s_{x+1}) \in \delta_p$). Furthermore, since $l \geq k$ and the fact that p is contained k-active stabilizing with adv for S, if $count_{adv_k}$ can execute in $\langle s, l\rangle$, then $s \in S$. Moreover, by Definitions 12 and 14, the resulting state is also in S'.
3. For any sequence σ (=$\langle s_0, s_1, s_2, ...\rangle$), if σ is a $\langle p, adv, k\rangle$-computation, then there exists l such that $s_l \in S$. This follows trivially from the definition of contained k-active stabilization. □

Finally, the following theorem trivially holds from the definition of active and contained active stabilization.

Theorem 5. *If p is k-active stabilizing with adversary adv for invariant S Then p is contained k-active stabilizing with adversary adv for invariant S.*

4 Comparing the Cost of Automated Verification for Active and Passive Stabilization

The problem of verifying stabilizing programs involves two parts: The first part relates to proving that in legitimate states (i.e., invariant), the program satisfies the specification at hand. And, the other part relates to proving that starting from an arbitrary state, the program recovers to legitimate states. Since the goal of this section is to compare the cost of verification for passive stabilization and active stabilization, we only focus on the second part. In other words, we compare the complexity of verification of *convergence* for passive and for active stabilization. We introduce instance of verification for passive and active stabilization. Then, we present the corresponding complexity results.

Instance. A program $p = \langle S_p, \delta_p\rangle$ and a state predicate S of p.
Verifying passive stabilization decision problem (VPS). Is p passive stabilizing for invariant S?

Theorem 6. *VPS can be solved in polynomial-time in $|S_p|$.*

Proof. This is a well-known result that can be proved with the following simple algorithm:

1. Closure property can be trivially verified by considering each transition in δ_p.
2. For convergence, if δ_p included any states outside S where there are no outgoing transitions, the answer to the decision problem is false. Assuming that this is not the case, we begin with program q which has the state space

$S_q = S_p - S$ and transitions $\delta_q = \delta_p - \{(s_0, s_1) \mid s_0 \in S \vee s_1 \in S\}$. Since we have removed some transitions, q may contain a deadlock state. If so, we remove that state from S_q and the corresponding transitions that enter and exit that state. Upon termination, if S_q is empty, p is passive stabilizing for invariant S. If not, there is a cyclic-computation that does not include any state in S. In other words, p is not passive stabilizing for invariant S. □

Next, we present the results for verification of active stabilization.

Instance. A program $p = \langle S_p, \delta_p \rangle$, an adversary adv for p, a state predicate S of p, and an integer $k \geq 2$.
Verifying k-active stabilization decision problem (VkAS). Is p k-active stabilizing with adversary adv for invariant S?

Theorem 7. *VkAS can be solved in polynomial-time in $|S_p|$.*

Proof. First, as stated earlier, closure proofs can be performed in polynomial time in $|S_p|$. The remaining proof is predicated under the assumption that the closure properties are satisfied. In particular, to prove convergence, we map the problem of verifying active stabilization to the problem of verifying passive stabilization. Specifically, we construct program p_1 as follows.

$$p_1 = \{(s_0, s_1) | \ \exists l : l \geq k-1 : reach(s_0, s_1, l) \vee \\ (\exists s_2 :: reach(s_0, s_2, l) \wedge (s_2, s_1) \in adv)\}, \text{ where}$$

$reach(s_0, s_1, l)$ denotes that s_1 can be reached from s_0 by execution of exactly l transitions of p

Intuitively, program p_1 executes $k-1$ or more transitions of program p and then (optionally) one transition of the adversary. Next, we show that p is k-active stabilizing with adversary adv for invariant S iff p_1 is passive stabilizing for invariant S

1. $\Rightarrow$ Let σ be a pure-computation of p_1. We construct a corresponding $\langle p, adv, k \rangle$-computation as follows: For each transition (s_0, s_1) in σ, we replace it by a sequence (that begins in s_0 and ends in s_1) of $k-1$ or more transitions of p followed by an optional transition of adv. By construction of p_1, this is always feasible. Let the resulting sequence be σ_1. Since σ_1 is a $\langle p, adv, k \rangle$-computation, it contains a suffix where all states are in S. Hence, σ also contains a state in S.
2. $\Leftarrow$ Let σ be a $\langle p, adv, k \rangle$-computation of p. We construct a corresponding pure-computation of p as follows: If σ contains a transition by the adversary in the first k transitions, we consider the suffix that begins in the state after the transition of the adversary. Hence, without loss of generality we can assume that the first $k-1$ transitions in σ are transitions of p. Now, we obtain a pure-computation of p_1 as follows: The initial state in σ_1 is the same as that in σ. Let this state be s_0. Now, to obtain the next state in σ_1, we identify the first occurrence of the transition of the adversary in σ.

If such a transition, say (s_a, s_b), exists then the successor state is s_b. If such a transition does not exist then the successor state is the one obtained by executing $k-1$ transitions of p. It follows that σ_1 is a pure-computation of p and, hence, includes a suffix that is entirely within S. Hence, σ also contains a state in S. □

5 Methodology for Designing Active Stabilizing Programs

In this section, we identify an approach for designing a program to be self-stabilizing. This approach is based on the convergence stair [10] approach for designing self-stabilizing programs. In particular, the problem in designing an active stabilizing program lies in the fact that an adversary can disrupt the progress made by the program towards recovering to the invariant. However, if we can prove that the program manages these disruptions in a suitable fashion, it can be proved that the program is active stabilizing to the adversary. Specifically, we prove the following theorem.

Theorem 8. *Let $p = \langle S_p, \delta_p \rangle$ be a program, adv be an adversary for p, and $S_0, S_1, \ldots, S_n$ be a sequence of state predicates of p. If*

- $S_0 = S_P$
- $\forall j : 0 \leq j < n : (S_{j+1} \Rightarrow S_j)$,
- $\forall j : 0 \leq j \leq n : S_j$ *is closed in p,*
- $\forall j : 0 \leq j \leq n : S_j$ *is closed in adv,*
- *For any finite sequence $\alpha = \langle s_1, s_2, ..., s_k \rangle$, if $s_1 \in S_j$ and $\forall l : 0 < l < k : (s_l, s_{l+1}) \in \delta_p$ then $s_k \in S_{j+1}$.*

Then

- *p is k-active stabilizing with adversary adv for S_n.*

Proof. The closure requirements for active stabilization are trivially satisfied. Regarding the last requirement in Definition 7, consider any $\langle p, adv, k \rangle$-computation, say σ. Based on the last constraint in this theorem and definition of $\langle p, adv, k \rangle$-computation, there exists a state, say s_a in σ such that $s_a \in S_1$. Since S_1 is closed in p and adv, all states in the suffix of σ that starts from s_a are in S_1. Again, by the same argument, σ contains a state, say s_b, in S_2, and so on. Thus, σ contains a state in S_n. □

The above theorem suggests the following approach to design active stabilizing programs. First, we identify a sequence of stair predicates $S_1, S_2, \ldots, S_{n-1}$ such that each of these predicates is closed in adv. Then, we ensure that the recovery from any of these predicates to the next state predicate is achieved before the adversary can perturb the program. Observe that in this fashion, the adversary can in fact execute several times before the program reaches the invariant. However, intuitively, the disruption by the adversary is less than the progress made by the program.

Moreover, if we focus on last condition in Theorem 8, it only focuses on pure-computations where the adversary is not allowed to execute. Thus, design of each stair is equivalent to the design of passive stabilization where the *convergence steps* are bounded. Thus, each stair of the active stabilizing program can potentially be constructed out of a collection of passive stabilizing programs.

6 Relation between Active Stabilization and Other Stabilization Techniques

In this section, we compare active stabilization with other stabilization techniques, namely *fault-contained stabilization* and *Byzantine self-stabilization* in Subsections 6.1 and 6.2, respectively.

6.1 Fault-contained Stabilization

The problem of fault-containment stabilization has been studied (e.g., [8, 9, 16]) in the literature to deal with two problems with stabilizing programs. The first problem is that stabilizing programs do not typically differentiate between an arbitrary global state and a state that is "almost legitimate" [8]. Hence, the goal of these algorithms is that if the state is "almost legitimate" then it reaches a legitimate state within a small number of steps. However, the recovery from an arbitrary state may take longer. The second problem is that after the program reaches a legitimate state, there is a high probability that transient faults will only perturb it to a state that is "almost legitimate" as opposed to an arbitrary state.

With this intuition, in [8,9,16], the authors introduce a notion of limited fault class, lf_p, for program p. lf_p is a subset of $S_p \times S_p$. Moreover, if the program is perturbed by lf_p in a legitimate state, a quick recovery is provided if no additional faults occur. Moreover, if multiple faults from lf_p occur or if faults outside lf_p occur then stabilization is still provided. Thus, fault-contained stabilization can be defined as follows:

Definition 15 (Fault-contained Stabilization). *Let $p = \langle S_p, \delta_p \rangle$, S be a state predicate of p, and lf_p be a subset of $S_p \times S_p$. And, let $w \geq 1$ be an integer. p is* fault-contained stabilizing *for lf_p with w steps for invariant S iff*

- *p is passive stabilizing for invariant S, and*
- *For every sequence $\sigma = \langle s_0, s_1, \ldots \rangle$, if $s_0 \in S$, $(s_0, s_1) \in lf_p$ and $\forall j > 1$: $(s_j, s_{j+1}) \in \delta_p$ then $s_w \in S$.*

Now, we can show that if p is contained k-active stabilizing then it provides fault-contained stabilization. Note that the converse of this theorem is not correct since a fault-contained stabilizing program may not recover to legitimate states if it is continuously perturbed by faults.

Theorem 9. *If p is contained k-active stabilizing with adversary adv for S, then p is fault-contained stabilizing for adv with k steps for S.*

Proof. Follows trivially from Definitions 9 and 15. □

Corollary 2. *If p is k-active stabilizing with adversary adv for S, then p is fault-contained stabilizing for adv with 1 steps for S.*

6.2 Byzantine Self-stabilization

While all the formalization in Byzantine self-stabilization work cannot be presented here, we give a brief approach considered in these papers and its relation to active stabilization. In [14], the program is viewed in terms of a set of, say n, processes. Thus, the state of the program is of the form $\langle v_1, v_2, ..., v_n\rangle$, where v_j denotes the state of process j. (If channels are used corresponding entry is added for channel contents as well.) Thus, the state space of the program, (S_p in Definition 1) is obtained by considering all possible tuples of $\langle v_1, v_2, ..., v_n\rangle$, where the domain of v_j depends on the application at hand.

Moreover, some processes can be Byzantine. If process j is Byzantine, it can change the value of v_j arbitrarily. Thus, transitions of p (cf. Definition 1), δ_p is of the form: $\delta_p = \delta_{p_g} \cup \delta_{p_b}$, where δ_{p_b} denotes the transitions that correspond to the transitions executed by Byzantine process(es) and δ_{p_g} denotes the transitions executed by non-Byzantine processes. Furthermore, δ_{p_g} and δ_{p_b} are disjoint. It is also assumed that Byzantine processes do not prevent non-Byzantine processes from executing. However, Byzantine processes can disrupt the recovery process. Thus, the definition of stabilization in the presence of Byzantine faults is adapted as follows.

Definition 16. *A program $p = \langle S_p, \delta_p\rangle$, where $\delta_p = \delta_{p_g} \cup \delta_{p_b}$, is said to be stabilizing in the presence of Byzantine faults for invariant S iff*

- *S is closed in p, and*
- *for any state sequence, say σ, of the form $\langle s_0, s_1, s_2, ...\rangle$, if*
  - $\forall j \geq 0 : (s_j, s_{j+1}) \in \delta_p$
  - *Number of transitions of δ_{p_g} is σ is infinite.*
- *then*
  - *there exists l such that $s_l \in S$.*

Theorem 10. *If program $p = \langle S_p, \delta_p\rangle$, where $\delta_p = \delta_{p_g} \cup \delta_{p_b}$ is stabilizing in the presence of Byzantine faults for invariant S, then $\langle S_p, \delta_{p_g}\rangle$ is 2-active stabilizing with adversary δ_{p_b} for S.*

Proof. Closure proofs are trivially satisfied from Definition 16. In a $\langle\langle S_p, \delta_{p_g}\rangle, \delta_{p_b}, 2\rangle$-computation, there is at least one transition of δ_{p_g} between any two transitions of δ_{p_b}. Hence, the number of occurrences of transitions in δ_{p_g} is infinite. Hence, in any $\langle\langle S_p, \delta_{p_g}\rangle, \delta_{p_b}, 2\rangle$-computation, a state in S is reached. □

To prove the converse of the above theorem, we recall the standard definition of transitive closure.

Definition 17 (Transitive Closure). *A set of transitions δ_{p_b} is transitive closed iff* $\forall a, b, c :: ((a, b) \in \delta_{p_b} \wedge (b, c) \in \delta_{p_b}) \quad \Rightarrow \quad ((a, c) \in \delta_{p_b})$

Theorem 11. *Let p be a program whose transitions are partitioned in terms of δ_{p_g} and δ_{p_b}, where δ_{p_b} are the transitions executed by Byzantine processes. If $\langle S_p, \delta_{p_g} \rangle$ is 2-active stabilizing with adversary δ_{p_b} for S, and δ_{p_b} is transitive-closed, then p is stabilizing in the presence of Byzantine faults for S.*

Proof. The closure proof is trivially satisfied. Consider any computation, say σ, of p where transitions of δ_{p_g} execute infinitely often. Consider a compacted version of σ, say c_σ that is obtained as follows: If σ contains two consecutive transitions, say (s_j, s_{j+1}) and (s_{j+1}, s_{j+2}), in δ_{p_b} then we replace them by (s_j, s_{j+2}). And, repeat this process until there are no two successive transitions in δ_{p_b}. Since δ_{p_b} is transitive closed, c_σ is a $\langle \langle S_p, \delta_{p_g} \rangle, \delta_{p_b}, 2 \rangle$-computation. Hence, it includes a state in S. Thus, σ includes a state in S. □

7 Related Work

There are several variations of stabilization (denoted by passive stabilization in this paper) that are considered in the literature. These include fault-containment stabilization, byzantine stabilization, FTSS, multitolerant stabilization, weak stabilization, probabilistic stabilization, and nonmasking fault-tolerance.

Fault-containment stabilization refers to stabilizing program that ensure that if only one (respectively, small number of) fault occurs then quick recovery is provided to the invariant. Examples of such programs include [8, 16]. Byzantine stabilization refers to stabilizing programs that tolerate the scenario where a subset of processes is Byzantine. Examples of such programs include [13, 14]. FTSS refers to stabilizing programs that tolerate permanent crash faults. Examples of such programs include [3]. Multitolerant stabilizing systems ensure that in addition to stabilization property, the program ensures that the safety property is never violated when only a limited class of faults occur. Examples of such systems include [12]. As discussed in the last two sections, fault-containment stabilization and Byzantine stabilization are closely related to Active stabilization.

Weak stabilization [5, 11], as the name suggests, is a weaker version of stabilization. In weak stabilizing programs, from every state, there is a path to reach a state in the invariant. However, the program may contain loops that are outside legitimate states. In [11], it is shown that under certain fairness condition, a weak stabilizing program is also a stabilizing program. In probabilistic stabilization, the program recovers to legitimate states with high probability. Finally, nonmasking fault-tolerance [1, 2] refers to programs where the program recovers from states reached in the presence of a limited class of faults. However, this limited set of states may not cover the set of all states.

8 Conclusion

In this paper, we proposed the concept of *active stabilization*, where program's state can be perturbed by *faults* to any arbitrary state and recovery is accomplished in the presence of constant perturbation by an *adversary*. We introduced different types of active stabilizing programs depending upon the behavior of the adversary and the ability of program to recover. We evaluated the cost of verification for passive and active stabilization. We also argued that active stabilization is a highly expressive concept by presenting comparison to fault-contained stabilization and Byzantine self-stabilization.

For future work, we are considering several research directions. We are currently working on developing efficient techniques for verification of active stabilization as well as results about composition of active stabilizing programs.

References

1. Anish Arora. Efficient reconfiguration of trees: A case study in methodical design of nonmasking fault-tolerant programs. In *Science of Computer Programming*. 1996.
2. Anish Arora, Mohamed G. Gouda, and George Varghese. Constraint satisfaction as a basis for designing nonmasking fault-tolerance. In *ICDCS*, pages 424–431, 1994.
3. Joffroy Beauquier and Synnöve Kekkonen-Moneta. On ftss-solvable distributed problems. In *WSS*, pages 64–79, 1997.
4. M. Demirbas, A. Arora, and M. Gouda. A pursuer-evader game for sensor networks. In *Symposium on Self-Stabilizing Systems(SSS)*, pages 1–16, 2003.
5. Stéphane Devismes, Sébastien Tixeuil, and Masafumi Yamashita. Weak vs. self vs. probabilistic stabilization. In *Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems*, ICDCS '08, pages 681–688, Washington, DC, USA, 2008. IEEE Computer Society.
6. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), 1974.
7. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ., 1990.
8. S. Ghosh and A. Gupta. An exercise in fault-containment: Self-stabilizing leader election. *Information Processing Letters*, 1996.
9. S. Ghosh, A. Gupta, T. Herman, and S. Pemmaraju. Fault-containing self-stabilizing algorithms. In *Principles of distributed computing (PODC)*, pages 45–54, 1996.
10. M. G. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4):448–458, 1991.
11. Mohamed G. Gouda. The theory of weak stabilization. In *In WSS01 Proceedings of the Fifth International Workshop on Self-Stabilizing Systems, Springer LNCS:2194*, pages 114–123, 2001.
12. Sandeep S. Kulkarni and Anish Arora. Multitolerance in distributed reset. *Chicago J. Theor. Comput. Sci.*, 1998, 1998.
13. Mahyar R. Malekpour. A byzantine-fault tolerant self-stabilizing protocol for distributed clock synchronization systems. In *SSS*, pages 411–427, 2006.
14. M. Nesterenko and A. Arora. Local tolerance to unbounded byzantine faults. In *IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 22–31, 2002.
15. S.Dolev, A.Israeli, and S.Moran. Uniform dynamic self-stabilizing leader election. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):424–440, 1997.
16. H. Zhang and A. Arora. Guaranteed fault containment and local stabilization in routing. *Computer Networks (Elsevier)*, 2006.