Distrib Comput (1993) 7: 17–26

DISTRIBUTED COMPUTING

Self-stabilizing extensions for message-passing systems

Shmuel Katz[1,*], **Kenneth J. Perry**[2]

[1] Technion, Israel Institute of Technology, Haifa 32000, Israel
[2] IBM Research, P.O. Box 704, Yorktown Heights, NY 10598, USA

Received February 1990 / Accepted November 1991

Shmuel Katz received his B.A. in Mathematics and English Literatue from U.C.L.A., and his M.Sc. and Ph.D. in Computer Science (1976) from the Weizmann Institute in Rechovot, Israel. From 1976 to 1981 he was a researcher at the IBM Israel Scientific Center. Presently, he is an Associate Professor in the Computer Science Department at the Technion in Haifa, Israel. In 1977–78 he visited for a year at the University of California, Berkeley, and in 1984–85 was at the University of Texas at Austin. He has been a consultant and visitor at the MCC Software Technology Program, and in 1988–89 was a visiting scientist at the IBM Watson Research Center. His research interests include the methodology of programming, specification methods, program verification and semantics, distributed programming, data structures, and programming languages.

Kenneth J. Perry has performed research in the area of distributed computing since obtaining Masters and Doctorate degrees in Computer Science from Cornell University. His current interest is in studying problems of a practical nature in a formal context. He was graduated from Princeton University in 1979 with a B.S.E. degree in Electrical Engineering and Computer Science.

Summary. A self-stabilizing program eventually resumes normal behavior even if execution begins in an abnormal 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 one program being a *self-stabilizing extension* of another; (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.

Key words: Self-stabilization – Message-passing – Superimposition

* The research of this author was partially supported by Research Grant 120-749 and the Argentinian Research Fund at the Technion

Correspondence to: K.J. Perry

1 Introduction

The concept of a *self-stabilizing* distributed program, introduced by Dijkstra [6], requires that a program executing with an arbitrary initial state (including arbitrary control locations) eventually reaches a *legitimate* state and thereafter remains in legitimate states. In other words, the property "the program is in a legitimate state" is stable (see [4]) and eventually true. The arbitrary initial state represents the situation immediately following a fault or reconfiguration of a dynamic system. This contrasts with the normal situation in which a program's correct behavior is predicated on the initial state satisfying particular properties.

Designing self-stabilizing programs is difficult because a process cannot always distinguish between an arbitrary initial state and a legitimate state that occurs during a normal computation. For example, a flag that is ordinarily set only following some action may be true in the initial state without the action having occurred. Thus, reasoning that is correct for ordinary programs may not be appropriate when self-stabilization is considered.

In spite of the difficulties, many self-stabilizing algorithms have been published [6, 13, 7, 3, 2]. The elegance of such custom-made self-stabilizing programs is often

impressive. However, a more generally applicable approach in our view requires separating the self-stabilization from the original algorithm design. This separation reduces the complexity of design and encourages the reuse of valuable techniques that are otherwise hidden when encoded in particular algorithms.

The goal of this work is to show that it is possible to transform an existing algorithm into a self-stabilizing version in a uniform manner. To accomplish this goal, we define the concept of one program being a *self-stabilizing extension* of another program and characterize the properties that may eventually hold in such an extension. We then demonstrate the possibility of semi-automatically creating such extensions via the superimposition of self-stabilizing distributed algorithms for taking global snapshots and performing resets.

1.1 Background

Dijkstra [6] first 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 [7, 3, 2] follow Dijkstra in that they deal with mutual exclusion or token-passing and create self-stabilizing programs from scratch.

Lamport's work on self-stabilizing mutual exclusion [13] differs from previous work in that the algorithms are created by adding statements to existing programs. It is also among the few results providing a precise semantics for self-stabilization. Lamport [13] defined *transient malfunction* behavior to be an execution in which each process initially assigns arbitrary values to its variables and location counter and subsequently executes according to its program. In Lamport's paper, an algorithm is *self-stabilizing for a property* if the property eventually holds for every such behavior. He leaves open the domain of values in the malfunction operation.

1.2 A message-passing system

Previous results on self-stabilization generally employ a shared-memory model of computation. In contrast, the computational model used in this paper is that of an *asynchronous message-passing system.* A *message-passing system* is a collection of *processes* (named 0 through $n-1$) that are connected by FIFO communication *channels.* The interprocess connections are described by a set **E** of ordered pairs of process names. Processes may exchange values only by sending and receiving *messages* on the channels; it is assumed that every message in a channel is eventually received. However, the system is *asynchronous* in that there are no bounds on message delivery time, channel capacities or relative process speeds.

What is unusual about a self-stabilizing system is that in the initial state of such a system, process states, location counters and channels may have arbitrary values. No assumptions are made about these values other than that they are from the appropriate domain. This introduces several problems that are not present when self-stabilization is considered in the shared-memory model. Note that, although simulations of message-passing by shared-memory (and vice-versa) exist, they are not self-stabilizing.

The problem of the *apparently sent message* arises from an initial state in which a channel between neighboring processes is empty but the local process states are consistent with ones in which each awaits a reply to a message that it has just sent. In such an initial state, a program that is deadlock-free when execution starts in a "normal" initial state may find itself deadlocked. Unlike a shared variable, a channel cannot be read by the sender so the sender is unable to detect when this situation has occurred.

Another problem can be caused by an initial state in which messages are present in the channels. These messages, and any messages generated as a result of receiving them, may prevent the attainment of a legitimate global state. At some point any harmful message resulting from the channel contents of the initial state must be purged from the system.

2 The semantic model and its difficulties

We adopt the usual definitions of the interleaving execution model of concurrent systems: a *local state* of a process is an assignment of values to local variables and the location counter; a *global state* of a system of processes is the cross product of the local states of its constituent processes plus the contents of the channels; the *underlying semantics of program operations* is the set of atomic steps and their associated state changes; an *execution sequence* of a program is an infinite sequence of global states in which each element follows from its predecessor by execution of a single atomic step of the program. (As a technical convenience, the execution sequence of a terminating program is the sequence derived by repeating the final state forever.)

A program is expressed as a collection of *guarded statements*, each consisting of a condition (called a *guard*) and a sequence of commands that implement atomic steps. When the guard of a statement is true, the associated sequence of commands may be executed. Nondeterministic choice is assumed over statements with true guards. We also assume that every statement whose guard remains true is eventually selected for execution.

Since our goal is to show how a self-stabilizing program may be derived from an existing program, we now make precise the definition of a self-stabilizing program and a self-stabilizing extension of a program. Let **P** be a program and let *sem* (**P**) ("the semantics of **P**") denote the set of all possible execution sequences of **P**. The "legal" states of **P** could be defined in either of two ways: as the states satisfying a predicate (called **P**'s *specification*) or as the states reachable from a normal initial state. For the purpose of defining a self-stabilizing extension of a program we choose the latter. The *normal initial states of* **P** are those states in which the location counter

of each process is 0 and all channels are empty; the *legal semantics of* **P** (i.e., the semantics intended by the programmer), denoted by *legsem* (**P**), is the subset of *sem* (**P**) containing exactly those sequences with a normal initial state. Every sequence in *legsem* (**P**) is defined to be a *legal execution sequence* and every state in such a sequence is defined to be a *legal state.*

Note that, in general, not all global states are legal since there are many combinations of values that cannot arise in any legal execution sequence. The *illegal execution sequences* are those with initial states that are not legal. A third class of execution sequences are those with initial states that are legal but not normal initial states, i.e., states that ordinarily occur only in the middle of an execution sequence. Such sequences 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* iff 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, every execution of a self-stabilizing program eventually resembles a legal execution sequence. Now we turn to the relationship between a program and an extension of a program. An extension of an existing program is typically (but not necessarily) derived by augmenting the original program with new variables and message types. We therefore need to distinguish between variables/message types that are common to the original program and its extension, and those unique to either. Toward this end assume that, by convention, each message is adorned with a "type field". The *projection of a state onto a set of variable names and message types* is the state obtained by including only those variables and messages with names and types that are in the set. Similar notions of projection for sequences of states and sets of sequences of states are defined in the obvious manner.

Definition 2. Program **Q** is an *extension* of program **P** iff the projection of *legsem* (**Q**) onto the set of variables and message types of **P** is identical to *legsem* (**P**), up to stuttering[1].

Definition 3. Program **Q** is a *self-stabilizing extension* of program **P** iff **Q** is self-stabilizing and 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 finite repetition of states (stuttering) is ignored. Moreover, **Q** must be self-stabilizing for all its computations. When started 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 initial state that is either illegal or not normal. Note that no relationship is required between either the illegal computations of **P** and **Q** or among the location counters.

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

No correspondence between location counters is necessary because **Q** need not have the same control structure as **P**. In particular, **Q** generally has no state in which all processes have halted even if **P** is a terminating program. If **Q** had such a state then the initial state of an execution sequence could have all processes halted even though **P** might not be in a legal state. Therefore, the projection onto **P**'s variables and message types of a self-stabilizing extension of a terminating program **P** contains execution sequences in which **P**'s final state repeats forever.

3 Limitations on self-stabilizing extensions

We now consider which properties can hold in a self-stabilizing extension. A simple example illustrates the potential problems. Suppose the legal semantics of **P** restricts the values "printed" to the sequence "1, 0, 0,...". Then a program **Q** that prints this sequence when started in a normal initial state but which prints only all-zero sequences when started in any other state could be a self-stabilizing extension of **P** even though it does not share one of the key properties of **P**'s legal computations: "a 1 is always printed". This discrepancy arises because *legsem* (**P**) has a sequence with a suffix that does not have the property. 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 over variables and messages of* **P** *which is in the future fragment of linear temporal logic, then:*
A holds for a suffix of every sequence of sem (**Q**) *iff for each sequence in legsem* (**P**)*, A is true infinitely often.*

Proof. The future fragment of linear temporal logic [15, 12] restricts an assertion over execution sequences to refer only to states in the sequence that include and follow the state in which the assertion is evaluated. So the truth of A over a suffix S of sequence T does not depend on the states of T preceding the initial state of S.
$\Rightarrow$. Since A refers only to the suffix of a sequence and the suffix of any sequence in *sem* (**Q**) is itself a sequence in *sem* (**Q**), then if A is true over some suffix of each sequence in *sem* (**Q**), A is true infinitely often over each sequence in *sem* (**Q**). Because **Q** is an extension of **P**, *legsem* (**P**) is contained in the projection onto **P**'s variables and message types of *sem* (**Q**). Thus, A is true infinitely often over each sequence in *legsem* (**P**).
$\Leftarrow$. If A is true infinitely often over each sequence in *legsem* (**P**), then A is true infinitely often over each suffix of each sequence in *legsem* (**P**). Since **Q** is a self-stabilizing extension of **P**, the projection onto **P**'s variables and message types of some suffix of each sequence in *sem* (**Q**) is identical to the suffix of some sequence in *legsem* (**P**). Assertion A refers only to the variables and message types in **P**; therefore A is true infinitely often over some suffix of each sequence in *sem* (**Q**) and thus is true infinitely often over each sequence in *sem* (**Q**). □

Note that if **P** is a terminating program, an assertion A is true infinitely often for each sequence in *legsem* (**P**) precisely when A is true in the repeating final state.

Until recently only temporal logics with future modalities were used in specifications. Although logics with past modalities have also been introduced [14] and seem to aid modularity of specification, in the specialized context of self-stabilization the versions based on future modalities seem preferable, due to the above theorem. Note that properties that do not hold in a self-stabilizing extension may be unnecessarily strong and can often be replaced by ones that do hold. For example, the assertion that "eventually, infinitely often the number of responses equals the number of queries" may never become true for an execution sequence in which there are already 2 responses in the inital state. However, the assertion that "eventually, infinitely often a query is sent and a matching response is returned in a later state" can eventually be true and may sufficiently capture the desired behavior.

Another interesting limitation regarding self-stabilization is that, in general, no process executing a self-stabilizing program can ever *know* (in the precise sense defined in [9, 8]) that the system is in a legal state even though the eventuality of a legal state is guaranteed. Suppose there were some local state of process i that indicated a legal global state. Then, except for trivial cases, there are many illegal initial global states that are consistent with this local state of i, contradicting the validity of i's indicator. The effect of this limitation is to prevent a self-stabilizing algorithm from using knowledge of the occurrence of a legal state to intentionally switch to a more efficient mode of operation that cannot recover from an illegal state.

4 Creating self-stabilizing extensions

In this section we show how to semi-automatically create a self-stabilizing extension of a distributed program **P**. Our technique is to create an extended program by *superimposing* [1, 10] onto **P** a self-stabilizing "supervisor" program. By making the supervisor self-stabilizing, we guarantee that eventually all global states of the extended program have projections onto the supervisor's variables and message types that are legal states of the supervisor. From that point on, the supervisor acts to ensure that eventually all future global states of the extended program have projections onto the variables and message types of **P** that are legal states of **P**. Once the extended program's state is legal for both the supervisor and **P**, the program's actions do not interfere with **P** in any functional manner. Thus the extended program is a self-stabilizing extension of **P**.

The essence of the supervisor is to repeatedly obtain a snapshot of **P**'s global state, evaluate whether the snapshot indicates an illegal state of **P**, and, when it does, reset the global state to one whose projection is a normal initial state of **P**. Toward this end, in the following subsections we describe self-stabilizing global snapshot and global reset algorithms. Implementing these algorithms is difficult since they too must function correctly no matter what the initial state.

Note that some number of inaccurate or incomplete snapshots may occur initially when execution starts in a state that is not a normal initial state of the extended program. We relate the number of such snapshots to properties of the initial state in Sect. 6.

4.1 Overview of the snapshot algorithm

Our supervisor program is derived from the single-snapshot algorithm of Chandy and Lamport [4]. That algorithm may be superimposed onto **P** to enable an "evaluation process" to obtain an accurate snapshot for the state in which it begins executing. (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.)

In the Chandy-Lamport algorithm, a process either spontaneously initiates the algorithm or begins participating upon receiving a first *marker* message from one of its neighbors. The first action taken by a process during the single-snapshot algorithm is to record its local state and send markers to all neighbors. Subsequently, for each neighbor from which no marker has yet been received, the process records on a queue all messages of **P** received from that neighbor. Once a marker has been received from a neighbor, recording of that neighbor's messages stops; once markers have been received from all neighbors, the process has completed its participation in the snapshot. As a final action, the process sends a *report*, which contains the state and queue data that it has recorded, to a central evaluation process that assembles the reports into a snapshot of the global state. Other than being recorded in a queue, **P**'s messages are not delayed or altered, so the superimposition of the single-snapshot algorithm does not affect **P**'s correctness. Note that this algorithm is guaranteed to terminate with an accurate snapshot only for legal execution sequences.

Our supervisor causes the evaluation process to repeatedly initiate iterations of the single-snapshot algorithm. It also adapts the snapshot algorithm to allow it to reset the global state as well as record it. To cope with states with undesired snapshot messages or values of variables that do not reflect the intended properties of the state, the supervisor also implements techniques that distinguish between successive iterations of the snapshot, ensure that message are not copied without bound (to avoid flooding the system with outdated messages), and, of course, guarantee that eventually snapshots will be accurate. The following subsections explain how each of these goals is achieved.

4.1.1 Distinguishing between iterations. In order to distinguish between separate iterations of the snapshot we use round numbers. Each snapshot message is stamped with a round number and each process maintains a variable containing a round number; upon receiving a message a process compares the values stamped on the message and stored in the variable. A process interprets a message as belonging to the ongoing iteration if the values are equal; when the message's value is greater than the variable's, the message is interpreted as a signal to begin participating in a new iteration; if the variable's value

exceeds the message's, the message is interpreted as being left over from a previous iteration. To ensure termination of each iteration, the contents of the message are propagated to all neighbors in both of the latter cases. Note that these interpretations are consistent with the behavior of processes when the state of the extended program is legal. In an initial state that is not normal, the round values stamped on messages and stored in variables are arbitrary, so initially the intepretations may not be consistent with the actual behavior of processes.

Our description of the single-snapshot algorithm distinguished the role of one process: the central evaluation process that detects the end of the snapshot and assembles reports into a global snapshot. The supervisor further endows this process with the distinction of being the only one that spontaneously initiates a new iteration of the single-snapshot. As a consequence, the evaluation process can choose which round number to stamp on a message while the other processes are constrained to pass on the round value that was stamped on the last message received. We will show that once the evaluation process chooses a round number that exceeds the value stamped on any message or stored in the round variable of any process, it will eventually obtain an accurate snapshot.

4.1.2 Preventing copying and ensuring progress. In order to prevent the information in a message from being passed from process to process without limit, each snapshot message contains a list of process identifiers. If a process receives a message containing a list in which its own identifier appears, the information is not passed on to any neighbor; otherwise, the information is passed on only after the process has inserted its own identifier into the list. Thus a process is prevented from passing on the information contained in a message more than once.

A supervisor deadlock in an illegal state would be possible if every process had a local state in which it could send no message until it received a new message. This is avoided by causing the evaluation process to be the only one able to repeatedly generate spontaneous messages, an act that we call *prodding*. The evaluation process prods by sending (perhaps not for the first time) a marker message stamped with a round number equal to the value stored in its own round variable.

4.1.3 Evaluating snapshots and resetting the global state. A predicate the recognizes legal states of **P** is needed so that the evaluation process can determine when a reset of the global state is necessary. This predicate must be supplied as input to the transformation, which is why the transformation is not completely automatic. Note that even with such a predicate a process cannot determine whether the snapshot is accurate (i.e., reflects the actual state of the system) or is due to artifacts of an initial state that is not a normal initial state of the extended program.

In the best case the predicate would compare the snapshot to a precise characterization of the legal global states of **P**. But this characterization may be difficult to specify and may be more precise than necessary (e.g., involving "temporary" variables of processes). A weaker characterization relating only key variables may often suffice. The only requirements needed to ensure that the extended program is a self-stabilizing extension of **P** are that the predicate be true for legal states and that every illegal computation of **P** either eventually leads to a legal state on its own or results in an unbounded sequence of illegal states that violate the predicate (in which case a reset establishes a legal state). A possible relaxation of this requirement is discussed in the final section. For some properties such as liveness it may be necessary to accumulate a sequence of successive snapshots.

Once the evaluation process determines that a snapshot represents an illegal state, it initiates a change of the global state to one whose projection is a normal initial state of **P**. With a few modifications, the techniques that are used to obtain a snapshot may be used to create this normal initial state. The main difference between a snapshot and reset iteration is that, in a reset, a process alters its local state (to its local part of a normal initial state) before recording it. It then suspends execution of **P** and discards incoming messages of **P** while continuing execution of the snapshot. If the preceding snapshot was accurate, when the reset iteration ends the state of every process is as it would be in a normal initial state of **P**. The next iteration of the snapshot algorithm frees each process to resume executing **P**.

4.2 A self-stabilizing snapshot algorithm

4.2.1 Message fields and variables. The tasks described abstractly above are concretely implemented in an algorithm we call StableSnap using multi-part messages. Each of StableSnap's messages has a RND and ID field. The RND field implements the round number that is stamped on a message; the ID field implements the list of process identifiers that is contained in a message. A local variable named R implements the round value stored by each process and a Boolean variable *reset* indicates whether the process is to behave as in the reset or snapshot algorithm.

A message also has a TYPE field that further specializes it. A value of **snapshot** in this field indicates the presence of a field called RESET, which is used by the reset algorithm; a value of **report** indicates the presence of a field called SNAP, which contains a report (i.e., the local state and queue contents recorded by a process during the single-snapshot iteration). We assume that **P**'s statements create messages with a TYPE field value of **basic** so that **P**'s messages may be distinguished from the supervisor's. In the tuple notation used in StableSnap, the fields appear in the following order: RND, ID, RESET or SNAP, and TYPE.

Variables used by the single-snapshot algorithm include s – which is used to record the local state, array q – which implements the queues used to record messages, and Boolean array *mark*, where $mark[j]$ being true indicates that a marker has been received from neighbor j. Process 0 implements the evaluation process and therefore has additional variables $snap[k]$, which store the report sent by each process k. In a legal state of the extended program, when $snap[k] \neq \textbf{nil}$ is true for all k, the *snap* array contains a snapshot of the global state.

4.2.2 Notation. When used in the code of StableSnap, i denotes a process's identifier and a field's name denotes the value contained in the field of the message just received. When the value of a field is not in the domain appropriate for that field, the message is ignored since it is obviously an artifact of an illegal initial state. Similarly, if no guard holds for a message, that message is ignored.

A shorthand is used for complex predicates and groups of statements. Predicates **Finished**, which recognizes either inconsistent round values or local termination of the single-snapshot algorithm, and **Seen**, which recognizes a message with information that a process has already passed on, are defined as:

- **Finished** $\equiv (\mathrm{RND} \neq R) \vee (\wedge_{k:(k,i)\in \mathbf{E}} mark[k])$
- **Seen** $\equiv (i$ appears in sequence ID$)$

A new iteration of the single-snapshot algorithm is started by the evaluation process by executing the statement labelled *Start*, which sends a marker to each neighbor. The statements labelled *Propagate* and *RelayReport* are respectively used to pass markers and reports on to neighbors. Finally, the statement labelled *Report* creates a report by packaging the local state and queues recorded during the snapshot into a message that is sent to all its neighbors (and eventually relayed to the evaluation process). The statements are defined as follows:

- *Start:* /* Initiate (or prod) a snapshot iteration */
  $\|_{k:(0,k)\in \mathbf{E}}$ **send** $\langle R, 0, reset,$ **snapshot**$\rangle$ **to** k
- *Propagate:* /* Pass **snapshot** message to all neighbors */
  $\|_{k:(i,k)\in \mathbf{E}}$ **send** $\langle$RND, **append** (ID, i), RESET, **snapshot**$\rangle$ **to** k
- *RelayReport:* /* Pass **report** message to all neighbors */
  $\|_{k:(i,k)\in \mathbf{E}}$ **send** $\langle$RND, **append** (ID, i), SNAP, **report**$\rangle$ **to** k
- *Report:* /* Send local snapshot to process 0 via neighbors */
  $local\_snap := (s, \{\langle l, q[l]\rangle \mid (l,i) \in \mathbf{E}\})$;
  $\|_{k:(i,k)\in \mathbf{E}}$ **send** $\langle$RND, i, $local\_snap$, **report**$\rangle$ **to** k

4.2.3 Behavior of processes. Since we have defined the evaluation process to be the only one able to spontaneously initiate a new iteration of the single-snapshot algorithm, every other process waits to receive a message before acting. The code of these processes is shown in the guarded statements of Fig. 1. The first four guarded statements correspond to receiving a message that is respectively interpreted as: a marker of the ongoing iteration, a message containing information that has already been passed on by the process, the first marker received from any process in a new iteration of the single-snapshot algorithm, and a message left over from a previous iteration. The statement labelled (NEXT) includes the first

On receiving a message with TYPE = **snapshot** from process j:

(SNAP) $\mathrm{RND} = R \wedge \neg mark[j]$
$\longrightarrow mark[j] := \mathbf{true}$;
if **Finished then** *Report* **fi**

(DEJA) $(\mathrm{RND} \neq R \vee mark[j]) \wedge$ **Seen**
$\longrightarrow$ **skip**

(NEXT) $\mathrm{RND} > R \wedge \neg$**Seen**
$\longrightarrow R, reset := \mathrm{RND}, \mathrm{RESET}$;
if *reset* **then** local state := *NS* **fi**;
/* Actions of single-snapshot alg. */
$s :=$ local state;
$q[j], mark[j] :=$ **nil, true**;
$\|_{k:(k,i)\in \mathbf{E},\ k\neq j}$ $q[k], mark[k] :=$ **nil, false**;
Propagate

(PROD) $(\mathrm{RND} < R \vee (\mathrm{RND} = R \wedge mark[j])) \wedge \neg$**Seen**
$\longrightarrow$ *Propagate*;
if Finished then *Report* **fi**

On receiving a message with TYPE = **report** from process j:

(PASSREP) $\neg$**Seen**
$\longrightarrow$ *RelayReport*

On receiving a message m with TYPE = **basic** from process j:

(BASIC) $\neg reset$
$\longrightarrow$ **if** $\neg mark[j]$
then $q[j] :=$ **append** $(q[j], m)$ **fi**; **P**

Fig. 1. Snapshot algorithm for process $i > 0$

actions that a process takes when participating in a new iteration of the single-snapshot algorithm (i.e., recording the local state, initializing the queues, and propagating the contents of the message). It also extracts the RESET field from the message and, if its value is true, behaves as it would in the reset algorithm (i.e., changes its local state to *NS*, which denotes the state it would have in a normal initial state).

The guarded statement labelled (PASSREP) relays reports on to its neighbors. When *reset* is false, statement (BASIC) appends one of **P**'s messages to a queue (providing no marker has yet been received on the channel during this iteration) and invokes **P**, as is done when the single-snapshot algorithm is superimposed onto **P**.

The code of the evaluation process (process 0) is shown in Fig. 2. Since it is the process that assembles reports into a global snapshot of the ongoing iteration, upon receiving a report it executes statement (REPT), which copies the report to the local variable $snap[k]$.

To begin a new iteration, the process executes statement (ITERATE). Note that this statement is enabled only when the local *snap* variables indicate that the ongoing iteration has terminated. Predicate **legal** is applied to the collection of reports in order to evaluate whether the snapshot is a legal state of **P** and variable *reset* is set accordingly.

Statement (ITERATE) is analogous to statement (NEXT) in that it includes the first actions of a process that spontaneously begins participating in a snapshot. Similarly, (SNAPZ) is analogous to statement (SNAP) in that it records the fact that a marker has been received

On receiving a message with TYPE = **snapshot** from process j:

(SNAPZ) RND = $R \wedge \neg mark[j]$
$\longrightarrow$ $mark[j] :=$ **true**;
if Finished then
$snap[0] := (s, \{\langle l, q[l]\rangle \mid (l, 0) \in \mathbf{E}\})$
fi

(SELFCNTL) RND $\neq R \vee mark[j]$
$\longrightarrow$ **skip**

On receiving a message with TYPE = **report** from process j:

(REPT) RND = R
$\longrightarrow$ $k := first(\mathrm{ID})$;
if $(snap[k] =$ **nil**) **then** $snap[k] :=$ SNAP **fi**

On receiving a message m with TYPE = **basic** from process j:

(BASICZ) $\neg reset$
$\longrightarrow$ **if** $\neg mark[j]$
then $q[j] :=$ **append** $(q[j], m)$ **fi**; **P**

Process 0, spontaneously:

(GENPROD) **true**
$\longrightarrow$ *Start*;
if Finished then
$snap[0] := (s, \{\langle l, q[l]\rangle \mid (l, 0) \in \mathbf{E}\})$
fi

(ITERATE) $\wedge_{k=0}^{n-1}(snap[k] \neq$ **nil**)
$\longrightarrow$ $R, reset := R + 1, \neg$ **legal** $(snap)$;
$\|_{0 \le k < n}$ $snap[k] :=$ **nil**;
if *reset* **then** local state := NS **fi**;
/* Actions of single-snapshot alg. */
$s :=$ local state;
$\|_{k:(k,0) \in \mathbf{E}}$ $q[k], mark[k] :=$ **nil**, **false**;
Start

Fig. 2. Snapshot algorithm for process 0

from a process during the ongoing iteration. Statement (GENPROD) is responsible for the spontaneous generation of prod messages that ensure that StableSnap cannot deadlock with every process waiting to receive a message. (GENPROD) and (SNAPZ) also include code to copy process 0's part of the snapshot to $snap[0]$, thereby simulating process 0 sending and receiving its own report. For the purposes of the correctness proof in the following section, this is considered to be equivalent to process 0 executing *Report*.

5 Correctness

Our goal is to demonstrate that the superimposition of the supervisor program onto **P** is a self-stabilizing extension of **P**. Since StableSnap does not interfere with **P** (except when performing a reset in an illegal state) our primary obligation is in showing that StableSnap eventually repeatedly obtains accurate snapshots. An essential part of the proof is showing that the values present in the initial state as values of the variables and messages of StableSnap are eventually "flushed" from the system and leave no after-effects. We do this by proving that every state has a successor in which statement (ITERATE) is executed, thereby increasing the value of process 0's R variable.

For the sequel, choose any state and let r_0 be the value of process 0's R variable in this state. The initial lemmas demonstrate that, unless statement (ITERATE) is executed in a successor of the chosen state, execution of StableSnap leads to a state in which each message has a RND field equal to r_0.

Lemma 1. *Either statement* (ITERATE) *is eventually executed, or eventually the* RND *field of every* **snapshot** *message is equal to r_0 in all states until the next execution of statement* (ITERATE).

Proof. Suppose (ITERATE) is not executed. Since only the actions *Propagate* and *Start* send **snapshot** messages, the RND field of any **snapshot** message that subsequently may be sent is equal to either the RND field of an existing **snapshot** message (if the new message is sent by *Propagate*) or equal to r_0 (if the new message is sent by *Start*). Using a multiset ordering [5] we show that execution of StableSnap leads to a state in which no **snapshot** message has a RND field different than r_0 and that this remains true as long as (ITERATE) is not executed. For each state and each $r \neq r_0$, define S_r to be a multiset that has one element for each message in the state that has a RND field equal to r, and no other elements. The element of S_r corresponding to a message is the set of names of processes in the system that are not included in the ID field of the message.

Consider the effect on S_r of the statement executed upon receiving a **snapshot** message with a RND field equal to r. Either no **snapshot** message is sent by the statement, thereby reducing the size of S_r, or a finite number of **snapshot** messages each with a RND field equal to r are sent (by *Propagate*). In the latter case, by definition of *Propagate* and S_r, one set in S_r is removed (corresponding to the message received) and replaced by a finite number of sets each of cardinality less than that of the set removed. Since every message with a RND field equal to r is enventually received and both possibilities result in an S_r that is smaller in the multiset ordering, eventually S_r will be empty for every $r \neq r_0$, by the well-foundedness of the ordering. Therefore, there will be no **snapshot** message with a RND field different than r_0. As long as (ITERATE) is not executed, this will remain true. □

Lemma 2. *Either statement* (ITERATE) *is eventually executed, or eventually the* RND *field of every* **snapshot** *and* **report** *message is equal to r_0 in all states until the next execution of statement* (ITERATE).

Proof. Suppose (ITERATE) is not executed. Then, by the preceding Lemma, execution of StableSnap leads to states in which every **snapshot** message has a RND field equal to r_0. By definition of StableSnap, the RND field of any report message that subsequently may be sent is equal to either the RND field of an existing **report** message (if the new message is sent by *RelayReport*) or equal to r_0 (if the new message is sent by *Report*).

For each state in which any **snapshot** message has a RND field equal to r_0, define multisets R_r as in the pre-

ceding proof, but this time based on **report** messages. Now consider the effect on R_r of the statement executed upon receiving a **report** message with a RND field equal to r. As in the previous proof, the value of R_r in the multiset ordering decreases, because either no message is sent by the statement or the sets corresponding to the **report** messages sent (by *RelayReport*) are of smaller cardinality than the set corresponding to the message received.

Messages of type **report** may also be sent by *Report* upon receiving a **snapshot** message. But each such message has a RND field equal to r_0 so by definition of *Report*, any **report** message sent as a result has no effect on any R_r. Thus, eventually there will be no **report** message with a RND field different than r_0, by the well-foundedness of the multiset ordering. □

The preceding lemma enables us to focus on states in which each message has a RND field equal to r_0. The next series of lemmas apply only to such states.

Lemma 3. *Either statement* (ITERATE) *is eventually executed, or eventually there is a state in which every process has sent to each neighbor at least one* **snapshot** *message with a* RND *field equal to* r_0.

Proof. Suppose (ITERATE) is not executed. Then by the preceding lemma, execution of StableSnap leads to states in which any message has a RND field equal to r_0. We begin by observing a simple fact about such states. If a process $i > 0$ receives l **snapshot** messages from a neighbor k, and each message has a RND field equal to r_0 and an ID field that does not contain i, it will relay at least $l-1$ of these messages to its neighbors. This follows since (SNAP) is the only statement that does not relay a **snapshot** message and its guard is falsified after the first **snapshot** message with RND field equal to r_0 is received from k (i.e., R becomes stable and $mark[k]$ becomes true).

Let the *level* of process i be the length of the shortest path from process 0 to i. If process 0 has executed statement (GENPROD) l times in a state in which its R variable equals r_0 and in which any message has a RND field equal to r_0, then each process on level L will eventually send to each neighbor at least $l-L$ **snapshot** messages, each of which has a RND field equal to r_0. This follows by induction on L: if $L=0$, the claim is trivial. For $L > 0$, a process i at level L has a neigbor j at level $L-1$ that, by the inductive hypothesis, eventually sends it $l-(L-1)$ such messages. By the eventual message-delivery assumption and the above fact, process i eventually relays $l-L$ of these messages.

Note that (GENPROD), the prodding statement of process 0, is continuously enabled so if (ITERATE) has not executed again, (GENPROD) will eventually execute $D+1$ times, where D is the maximum level. Thus, every process eventually sends to each neighbor at least one **snapshot** message with RND field equal to r_0. □

Lemma 4. *Either statement* (ITERATE) *is eventually executed, or eventually there is a state in which every process has executed Report and every process other than process* 0 *has sent to each neighbor at least one* **report** *message with* RND *field equal to* r_0 *and an* ID *field equal to its own name.*

Proof. By the previous lemma, eventually either (ITERATE) is executed or execution of StableSnap leads to a state in which each **snapshot** message has a RND field equal to r_0 and in which each process has sent to each of its neighbors at least one **snapshot** message with RND field equal to r_0. Since all messages are eventually received, process i receives at least one of these **snapshot** messages from each neighbor. Predicate **Finished** is true in the state in which i receives the **snapshot** message from the last of its neighbors from whom it had not previously received one of these messages, either because $mark[j]$ is true for each neighbor j or because its R variable is different than r_0, the RND field of the message. Thus process i executes *Report* and, if $i > 0$, the process sends a **report** message to each neighbor with a RND field equal to r_0 and an ID field equal to i. □

Lemma 5. *Either statement* (ITERATE) *is eventually executed, or eventually there is a state in which* ($snap[k] \neq$ **nil**) *is true for each k (and thus,* (ITERATE) *is enabled).*

Proof. By the previous lemma, eventually either (ITERATE) is executed or execution of StableSnap leads to a state in which process 0 has executed *Report* (thereby setting $snap[0]$) and each process $i > 0$ has sent to each neighbor a **report** message with a RND field equal to r_0 and an ID field equal to i. This is the basis for an inductive proof that each process $j > 0$ on a simple path from i to 0 eventually sends a **report** message with RND field equal to r_0 and ID field equal to a simple path from i to j. For the inductive step, note that statement (PASSREP) is enabled when j receives the message and therefore it executes *RelayReport*, appending j to the ID field. Thus, eventually process 0 receives the **report** message originated by i and sets $snap[i]$, following the actions of statement (REPT). □

We now have established enough facts to prove that StableSnap continuously iterates.

Theorem 2 (Iteration Liveness). *Every state has a successor in which statement* (ITERATE) *is executed, thereby increasing the value of process* 0*'s R variable. Therefore, this variable increases without bound.*

Proof. By the previous lemma, eventually either (ITERATE) is executed or execution of StableSnap leads to a state in which (ITERATE) becomes continuously enabled until it next executes. By the assumption of fairness (ITERATE) will be executed, resulting in the value of process 0's R variable being increased by one. □

Theorem 3 (Good Snapshot). *In any execution of* StableSnap, *eventually process* 0 *repeatedly obtains accurate snapshots for the state in which each* (ITERATE) *statement is executed.*

Proof. For any finite prefix of an execution sequence of StableSnap, let H denote the finite set such that $h \in H$ if and only if, in some state of the prefix, h is the value of either the R variable of some process or the RND field of some message. A state is defined to be *good* if and only

if the value of process 0's R variable in the state is equal to the maximum of H. Since (ITERATE) is the only statement that adds new elements to H, the R variable of each process is monotonically increasing (by the action of statement (ITERATE) and the guard of (NEXT)) and thus *good* is a stable property. Moreover, the eventuality of a good state is ensured by Theorem 2.

We will show that in the interval between consecutive executions of statement (ITERATE) in good states, when a reset is not being done, the behavior of the processes mimics an execution of the Chandy-Lamport single snapshot algorithm. Therefore an accurate snapshot is obtained in each such interval.

In the state immediately following execution of statement (ITERATE) in a good state, statement (ITERATE) is not enabled and r_0, the new value of process 0's R variable, is strictly larger than the R variable or RND field of any message of any previous state. Thus, (ITERATE) cannot become re-enabled until each process has executed *Report* at least once. By Lemma 3, eventually every process receives a **snapshot** message with RND field equal to r_0 from each neighbor.

Upon receiving the first **snapshot** message from any neighbor (call this neighbor j), statement (NEXT) is enabled at process $i > 0$. By definition of statement (NEXT), the value of process i's R variable becomes equal to the value of process 0's R variable (similarly for the *reset* variables) and process i executes the first statements of the Chandy-Lamport algorithm (recording the local state, initializing the queues, and propagating the message). Upon subsequently receiving a **snapshot** message for the first time from any neighbor $j' \neq j$, statement (SNAP) is enabled, resulting in recording of that channel being terminated and reporting if finished. Should i receive more than one **snapshot** message from the same neighbor (e.g., j) there is no effect on the snapshot (the guard of (SNAP) is false for the additional messages since $mark[j]$ is made true when the first message from j is received). Identical reasoning holds for the guards of process 0. Thus, the actions of StableSnap mimic those of the original Chandy-Lamport algorithm so the part of the state recorded by a process is as in that algorithm. By Lemma 4 and Lemma 5, each process sends this local snapshot via *Report*, process 0 receives it and thereby has a representation of an accurate snapshot in its *snap* variables, and statement (ITERATE) is once again enabled. □

Reasoning that is nearly identical to the above can be used to show that, if an iteration of StableSnap is started in a good state in which process 0's *reset* variable is equal to true, the iteration establishes a state whose projection onto **P**'s variables and message types is a legal state of **P**. Observe that since variables R and *reset* are altered only in the same simultaneous assignment statements, the preceding proofs could trivially be adapted to refer to the pairs $(R, reset)$ and (RND, RESET) rather than the individual R variable and RND field. As a result, Lemma 3 would show that the *reset* variables of all processes are equal at the end of an iteration that begins in a good state. Thus an iteration that begins in a good state also ensures that either all processes exhibit reset behavior or all exhibit normal snapshot behavior, never a combination of the two.

Suppose a good state has been achieved and statement (ITERATE) sets process 0's *reset* variable to true for a new iteration. Then at the end of the iteration, each process reports that it has recorded a local state of *NS* and that the queues are empty. Together the reports indicate a normal initial state of **P** so **legal** is true in the next iteration. In fact, the reports reflect the projection of the actual global state at the end of the iteration and therefore no further resets occur. To see that this is the case, observe that the iteration causes all processes to suspend **P** in state *NS* and send a marker to each neighbor. Any of **P**'s messages that remain in a channel must have been sent before the last process suspended itself and sent a marker. Since the iteration cannot end before each process has received a marker from each neighbor, any such left-over message of **P** is discarded because it is received at a process that is already suspended.

6 Complexity

The proof of Theorem 3 shows that accurate snapshots are obtained once the system is in a "good" state (as defined in the proof). So that we may relate the time needed to establish the first good state to the values present in the initial state, let V denote the set of values present in the R variable of any process or in the RND field of any message in the initial state, let r_0 denote the value of process 0's R variable in the initial state, and let d denote the difference between r_0 and $max(V)$. Then execution of StableSnap leads to a good state in no more than d iterations so, in the terminology of Gouda and Evangelist [7], the "convergence span" of StableSnap is d, where statement (ITERATE) is identified as the "critical" step. The convergence span may be reduced by creating an additional field in **report** messages and having *Report* set this field to the value of the local R variable; process 0, in executing statement (ITERATE), can change its R variable to one that exceeds the maximum value observed in this field in the last set of **report** messages. The convergence span is thus reduced to no more than the size of the subset of V containing values larger than r_0. If by chance there are no **snapshot** or **report** messages in the initial state, execution of the modified StableSnap leads to a good state at the start of the second iteration.

Because the model of computation employed in this paper is asynchronous and there is a statement (GENPROD) that is always enabled to send messages, we cannot place an upper bound on the number of messages actually sent during a snapshot. However once StableSnap has stabilized, (GENPROD) need not execute at all, although we cannot take advantage of this fact since the stabilization cannot be infallibly detected by any process. A more realistic model would use absolute time intervals and time-outs to restrict execution of (GENPROD).

It is also possible to adjust the frequency with which snapshots are taken. If snapshots were taken after a small numbers of steps of **P**, then few steps of **P** would execute

in an illegal state. However the number of steps devoted to taking snapshots would be a significant fraction of the total steps executed even when a legal state has been established. On the other hand, if snapshots were taken after a large number of steps of **P**, then only a small percentage of the total steps executed would be devoted to taking snapshots. The price paid is that many steps of **P** would execute in an illegal state before a reset occurs. Of course, this assumes that **P** does not deadlock even in illegal states.

7 Discussion

In this paper we have explored the limits of self-stabilization: we have shown the impossibility of attaining certain properties in self-stabilizing extensions and demonstrated a transformation to create a self-stabilizing extension of a program. Our aim was to demonstrate the possibility of a semi-automated transformation, not to contend that it is always desirable to use one. Several issues regarding the transformation's utility should be noted.

The advantage of our transformation is its generality: its correctness does not depend in any way on the behavior of the program that is being extended (only the correctness of the predicate that recognizes illegal states of the program). A programmer may design an algorithm as usual (i.e., ignoring the possibility of abnormal behavior) and then apply the transformation, thereby separating the concerns of achieving self-stabilization from those involving "normal" correctness. In fact, given only the predicate to recognize illegal states, an automatic preprocessor that converts programs into self-stabilizing versions can be implemented.

One may ask whether it would be better to use a transformation that takes advantage of properties specific to the program being extended in order to create an extension that is more efficient or elegant. For example, the self-stabilizing repeated snapshot algorithm that we presented was derived by imposing a self-stabilizing "control" algorithm (to regulate iteration) on top of the Chandy-Lamport single-snapshot algorithm. Because an invariant of the single snapshot algorithm is that at most one "marker" may be received on each channel, the "control" algorithm insulates the single-snapshot algorithm from any message that would violate this invariant. Perhaps this technique of imposing an algorithm that acts as a filter can be extended to a wider class of programs.

A possible disadvantage of our approach is due to the fact that the self-stabilizing program that we derive is an extension of the original program. Consider a program that produces two types of execution sequences, one establishing property "A" and the other establishing property "B". A self-stabilizing program that only establishes property "A" may be satisfactory in a particular situation even though it is not an extension of the original. Therefore, extending an existing program may not produce the simplest solution in all situations.

Another possible drawback may be the difficulty in formulating a predicate that precisely recognizes illegal states of the program being extended. One could weaken this requirement so that the predicate may falsely recognize a small number of legal states or fail to recognize some improbable illegal states. The penalty for doing so would be that unnecessary resets could occur or that self-stabilization would be achieved only with high probability.

There is also the question of efficiency. Our choice of flooding as the method of routing **report** messages to process 0 appears wasteful. A dynamically created spanning tree would be more desirable but construction of this tree must be self-stabilizing as well. Alternatively, we could route messages along a constant fixed path but the resulting algorithm could not accommodate dynamic changes in the system configuration such as the addition of a new process.

It is also undesirable that the value of the R variable and RND field of a message grows without bound. By assuming an upper bound on the number of messages present in the channels of initial states (but not any subsequent states) we have obtained a version of the algorithm that bounds these values [11].

References

1. Bouge L, Francez N: A compositional approach to superimposition. Principles of Programming Languages. ACM 1988
2. Brown GM, Gouda MG, Wu CL: Token systems that self-stabilize. IEEE Trans Comput 38: 845–852 (1989)
3. Burns J, Pachl J: Uniform self-stabilizing rings. ACM Trans Programm Lang Syst 11: 330–344 (1989)
4. Chandy KM, Lamport L: Distributed snapshots: determining global states of distributed systems. ACM Trans Comput Syst 3 (1): 63–75 (1985)
5. Dershowitz N, Manna Z: Proving termination with multiset orderings. Commun ACM 22 (8): 465–476 (1979)
6. Dijkstra EW: Self-stabilizing systems in spite of distributed control. Commun ACM 17 (11): 643–644 (1974)
7. Gouda M, Evangelist M: Convergence/response tradeoffs in concurrent systems. Tech Rep TR 88-39, University of Texas at Austin, 1988
8. Halpern J, Fagin R: Modelling knowledge and action in distributed systems. Distrib Comput 3: 159–177 (1989)
9. Halpern J, Moses Y: Knowledge and common knowledge in a distributed environment. 3rd ACM Symposium on Principles of Distributed Computing, pp 50–61, 1984
10. Katz S: A superimposition control construct for distributed systems. ACM Trans Program Lang Syst 15 (2): 337–356 (1993)
11. Katz S, Perry KJ: Self-stabilizing extensions for message-passing systems. 9th ACM Symposium on Principles of Distributed Computing, pp 91–101, 1990
12. Lamport L: What good is temporal logic. 9th World Congress, IFIP 1983, pp 657–668
13. Lamport L: The mutual exclusion problem: Part II – statement and solutions. J ACM 33 (2): 327–348 (1984)
14. Lichtenstein O, Pnueli A, Zuck L: The glory of the past. Conference on Logics of Programs. Lect Notes Comput Sci, vol 193. Springer, Berlin Heidelberg New York 1985, pp 196–218
15. Pnueli A: The temporal logic of programs. 18th Annual Symposium on Foundations of Computer Science, IEEE 1977, pp 46–57