Self-stabilizing Extensions for Message-passing Systems

Shmuel Katz
Computer Science Department, The Technion

Kenneth J. Perry
IBM T.J. Watson Research Center

Abstract

Self-stabilization is an abstraction of fault tolerance for transient malfunctions. Intuitively, a self-stabilizing program resumes normal behavior even if execution begins in an illegal initial state. In this paper, we explore the possibility of extending an arbitrary program into a self-stabilizing one. Our contributions are: (1) a formal definition of the concept of a program being a *self-stabilizing extension* of a non-stabilizing program; (2) a characterization of what properties may hold in such extensions; (3) a demonstration of the possibility of mechanically creating such extensions.

The computational model used is that of an asynchronous distributed message-passing system whose communication topology is an arbitrary graph. We contrast the difficulties of self-stabilization in this model with those of the more common shared-memory models.

1 Introduction

1.1 Problem Statement

Self-stabilization is an abstraction of fault-tolerance for a model in which transient faults arbitrarily corrupt data, messages, and location counters (but not the program code). Each such fault is assumed to be followed by a long period without additional faults. The concept of a *self-stabilizing* distributed program, introduced by Dijkstra [Dij74], requires that a program executing with an arbitrary initial state (including arbitrary control locations) eventually must reach a *legitimate* state and thereafter remain in legitimate states. In other words, the property "the program is in a legitimate state" is stable (see [CL85]) and eventually true. The arbitrary initial state represents the situation immediately after a fault.

The difficulty of self-stabilization lies mainly in the fact that a process has no way of distinguishing between an initial state and one that occurs during the computation. For example, the assertion $\{x = 0\}$ does not always hold following the code $x := 0$ since the initial control state may be just after this statement, without it having actually executed. Similarly, control of a process can be just after a **send** instruction in a message-passing model, without a message having actually been sent and with no indication of this fact. Thus, no process can ever "depend" on the accuracy of any value stored in its memory.

The contributions of this paper are (1) to formally define the concept of a program being a *self-stabilizing extension* of a non-stabilizing program; (2) to characterize which properties may eventually hold in such an extension; (3) to examine the limits of self-stabilization in the asynchronous distributed message-passing model of computation; (4) to demonstrate the possibility of automatically creating such extensions, via the superimposition of self-stabilizing algorithms for taking global distributed snapshots and performing resets.

The self-stabilizing version of the snapshot algorithm has repeated sending of messages to ensure liveness, and round numbers to ensure eventual consistency of the results. A technique is also introduced to avoid clogging the system with outdated messages. The proof of self-stabilization is based on showing that all messages initially in the channels (including snapshot messages that were never actually sent) and the effects of those messages will eventually be 'flushed' from the system, as will the effects of the arbitrary initial values in the processes.

We also show that although self-stabilization is globally achievable, no process in the system can ever *know* that the system has self-stabilized. This rules out the possibility of algorithms that switch from an inefficient but self-stabilizing mode to one that is more efficient but non-stabilizing.

1.2 Previous Work

Previous results on self-stabilization generally employ a shared-memory model of computation, with self-stabilization being a primary consideration in the construction of the program. In contrast, our results examine a different computational model (message-passing in a communication graph with arbitrary topology), and focus on the mechanical transformation of a program into an extension that is also self-stabilizing. This necessitates formulating a precise semantics for self-stabilization and using it to define when a program is "a self-stabilizing equivalent" (i.e, extension) of a non-stabilizing program.

Dijkstra[Dij74] demonstrated the concept of self-stabilization through an example that identified the legitimate states as those with exactly one enabled operation (called a *privilege*). His solutions thus are examples of self-stabilization for a form of mutual exclusion (or, equivalently, for a token ring with a single token). Subsequent results [Dij74,Lam84,GE88,BP89,BGW89] follow Dijkstra in that they deal with mutual exclusion or token-passing and create self-stabilizing programs from scratch. Lamport's mutual exclusion algorithm[Lam84] is the exception in that he creates a self-stabilizing program by inserting statements into a non-stabilizing program.

There are few results providing a precise semantics for self-stabilization. Lamport[Lam84] defined a *transient malfunction* behavior as an execution in which each process initially assigns arbitrary values to its variables before following the code of the algorithm. Then an algorithm is self-stabilizing for a property A if A eventually holds for every such behavior. He leaves open the domain of the values in the malfunction operation.

2 The Model and Its Difficulties

An outline of a formal semantics for the model is given in the following section. Intuitively, the computational model used in this paper is that of an *asynchronous message passing system.* A *message passing system* is a collection of *processes* that are connected by FIFO communication *channels.* Processes may exchange values only by transmitting *messages.* The system is *asynchronous* in that there are no bounds on either relative process speeds, message delivery time, or channel capacities. We do assume that every message sent is eventually received and that every statement whose guard remains true is eventually selected for execution. In the initial state of such a system, process states, location counters, and channels may have arbitrary values.

Although simulations of message-passing by shared-memory (and vice-versa) exist, they are not self-stabilizing. With regard to self-stabilization, the asynchronous message-passing model introduces several phenomena absent from the shared-memory model.

The primary one is *apparently sent messages.* In a local process state intended to follow the sending of a message, there is no way to determine whether (a) the desired message has actually been sent or (b) this is merely a false impression, and the local state is part of the initial global state. This could cause deadlock, with a process waiting for a response that will never come because the request message was never actually sent. In contrast, shared memory allows a process to determine (by reading) that a variable has an unexpected value.

Another difficulty is of *infinite propagation* of false channel information. Since the channels can initially have arbitrary messages on them, the effects of these messages must not be to indefinitely generate new messages that are possible only in response to the misleading incoming messages. Otherwise an acceptable global state might never be reached. At some point, these injurious initial messages and injurious new messages generated due to them must be purged from the system.

3 Semantics of self-stabilization

We precisely define the notion of what it means for one program to be both self-stabilizing and "an extension" of another program. The definitions below are for the interleaving execution model of concurrent systems.

We adopt the usual definitions of: a *local state* of a process (an assignment of values to the local variables and the location counter); a *global state* of a system of processes (the cross product of the local states of its constituent processes, plus the contents of the FIFO channels); the semantics of program operations (the possible atomic steps and their associated state changes); an *execution sequence* of program **P** (a possibly infinite sequence of global states in which each element follows from its predecessor by execution of a single atomic step of **P**).

The set of all possible execution sequences of program **P** is denoted by *sem*(**P**) and defines the semantics of **P**. Note that no assumption is made about the initial state of an execution sequence, except that all values are from the appropriate domain.

The definition of self-stabilization depends on what are considered the "legitimate" states of a program. We could define the legitimate states of program **P** as those satisfying a predicate (called **P**'s *specification*). Alternatively, these states could be defined as those obtainable from a "normal" execution. For the purpose of defining self-stabilizing extensions of a program, we choose the latter. Of course, the normal initial states could also be defined using a predicate but, for the sake of concreteness, we choose a particular common possibility. Those initial states in which the location counter of each process is 0 and all channels are empty are said to be *nor-*

mal; the *legal* (i.e., intended) semantics of program **P** is the subset of *sem*(**P**) containing only sequences with normal initial state and is denoted by *legsem*(**P**). Every global state in a sequence from *legsem*(**P**) is also defined to legal.

Note that the set of legal global states is, in general, much smaller than the set of possible global states, since the latter includes many combinations of values that do not arise in any legal execution sequence. The *illegal execution sequences* are those that have initial illegal states. There is yet a third class of execution sequences: those with initial states that are legal, but not normal, e.g., with control not at the beginning of the code. These are clearly suffixes of legal sequences. In the continuation, note that a suffix of a sequence can be the sequence itself.

Definition 1 *Program* **P** *is* self-stabilizing *if each sequence in sem*(**P**) *has a non-empty suffix that is identical to a suffix of some sequence in legsem*(**P**).

In other words, from some point on every computation is identical to a legal one. Now we turn to the relation between a program and an extension. A *projection* of a global state onto a subset of the variables and the messages on the channels is the value of the state for those variables and messages.

Definition 2 *Program* **Q** *is an* extension *of program* **P** *if for each global state in legsem*(**Q**) *there is a projection onto all variables and messages of* **P** *such that the resulting set of sequences is identical to legsem*(**P**), *up to stuttering*[1].

Definition 3 *Program* **Q** *is a* self-stabilizing extension *of program* **P** *if* **Q** *is self-stabilizing and also is an extension of* **P**.

That is, considering only those portions of **Q**'s global state that correspond to **P**'s variables and messages, the legal semantics of **P** and **Q** are identical if repetitions of states are ignored. Moreover, **Q** is self-stabilizing for all its computations. When begun in normal initial states, **P** and **Q** have the same possible executions (relative to **P**'s state, ignoring location counters) and **Q** resumes the intended semantics when begun in an illegal initial stat For legal but not normal initial states, **Q** merely executes the suffix of a legal computation, relative to **P**. Note that no correspondence is required among the illegal computations of **P** and **Q**, or among the location counters.

[1] When comparing sequences, adjacent identical states are eliminated; this is sometimes called the elimination of stuttering.

In particular, program **P** may terminate with control locations after all statements of its program (or, equivalently, at *halt* statements), but **Q** generally has no such halting locations. Otherwise, **Q** could have an initial state with its control halted, but with an illegal global state relative to the messages and variables—and thus not be self-stabilizing. The extension for a terminating computation of **P** has execution sequences that eventually repeat a final state of **P** forever, changing only variables not present in **P**. By similar reasoning, except for trivial cases, no process can ever have a local state that guarantees a legal *global* state. (The process could intially have that state, even when the global state is illegal.) This means that no process can ever *know* that the system has self-stabilized, even though self-stabilization can be guaranteed to eventually occur.

4 Limits on Self-Stabilizing Extensions

Before demonstrating how a self-stabilizing extension of a program is created, we consider which properties can hold in the extension. A simple example illustrates the potential problems. Suppose that the specification of program **P** is "a 1 is eventually output" and that "1,0,0,..." is the only legal output sequence of **P**. Furthermore, suppose that program **Q** outputs only all-zero sequences when started in an illegal initial state and otherwise produces the same sequence as **P**. Then **Q** is a self-stabilizing extension of **P** but does not eventually satisfy **P**'s specification. The problem is that some sequences in *legsem*(**P**) satisfy the specification but have suffixes that do not.

The following theorem characterizes properties that can hold in self-stabilizing extensions.

Theorem 1 (Characteristic)
If **Q** *is a self-stabilizing extension of* **P**, *and A is an assertion in (future) linear temporal logic over variables and messages of* **P**, *then:*
A holds for a suffix of every execution sequence of **Q** *iff for each sequence in legsem*(**P**) *either A is true in the final state (for finite sequences) or A is infinitely often true.*

Proof Since A is in the future fragment of linear temporal logic, for a suffix S of a sequence T, the truth of A for S is independent of the states in T but not S. That is, the states before S are irrelevant to the truth of A on S.

$\Rightarrow$. In *sem*(**Q**), every suffix of an execution sequence is in itself an execution sequence. Since A is true for a suffix of every execution sequence of **Q**, if the sequence is infinite, A is true infinitely often along the sequence, because it only relates to the suffix, as noted above. Similarly, if the state relative to **P** repeats forever from some point on, A must hold in this "final" state, since it too is an execution sequence. Since **Q** is a self-stabilizing extension of **P**, by definition for each sequence in *legsem*(**P**) there is a sequence in *legsem*(**Q**) identical w.r.t. **P**'s state. A must be true for these sequences of *legsem*(**Q**) and their suffixes, since they are a subset of the sequences in *sem*(**Q**), and thus it must also hold infinitely often (or in the final state) for the sequences in *legsem*(**P**).

$\Leftarrow$. If A is infinitely often true for the infinite sequences in *legsem*(**P**), then it is infinitely often true for the infinite sequences in *legsem*(**Q**) that correspond to them. Similarly, for finite sequences, if A is true in the final state, then **Q** will have a sequence in *legsem*(**Q**) that eventually repeats the final state of **P** forever, changing only variables and messages not in **P**, and thus not affecting the continued truth of A in that sequence. Every sequence in *legsem*(**Q**) has a projection onto one in *legsem*(**P**). Since **Q** is self-stabilizing, every sequence in *sem*(**Q**) has a suffix identical to one in *legsem*(**Q**), and thus satisfying A. ∎

Note that properties that cannot hold in a self-stabilizing extension can often be rephrased as ones that do. For example, if *legsem*(**P**) contains only the infinite sequence of values (0,1,0,1,...), then an assertion that eventually the number of 1's and of 0's will be equal may never become true (e.g., if in **Q** there are sequences that begin with (1,1,1,0,1,0,1,...)). On the other hand, an assertion that infinitely often there is a state followed by a later state for which in the states between them there are equal numbers of 1's and 0's is true for every sequence in a self-stabilizing extension.

5 Automatically creating self-stabilizing extensions

In this section, we give a means for automatically creating a self-stabilizing extension of a distributed program **P**. This demonstrates the possibility of such an approach. Our technique is to *superimpose* onto **P** a self-stabilizing "control" program. The "control" program interleaves steps of **P** with steps of an intuitively simple task that repeatedly (a) takes "snapshots" of the global state; (b) tests whether these snapshots indicate an illegal global state; (c) resets the memory of each process to some default legal state, if a problem is detected. The difficulty in implementing this simple task arises because it too must function correctly no matter what the initial state. In particular, messages and variable values indicating partially completed snapshots or inconsistent fragments of several snapshots can be present in the initial state. Toward this end, our main contribution is the creation of self-stabilizing global snapshot and global reset algorithms.

The extended program resulting from the superimposition has the following properties, which guarantee self-stabilization of **P**: (1) eventually an accurate snapshot completes (even though the accuracy cannot be known by any process) (2) eventually, if an illegal state is revealed in an accurate snapshot, the reset algorithm is invoked and establishes a legal global state, and (3) thereafter, snapshots remain accurate and no further resets occur because legal global states only have legal successors. Thus, the extended program eventually continues to take snapshots but in no other way interferes with **P**.

5.1 A Self-Stabilizing Snapshot Algorithm

5.1.1 Overview

We now present a self-stabilizing algorithm to take accurate snapshots of the global state. This (and the reset algorithm) are imposed onto program **P** (called the **basic** program; **P**'s messages are called **basic** as well).

Definition 4 *At any global state σ, a process is said to have an* accurate snapshot for α *if local variables of the process contain a representation of a global state that is a possible successor of α and a possible predecessor of σ.*

StableSnap (Figures 1, 2, 3) is a self-stabilizing algorithm that permits process 0 to iteratively obtain accurate snapshots for the state that **P** had when each iteration began. Process 0 is arbitrarily chosen to have the special role of being both the initiator of snapshots and the process whose variables contain its results. We assume that the network topology is described by a set **E** of ordered pairs of process-identifiers. The network topology is unrestricted (except for being strongly-connected, which is needed if a single process both initiates and collects information).

We emphasize that a self-stabilizing algorithm is only required to *eventually* establish the desired property, and not to establish it *immediately*. It is both acceptable and likely (e.g, when the initial state is not a normal initial state) that some number of inaccurate snapshots will initially occur. The sole requirement is that, from some point onwards, only accurate snapshots of **P** are obtained by process 0.

StableSnap is based on an iterated version of the single snapshot algorithm of Chandy and Lamport[CL85] that appears in Figure 1 as a macro[2] *CL* to be invoked by each process i upon receiving a message (called a *marker*) from process j. The first *marker* received by a process causes it to save its local portion of **P**'s state in a variable, begin recording basic messages received on each incoming channel, and propagate a *marker* to its neighbors. Subsequent *markers* cause the channel recording to terminate. A process terminates when it has received one *marker* on each incoming channel.

StableSnap is defined to iteratively invoke the Chandy-Lamport algorithm and collect the recorded information. It both regulates the iteration (by initiating waves of *token* messages) and insulates the Chandy-Lamport algorithm from unexpected messages. For example, an invariant of the single snapshot algorithm is that exactly one *marker* is received on each channel; this assumption can clearly be violated by *marker* messages that may be present in initial states that are not normal. Lastly, it also establishes the initialization condition (*initiated* = `false`) for each iteration of the single snapshot algorithm.

The difficulty of correctly defining StableSnap arises because the variables and messages of the initial state are completely arbitrary. For example, initially two processes may have different values in the local variable (*Current*) that defines the iteration in which each is participating. This introduces the potential problems of deadlock and the infinite propagation of messages. We avoid deadlock by introducing a non-reactive statement (GENPROD) to create prods (spontaneously-sent messages); these prods themselves introduce problems in that it is now possible for messages created in different iterations to simultaneously be in the system. To distinguish between them, we add to each *token* and *report* message an integer field VAL, which contains an iteration number. Infinite propagation is avoided by adding to each message a field PATH, which is a sequence of process identifiers.

5.1.2 Behavior of processes

At a high-level, the behavior of each process other than 0 in executing StableSnap is to "react" to the receipt of a message m, containing the values v, p, and r in the VAL, PATH, and PIECE fields respectively. If predicate **IsNext** recognizes m as a *token* meant to start a new iteration, the process executes statement (NEXT), which changes its iteration counter and invokes macro *CL*. Should m be recognized as a *marker* message (i.e., predicate **IsMarker** holds), the process executes statement (RECSNAP) and invokes the *CL* macro. If neither of the above hold, the message is recognized as a prod (statement (RECPROD)) and is passed on to its neighbors. Additionally, the prod may cause a *report* message to be sent to process 0 if predicate **Finished**, which determines whether local termination has occurred, is satisfied. The PIECE field of a *report* message contains the state and channel information recorded by macro *CL*; this enables process 0 to obtain a representation of the global state. The correct implementation of these predicates is made non-trivial because of the multitude of global states (particularly the illegal ones) in which they may be evaluated.

One possibility for the *report* messages is to broadcast them like the *token* messages (with all the components to prevent indefinite flooding). Another is to send them on a fixed (constant, built-in) path to process 0. The option usually employed in the Chandy-Lamport algorithm – of using the edges along which the initiating token for the present round was first received to get a spanning tree to the origin– is not possible in our context. The initiating edge is recorded as data, and could be inaccurate in an initial state, leading to a non-terminating reporting stage. There are additional options such as building a self-stabilizing spanning tree algorithm for our model, but we do not deal with this aspect here, and assume one of the two possibilities above.

Process 0's behavior is different in that it (a) initiates each new iteration of the single snapshot algorithm (by executing statement (START)) (b) non-reactively creates prods by executing statement (GENPROD) (c) saves the information contained in *report* messages by executing statement (RECREPT). Statement (START) detects termination of an iteration when *reported*[k] becomes true for all k and initiates a new iteration of the single snapshot algorithm by creating a new *token*. (Solely for the purpose of avoiding awkward coding, we assume a channel from process 0 to itself; this allows it to "react" to its own messages, just like the other processes.)

6 Correctness of the Snapshot Algorithm

The proof that StableSnap is a self-stabilizing algorithm that eventually permits process 0 to iteratively obtain accurate snapshots proceeds by defining a certain class of *good* states, demonstrating that such states are eventually repeated (Lemma 3), and demonstrating that an accurate snapshot is obtained when the single snapshot iteration begins in such a state (Theorem 2). Intuitively, the good states are the ones in which an iteration is guaranteed to terminate with an accurate snapshot.

Definition 5 *A global state σ is* good *if and only if*

[2] We look at macro invocation as an abbreviation for the statements contained in its body, after parameter substitution has been performed.

```
macro CL(j):
Process $i, i \geq 0$
        $\neg initiated$                 $\Rightarrow$  [$initiated$ := true;
                                            $eoc[j]$ := true;
                                            $eoc[k]$ := false, for all $k \neq j : (k,i) \in \mathbf{E}$;[a]
                                            $record[k]$ := $nil$, for all $k : (k,i) \in \mathbf{E}$;
                                            $basic\_state$ := state of $\mathbf{P}$
                                            /* Send[b] a marker to all $k : (i,k) \in \mathbf{E}$ */
                                            ]

        $initiated \wedge \neg eoc[j]$   $\Rightarrow$  $eoc[j]$ := true

        $initiated \wedge eoc[j]$        $\Rightarrow$  skip /* Illegal state */
```

[a]A side effect not shown is that process i starts to record in variable $record[k]$ all basic messages of $\mathbf{P}$ that are received from each neighboring process k when $eoc[k]$ is **false**.

[b]In a legal state, the *marker* will be piggy-backed onto a *token* message of the algorithm that invokes this macro.

Variables:

- *basic_state* is state of the basic program **P**.
- *initiated* is a Boolean indicating that initialization has been performed.
- $eoc[j]$ is a Boolean indicating that a *marker* message has been received from j.
- $record[k]$ is the sequence of **basic** messages received from k while $eoc[k]$ was `false`.

Figure 1: Chandy-Lamport snapshot algorithm.

1. *The value of the Current variable in each process is identical.*
2. *No message in σ can henceforth cause* **IsNext** *to become true, for any process i.*
3. *$reported[k]$ is true for all $0 \leq k < n$.*

We first show that nothing prevents a new iteration from beginning.

Lemma 1 (Iteration Liveness) *In any execution of* StableSnap, *for any state α, there is some successor state σ in which the guard of statement* (START) *is satisfied. Moreover, this guard remains satisfied in each successor of σ until the next* (START) *statement is executed.*

Proof Let v_α denote the value of process 0's *Current* variable in state α. We claim that eventually, there is some successor state of α in which either "$reported[k]$ is true for all k" already holds (in which case the lemma holds) or in which every message is a copy of a message created by executing statement (GENPROD) in a successor state of α. In the latter case, each of these messages has field VAL $= v_\alpha$ because process 0 cannot change its *Current* variable until $reported[k]$ becomes true for all k. By the continued enabling and execution of statement (GENPROD) and the assumptions of FIFO channels and eventual message delivery, we see that repeated reception of the messages arising from statement (GENPROD) results in each process eventually sending a *report* message with VAL $= v_\alpha$. As each *report* is received by process 0, $reported[k]$ become true for each k in turn until finally a state σ results in which $reported[k]$ is true for all k. ∎

In the next lemma, we establish that reaching a good state is unavoidable, no matter what the initial state.

Lemma 2 (Eventuality of good) *In every execution of* StableSnap, *every state eventually has a good successor state. Moreover, each successor of a good state remains good until the subsequent* (START) *statement is executed.*

Proof Let α be any state and let V denote the finite set of values that are present in α either as values of *Current* variables or as VAL fields of messages. By Lemma 1, inspection of the guards, and the assumptions of eventual message delivery and fairness, a state β occurs in which process 0 has just executed a (START) statement and for which v_β, the value of *Current* for process 0, exceeds the maximum value in V.

Eventually, each process in turn receives a copy of the *token* with VAL $= v_\beta$. By construction of v_β and

Process $i > 0$, upon receiving *token* message m with VAL = v, and PATH = p from process j:

(RECSNAP)	**IsMarker**	$\Rightarrow$	[$CL(j)$; **Finished** $\Rightarrow$ *Report*]
(DEJAVU)	**IsCntl** $\wedge$ **Seen**	$\Rightarrow$	**skip**
(NEXT)	**IsCntl** $\wedge \neg$**Seen** $\wedge$ **IsNext**	$\Rightarrow$	[*Current* := v; *Propagate*; *initiated* := `false`; $CL(j)$]
(RECPROD)	**IsCntl** $\wedge \neg$**Seen** $\wedge \neg$**IsNext**	$\Rightarrow$	[*Propagate*; **Finished** $\Rightarrow$ *Report*]

Upon receiving a *report* message:

(PASSREP) **InReportForm** $\wedge \neg$**Seen** $\Rightarrow$ /* pass it on */

Variables and macro specifications:

- *Current* is the "iteration number".
- *Propagate* sends a *token* message with PATH = append(p, i) and VAL = v to each process k such that $(i, k) \in \mathbf{E}$.
- *Report* sends a *report* message with PATH = i, VAL = v, and

$$\text{PIECE} = \langle \textit{basic\_state}, \textit{record}[k], \text{for each } k : (k, i) \in \mathbf{E} \rangle$$

  to process 0 in one of the ways described in the text.

Definition of predicates:

- **IsMarker** $\equiv \neg eoc[j] \wedge (\text{v} = \textit{Current})$
- **IsCntl** $\equiv \neg$**IsMarker** $\equiv eoc[j] \vee (\text{v} \neq \textit{Current})$
- **Seen** $\equiv$ (i appears in sequence p) $\wedge((i \neq 0) \vee (\text{p} \neq 0))$
- **IsNext** $\equiv (\text{v} > \textit{Current})$
- **Finished** $\equiv (\text{v} \neq \textit{Current}) \vee (\bigwedge_{k:(k,i)\in \mathbf{E}} \; eoc[k])$
- **InReportForm** is true only on *report* messages.

Figure 2: Snapshot algorithm for process $i > 0$.

Process 0, upon receiving *token* (or *report*) message m with VAL = v, and PATH = p, (and PIECE = r) from process j:

```
(SELFSNAP)   $\mathbf{IsMarker}$                                        $\Rightarrow$  $[CL(j); \mathbf{Finished} \Rightarrow \mathit{Report}]$
(SELFSENT)   $\mathbf{IsCntl} \wedge \mathbf{Seen}$                        $\Rightarrow$  $\mathbf{skip}$
(RECREPT)    $\mathbf{IsCntl} \wedge \neg\mathbf{Seen} \wedge \mathbf{IsReport}$    $\Rightarrow$  $[k := \mathit{first}(\mathrm{p});$
                                                               $\neg \mathit{reported}[k] \Rightarrow$
                                                                 $[\mathit{reported}[k] := \mathtt{true}; \mathit{piece}[k] := \mathrm{r}]$
                                                               $]$
(SELFPROD)   $\mathbf{IsCntl} \wedge \neg\mathbf{Seen} \wedge \neg\mathbf{IsReport}$   $\Rightarrow$  $[\mathbf{Finished} \Rightarrow \mathit{Report}]$
```

Process 0, spontaneously:

```
(GENPROD)   $\mathtt{true}$                                 $\Rightarrow$  $\mathit{Start}$
(START)     $\bigwedge_{k=0}^{n-1} \mathit{reported}[k]$   $\Rightarrow$  $[\mathit{reported}[k] := \mathtt{false}$, for all $k \geq 0;$
                                                 $\mathit{Current} := \mathit{Current} + 1;$
                                                 $\mathit{Start};$
                                                 $\mathit{initiated} := \mathtt{false}; CL(0)$
                                                 $]$
```

Variables and macro specifications:

- *reported*$[k]$ is a Boolean indicating that process k has ended its participation in the current iteration and that *piece*$[k]$ is k's portion of the global state recorded.
- *Start* sends a *token* message with PATH = 0 and VAL = *Current* to each process k such that $(0, k) \in \mathbf{E}$.

Definition of additional predicate:

- $\mathbf{IsReport} \equiv (\mathrm{v} = \mathit{Current}) \wedge \mathbf{InReportForm}$

Figure 3: Snapshot algorithm for process 0

inspection of the guards, it can be shown that eventually there is a state γ in which the *Current* variable of each process i is equal to v_β and for which no message in state γ can henceforth cause **IsNext** to become true when it is received by any process. Both these properties are stable until the next (START) statement is executed. Therefore, γ has a successor in which $reported[k]$ is also true for all $0 \leq k < n$. ▮

Invoking Lemma 2 inductively gives

Lemma 3 (Recurrence of good) *In any execution of* StableSnap, *good states repeatedly occur.*

Theorem 2 (Good Snapshot) *In any execution of* StableSnap, *eventually process* 0 *repeatedly obtains accurate snapshots for the state in which each* (START) *statement is executed.*

Proof Since Lemma 3 shows that good states repeat, we need only show that accurate snapshots are obtained for them. This is demonstrated by a careful analysis that shows that the (START) statement executed at a good state results in each process receiving the same messages as it would were it executing the Chandy-Lamport algorithm in isolation. ▮

7 The reset algorithm

It is assumed that, along with the text of **basic** program **P**, we are given a predicate that recognizes representations of the legal global states of **P**. By applying this predicate to each representation of the global state obtained by StableSnap, process 0 may invoke a *reset* algorithm when an illegal state of **P** is detected. The reset algorithm is imposed onto StableSnap by associating "flavors" with the VAL field of each *token* and *report* message. To begin a normal snapshot, "vanilla" *tokens* are created; to begin a reset, "reset" tokens are created. Upon receiving a "reset" token, in addition to the actions described in StableSnap, each process suspends execution of **P** (discarding any **basic** messages that subsequently arrive) and changes its local part of **P**'s state to the values it would have in some default legal global state. Vanilla tokens are handled exactly as in StableSnap, with the addition that receiving a token that starts a new snapshot causes a process to resume execution of **P** if it were suspended. The correctness proof is deferred to a fuller version of the paper.

8 Complexity

Gouda and Evangelist have defined the *convergence span*[GE88] of a self-stabilizing system to be the maximum number of "critical" steps that must be executed before a legal state is reached. For StableSnap, consider statement (START) to be a critical step. Then because each critical step increments process 0's *Current* variable by 1, the convergence span of StableSnap may be as much as the difference between the initial value of process 0's *Current* variable and the maximum of V, where V is the set of values that are present in the initial state as values of *Current* variables or VAL fields of messages.

The convergence span may be reduced by having a process include its *Current* value in each *report* message that it creates. Process 0, in executing statement (START), may then simply choose a value that exceeds the maximum value received in the last set of *report* messages. This reduces the convergence span to at most the cardinality of the subset of V that contains values greater than or equal to the initial value of process 0's *Current* variable. If by chance there are no initial *token* or *report* messages (only arbitrary *Current* values), the modified version will converge on the second iteration to an accurate snapshot, followed by an accurate reset, if necessary.

Due to the completely asynchronous model, and the need for prod messages to avoid deadlock, under the assumptions seen here there is no way to bound the number of messages used for snapshots. The initiating process could generate arbitrarily many prod messages before any other process completes its snapshot. A more realistic model would use absolute time intervals and time-outs to control the frequency of snapshots and prodding.

There is also a possible trade-off on the frequency of snapshot operations, relative to operations of the **basic** computation. If snapshots are taken frequently, the system will self-stabilize when necessary, without wasting significant **basic** computation time. In this case, the number of steps needed afterwards to keep taking snapshots is large relative to the **basic** computation. On the other hand, if snapshots are only taken after some large number of **basic** computation steps, the self-stabilization will be slower, but the subsequent cost will be lower. Of course, this assumes that the **basic** computation cannot deadlock, even from illegal initial states.

9 A bounded implementation of StableSnap

In this section we describe a Bounded StableSnap that differs from the original in that it eliminates the unbounded growth in value of the *Current* variables. Bounding these values by a constant M complicates the algorithm because copies of *token* and *report* messages with identical values are created on different iterations (since, modulo M, the values v and $v + m \cdot M$ are indistinguishable for any $m > 0$). Therefore, care must be taken to ensure that, in any state, all *token* and *report* messages with VAL = v were created on the *same* iteration. We achieve this mainly by restricting the conditions under which a process may send a *report* message. The restriction will result in a bound on the number of iterations that a copy of a *token* or *report* message may remain in the system. Once this bound is exceeded, no ambiguity results in having process 0 create a new *token* with an identical VAL field.

For this section, we retain all of the assumptions of the original model but assume an upper bound N on the number of messages present in the initial states. Note that no bound on the capacities of channels is necessary.

The following modifications to StableSnap suffice:

1. The *Current* variable of process 0 is incremented modulo M (in statement (START)), where $M = 1 + n \cdot (N + 2)$.

2. Each process i maintains *Recent*$[k]$, $0 \leq k < 2 \cdot n$ as the k^{th} most recent value it has received in the VAL field of a *token* message.

3. Predicate **IsNext** is re-defined to recognize only values that are different from *Recent* values:
$$\textbf{IsNext} \equiv \left(\bigwedge_{k=0}^{2 \cdot n - 1} v \neq Recent[k] \right)$$

4. Predicate **Finished** is strengthened such that non-*Current* values satisfy the predicate only if identical values are received on each incoming channel:
$$\begin{aligned} \textbf{Finished} \equiv \; & (v \neq Current \wedge \textbf{AllEqual}(v)) \\ & \vee (v = Current \wedge \textstyle\bigwedge_{k:(k,i) \in \mathbf{E}} \; eoc[k]) \end{aligned}$$
where
$$\textbf{AllEqual}(w) \equiv \left(\bigwedge_{k:(k,i) \in \mathbf{E}} w = LastProd[k] \right)$$
and variable *LastProd*$[j]$ is maintained by each process i as the value of the VAL field of the most recent "prod" message received from process j.

10 Proof of correctness of Bounded StableSnap

We briefly sketch the main ideas behind the proof. Most of the work is in establishing a bound on the values that variables and VAL fields may have in any system state. As a convenience, define the *start of an iteration* to be the state in which a (START) statement is executed, the *end of an iteration* to be the first successor state in which *reported*$[k]$ is true for all k, and the "iteration number" to be the number of times process 0 has executed a (START) statement (which is equal to its *Current* variable, assuming w.l.o.g. that it begins at 0). We start by defining a relationship between messages.

Definition 6 *Message m' is an* immediate copy *of message m if*

- *m' is sent by some process i as part of the action that it executes upon receiving message m, and*
- *m' is identical to m except that the* PATH *of m' is the concatenation of i to the* PATH *of m.*

"Copy" is the reflexive, transitive closure of the "immediate copy" relation.

We first show that in every execution sequence, all messages are eventually copies of ones actually sent. States with this property are called *purged.* Recall that the initial state may have messages in the channels; purged states exclude copies of such messages.

Lemma 4 (Eventuality of a Purged State) *In every execution sequence, there is a $P < M$ such that a purged state occurs by the end of iteration P.*

Proof An iteration that begins in a non-purged state may end only when process 0 receives a *report* from each process. By analyzing the conditions under which a *report* may be received, and by the FIFO channel assumption, it is easy to show that some process receives a copy of a message that existed in the initial state. Because a message may be copied at most n times, and by assumption there are at most N messages in any initial state, a purged state occurs by iteration $n*N$. ∎

Next we show that no message may forever remain in a channel; it must eventually be received (and perhaps copied).

Lemma 5 (Aging of Tokens) *In every execution sequence, at the end of each iteration after iteration number P, there are no token or report messages that were present at the beginning of the iteration. (Although* copies *of these messages may be present.)*

Proof Let α be the start of iteration number $v > P$ and assume (for the moment) that immediately prior to α, no *report* or *token* message with VAL = (v mod M) exists. Then, by construction of the protocol, the end of the iteration cannot occur unless each process i has received from each j (such that $(j, i) \in \mathbf{E}$) at least one copy of a *token* created after α. Since channels are FIFO, i must first receive (and perhaps copy) any message that was already present in channel (j, i) in α. Hence, the claim will follow once the assumption is proved.

The assumption is proved by induction on v. As a basis, consider iterations numbered v, where $P < \mathrm{v} \leq P + 2 \cdot n$, The claim holds trivially since for iterations numbered $P < \mathrm{v} \leq M$, no message with VAL = v can exist until the start of iteration v. For iterations $\mathrm{v} > P + 2 \cdot n$, assume the claim holds inductively for all lesser numbered iterations and suppose, to the contrary, that a *report* or *token* message with VAL = (v mod M) existed in α. By construction of the protocol, this message could have been created no later than iteration $\mathrm{v} - M$. But, by the inductive hypothesis there have been at least $2 \cdot n$ iterations since iteration $\mathrm{v} - M$ in which every message in each channel has been copied. Since a message may be copied no more than n times, the claim follows. ∎

With this result, we can bound the values of the VAL fields of messages and the contents of the *Recent* variables. Then good states can be shown to repeatedly occur, and accurate snapshots to be eventually taken, as in the previous proof.

References

[BGW89] G. M. Brown, M. G. Gouda, and C. L. Wu. Token systems that self-stabilize. *IEEE Transactions on Computers*, 1989. to appear.

[BP89] J. Burns and J. Pachl. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11:330–344, 1989.

[CL85] K.M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[Dij74] E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[GE88] M. Gouda and M. Evangelist. *Convergence/Response tradeoffs in concurrent systems*. Technical Report TR88-39, University of Texas at Austin, 1988.

[Lam84] L. Lamport. The mutual exclusion problem: part ii - statement and solutions. *Journal of the ACM*, 33(2):327–348, 1984.