Self-Stabilization

MARCO SCHNEIDER

Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712-1188

In 1973 Dijkstra introduced to computer science the notion of self-stabilization in the context of distributed systems. He defined a system as *self-stabilizing* when "regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps." A system which is not self-stabilizing may stay in an illegitimate state forever. Dijkstra's notion of self-stabilization, which originally had a very narrow scope of application, is proving to encompass a formal and unified approach to fault tolerance under a model of transient failures for distributed systems. In this paper we define self-stabilization, examine its significance in the context of fault tolerance, define the important research themes that have arisen from it, and discuss the relevant results. In addition to the issues arising from Dijkstra's original presentation as well as several related issues, we discuss methodologies for designing self-stabilizing systems, the role of compilers with respect to self-stabilization, and some of the factors that prevent self-stabilization.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols—*protocol verification*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; *network operating systems*; D.1.1 [**Programming Techniques**]: General; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.4 [**Software Engineering**]: Program Verification—*correctness proofs*; *reliability*; D.2.5 [**Software Engineering**]: Testing and Debugging—*error handling and recovery*; D.2.10 [**Software Engineering**]: Design—*methodologies*; D.4.1 [**Operating Systems**]: Process Management—*concurrency*; *deadlocks*; *mutual exclusion*; *synchronization*; D.4.5 [**Operating Systems**]: Reliability—*fault-tolerance*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Algorithms, Design, Reliability, Verification

Additional Key Words and Phrases: Convergence, fault tolerance, stabilization, self-stabilizing systems, self-stabilization, transient errors, transient failures

INTRODUCTION

The notion of self-stabilization was introduced to computer science by Dijkstra [1973, 1974]. Uncertain as to whether a nontrivial self-stabilizing system under distributed control could exist at all, Dijkstra limited his attention to a ring of finite-state machines. He defined a system as self-stabilizing when "regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps." A system which is not self-stabilizing may stay in an illegitimate state forever. Dijkstra observed that "The complication is that a node's behaviour can only be influenced by the part of the total system state description that is available in that node: local actions taken on account of local informa-

CONTENTS

tion must accomplish a global objective." While much of the difficulty of achieving self-stabilization, as well as its benefits, arise out of concurrency, as we shall discuss, it has applications in sequential systems too.

Dijkstra did not address the significance of the property of self-stabilization. Appreciation of the results were left to the reader. In the years following his introduction, very few papers were published in this area, as one can see by glancing at our bibliography. This fact was belabored by Lamport, who said the following at his invited address in 1983 to the 3rd ACM Symposium on Principles of Distributed Computing [Lamport 1984]:

> I regard this as Dijkstra's most brilliant work—at least, his most brilliant published paper. It's almost completely unknown. I regard it to be a milestone in work on fault tolerance.

More recently, the investigation and use of self-stabilization as an approach to fault-tolerant behavior has been undergoing a renaissance. Within the past few years there has been a flurry of papers, as well as a workshop devoted entirely to this area at the Microelectronics and Computer Technology Corporation in August, 1989. Dijkstra's notion of self-stabilization, which originally had a very narrow scope of application, is proving to encompass a formal and unified approach to fault tolerance under a model of transient failures for distributed systems. In this paper we survey the emerging field of self-stabilization.

Our presentation is organized as follows. First, we define self-stabilization. Then, we discuss its relationship to fault tolerance and argue that it represents a departure from previous approaches to this area. Next, we discuss Dijkstra's seminal work and use it to motivate our survey of the field. We discuss the issues arising from Dijkstra's original presentation as well as several related issues. Finally, we discuss three important themes that have emerged in this area of research. In particular, we discuss the methods that have been used to design more complex self-stabilizing systems; we discuss the relationship between compilers and self-stabilization; and we enumerate some of the factors that have been found to interfere with self-stabilization. We conclude with some suggestions for future research.

1. DEFINITION OF SELF-STABILIZATION

Before defining self-stabilization, some preliminary definitions are in order. A program specifies a system through the combination of its statements and an implicit model of computation or type of architecture (e.g., asynchronous message passing, CSP, Turing Machine, etc.) for which the statements are written. In systems that consist of more than one machine, we refer to each such machine as a process. A system is composed of two types of *components*: processes and interconnections between processes (such as shared memory or message channels). The *topology* of a system is the directed graph formed from its components by denoting processes as nodes and their interconnections (via shared memory or message channels) as directed edges. Each component of a system has a *local state*. We define the *global state* of a system (likewise a program) as the union of the local states of its components. The *behavior* of a system consists of a set of states, a transition relation between them, and a set of fairness criteria on the transition relation.

We define *self-stabilization* for a system S with respect to a predicate P, over its set of global states, where P is intended to identify its correct execution. S is *self-stabilizing* with respect to predicate P if it satisfies the following two properties:

(1) **Closure**—P is closed under the execution of S. That is, once P is established in S, it cannot be falsified.

(2) **Convergence**—Starting from an arbitrary global state, S is guaranteed to reach a global state satisfying P within a finite number of state transitions.

States satisfying (not satisfying) P are called *legitimate* (*illegitimate*) states respectively. We use the terms *safe* and *unsafe* interchangeably with *legitimate* and *illegitimate*, respectively. A program may be defined to be self-stabilizing in a corresponding manner. Thus, a self-stabilizing program specifies a self-stabilizing system.

We introduce a generalization of self-stabilization based on Arora and Gouda [1992] and Arora [1992] which will prove to be quite useful. We define *stabilization* for a system S with respect to predicates P and Q, over its set of global states. S satisfies $Q \rightsquigarrow P$ (read as Q stabilizes to P) if it satisfies the following two properties.

(1) **Closure**—P is closed under the execution of S. That is, once P is established in S, it cannot be falsified.

(2) **Convergence**—Starting from any global state satisfying Q, S is guaranteed to reach a global state satisfying P within a finite number of state transitions.

Note that if S is self-stabilizing with respect to P then this may be restated as $TRUE \rightsquigarrow P$ in S.

It is often the case that in writing a program, the author does not have a particular definition of safe and unsafe states in mind but has designed the program to function from a particular set of start states. Under these circumstances, it is reasonable to define as safe those states that are reachable under normal program execution from the set of legitimate start states (hereafter referred to as the *reachable set*). By default, when we say that a program is self-stabilizing, without mentioning a predicate, we mean with respect to the reachable set. By definition, the reachable set is closed under program execution, and it corresponds to a predicate over the set of states. Such a definition seems all the more natural when we are dealing with algorithms whose purpose is to compute a function as opposed to ensuring some form of coordination and control, in which case the start state incorporates the input to the function.

We introduce the model of failure that we will use in our discussion on fault tolerance. A *transient failure* is an event that may change the state of a system, but not its behavior. We assume that the state of a system is violable, whereas its behavior is not. Transient failures may change the global state in a system by corrupting the local state of a process as represented by memory or program counter (how states are realized is not part of the model) or by corrupting message channels or shared memory. The property of self-stabilization models the ability of a system to recover from transient failures under the assumption that they do not continue to occur.

In the design of systems with dynamic topologies, the safety of a state may also depend on the current topology. The property of self-stabilization may be used to model the ability of a dynamic system to tolerate changes to its topology brought about by failures or repairs of its components. This is done by defining the global state of a dynamic system to include its current topology. That is, "up" or "down" with respect to each component may be defined as part of the system state. More generally, a program for dynamic systems might require a specific type of topology to work correctly. For instance a distributed message-passing program might require a strongly connected graph. In such a case, we could define the set of "safe" topologies by a separate predicate Q. The property $Q \rightsquigarrow P$ in S would denote the property that for any topology satisfying Q, S is self-stabilizing with respect to P.

A useful device for understanding the difficulty of achieving self-stabilization in a dynamic system is an adversarial argument which we will abstract in the form of a *malicious adversary* or *evil demon* engaged in a suicide mission. The goal of the malicious adversary is to sabotage a system in any way possible. Such an attack can arbitrarily and even maliciously alter the system state. In addition it is possible that the adversary may wipe out some of the system's components (which may be modeled as part of its state also). What makes such an attack all the more malicious is that it may not be possible for a system to "know" it has been attacked. To be self-stabilizing, a system must have the ability to recover from such malicious attacks assuming that they cannot repeatedly occur and that a functional subset of the systems components are left intact. This last part is crucial since a real attack could destroy a system by obliterating some or all of its components and hence altering its topology in addition to its state. Should the topology be altered, recovery cannot always be guaranteed by any system. In particular, two possibilities arise.

(1) The system is destroyed so that it can no longer meet the functional requirements of its mission. Processes and/or interconnections have been destroyed. No form of fault tolerance can overcome this situation once it has occurred. The malicious adversary has won.
(2) While some components of the system may be destroyed, enough are left intact so that it can complete its mission. It is up to the designer of a system to decide under what set of topologies (conditions) the system should be able to complete its task. The system must be able to stabilize under such conditions.

We now further elaborate upon the relationship between self-stabilization and fault tolerance.

2. SELF-STABILIZATION AS AN APPROACH TO FAULT TOLERANCE

A self-stabilizing system has two useful properties which can informally be seen as restatements of one another:

(1) It need not be initialized.
(2) It can recover from transient failures.

If we are willing to tolerate the temporary violation of a system specification, then we need not specify an initial state for a self-stabilizing system: any state will do. The requirement that execution commence from an initial state is problematic at best for multiprocess systems. Consider, for example, computer networks. It would be unreasonable to stop a network and reinitialize it when adding or removing nodes. In this section we focus on the second property above, that of fault tolerance.

We first argue that self-stabilization represents a departure from previous approaches to fault tolerance. Historically, researchers have tended to address the wide variety of phenomena within fault tolerance by countering the effects of their individual causes. In doing so they have neglected their commonality and ended up with a piecemeal approach. Self-stabilization provides a unified approach to transient failures by formally incorporating them into the design model. Again we quote Lamport on Dijkstra's seminal work [Lamport 1984]:

> I regard it to be a milestone in work on fault tolerance. The terms "fault tolerance" and "reliability" never appear in this paper.

"Fault tolerance" and "reliability" are an integral but implicit part of the design and not afterthoughts. Given that we can never eliminate transient failures, a self-stabilizing system meets a stronger, more satisfying notion of correctness. That is, should a transient failure occur, resulting in an inconsistent system state, then regardless of the failure's origin, the system will eventually correct itself without any form of outside intervention.

By way of example, we examine coordination loss within distributed systems. Coordination loss may be viewed as a transient failure. This example is adapted from Gouda and Multari [1991]. "Informally, coordination is said to be lost at a given global state of a distributed program if and only if the local states of the different processes in the program, though each of them may be correct in its own right, are inconsistent with one another in the given global state." Any program that is self-stabilizing can recover from loss of coordination. This phenomenon has numerous causes, many of which are indistinguishable once such an event has occurred. These include:

(1) *Inconsistent initialization*: The different processes in the program may be initialized to local states that are inconsistent with one another.
(2) *Mode change*: A system may be designed to execute in different modes. In changing the mode of operation it is impossible for all of the processes to effect the change at the same time. The program is bound to reach a global state in which some processes have changed while others have not.
(3) *Transmission errors*: The loss, corruption, or reordering of messages may result in an inconsistency between the states of sender and receiver.
(4) *Process failure and recovery*: If a process returns to service after "going down," its local state may be inconsistent with the rest of the program.
(5) *Memory crash*: The local memory of a process may crash, causing its local state to be inconsistent with the rest of the program.

Traditionally, each of these issues has been handled separately, one at a time, and yet these seemingly disparate failure phenomena all have a common antidote, that of the self-stabilizing system. The traditional incremental and ad hoc approach is analogous to the use of exception handlers for the purpose of fault-tolerant software. Each addition of an exception condition may indeed reduce the possibility of a failure, but without a formal basis its elimination can never be guaranteed.

We have argued that self-stabilization provides a formal and unified approach to fault tolerance with respect to a model of transient failures and that this represents a departure from previous approaches to fault tolerance. We now briefly address how one might apply our transient failure model in a useful manner. Implicit in the notion of a self-stabilizing system, and hence our transient failure model, is the assumption that while the abstract state of a program or system may be corrupted, the program or system itself is inviolable (its behavior remains intact). The appropriateness of such an assumption depends on the fault or source of failure we wish to address, as well as the construction of the system in question.

Consider an actual computer system upon which we wish to implement a self-stabilizing program. If program and state are both realized in memory, then clearly the property of self-stabilization does not address the issue of memory corruption due to outside environmental events. However, if the program is placed in read-only memory, then the property of self-stabilization more properly addresses such faults. Memory corruption will only affect the abstract state of the program. In addition to memory corruption, we listed four other causes for loss of synchronization in distributed systems. Events such as transmission errors, process failures, mode change, and inconsistent initialization can alter the abstract state of a system without otherwise affecting its proper behavior.

We note an important distinction between the use of self-stabilization and traditional methods such as replication or error correction: whereas the latter methods attempt to mask the occurrence of errors and thus prevent failure, self-stabilization guarantees recovery should a transient failure occur. In this way, self-stabilization provides a complementary approach to other methods of fault tolerance.

One criticism of self-stabilization as a design goal is that it is too strong a property and thus either too difficult to achieve or achieved at the expense of other goals. As transient failures are not always arbitrary, it is useful to consider recovery from restricted transient failures. Such a notion is modeled by the property of *stabilization*. Instead of considering recovery from an arbitrary state, we can consider recovery from a restricted set of states as specified by a predicate Q (corresponding to the set of states that may result from a particular set of failures). The property of $Q \leadsto P$ may be used in place of self-stabilization with respect to P under the assumption that a transient failure will result in a state satisfying Q.

We conclude with some remarks on how the formal property of self-stabilization may be used to specify additional fault-tolerant behavior. Self-stabilization has historically been viewed in terms of the ability of a system to recover from transient failures under the assumption that they do not continue to occur. However, the two formal properties of self-stabilization and stabilization may be used to specify the ability of a system to tolerate all types of faults. This is achieved by considering faults to be transitions on an augmented state space and transition relation. The original transitions of the system cannot alter the augmented state but may depend on it. The properties of self-stabilization and stabilization as applied to the augmented system can be used to specify fault tolerance within the original system. An important application of this technique in the literature on self-stabilization is the case of dynamic topologies. Self-stabilization may be used to specify a system's ability to tolerate permanent as well as temporary changes to its topology. Further discussion is beyond the scope of this survey. For a formal approach to fault tolerance and an elaboration on these ideas, we direct the reader to Arora and Gouda [1992] and Arora [1992] as well as Cristian [1985].

3. AN OVERVIEW OF SELF-STABILIZATION

3.1 The Original Introduction

Dijkstra's original introduction of self-stabilization to computer science [Dijkstra 1973, 1974] continues to be important both because of the historical perspective it provides and because, as we shall see, many of the issues it raised continue to be the focus of papers on the subject today. We follow the presentation given in Dijkstra [1974].

The context of his presentation was a connected graph of finite-state processes, where processes placed in directly connected nodes were called neighbors. He defined a *privilege* as a boolean function of the state of a process and its neighbors indicating whether a particular transition was enabled for the process. Self-stabilization was defined in two phases. He first defined as *legitimate* those states meeting a global-correctness criterion with the following four additional constraints:

(1) In each legitimate state one or more privileges will be present.
(2) In each legitimate state each possible move will bring the system again in a legitimate state.
(3) Each privilege must be present in at least one legitimate state.
(4) For any pair of legitimate states there exists a sequence of moves transferring the system from the one into the other.

He then used the following definition for self-stabilization:

> We call the system "self-stabilizing" if and only if, regardless of the initial state and regardless of the privilege selected each time for the next move, at least one privilege will always be present and the system is guaranteed to find itself in a legitimate state after a finite number of moves.

The definition above was provided in the context of mutual exclusion, and thus it is restrictive in comparison to our own. The four constraints on legitimate states combine the requirement of closure (number two) with several other unnecessary properties. Constraint one is a progress requirement that is in principle unnecessary. It is equivalent to requiring that all legitimate computations be infinite, or in other words, the system should not terminate. This is a reasonable constraint in the context of the original presentation. As we shall see, termination is one of the factors that can prevent self-stabilization. Constraint two is a restatement of our closure definition. Constraint three does not appear to be relevant to the notion of legitimate and is a restriction on the program. Constraint four produces a restrictive definition in comparison to ours since it precludes the possibility that the legitimate states are divided into two or more disjoint sets each of which is closed under the transition relation. Finally, the explicit requirement that one privilege always be present is unnecessary.

Uncertain as to whether a nontrivial (all states legitimate is considered trivial) self-stabilizing system under distributed control could exist at all, Dijkstra narrowed his attention to a ring of processes under the control of a central demon which selected in "fair random order" one privileged process at a time allowing it to make a transition that was a function of its old state and that of its neighbors. He defined as legitimate those states in which exactly one privilege was present. In Dijkstra [1973], he additionally required that the privilege rotate around the ring. Thus, he was attempting to produce a self-stabilizing algorithm for mutual exclusion via a token ring with one token.

Dijkstra [1973] observed that if the processes are required to be identical, then in general the problem could not be solved. More specifically, in the case of a nonprime number of processes N, should the system start in a state with a cyclic symmetry of degree n, where $(2 \leq n \leq N/2)$, then the demon is free to give its first n commands equally spaced around the ring, thus preserving a state of cyclic symmetry. This behavior may be continued such that the system will never converge to a legitimate state.

Two extremes arise in the solution to this problem. One may choose to make all of the processes different or to make one "distinguished" and the others identical. Dijkstra chose the latter approach. Dijkstra [1974] presents two additional solutions for which privilege is not required to rotate around the ring (instead it travels back and forth). These solutions use four and three states respectively to achieve the desired result. Both of these solutions are *bidirectional.* We call a ring *unidirectional* if the transition relation for each machine only depends on one neighbor; otherwise it is *bidirectional.* We present his first solution based on the presentation in Dijkstra [1973]. This solution is *unidirectional.*

We are given N K-state processes $(K > N)$ which are labeled M_0 through M_{N-1}, where M_0 is distinguished, and for $1 \leq i < N$, M_i are identical.

If the state of M_i, $1 \leq i < N$, is different from that of its left-hand neighbor, a transition is enabled which will result in M_i taking on the state of its left-hand neighbor:

$$\langle \forall i | 1 \leq i < N: \quad M_i := M_{i-1} \ \mathit{if}\, M_i \neq M_{i-1} \rangle$$

If the state of M_0 is equal to that of its left-hand neighbor M_{N-1} (call it state S), a transition is enabled that will bring it into state $(S + 1) \bmod K$:

$$M_0 := (M_0 + 1) \ \mathit{mod}\, K \quad \mathit{if} \quad M_0 = M_{N-1}$$

Since these are the only possible transitions, after a process makes a transition, it is no longer privileged. It is easy to see that if the system starts with all processes in the same state, then the privilege will rotate around the ring. Furthermore it can be shown quite easily that if started in an arbitrary state, the system will stabilize.

A number of important issues are raised by Dijkstra's search for an example of a self-stabilizing system, both in the choices by which he restricted that search and in his final solution. These issues include the necessity of a central demon, the number of states in a solution, the importance of asymmetry, and the use of criteria for legitimate states other than mutual exclusion. Subsequent papers have introduced a number of new issues including the costs of self-stabilization, probabilistic self-stabilization, and self-stabilizing communication protocols (programs). In the remainder of this section we give an overview of subsequent work in this area organized according to these issues.

3.2 The Role of the Central Demon

The requirement of a central demon is an unreasonable constraint for a truly distributed system. In particular, its implementation requires some form of centralized control. Dijkstra used such a mechanism because the transitions of neighboring processes were interfering. That is, by firing its enabled transition, one process could disable an enabled transition in another process. In general, a system in which transitions are executed atomically and only sequentially (as provided by a central demon) will not behave the same as one in which transitions are fired in parallel, or one in which reads and writes are interleaved.

In response to the restrictions of the central demon, Burns [1987] and Brown et al. [1989] both introduce classes of self-stabilizing token systems whose state transitions are noninterfering. Burns considers a ring of finite-state processes, and Brown et al. consider an array of finite-state processes with synchronization primitives between neighbors. By virtue of noninterference, if a state transition is enabled at any instant, then it will continue to be enabled until it is executed. Once a transition is enabled in a process, it continues to be enabled regardless of what other enabled transitions are fired, including ones by that process. The noninterference property of these systems makes their implementation in hardware easier and allows them to be realized with a read/write level of atomicity. These systems are self-stabilizing in the sense that regardless of the number of tokens that exist initially, the system is guaranteed to reach a state wherein only one token exists.

Thus a central demon (or centralized control of any form) is not necessary for a self-stabilizing single-token system.

It is interesting to note that Dijkstra's three-state and four-state solutions are still correct if transitions are allowed to fire synchronously [Burns et al. 1989].

3.3 The Role of Asymmetry and Probabilistic Self-Stabilization

While Dijkstra showed that a self-stabilizing token ring with a nonprime number of processes must have an asymmetric design, Burns and Pachl [1989] demonstrate that there is a self-stabilizing ring with no distinguished process (a symmetric solution), if the size of the ring is prime. It should be noted, however, that their solution requires the use of a demon to prevent the possibility of a livelock in which all of the processes move in lockstep forever. In other words, the ring cannot be synchronous.

Asymmetry has to be maintained in systems where processes may synchronize with one another such as mutual exclusion, dining philosophers, drinking philosophers, and resource allocation systems under deterministic rules. This point was brought to prominence by Lehman and Rabin [1981] who show that there is no deterministic symmetric solution to the dining philosophers problem. Gouda [1987] demonstrates a self-stabilizing asymmetric solution to this problem.

Because such systems are important targets for self-stabilizing implementations, an understanding of the relationship between self-stabilization and symmetry (or asymmetry) is important. There are two methods by which asymmetry may be maintained in systems of synchronizing processes for which we introduce the terms *state* and *identity*. (Gouda [1987] introduces the terms **memory** and **identity** respectively to denote these two types of asymmetry.) A system is *asymmetric by state* when all of its machines are identical, but they have different initial local states. This may be realized by identical programs. A system is *asymmetric by identity* when not all of its machines are identical. This may be realized by identical programs that are parameterized by a local id. These two methods may be rephrased as a distinction between the use of a homogeneous set of processes with different start states and the use of a heterogeneous set of processes.

In general, systems asymmetric by state (only) cannot be self-stabilizing, while those that are asymmetric by identity can be. This follows from the fact that if a system that is asymmetric by state is put into a symmetric global state, it may not converge to an asymmetric global state. Thus Gouda's self-stabilizing solution to the dining philosophers problem is by necessity asymmetric by identity.

As pointed out, these results hold only for deterministic solutions. In their paper, Lehman and Rabin [1981] demonstrate a probabilistic symmetric solution to the dining philsophers problem. By incorporating randomized actions, they break symmetry. We define *probabilistic self-stabilization* for a system S with respect to a predicate P, over its set of global states. S is *probabilistically self-stabilizing* with respect to predicate P if it satisfies the following two properties:

(1) **Closure**—P is closed under the execution of S. That is, once P is established in S, it cannot be falsified.

(2) **Convergence**—There is a function f from the natural numbers to the interval $[0, 1]$ satisfying $\mathrm{Lim}_{k \to \infty} f(k) = 0$ such that the probability that starting from an arbitrary global state, S will reach a global state satisfying P within k transitions is at least $1 - f(k)$.

Israeli and Jalfon [1989] present a probabilistic self-stabilizing algorithm for orientation of an asynchronous bidirectional ring of processes, and Herman [1990] presents a probabilistic self-stabilizing algorithm for a synchronous ring with an odd number of identical processes and one token.

3.4 Reducing States in a Token Ring

In this section we discuss results concerning the number of states per machine required to achieve self-stabilization in token rings. The goal of producing systems with a finite number of states per machine is of particular interest because such machines may have direct implementations in hardware.

We first discuss the quest for self-stabilizing token rings with two states per machine. If we do not require the property of self-stabilization, then there exists an asymmetric token ring with two states per machine. Consider the following solution where initially all machines have the same value:

$$\begin{aligned} &\langle \forall i | 1 \le i < N: \\ &\quad M_i := M_{(i-1)} \quad \mathit{if}\, M_i \ne M_{(i-1)} \rangle \\ &\quad M_0 := \neg M_0 \quad \mathit{if} \quad M_0 = M_{n-1} \end{aligned}$$

Ghosh [1990b] shows that the minimum number of states per machine possible in a self-stabilizing token ring, assuming a central demon and deterministic execution, is three. However, Ghosh [1990a] demonstrates the existence of a nontrivial self-stabilizing system which requires two states. In order to achieve this solution a high atomicity is used in each action. In particular, each of the nonexceptional machines must read from three of its neighbors. A nonring topology and a higher atomicity is traded for a reduced number of states.

In Herman [1990], a solution requiring only two states is presented for a probabilistically self-stabilizing synchronous token ring with randomized actions. This solution is unidirectional and symmetric. Flatebo and Datta [1992] provide a two-state probabilistically self-stabilizing token ring with randomized actions under the assumption of a "randomized central demon" (the choice among privileged machines is made randomly). The assumption of randomness allows them to ignore malicious scheduling on the part of the demon. Their solution is unidirectional and asymmetric. We now present their solution requiring two exceptional machines.

```
{Exceptional Machine 0}
$M_0 := \neg(M_0) \quad if\, M_0 = M_{N-1}$
{Exceptional Machine $N - 1$}
$M_{N-1} := \neg(M_{N-1})$ with some probability
    $if\, M_{N-1} = \neg M_{N-2}$
{All Other Machines}
$\langle \forall i | 1 \le i < N - 1:$
    $M_i := \neg M_i \quad if\, M_i = \neg M_{i-1} \rangle$
```

In order for self-stabilizing systems with two states to be obtained, it is necessary to either relax the requirement of self-stabilization to probabilistic self-stabilization and use randomized actions, or use a nonring topology with higher atomicity. The exact relationships between these properties has yet to be determined.

We conclude with some additional work concerning the number of states required per machine in self-stabilizing token rings. While Dijkstra's bidirectional solutions required four and three states respectively, his unidirectional solution required $N + 1$ states per machine. Haddix [1991] demonstrates two unidirectional token rings that only require a constant number of states. His solutions work under a distributed demon (any number of machines may be selected at each step). In Israeli and Jalfon [1990] dynamic token rings are investigated. It is shown that the number of states per machine in a dynamic token ring whose size belongs to the range $[2 \cdots N]$ is $\Omega(N - 1)$. As a direct consequence, any solution for dynamic token rings of arbitrary size must have an infinite number of states per machine.

3.5 Costs of Self-Stabilization

While it is agreed that self-stabilization is a desirable property, its definition does not preclude the possibility of unbounded recovery in a system. The definition does not include a bound on the number of steps a system takes to converge to a safe state. Furthermore, one would expect there to be tradeoffs between how fast a system stabilizes and how fast it executes. A study of these costs is very important if this model is to have applications in real-time systems, that is, systems where goals must be accomplished within a specific amount of time, or by a certain time.

Following the terminology of Gouda and Evangelist [1990], we introduce two important concepts relating to the costs of self-stabilization: *Convergence Span* and *Response Span*. We define the *Convergence Span* as the maximum number of transitions that can be executed in a system, starting from an arbitrary state, before it reaches a safe state. We define the *Response Span* as the maximum number of transitions that can be executed by a system starting from some initial state and ending in some target state. The choice of initial and target state sets depends on the intended functionality of the system.

Gouda and Evangelist [1990] present a class of self-stabilizing systems for detecting termination on a unidirectional ring, where, for a fixed number of processes n, (1) the convergence spans of n/k are inversely proportional to the response spans of nk and (2) k is a parameter whose value can be chosen arbitrarily from the domain 1 through n.

They define *Convergence Span* as the maximum number of *critical* transitions that can be executed in a system, starting from an arbitrary state, before it reaches a safe state. Our definition assumes implicitly that all state transitions are *critical*. The motivation behind *critical* transitions is that only transitions that may lead to a violation of safety requirements are important. Their weaker definition of self-stabilization allows a system to remain in an unsafe state indefinitely if none of its critical transitions are executed infinitely often.

Whitbty-Strevens [1979] investigates the three algorithms proposed by Dijkstra in terms of their Convergence Span and their cost in network communication. He uses the term "*Pseudo-Legitimate States*" for states which are not legitimate, but will converge to a legitimate state and have already met the aim of mutual exclusion. Under our definition of self-stabilization, we can identify all such states as legitimate, since the property of mutual exclusion is closed in Dijkstra's algorithms. Thus, the distinction between *legitimate* and *pseudo-legitimate* is unnecessary when applied to our definitions.

Chang et al. [1987] evaluate a restricted case of Dijkstra's K-state solution and find that the expected number of moves which processes make to reach a legitimate state is $O(n^{1.5})$, and the expected number of messages passed is $O(n^2)$. Tchuente [1981] is also concerned with convergence span. He provides an algorithm that converges twice as fast as Dijkstra's.

3.6 Self-Stabilizing Communication Protocols

Gouda and Multari [1991] and Multari [1989] define a communication protocol as existing over a distributed system with FIFO message channels and consisting of a set of actions (each corresponding to a particular process), where the variables of each process (excluding the message channel) are local to that process. They prove that the following three properties are necessary for a communication protocol to be self-stabilizing:

(1) Nontermination.
(2) An infinite number of safe states (in the form of unbounded local variables) in a nonempty subset of its processes.
(3) Timeout actions in a nonempty subset of its processes.

They show that the standard formulations of the sliding-window protocol and the two-way handshake are not self-stabilizing and present self-stabilizing versions. Because it is finite state, the standard formulation of the alternating-bit protocol is not self-stabilizing as a direct consequence of the second requirement that a self-stabilizing communication protocol must have an infinite number of safe states.

In response to this last result, Afek and Brown [1989] present a probabilistic version of the alternating-bit protocol that is self-stabilizing. Finally, Burns et al. [1990] introduce a notion weaker than self-stabilization, that of *pseudo-stabilization*. They are able to show that the alternating-bit protocol does pseudo-stabilize to its intended specification. Informally, system S *pseudo-stabilizes* to predicate P, if and only if under all computations, starting from an arbitrary state, eventually, P holds forever. In other words, P may hold and not hold a finite unbounded number of times, but eventually it will continue to hold. By contrast, our definition of self-stabilization requires that once predicate P becomes true it continues to hold forever. The stronger requirement of self-stabilization is advantageous over pseudo-stabilization in finite-state systems, since the former property implies a bounded convergence span while the latter does not. This follows as a consequence of the pigeonhole principle (the same unsafe state cannot occur more than once). Formally, pseudo-stabilization is defined in terms of sets of computations rather than state predicates. It is expressed in temporal logic as eventually always P. Further discussion is beyond the scope of this survey.

4. METHODOLOGIES FOR DESIGNING SELF-STABILIZING SYSTEMS

Although Dijkstra's original work on self-stabilization was presented in the context of mutual exclusion on a ring of finite-state machines, the issues of algorithm and architecture are orthogonal to the property of self-stabilization. Kruijer [1979] is the first work to separate these issues from the property of self-stabilization. He presents a self-stabilizing program for a tree-structured system of finite-state machines in which only one process along any path from the root to a leaf node may be privileged at a time. Thus, more than one process may be privileged at a time. Essentially, the way the algorithm works is that the root is moved one state ahead resulting in a wave that is propagated through the branches of the system to its leaves and then reflected back to the root. This corresponds to a diffusing computation. The state transitions of Kruijer's system are interfering.

If we wish to design more complex self-stabilizing systems for arbitrary topologies, then we ought to apply traditional methodologies such as layering and modularization. We would like to divide a program into distinct components, make each component self-stabilizing independently and then compose them. Recent research on self-stabilization has been addressing these issues. As it turns out, self-stabilization is amenable to the technique of layering. Consider the property $Q \leadsto P$ that we introduced in Section 1 as a generalization of self-stabilization. We noted that if S is self-stabilizing with respect to P, then $TRUE \leadsto P$ in S. Furthermore, the relation $\leadsto$ is transitive. If $Q \leadsto P$ and $P \leadsto R$, then $Q \leadsto R$. Informally we can see how transitivity corresponds to the technique of layering. Given S_1 satisfying $Q \leadsto P$ and S_2 satisfying $P \leadsto R$, we combine S_1 and S_2 such that S_2 reads from the variables of S_1 to produce a new program satisfying $Q \leadsto R$. For a formal treatment of these techniques as well as related ones see Arora [1992], Arora and Gouda [1992], as well as Gouda and Herman [1991].

A logical first step in addressing the design of self-stabilizing systems through program layering is to develop primitives that provide structure on which other algorithms may be built. Two basic structuring mechanisms for a concurrent system are a common clock and the topology by which its processes are interconnected. We discuss clocks first.

Maintenance of time through the use of local clocks for shared-memory systems is called *unison* by Gouda and Herman [1990] and Couvreur et al. [1992]. Unison is achieved when the clocks of a system are in agreement and remain so under incrementation. Unison may be specified as a safety property and a progress property. More specifically for a synchronous shared-memory system we have:

- Safety: All clocks have the same value.
- Progress: All clocks are incremented at each step.

For an asynchronous shared-memory system we have:

- Safety: For any two neighboring nodes, the values of their clocks differ by at most one.
- Progress: A clock is incremented to $i + 1$, when the clocks in all neighboring nodes have the value i or $i + 1$.

A self-stabilizing algorithm for synchronous unison with bounded clocks is provided in Arora et al. [1991]. Gouda and Herman [1990] provide a self-stabilizing algorithm for synchronous unison with unbounded counters. Finally, Couvreur et al. [1992] provide self-stabilizing algorithms for both bounded and unbounded asynchronous unison in a shared-memory model. An important application of asynchronous unison is to simulate a synchronous system.

Perhaps the most basic primitive with respect to an arbitrary dynamic topology is leader election. An algorithm for self-stabilizing leader election may be found in Afek et al. [1990]. Given the presence of a leader, another basic primitive that can be provided is a spanning tree. Dolev et al. [1990] and Arora and Gouda [1990] layer self-stabilizing mutual exclusion and reset algorithms respectively on top of self-stabilizing spanning-tree algorithms for arbitrary connected graphs. An implicit part of the spanning-tree algorithm of Arora and Gouda [1990] is a leader election. We now discuss these two examples of self-stabilizing programs produced by layering techniques.

Dolev et al. [1990] separate the property of self-stabilization from the task of achieving mutual exclusion on an asynchronous shared-memory system by utilizing a compositional approach in their design wherein one self-stabilizing layer is built on another. The algorithm they present is dynamic, allowing the topology to change, with the exception of a distinguished process. They use a weaker model of shared memory than in previous works, wherein read and write actions are integrated into one state transition. In their model, reads and writes are independent steps. This allows more general schedules such as P1 reads, P2 reads, P2 writes, P1 writes. Transition systems that are noninterfering can also afford such schedules since once a transition is enabled, it continues to be enabled.

The first layer of their algorithm is a self-stabilizing spanning-tree protocol for an arbitrary topology which may change dynamically with the exception of the distinguished process. Their protocol is based on a breadth-first search of the graph rooted at the distinguished process. The distinguished process is needed in order to break symmetry, and must always be present. The programs of all other processes are identical, but parameterized by their local topology.

The second layer of their algorithm is a self-stabilizing protocol to achieve mutual exclusion on a dynamic tree-structured system. It works as follows. When a process becomes privileged it executes its critical section. Once the privileged process completes execution of its critical section, it passes the privilege to its children in left-to-right order. This results in a depth-first tour of the spanning tree.

Finally, they combine the two protocols by superposing their respective programs on each process in order to obtain a single self-stabilizing dynamic protocol for mutual exclusion on arbitrary connected graphs.

Arora and Gouda [1990] introduce a self-stabilizing reset algorithm for asynchronous shared-memory systems. They also utilize a compositional approach in their design wherein one self-stabilizing layer is built on another. Their algorithm is dynamic, allowing the topology to change, as long as the corresponding graph is connected. Unlike in Dolev et al. [1990], the assumption of a distinguished process (leader) for spanning-tree construction is not made. However, their system is asymmetric by identity. Each process has a unique identifier.

The system consists of three layers, a self-stabilizing spanning-tree layer, a self-stabilizing wave correction layer, and an application layer. In the spanning-tree layer, a leader or root is elected, leading to a rooted spanning tree. In the wave layer, reset requests are forwarded to the root which initiates a diffusing computation in which the reset propagates to the leaves of the tree and is "reflected" back to the root. Once it returns to the root, the reset is complete.

A slightly different layering technique is found in Katz and Perry [1990]. (We will discuss this work in greater detail in Section 5.2.) They provide a self-stabilizing "'platform" by which self-stabilization may be forced onto asynchronous message-passing systems for which a decidable predicate describing the set of safe states has been provided. The purpose of their "platform" is to reset the system upon the detection of an illegitimate state. The "platform" only writes to variables of the original program if an illegitimate state is detected. Thus, during normal execution the "platform" does not affect the original program.

The work of Katz and Perry [1990] suggests the next logical methodological step for constructing self-stabilizing systems. The property of self-stabilization can be separated from the algorithms themselves. We now discuss the role of compilation in the construction of self-stabilizing systems.

5. THE ROLE OF COMPILERS WITH RESPECT TO SELF-STABILIZATION

The purpose of a compiler is to take a program written in one language and produce an "equivalent" program in another language. We typically think of a compiler as taking a source program in a high-level language and producing an object program to be run on a particular architecture. However, in either case the architecture is represented by a language, and thus a high-level language may be thought of as an architecture in its own right. Additionally, the source and target may both be on the same architecture, albeit abstract.

More formally, a compiler is a homomorphism $f: A \rightarrow B$ between classes of architectures or systems, such that for each system $M \in A$, $f(M)$ mimics in some well-defined manner the computations of M.

An operational definition of "equivalence" is that the object program preserves those properties of the source program that are important to the designer. In a sequential paradigm under termination we would expect that both programs compute the same result. In practice, the object program mimics the source program in a well-defined manner. For instance, in the case of imperative languages, individual actions of the source program map to one or more specific actions in the object program. In a distributed or concurrent paradigm, we would expect the preservation of qualitative properties due to the need for coordination and control among the processes of a system. This is true for nonterminating sequential systems, such as operating systems, as well.

When we design a program that is self-stabilizing, we wish the compiler to produce an object program that is self-stabilizing on its target architecture. Otherwise, it would be pointless to go through the extra effort necessary to make the original program self-stabilizing. Given the importance of self-stabilization as a program property, it is prudent to ask whether it is preserved by the compilation process. Better yet, should our program not be self-stabilizing, we would like a compiler that produces an object program that is self-stabilizing on its target architecture. In doing so we would be able to abstract the difficulties of self-stabilization into the compilation process and allow systems designers to focus on other issues. Such capabilities would be very desirable as standard options on a compiler. If the full benefits of self-stabilizing programs are to be realized, they must be studied in the context of compilers.

Let us review what has been accomplished with respect to self-stabilization and compilers. Dijkstra's [1973] seminal work in self-stabilization may be interpreted as a demonstration of the limits of the compilation process with respect to self-stabilization, although he did not present it in this fashion. In particular, he showed that in general one cannot construct a self-stabilizing ring of processes consisting of identical (symmetric) processes. We may interpret this result as follows. There does not exist a compiler from asymmetric rings to symmetric rings that forces or preserves self-stabilization. However, if self-stabilization is not required, we can compile an asymmetric ring into a symmetric ring. We combine the processes of the asymmetric ring into one new process. Each copy of this new process is started in the component corresponding to the appropriate original process. The "symmetric ring" will be asymmetric by state. This observation is due to Gouda et al. [1990].

In the work including and subsequent to Dijkstra's original presentation, up to the past few years, self-stabilizing systems have been produced by "handcrafting" them individually from scratch. The one exception to this is Lamport's [1986] mutual exclusion algorithm wherein he inserts additional statements into a nonstabilizing program to yield a self-stabilizing version. Only recently has the subject of compilation been raised with respect to self-stabilization. One of the first papers in this area suggested that the task of forcing self-stabilization is often difficult to achieve, especially when compiling from one abstract architecture to another. Subsequent results would appear to contradict these conclusions. With this in mind, we summarize the important contributions of this early paper, followed by detailed discussion of later work in subsequent subsections.

Gouda et al. [1990] argue that self-stabilization is in principle unstable across architectures. In particular, they demonstrate pathological cases for a variety of abstract architectures under which there cannot exist a compiler that preserves or forces self-stabilization, given a mostly liberal definition of program equivalence. Their proofs rely on exhibiting programs for which there does not exist an equivalent self-stabilizing version under the architecture in question. The classes of concurrent systems they consider include cellular arrays, communicating finite-state machines, CSP systems, systems of Boolean programs communicating via 1-reader/1-writer shared variables, and Petri Nets.

The research of Gouda et al. [1990] has led to greater insight into the factors that contribute to the instability of self-stabilization. In addition to termination, two new phenomena that prevent self-stabilization, isolation, and look-alike configurations, were characterized. These factors are discussed in Section 6. Beyond characterizing some of the factors that prevent self-stabilization within systems, the major contribution of this work is the methodological issues it raised and not the individual results. In particular, the results demonstrate that the ability to force or preserve self-stabilization is highly dependent on how we require properties such as termination, concurrency, and fairness to be preserved when compiling from one system to another. For instance if we assume that termination must be preserved, then self-stabilization cannot be forced for most of the systems examined. We direct the reader to the paper for further details.

In the remainder of this section we discuss compilers that force self-stabilization onto sequential programs, message-passing systems, and shared-memory systems.

5.1 Compilers for Sequential Programs

The primary focus of research on the property of self-stabilization has been in the context of concurrent and distributed systems. Within such systems the goals of algorithms are qualitative as well as quantitative. In this section we discuss the work that has been done with respect to sequential programs with quantitative goals (i.e., they compute a function or a relation). The functional or relational (under nondeterminism) specification of a sequential program, as represented by a mapping between initial and final states, is trivial to preserve during normal compilation in contrast to other specifications. However, the requirement of termination makes self-stabilization more difficult to achieve. The work in this area has focused on the tradeoffs and relationships between compilers and their object programs with respect to their complexities. These relationships have been examined with respect to compilers that do and do not preserve termination.

This area has been studied by Browne et al. [1990] and Schneider [1992]. The work of Browne et al. was done in the context of reactive systems, more specifically real-time decision systems which react to periodic sensor readings. We briefly describe the rule-based program model as it appears in the two papers.

Both models are similar to that of UNITY [Chandy and Misra 1988]. For specific details see the respective papers. A *rule-based program* consists of an initialization section and a finite set of rules. A *rule* is a multiple assignment statement with a guard (enabling condition) which is a predicate over the variables of the program and thus over its state. In a state for which the guard of a rule is true, we say that the rule is *enabled.* A *computation* is a sequence of rule firings wherein at each step an enabled rule is nondeterministically selected for execution. Because statements are nondeterministically chosen to be executed, there is not a program counter. This structure provides a theoretically satisfying model from which to study programs, as the state space of a program is composed entirely from its variables.

A program is considered to have terminated when a fixed point is reached. A *fixed point* is a state in which the values of the variables (hence the state) cannot change. This is the case if for every rule its guard is false or the execution of its assignment does not change the values of its target variables. We define a *partial fixed point* as a state from which a subset of the program variables cannot change.

The variables of a program consist of its input and output variables respectively. Self-stabilization would not be possible were we to allow inputs to be corrupted; thus input variables are considered incorruptible. This may be motivated by the fact that in a reactive system, the values of input variables are determined by updates from an external environment.

We consider two definitions of program equivalence by which an object program may be said to implement its source. In each case we allow the object program to use additional variables. An object implements its source with respect to fixed points (partial fixed points) if on any input, the object is guaranteed to reach a fixed point (partial fixed point) corresponding to a fixed point in the source. Thus we consider compilers that do and do not preserve termination. These definitions, though they allow the use of additional variables, are not the same as Katz and Perry's [1990] *extension* (see Section 5.2), since no requirement is placed on how a computation proceeds in the object program. We are now ready to present the results in this area.

In order for a terminating program to be self-stabilizing, the relation it computes must be verifiable in one step; otherwise it could terminate in an unsafe state. This is not the case for most useful programs. Consider, for example, sorting a list of n elements. In order for a sort program to terminate in a self-stabilizing manner, it must verify in one step that the output is indeed a sorted list and that it is a permutation of the input. Such a check requires $\Omega(n)$ comparisons in an array-based model and thus cannot be achieved in one step. By virtue of this fact, we can conclude that there does not exist a compiler for arbitrary programs that forces self-stabilization while preserving termination. An example of a relation that is verifiable under a standard model is an algorithm that takes a composite x as input and outputs a divisor d. We need only verify that d divides x.

Browne et al. [1990] demonstrate a class of programs for which there is a compiler that forces self-stabilization, while preserving termination. Their object programs have a runtime and size within a constant factor of their source. They also assume that inputs are not corruptible. These programs, which they call "acyclic," fit the following criteria:

(1) Their data dependency graphs are acyclic.
(2) Each rule assigns only one variable.
(3) For any pair of enabled rules with the same target variable, both rules will assign the same value to that variable.

Since general results are not possible for infinite-state programs it is only natural to consider finite-state programs. However, when we investigate the class of programs restricted to boolean variables, we find that analogous results cannot be obtained. The straightforward solution to this problem is to create a state graph from which one can construct a lookup table. Using the table, under any possible state the system will act appropriately and in constant time. Since the number of states is exponential in the number of variables, the size of the resulting program is exponential, and thus its construction is not tractable. Schneider [1992] shows that if there exists a polynomial-time compiler that forces self-stabilization (while preserving termination) onto boolean programs then *PSPACE* = *NP*. These are two complexity classes which are not thought to be equivalent [Garey and Johnson 1979; Johnson 1990]. Should we require that source and object have the same set of variables, then under such assumptions *PSPACE* = *P*. This is even less likely. Thus it appears that we cannot do better than exponential runtime for such a compiler, if we wish to compile arbitrary programs.

We define a family $\{P_n | n \geq 1\}$ of boolean programs as *polynomial-time uniform* if given the input size n, P_n can be constructed in time polynomial in n. Schneider [1992] shows that self-stabilizing polynomial-time uniform families of boolean programs can be used to compute exactly the relations in $NP \cap co\text{-}NP$ and provides polynomial-time compilers to force self-stabilization onto polynomial-time uniform families of boolean programs in P as well as $NP \cap co\text{-}NP$.

Since it is clear that for arbitrary programs, one cannot obtain the same results as for acyclic ones, it is prudent to investigate the removal of the requirement that the object program reach a fixed point (terminate). Schneider [1991] shows that by weakening the definition of equivalence to partial fixed points (termination is not required), we can produce in quadratic time an equivalent self-stabilizing program with a time complexity that is the square of the original. Schneider [1992] further improves this result by reducing the time complexity of the object to a constant factor of the source. We describe informally how they achieve their results.

It is assumed that the program computes a function: otherwise it can be determinized. Thus it always reaches a unique fixed point with respect to any input. The basic idea is to make an extra copy of the state space (each of the variables) and use it to continuously regenerate the unique fixed point of the original program. If the original variables are corrupted, then eventually the new variables will generate a fixed point, and if their values are different from the original variables, the original variables are reset.

Two auxiliary variables, *step* and *ceil*, are introduced. They may be initialized to 0 and 1 respectively, although this is unnecessary. Every time one of the original rules is executed, *step* is incremented. When *step* ≥ *ceil*, the original variables are reset to their initial values; *step* is set to zero; and *ceil* is incremented by one. Thus the original program is continuously reexecuted for one more step each time. No matter how the variables are altered, eventually *step* will be greater than or equal to *ceil*, and the original program will be rerun for *ceil* + 1 steps; and eventually a fixed point will be generated.

The time complexity of the translation is linear if the original source program was functional and quadratic otherwise. The time complexity of the object, in terms of its source, for a computation of length n, has an upper bound of $O(n^2)$. This is reduced to $O(n)$ in Schneider [1992]. A further improvement is possible. If the source program has an execution time bounded by $f(I)$, a function of the input, then *ceil* may be replaced by that function. The object will then have the same time complexity as the source and will stabilize within a constant factor of that time as well. This technique may be applied to finite-state programs since they will have bounded execution time as well.

5.2 Compilers for Asynchronous Message-Passing Systems

The work of Katz and Perry [1990] provides a general methodology, realizable by a compiler, to convert nonstabilizing programs to self-stabilizing versions in an asynchronous message-passing system. This is accomplished through the introduction of a self-stabilizing platform which when superposed (interleaved) with a nonstabilizing program yields a self-stabilizing one. The resulting program is called an *extension* of the original program. Their methodology, as we shall see, is restricted to programs for which there exists a decidable predicate for determining whether a global state is legitimate or not.

There are three components to their algorithm: a self-stabilizing version of Chandy and Lamport's [1985] global-snapshot algorithm, a self-stabilizing reset algorithm which is superposed on it, and a nonstabilizing program on which they are superposed to yield a self-stabilizing extension. A *superposition* interleaves new code with original code, without interfering with the original codes normal execution.

Global snapshots are repeatedly activated from a distinguished initiator. It is assumed that there exists a decidable predicate that recognizes representations of the illegitimate global states of the source. Once the distinguished initiator process has obtained a snapshot, it evaluates this predicate. Should the predicate fail, indicating an illegitimate state, execution of the reset algorithm is initiated, setting the global state of the source program to an initial state.

The snapshot and reset algorithms may also be thought of as providing a self-stabilizing "platform" that may be layered on programs in order to produce self-stabilizing extensions. This methodology is suitable for a compilation process requiring both program and predicate (specifying the set of safe states) as input and producing a self-stabilizing version of the same program as output.

An extension may be defined in the following way: Program Q is an *extension* of program P if the projection onto all variables and messages of P from the set of legal execution sequences in Q is equal to the set of legal execution sequences in P, up to stuttering. The program counters need not be the same. Roughly speaking this means that the subset of Q corresponding to P acts exactly like P except that the same state may repeat. This last part is crucial because if P terminates, its extension Q will need to repeat the final state of P forever, changing only variables not present in P, in order to be self-stabilizing. If Q terminates, it cannot be a self-stabilizing extension of P, because otherwise it could be put in a final state with an illegal global state relative to messages and variables.

The assumption that there exists a decidable predicate to differentiate between legitimate and illegitimate states is not reasonable in all cases. Most programs are not designed with a precise definition of legitimate and illegitimate in mind, and in fact Katz and Perry [1990] define legitimate in terms of the reachable set. The membership problem for the reachable set is generally undecidable and even when it is not in a nondeterministic program, the complexity of deciding membership may be greater than the complexity of the program itself which need only take one execution path. If we restrict ourselves to the domain of finite-state programs then such problems become decidable, but they are still potentially intractable as we discussed in Section 5.1.

Another point worth noting about their methodology is that the global-snapshot algorithm does not produce the current state, but rather a possible successor to the state from which it was initiated. Thus it is possible to do a reset from a legitimate state if the original algorithm stabilizes of its own accord.

More recently, Awerbuch and Varghese [1991] provide a compiler from deterministic synchronous message-passing programs into self-stabilizing versions for asynchronous message-passing systems. Their compiler is for noninteractive protocols. A *noninteractive protocol* is a protocol specified by an input/output relation. The protocols they consider compute a relation over the current enviroment. In particular, they take as input the current network topology. This allows the assumption that inputs are not corruptible. Examples of the types of algorithms considered are leader election and spanning tree.

5.3 Compilers for Asynchronous Shared-Memory Systems

We briefly touch upon some techniques by which self-stabilization may be forced onto shared-memory systems. In the same way as was done for message-passing systems, one can compose snapshot and reset algorithms. A snapshot can be accomplished by a diffusing computation. If an illegitimate state is detected, a reset, such as appears in Arora and Gouda [1990], is initiated.

A self-stabilizing synchronous shared-memory system may be compiled into a self-stabilizing asynchronous shared-memory system as follows. We assume that each process can read and write in one atomic action and that each shared variable is written to by only one process. A self-stabilizing asynchronous unison algorithm (as defined in Section 4) is used to simulate the steps of the synchronous system. Two copies of each shared variable are maintained corresponding to the current value and the previous value. In this way, a process can read the appropriate values of its neighbors in order to compute its own next value. When the local clock of a process is incremented from i to $i + 1$, it simultaneously executes one step of the synchronous system, updating the current and previous values for each of the shared variables that it owns.

6. SOME FACTORS THAT PREVENT SELF-STABILIZATION

In this section we discuss some of the factors that prevent self-stabilization. The first factor discovered to prevent self-stabilization within a system, *symmetry*, was discussed in Section 3.3. The next factor, *termination*, was touched upon in our discussions on communication protocols (Section 3.6), as well as compilers for sequential programs and asynchronous message-passing programs (Sections 5.1 and 5.2). The adverse effects of *termination* are easily seen. If any unsafe global state is a final state, then a system will not be able to stabilize. Self-stabilization is generally incompatible with termination. The one case where self-stabilization can be achieved in the presence of termination is for finite-state sequential programs as we discussed in Section 5.1. Since the number of states are finite, a compiler can remove all of the unsafe states. While the property of termination is very natural when dealing with algorithms whose goal is quantitative (i.e., compute a function), it is unnatural in the domain of distributed systems, where computations are nonterminating by design and have qualitative goals such as coordination and control. One form of termination that occurs within distributed systems is deadlock wherein one or more processes wait for an event that will not occur. We now discuss how the use of synchronization primitives can lead to deadlock states within a system and thus prevent self-stabilization.

Within a distributed message-passing system, there are certain points of synchronization wherein one process must wait for a message to come from another process. Typically a process sends a message and then waits for a response. By way of a malicious adversary, control of a local process could be placed at a point just after a send instruction without a message actually having been sent. Thus at any local process state that follows the sending of a message, given the existence of a malicious adversary, it is impossible for that process to know whether a message has in fact been sent. This situation can lead to deadlock wherein one or more processes wait for messages that will never come. This difficulty is addressed by Katz and Perry [1990] through the use of message prods wherein a message is spontaneously re-sent from time to time. Thus, even under the impression that a message has been sent, it will continue to be sent again, unless some form of acknowledgment has been received. In this way, deadlock is avoided.

In a shared-memory system, a process can test the value of shared memory whenever it wants to. There is no need to wait for a message, and thus such deadlocks are not a problem. By contrast it is generally impossible to avoid deadlock in a CSP system as defined in Hoare [1978]. In CSP all communication is synchronized such that both the receiver of a message is blocked until it arrives and the sender of a message is blocked until it is received. By virtue of this property we can show that there cannot exist a compiler that forces self-stabilization onto CSP systems, and in fact only a very restricted set of CSP systems can be self-stabilizing at all. Consider the undirected graph formed from a CSP system wherein a send and receive between any two processes results in an edge between them. Assume we have a system for which the corresponding graph forms a cycle. There does not exist a self-stabilizing version of such a system wherein the same communication pattern exists. Consider that if the equivalent self-stabilizing version has a cycle in its graph then it contains a deadlock state wherein each process in the cycle is attempting to synchronize with its neighbor.

We now elaborate upon the two new factors that are discussed in Gouda et al. [1990]. An *isolation* occurs within a system when the local state and computation of each process is consistent with some safe global state and computation, but the resultant global state and computation is not safe. The system is unable to stabilize due to inadequate communication or coordination between its processes. For a precise formal definition, we refer to the reader to Gouda et al. [1990].

We give a simple example of isolation. Consider an asynchronous message-passing system with three processes connected as a tree consisting of a root process and two child processes. There is a communication channel from the root to each child process. Let each process have a counter. An action of the root consists of incrementing its counter and sending a message token to one of its children. An action of a child consists of receiving a message token and incrementing its counter. We define a state as safe when the sum of the children's counters is equal the sum of the root's counter plus the number of tokens in the channels. Since the children have no way to communicate and their actions are initiated by the root, a malicious adversary could put the system in an unsafe global state from which it could not stabilize. For each child the local computation and state are consistent with all safe global computations, but the resultant global state and computation are not. The preceding example is degenerate in that any local state will be consistent with a safe global state.

Look-alike configurations result when the same computation (sequence of actions) is enabled at two different states with no way to differentiate between them. Should one of the two states be unsafe then the system cannot guarantee convergence from the unsafe state. This phenomenon is inherent to Petri nets. For every infinite computation in a Petri net, there are an infinite number of look-alike configurations (states) from which that computation may take place.

We give a simple example of look-alike configurations. Consider an asynchronous message-passing system consisting of two processes, P and Q, with message channels between them in both directions, and only one type of message called "token." Each process has one action in which it receives a token and forwards it to its neighbor (the other process). All of the states in which there are one or more tokens in the channel from Q to P are look-alike configurations for the above system. Consider that from any such state, the computation $(pq)^* \equiv p, q, p, q, \cdots$ is enabled, where p and q denote the one action of P and Q respectively. Define a state as safe if there is exactly one token in the system. Clearly, the above system is not self-stabilizing.

7. SUMMARY

We have seen that it is indeed possible to achieve the property of self-stabilization within most systems. However, this continues to be a difficult task. Further development of techniques and heuristics for the design of self-stabilizing programs and their proofs will help in this effort.

We have defined self-stabilization, stabilization, probabilistic self-stabilization, and pseudo-stabilization. Further investigation into the relationships between these formalizations of the notion of self-stabilization may prove fruitful. Other useful definitions may exist as well. A definition that incorporates the notion of convergence span would be useful.

One weakness of our definition of self-stabilization is that it is a global property. No requirement is placed on how a system converges to a safe state. Failure in one component may lead to corrective actions across an entire system. Local detection and correction of failures is more "natural" and desirable than techniques such as global reset. Work in this area has been initiated by Awerbuch et al. [1991]. Further formalization of these ideas would provide helpful guidelines for the designer of self-stabilizing systems.

For finite-state machines, we have seen that there are tradeoffs between the number of states, the topology, and the level of atomicity, as well as how we define self-stabilization. The investigation into the exact relationships between these properties continues.

With respect to compilers, we have seen how the dimensions of program equivalence, type of refinement, and class of programs may influence our ability to force or preserve self-stabilization. For instance, self-stabilizing extensions may be used to force self-stabilization onto message-passing programs for which the set of safe states can be described by a decidable predicate, but they do not preserve termination. These dimensions affect complexity measures as well. In addition, there are tradeoffs between the complexities of the source program, object program, and compiler. Further research is required to better understand these relationships.

The investigation of the compilation process benefits not only the compiler writer, but also the systems designer who seeks to "handcraft" self-stabilization. Understanding the sensitivity or instability of self-stabilization and the factors that contribute to it, is the first step in constructing systems with this property, whether we wish to handcraft or automatically produce or preserve it. Along these lines, a formal characterization of the factors that prevent self-stabilization would also be desirable.

ACKNOWLEDGMENTS

This survey is dedicated to the memory of Louis Emerson Rosier (1951–1991), friend and advisor, whose editorial feedback was very important in the early stages of this manuscript.

It was at his suggestion that we examined self-stabilization from the perspective of compilers. We also acknowledge the assistance of Mohamed Gouda as an invaluable source of information on self-stabilization. We thank Anish Arora for his insightful questions during discussions on the topic of self-stabilization. We extend our gratitude to Jay Misra and J. R. Rao for encouraging us to submit this work for publication. We thank Edgar Knapp, J. R. Rao, and Jacob Kornerup for providing comments that led to greater clarity in our presentation at various stages of its development. Finally, we acknowledge the contribution of our editor Richard Muntz and the anonymous referees. Their comments and suggestions contributed significantly to the overall clarity and structure of this work and inspired a number of improvements.

This work was supported in part by a grant from the Office of Naval Research, grant N00014-89-J-1913.

REFERENCES

ABADIR, M. S., AND GOUDA, M. G. 1992. The stabilizing computer. In *Proceedings of the 1992 International Conference on Parallel and Distributed Systems*. (Dec.).

AFEK, Y., AND BROWN, G. 1989. Self-stabilization of the alternating-bit protocol. In *Proceedings of the 8th Symposium on Reliable Distributed Systems*, 80–83.

AFEK, Y., KUTTEN, S., AND YUNG, M. 1990. Memory-efficient self-stabilization on general networks. In *Proceedings of the 4th International Workshop on Distributed Algorithms* (Bari, Italy, Sept.). In Lecture Notes in Computer Science, vol. 486. Springer-Verlag, New York, 15–28.

ARORA, A. 1992. A foundation for fault-tolerant computing. Ph.D. dissertation, Dept of Computer Sciences, Univ. of Texas at Austin.

ARORA, A., AND GOUDA, M. G. 1992. Closure and convergence: A foundation for fault-tolerant computing. In *Proceedings of the 22nd International Conference on Fault-Tolerant Computing Systems*.

ARORA, A., AND GOUDA, M. G. 1990. Distributed reset. In *Proceedings of the 10th Conference on Foundations of Software Technology and Theoretical Computer Science* (Dec.). Also in *Lecture Notes on Computer Science*, vol. 472. Springer-Verlag, New York.

ARORA, A., DOLEV, S., AND GOUDA, M. G. 1991. Maintaining digital clocks in step In *Proceedings of the 5th International Workshop on Distributed Algorithms* (Oct.). Also in *Parall. Process. Lett. 1*, 1 (Sept.), 11–18.

AWERBUCH, B., AND VARGHESE, G 1991 Distributed program checking: A paradigm for building self-stabilizing distributed protocols. In *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science* (Oct.). IEEE, New York

AWERBUCH, B., PATT, B. AND VARGHESE, G. 1991. Self-stabilization by local checking and correction. In *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science*, (Oct.). IEEE, New York.

BASTINI, F., YEN, I., AND CHEN, I. 1988. A class of inherently fault tolerant distributed programs. *IEEE Trans. Softw. Eng. 14*, 1432–1442.

BASTANI, F., YEN, I., AND ZHAO, Y. 1989. On self-stabilization, non-determinism, and inherent fault tolerance. In *Proceedings of the MCC Workshop on Self-Stabilizing Systems*. MCC Tech. Rep. STP-379-89.

BROWNE, J. C., EMERSON, A , GOUDA, M., MIRANKER, D., MOK, A., AND ROSIER, L. 1990 Bounded-time fault-tolerant rule-based systems. *Telematics Informat. 7*, 3/4, 441–454.

BROWN, G. M., GOUDA, M. G., AND WU, C.-L. 1989. Token systems that self-stabilize. *IEEE Trans. Comput. 38*, 6 (June 1989), 845–852.

BURNS, J. E. 1987. Self-stabilizing rings without demons. Tech. Rep. GIT-ICS-87/36, Georgia Inst of Technology.

BURNS, J. E., AND PACHL, J. 1989. Uniform self-stabilizing rings. *ACM Trans Programm Lang. Syst. 11*, 330–344

BURNS, J. E , GOUDA, M. G., AND MILLER, R. E 1990 Stabilization and pseudo-stabilization. Tech. Rep. TR-90-13, Dept. of Computer Sciences, Univ. of Texas at Austin

BURNS, J. E , GOUDA, M. G., AND MILLER, R. E 1989. On relaxing interleaving assumptions. In *Proceedings of the MCC Workshop on Self-Stabilizing Systems* MCC Tech. Rep. STP-379-89.

CHANDY, K. M., AND LAMPORT, L 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst. 3*, 1, 63–75.

CHANDY, K. M., AND MISRA, J. 1988. *Parallel Program Design: A Foundation*. Addison-Wesley, New York.

CHANG, E. J. H , GONNET, G. H. AND ROTEM, D. 1987. On the costs of self-stabilization. *Inf Process Lett. 24*, (1987), 311–316.

CHEN, N. S., YU, F. P., AND HUANG, S. T. 1991. A self-stabilizing algorithm for constructing spanning trees. *Inf. Process. Lett., 39*, (1991), 147–151.

CHENG, A. M. K. 1990. Analysis and synthesis of real-time rule-based decision systems. Ph.D. dissertation, Dept of Computer Sciences, Univ. of Texas at Austin.

COUVREUR, J.-M., FRANCEZ, N., AND GOUDA, M. G. 1992. Asynchronous unison. In *Proceedings of the 12th International Conference on Distributed Computing Systems* (Yokohama, Japan, June).

CRISTIAN, F. 1991. Understanding fault-tolerant distributed systems. *Commun. ACM 34*, 2 (Feb.), 56–78.

CRISTIAN, F. 1985. A rigorous approach to fault-tolerant programming. *IEEE Trans. Softw. Eng 11*, 1, (1985).

DIJKSTRA, E. W. 1986. A belated proof of self-stabilization. *Distrib. Comput., 1*, 5–6.

DIJKSTRA, E. W. 1974. Self-stabilizing systems in spite of distributed control. *Commun. ACM 17*, 643–644.

DIJKSTRA, E. W. 1973. Self-stabilization in spite of distributed control. In *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, Berlin, 1982, 41–46. Originally published in 1973.

DOLEV, S., ISRAELI, A., AND MORAN, S. 1990. Self-stabilization of dynamic systems. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing* (Quebec City, Canada, Aug.). ACM, New York

EVANGELIST, M. 1989. *Proceedings of the MCC Workshop on Self-Stabilizing Systems*. MCC Tech. Rep. STP-379-89.

FLATEBO, M., AND DATTA, A. 1992. Two-state self-stabilizing algorithms. In *Proceedings of the 6th International Parallel Processing Symposium* (Beverly Hills, Calif. Mar.), 198–203.

FLATEBO, M., DATTA, A., AND GHOSH, S. 1991 Self-stabilization in distributed systems. *IEEE Comput.*

GAREY, M., AND JOHNSON, D. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York.

GHOSH, S. 1990a. Self-stabilizing distributed systems with binary machines. In *Proceedings of the 28th Annual Allerton Conference*, 988–997.

GHOSH, S. 1990b Understanding self-stabilization in distributed systems. Tech. Rep. TR-90-02, Dept. of Computer Science, Univ. of Iowa.

GOUDA, M. G. 1989. The inevitable properties of programs. Dept. of Computer Sciences, Univ. of Texas at Austin.

GOUDA, M. G. 1991. The stabilizing philosopher: Asymmetry by memory and by action. Tech. Rep. TR-87-12, Dept. of Computer Sciences, Univ. of Texas at Austin.

GOUDA, M. G., AND EVANGELIST, M. 1990. Convergence/response tradeoffs in concurrent systems. In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing* (Dec.). IEEE, New York.

GOUDA, M. G., AND HERMAN, T. 1991. Adaptive programming. *IEEE Trans. Softw. Eng. 17*, 9 (Sept.).

GOUDA, M. G., AND HERMAN, T. 1990. Stabilizing unison. *Inf. Process. Lett. 35* (1990), 171–175.

GOUDA, M. G., AND MULTARI, N. 1991. Stabilizing communication protocols. *IEEE Trans. Comput. 40*, 4 (Apr.), 448–458.

GOUDA, M. G., HOWELL, R. R., AND ROSIER, L. E. The instability of self-stabilization. *Acta Inf. 27*, (1990), 697–724.

HADDIX, F. F. 1991. Stabilization of bounded token rings. Tech. Rep. ARL-TR-91-31, Applied Research Lab., Univ. of Texas at Austin.

HERMAN, T. 1990. Probabilistic self-stabilization. *Inf. Process. Lett. 15*, 63–67.

HOARE, C. A. R. 1978. Communicating sequential processes. *Commun. ACM 21*, 666–677.

ISRAELI, A., AND JALFON, M. 1990. Token management schemes and random walks yield self stabilizing mutual exclusion. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing* (Quebec City, Canada, Aug.). ACM, New York.

ISRAELI, A., AND JALFON, M. 1989. Self-stabilizing ring orientation. Tech. Rep., Dept. of Electrical Engineering, Technion-Israel.

JOHNSON, D. 1990. A catalog of complexity classes. In *Handbook of Theoretical Computer Science*. vol. A. North-Holland, Amsterdam.

JOHNSON, B. 1988. *The Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley, New York.

KATZ, S., AND PERRY, K. J. 1990. Self-stabilizing extensions for message-passing systems. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing* (Quebec City, Canada, Aug.). ACM, New York.

KRUIJER, H. S. M. 1979. Self-stabilization (in spite of distributed control) in tree-structured systems. *Inf. Process. Lett.*, *8*, 2, 2–79.

LAMPORT, L. 1986. The mutual exclusion problem: Part II—Statement and solutions. *J. ACM 33*, 327–348.

LAMPORT, L. 1984. Solved problems, unsolved problems and non-problems in concurrency. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*. ACM, New York, 1–11.

LAPRIE, J.-C. 1985. Dependable computing and fault tolerance: Concepts and terminology. In *Proceedings of FTCS-15*, 2–11.

LEHMAN, D., AND RABIN, M. 1981. On the advantages of free choice: A symmetric and fully distributed solution of the dining philosophers problem. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, 133–138.

LIN, X., AND GHOSH, S. 1991. Self-stabilizing maxima finding. In *Proceedings of the 28th Annual Allerton Conference*.

MULTARI, N. 1989. Towards a theory for self-stabilizing protocols. Ph.D. dissertation, Dept. of Computer Sciences, Univ. of Texas at Austin.

ÖZVEREN, C., WILLSKY, A., AND ANTSAKLIS, P. 1989. Stability and stabilizability of discrete event dynamic systems. MIT LIDS Pub., LIDS-P-1853, MIT, Cambridge, Mass.

SCHNEIDER, M. 1992. Compiling self-stabilization into sequential programs. Dept. of Computer Sciences, Univ. of Texas, Austin, Tex.

SCHNEIDER, M. 1991. Self-stabilization—A unified approach to fault tolerance in the face of transient errors. In Tech. Rep. TR-91-18, Dept. of Computer Sciences, Univ. of Texas at Austin.

TCHUENTE, M. 1981. Sur l'auto-stabilisation dans un réseau d'ordinateurs. *RAIRO Inf. Theor. 15*, 47–66. In French.

WHITBY-STREVENS, C. 1979. On the performance of Dijkstra's self-stabilising algorithms in spite of distributed control. In *Proceedings of the 1st International Conference on Distributed Computing Systems*. IEEE, New York.

Received 1991, final revision March 1992, accepted January 1993