Acta Informatica 27, 697–724 (1990)

Acta Informatica
© Springer-Verlag 1990

The Instability of Self-Stabilization *

Mohamed G. Gouda [1], Rodney R. Howell [2], and Louis E. Rosier [1]
[1] Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, USA
[2] Department of Computing and Information Sciences, Kansas State University, Manhattan, KS 66506, USA

Received July 3, 1989 / April 9, 1990

Summary. We argue that the important property of self-stabilization is, in principle, unstable across system classes. In particular, we first define a very broad notion of simulation. We then define what it means for a simulation to either preserve or force self-stabilization. Given these definitions, we then show that, for a variety of system classes, there is no simulation that preserves or forces self-stabilization.

1. Introduction

A system is said to be self-stabilizing if starting from any configuration the system is guaranteed to reach a "legal" configuration. The motivation behind this concept is that if, due to some unpredictable error, the system were to reach an "illegal" configuration, it would eventually correct itself, returning to some "legal" configuration. Thus, self-stabilizing systems are in some sense more robust than those that are not self-stabilizing. The notion of self-stabilization has been utilized in the fields of mathematics and control theory for many years (see, e.g., [CK80, OWA89]). Consider, for example, the Newton-Raphson method for finding roots of functions. For many functions, the Newton-Raphson method is self-stabilizing; i.e., no matter what initial estimate is made for the root, eventually the iteration will converge to a root. The concept of self-stabiliza-

* This work was supported in part by U.S. Office of Naval Research Grant No. N00014-86-K-0763 and National Science Foundation Grant No. CCR-8711579. Preliminary versions of this work have appeared in the Proceedings of the MCC Workshop on Self-Stabilizing Systems, MCC Technical Report No. STP-379-89, Austin, Texas, August 1989, and under the title "System Simulation and the Sensitivity of Self-Stabilization" in the Proceedings of the 14th International Symposium on Mathematical Foundations of Computer Science, LNCS 379, pp. 249–258, Porabka-Kozubnik, Poland, August-September 1989.

Offprint requests to: R.R. Howell

tion was introduced to the field of computer science by Dijkstra in [Dij73], and subsequently has gained much attention in computer science research, particularly in the area of distributed computing; see, e.g., [Dij74, Lam86, BP89, Gou90, BYC88, BGW89, Mul89].

The main purpose of this paper is to demonstrate how the potential for self-stabilization changes radically when one class of systems is simulated by another. As an example, consider a result of Dijkstra's given in [Dij73]. He gave a problem that can be solved by a self-stabilizing asymmetric ring of processes, and showed that no such solution exists if the ring is required to be symmetric (see also [LR81]). In particular, he showed that there is no self-stabilizing solution if all machines are required to the identical, though they are allowed to start in different configurations. This result can be interpreted in the following way: there is no simulation of an asymmetric ring by a symmetric ring that preserves self-stabilization. On the other hand, if self-stabilization is not required to be preserved, an asymmetric ring can be simulated by a symmetric ring as follows: simply compose all machines of the asymmetric ring into one new machine, and use copies of this new machine, each copy starting in the component corresponding to the appropriate original machine, to form the symmetric ring. The above interpretation suggests that the *simulation paradigm*, which is often used in analyzing and designing systems, may not be very robust when the issue of self-stabilization is involved. A simulation of a class A of systems by a class B of systems – the paradigm in action – is simply a function $f: A \rightarrow B$ such that for each system $M \in A$, the computations of $f(M)$ mimic in some well-defined manner the computations of M. An example of a simulation is the simulation of shared memory programs by CSP systems [Hoa78]. In order to examine the effect of self-stabilization on the simulation paradigm, we will demonstrate the presence or absence of various types of simulations on a wide variety of system classes. To make our negative results as strong as possible, our formal definition of simulation will be very weak. Thus, we are able to show that even for a very liberal notion of simulation (in particular, the simulations are not even required to be recursive) there are many cases in which simulations preserving or enforcing self-stabilization cannot exist. Furthermore, we do not even need to consider the effect of inputs to a system in order to achieve these negative results. On the other hand, ours is not the only conceivable "liberal" notion of simulation. Several choices had to be made in selecting a definition, and these choices may affect our results. We discuss such possibilities in the conclusion.

Compilers comprise perhaps the most important example of the simulation paradigm in action. In particular, we can view a compiler as computing a simulation of a source code program by an object code program on a specific architecture. In order for the compiler to be useful, it is imperative that the designer or programmer know what properties are preserved by the compiler. As an example, suppose a system designer works very hard to produce a self-stabilizing Ada program for some purpose. If the compiler were to generate object code that is not self-stabilizing on the target architecture, all of the designer's work would then be for naught. Thus, it is desirable that the compiler preserve self-stabilization, but if it does not, the users need to know. To go one step further, it would be even better if a compiler option could be selected to enforce self-stabilization. In such a case, the system designer would not need to worry about self-stabilization – it would be automatically enforced.

The simulation paradigm is also used in the analysis of systems to determine whether a particular property, such as self-stabilization, is present in a given system. A common scenario is that we have an algorithm or methodology for deciding the presence of the property in one class of systems, say systems of communicating finite-state machines (CFSMs), but do not yet have one for some similar class, such as Boolean CSP systems. In order to achieve such a methodology in a Boolean CSP system, we might simulate the CSP system by a system of CFSMs, analyze the CFSMs, and finally, conclude properties about the CSP system. More formally, let A and B be two system classes. The simulation paradigm is then applied to the analysis of systems in the following manner:

1. Find a simulation $f: A \to B$.
2. Given a machine $M \in A$, analyze $f(M)$.
3. Conclude properties about M.

In order for this procedure to work, the simulation clearly must preserve (the existence or absence of) the property being studied. This is usually not a problem because the standard simulations nearly always preserve most useful properties (e.g., deadlock freedom, reachability, liveness, fair nontermination, etc. [KM69, OL82, EL87, Car87, HRY88]). However, we will show that such is often not the case with respect to self-stabilization.

Hence, we specifically examine two questions related to the design and analysis of various classes of systems. Let A and B be two system classes. The two questions we ask are:

1. Does there exist a simulation of A by B that preserves self-stabilization?
2. Does there exist a simulation of A by B that forces self-stabilization?

The classes of concurrent systems we consider include cellular arrays, communicating finite-state machines, CSP systems, and systems of Boolean programs communicating via 1 reader/1 writer shared variables. In order to demonstrate that the difficulties are not simply products of concurrency, we also consider finite-state machines, Petri nets, Turing machines, and vector addition systems with states.

The main message here is that self-stabilization, as important a property as it is, is very sensitive to changes in the system classes under consideration. Understanding this sensitivity is the first step for a system designer, who seeks self-stabilization in systems. It is also important to keep the results of this paper in perspective. In particular, the definition of simulation is intentionally chosen to be very weak to support the proofs of very strong negative results. Thus, even though a number of positive results are included for completeness, they are not entirely satisfactory. In fact, by taking full advantage of the weakness of our definition, one might show the existence of some type of simulation by exhibiting a nonrecursive simulation. Such a simulation would obviously be unusable in the simulation paradigm. Therefore, in any study focusing on positive results, a stronger definition of simulation should be considered.

The remainder of the paper is organized as follows. In Sect. 2, we define much of the terminology used throughout the paper. In Sect. 3, we give an overview of the specific problems we examine and summarize the main factors that tend to disrupt self-stabilization. In Sect. 4, we examine the role of halting on self-stabilization. In Sects. 5 and 6, we examine two more subtle phenomena,

isolation and look-alike configurations. In Sect. 7, we discuss the assumptions made in this paper, and suggest directions for further research. Finally, in the Appendix we give detailed proofs of several theorems omitted from the body of the paper to improve readability.

2. Definitions

We include in this paper the examination of a wide variety of models of parallel computation. Although a particular model may have a more "natural" definition, for the sake of consistency, we define a *system of n concurrent processes* as a triple (Q, q_0, Δ), where Q is a (possibly infinite) set of *system configurations*, $q_0 \in Q$ is the *initial configuration*, and $\Delta = \{\delta_1, \ldots, \delta_n\}$, where each δ_i is a finite set of *transitions* and all δ_i's are pairwise disjoint. For systems of only one process, we also use the notation (Q, q_0, δ), where δ is a set of transitions. Intuitively, each δ_i represents the transitions of a process M_i. Each transition $t \in \delta_i$ is a partial function[1] $t: Q \rightarrow Q$. If $t(q_1) = q_2$, we also write $q_1 \xrightarrow{t} q_2$ or $q_1 \rightarrow q_2$. If $q_1 \xrightarrow{t_1} q_2 \xrightarrow{t_2} \ldots \xrightarrow{t_n} q_{n+1}$ and $\sigma = t_1 \ldots t_n$, $n \geqq 0$, then we also write $q_1 \xrightarrow{\sigma} q_{n+1}$ or $q_1 \xrightarrow{*} q_{n+1}$. We define the *reachability set* of (Q, q_0, Δ) as the set $R(Q, q_0, \Delta) = \{q \mid q_0 \xrightarrow{*} q\}$. The set $R(Q, q_0, \Delta)$ is distinguished from Q in that Q is the set of *possible* configurations (i.e., those allowed by the syntactic definition of the system), whereas $R(Q, q_0, \Delta)$ is that subset of Q consisting of those configurations that can actually be reached by the system. A *halting configuration* is a configuration $q \in Q$ such that $t(q)$ is undefined for all $t \in \bigcup_{i=1}^{n} \delta_i$. A *finite computation* of (Q, q_0, Δ) is a finite sequence σ of transitions in δ such that $q_0 \xrightarrow{\sigma} q$ for some $q \in Q$, and $t(q)$ is undefined for all $t \in \bigcup_{i=1}^{n} \delta_i$. An *infinite computation* of (Q, q_0, Δ) is an infinite sequence σ of transitions in $\bigcup_{i=1}^{n} \delta_i$ such that for each finite prefix σ' of σ, there is a $q \in Q$ for which $q_0 \xrightarrow{\sigma'} q$. A *computation* of (Q, q_0, Δ) is either a finite computation or an infinite computation. (Note that according to this definition, a proper prefix of a computation is *not* a computation, since there must be some transition defined at the last configuration of the prefix.) Let $C(Q, q_0, \Delta)$ denote the set of all computations of (Q, q_0, Δ).

All of the classes of systems we discuss in this paper can be defined by restricting the above definition of a concurrent system. (A sequential system is just a concurrent system of 1 process.) Usually, it is a straightforward matter to translate the standard definition of some particular system class to some restriction of a general concurrent system. In such cases, we will not give the translation, and we will use whichever characterization is more convenient.

We are now ready to formally define self-stabilization. In order to avoid many unnecessary details, we give a somewhat stricter definition than is typically done. In particular, we restrict the set of "legal" configurations to be exactly the reachability set of the system, rather than some specified superset of the

[1] In order to allow multiple transitions to cause the same action, we should technically define a transition as a symbol denoting a partial function. However, our definition allows us to avoid introducing additional notation that may lead to confusion. In any case, we do allow multiple transitions to cause the same action.

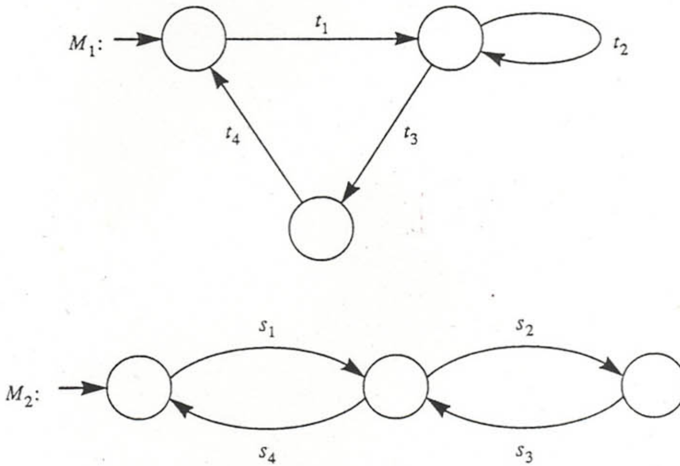

Fig. 1

reachability set. The impact of this restriction is to strengthen our results. More precisely, for each of our negative results, we will be showing a number of specific examples of systems on which self-stabilization cannot be forced, or for which self-stabilization cannot be preserved by some other type of system. If we were to use a more standard definition of self-stabilization, we would need to provide, along with the specific system, some formal description of the "legal" configurations; furthermore, this formal description would have to make sense when applied to an arbitrary simulating system. As it turns out, all of our negative results can be shown by choosing the set of "legal" configurations to be the reachability set. We therefore say that a system (Q, q_0, Δ) is *self-stabilizing* iff for every $q \in Q$, every computation σ of (Q, q, Δ) has a finite prefix σ' such that $q \xrightarrow{\sigma'} q'$ for some $q' \in R(Q, q_0, \Delta)$. It should be clear from the proofs that follow that dropping this restriction does not affect any of our results, either positive or negative.

Much of this paper involves the simulation of one system class by another. In order to conclude that some type of simulation does not exist, we need to formalize the concept of simulation. Before we do this, let us first consider somewhat informally what we mean by a simulation of one system by another. Consider the two finite-state machines M_1 and M_2 shown in Fig. 1. We claim that it makes sense to say that M_2 simulates M_1. To see why, let T be the set of all finite and infinite strings over $\{t_1, \ldots, t_4\}$, and let S be the set of all finite and infinite strings over $\{s_1, \ldots, s_4\}$. Consider the homomorphism[2] $h: T \to S$ defined by

- $h(\varepsilon) = \varepsilon$ (where ε denotes the empty string);
- $h(s_1) = t_1$;
- $h(s_2) = t_2$;
- $h(s_3) = \varepsilon$;
- $h(s_4) = t_3\, t_4$; and
- $h(t\sigma) = h(t)\, h(\sigma)$.

[2] Throughout this paper, we use the term homomorphism strictly in a language-theoretic sense, where the operation preserved is concatenation; i.e., $h(\varepsilon) = \varepsilon$ and $h(t\sigma) = h(t)\, h(\sigma)$.

It is easily seen that $h(C(M_2))=C(M_1)$. Note that since h is a homomorphism, we have for each transition s_i of M_2, a finite (possibly empty) sequence $h(s_i)$ of transitions of M_1 that s_i mimics. It is important that the two sets, $C(M_1)$ and $h(C(M_2))$, are identical: if there are computations in $C(M_1)$ that are not in $h(C(M_2))$, then M_2 may not capture the entire behavior of M_1, and if there are computations in $h(C(M_2))$ that are not in $C(M_1)$, then certain safety properties of M_1 may not hold for M_2.

In order to extend the above ideas to general concurrent systems, we make the following definitions. For a system (Q, q_0, Δ), $\Delta=\{\delta_1, \ldots, \delta_n\}$, let Δ^∞ denote the set of all finite and infinite sequences of transitions in $\bigcup_{i=1}^{n} \delta_i$. For a sequence $\sigma\in\Delta^\infty$, let the *projection* of σ onto process i, denoted $\pi_i(\sigma)$, $1\leqq i\leqq n$, be the homomorphism defined by

- $\pi_i(\varepsilon)=\varepsilon$ (where ε denotes the empty string);
- $\pi_i(t)=\varepsilon$ if $t\notin\delta_i$;
- $\pi_i(t)=t$ if $t\in\delta_i$; and
- $\pi_i(t\sigma)=\pi_i(t)\,\pi_i(\sigma)$.

Let $C_i(Q, q_0, \Delta)=\{\pi_i(\sigma)\,|\,\sigma\in C(Q, q_0, \Delta)\}$, $1\leqq i\leqq n$. We say that a concurrent system (Q', q'_0, Δ') of n processes *simulates* a concurrent system (Q, q_0, Δ) of n processes iff

1. there is a homomorphism $h\colon \Delta'^\infty\to\Delta^\infty$ such that
2. $h(C(Q', q'_0, \Delta'))=C(Q, q_0, \Delta)$,
3. $h(C_i(Q', q'_0, \Delta'))=C_i(Q, q_0, \Delta)$, $1\leqq i\leqq n$, and
4. for any computation σ' of (Q', q'_0, Δ'), $\pi_i(h(\sigma'))$ is finite iff $\pi_i(\sigma')$ is finite, $1\leqq i\leqq n$.

We then call h the *simulation homomorphism.* Note that not only do we require the behavior of the system as a whole to be preserved by a simulation, but that by condition 3, we also require the behavior of each process to be preserved by a corresponding process in the simulating system. This requirement forces the distributed nature of certain systems to be preserved by the simulation. Also, condition 4 prohibits infinite computations from simulating finite computations, and vice versa.

It is possible to weaken the above definition further without changing any of our results. Rather than considering the set of computations of a system as the specification of its behavior, we might wish to consider instead the projections of these computations onto certain "external" transitions. Modifying the definition in this way clearly does not change any positive results, since any simulation under our definition is also a simulation under this alternative definition. Furthermore, our negative results consist of exhibiting specific systems and showing that there is no simulation by any system in some given class that either preserves or forces self-stabilization. Since we are free to exhibit a system in which all the transitions are "external", none of these results are changed.

If $\mathscr{M}_1$ and $\mathscr{M}_2$ are two system classes, and there is a function $f\colon \mathscr{M}_1\to\mathscr{M}_2$ such that for all $M\in\mathscr{M}_1$, $f(M)$ simulates M, we say $\mathscr{M}_2$ *simulates* $\mathscr{M}_1$. We then call f the *simulation.* If for all $M\in\mathscr{M}_1$, $f(M)$ is self-stabilizing iff M is self-stabilizing, we say f is a *self-stabilization preserving simulation.* If for all $M\in\mathscr{M}_1$, $f(M)$ is self-stabilizing, we say f is a *self-stabilization forcing simulation.*

Table 1. Results involving simulations of one class by itself

System class	Forced	
	With halting	Without halting
Cellular arrays	no	yes
Linear cellular arrays	no	yes
Turing machines	no	yes
Finite-state machines	yes	yes
CSP	no	no
1 reader/1 writer shared memory programs	no	?
Communicating finite-state machine	no	?
Bounded communicating finite-state machines	no	no
Petri nets	no	no
Petri nets with capacities	no	no

The reader should note two important points from the above definitions. First, we make no requirements as to how easy it is to either find or compute each f and h. In particular, even though f must be computable in order to be used in the simulation paradigm, we do not make this requirement of a simulation in general. This generality provides for a very weak definition of simulation, yielding some very strong negative results. The second point is that our systems have no input. If inputs were to be considered, we would need to define self-stabilization so that for any input, from any configuration containing that input, all computations eventually reach a configuration reachable from the initial configuration having that input; i.e., we would assume the input to be incorruptible (a notion that suggests other issues that we will not discuss here). However, simply ignoring inputs serves to make our negative results even stronger (i.e., no simulation is possible even when inputs are not considered).

One of the factors that tends to disrupt self-stabilization concerns what we call "isolation". We say an *isolation* occurs at a configuration q of M if there exist a computation σ from q and distinct computations σ_1 and σ_2 from distinct configurations q_1 and q_2, respectively, such that for all $1 \leqq i \leqq n$, either $\pi_i(\sigma) = \pi_i(\sigma_1)$ or $\pi_i(\sigma) = \pi_i(\sigma_2)$, but σ is not enabled at either q_1 or q_2. Intuitively, the processes of M become partitioned into two nonempty sets S_1 and S_2 such that each process in S_i behaves as if it were executing σ_i from q_i, and any communication between the two sets is insufficient to correct this behavior. Another factor we examine concerns what we call "look-alike configurations". We say that configurations q_1 and q_2 are *look-alike configurations* if there is some computation σ enabled at both q_1 and q_2. Intuitively, the system does not have the power to differentiate between q_1 and q_2 because it may behave in exactly the same way upon entering either configuration.

Throughout this paper, we will use the notation σ^n to indicate the sequence σ iterated n times. Likewise, σ^ω will indicate σ iterated infinitely many times. Iteration will take precedence over concatenation, so that, for example, $\sigma\tau^3 = \sigma\tau\tau\tau$.

3. Summary of results

Throughout the remainder of the paper, we present our results concerning various classes of concurrent systems. These results are summarized in Tables 1

Table 2. Results involving simulations of one class by another

Simulated class	Simulating class	With halting		Without halting	
		Preserved	Forced	Preserved	Forced
Cellular arrays	Linear cellular arrays	–	–	yes	yes
Finite-state machines	Turing machines	no	no	yes	yes
1 reader/1 writer shared memory programs	Boolean CSP	no	no	no	no
Boolean CSP	Infinite CSP	yes	no	yes	no
Boolean CSP	Communicating finite-state machines	no	no	?	?
1 reader/1 writer shared memory programs	Communicating finite-state machines	no	no	?	?
Vector addition systems with states	Petri nets	no	no	no	no
Vector addition systems with states	Petri nets with capacities	yes	no	yes	no

and 2. In these tables, the "yes" and "no" entries indicate whether the specified simulation exists; a "?" indicates that we do not know whether the simulation exists; and the two "–" entries indicate that there is no simulation of cellular arrays by linear cellular arrays – even disregarding self-stabilization – when halting is allowed. The proofs of many of these results have the advantage of being rather short and fairly easy to follow. Unfortunately, proofs of this sort tend to give the (sometimes false) impression that the theorems are somewhat obvious. To the contrary, it has been our experience that problems involving self-stabilization are so different from other problems in distributed computing that the intuition developed by studying other problems is often misleading in the study of self-stabilization. To lend support to this claim, we reproduce the following comment of Dijkstra's concerning one of his related proofs in [Dij73]:

> Again I beg my intrigued readers to stop reading here and to try to solve the stated problem themselves, for only then will they (slowly!) build up some sympathy with my difficulties: the problem has been with me for many months, while I was oscillating between trying to find a solution – and many an at first sight plausible construction turned out to be wrong! – and trying to prove the non-existence of a solution. And all the time I had no indication in which of the two directions to aim, nor of the simplicity or complexity of the argument – if any! – that would settle the question.

In order to focus on the meaning of our results, rather than on the technical details of their proofs, we have omitted many of the proofs from the main body of this paper. The omitted proofs appear in the Appendix.

We have uncovered three main factors that tend to disrupt self-stabilization: halting, isolation, and look-alike configurations. Of these three, halting is the

most familiar (as far as we know, isolation and look-alike configurations are new concepts.) Furthermore, it is not hard to see that halting is very likely to inhibit self-stabilization, since the "bad" configuration the system might enter could conceivably be a halting configuration. The system would then have no way of recovering, since it would have already halted. Thus, it follows immediately from the definition of self-stabilization that in a self-stabilizing system, all halting configurations must be reachable.[3] For most system classes, this restriction causes a loss of computational power (see Table 1). A more interesting observation from Table 1, however, is that for some system classes, self-stabilization cannot be forced even when halting is disallowed. Hence, there must be other more subtle factors, such as isolation or look-alike configurations, interfering with self-stabilization. A look at Table 2 shows how halting can also interfere with simulations of one system class by another. First of all, as is shown in the first entry of Table 2, when individual processes are allowed to halt, there can be no simulation of arbitrary cellular arrays by linear cellular arrays (regardless of whether self-stabilization is preserved). The reason for this is that the communication connections in a linear cellular array form a linear chain, whereas in an arbitrary cellular array, arbitrary connections are allowed. Thus, if one process in a linear cellular array were to halt, it would split the system into two isolated components. However, Table 2 again shows entries which are unaffected by the presence of halting. Thus, it seems necessary to examine the extent to which halting affects self-stabilization before studying the more subtle issues of isolation and look-alike configurations.

4. Problems involving halting

Table 1 gives three system classes for which halting is the sole factor in preventing the forcing of self-stabilization: cellular arrays, linear cellular arrays, and Turing machines. (For standard definitions of cellular arrays and Turing machines, see, e.g., [IKM85, Kos74, Smi71] and [HU79], respectively.) Because cellular arrays are synchronous systems with multiple transitions executing simultaneously, they are rather awkward to define formally in terms of our definition of a system given in Sect. 2. For this reason, we omit in this section any formal discussion of cellular arrays, leaving their discussion to the Appendix (Theorems A.1, A.2, and their corollaries). The Turing machines we consider here are nondeterministic with $k \geqq 1$ tapes, each infinite to the right and having an unchangeable marker at the left end. We will now show that any self-stabilizing Turing machine must have an infinite computation; thus self-stabilization cannot be forced if we allow halting.

Theorem 4.1. *There is no self-stabilization forcing simulation of Turing machines by Turing machines.*

Proof. Consider a Turing machine M that starts in a halting configuration; i.e., its only computation is the empty computation. Suppose some self-stabilizing Turing machine M' simulates M. From the definition of simulation, all computations of M' must be finite; hence, M' has a halting configuration. Since Turing

[3] Under the more general notion of self-stabilization, all halting configurations must still be "legal".

machines have infinitely long tapes, we can modify the tape contents or move a tape head in any halting configuration of M' to generate infinitely many new halting configurations for M'. Because M' is self-stabilizing, all of these halting configurations must be reachable. From König's Infinity Lemma [Kon36], M' must have an infinite computation – a contradiction. □

In order to demonstrate the full effect of halting on the possibility of forcing self-stabilization on Turing machines, we now show that self-stabilization can be forced if no halting configurations are present. We should keep this result in its proper perspective. It is not a very strong result due to our weak definition of simulation. In particular, for any infinite computation, there is no bound on the maximum number of moves needed to simulate any transition in the computation. Furthermore, there is no bound on the number of moves made from an arbitrary configuration before a reachable configuration is reached. Nonetheless, it does show the existence of a self-stabilization forcing simulation.

Theorem 4.2. *There is a self-stabilization forcing simulation of Turing machines with no halting configurations by Turing machines.*

Proof. Let M be an arbitrary Turing machine with no halting configurations and k worktapes. We construct a self-stabilizing Turing machine M' that simulates M. M' contains $2k+1$ worktapes and operates as follows. M' simulates M on k tapes in a straightforward manner. After each simulated move of M, M' scans from left to right a special tape containing a list of transitions simulated so far. When M' encounters a symbol other than a transition, it overwrites that symbol with the last transition executed and overwrites the next symbol with a blank. Let n be the number of transitions in the list. M' then blanks the first n cells of the remaining k tapes and simulates the listed transitions on these k tapes. After the simulation of the list of transitions is completed, M' compares the first n symbols on each of the two sets of k tapes, verifying that the corresponding tapes match. Once this is verified, the next move of M is simulated, and the process continues. If at any time an unexpected symbol is encountered, all tapes are erased to a length equal to the number of transitions in the list, and the entire simulation is restarted (note that this restart is never done in a computation from the initial configuration). It is not hard to see that M' is self-stabilizing and simulates M. □

The only entry in Table 1 for which self-stabilization can be forced even in the presence of halting is for finite-state machines; such a simulation simply consists of removing all unreachable configurations. Concerning simulations of one system class by another, Table 2 shows that the only simulation for which we can show that halting interferes with the preservation or forcing of self-stabilization is the simulation of finite-state machines by Turing machines. In particular, we have the following theorem.

Theorem 4.3. *There is no self-stabilization preserving (forcing) simulation of finite-state machines by Turing machines; however, there is a self-stabilization preserving (forcing) simulation of finite-state machines with no halting states by Turing machines.*

Proof. If the tape is removed from machine M in the proof of Theorem 4.1, then this proof shows that there is no self-stabilization preserving (or forcing)

simulation of finite-state machines by Turing machines. On the other hand, let M be an arbitrary finite-state machine with no halting configurations. By adding a storage tape (that is always ignored) to M, we have a Turing machine M' with no halting configurations that simulates M. By Theorem 4.2, there is a self-stabilizing Turing machine M'' that simulates M' (and hence M). To show that there is a self-stabilization preserving simulation of M by a Turing machine, note that if M is self-stabilizing, M'' preserves self-stabilization; otherwise, by adding a new state q to the finite-state control of M'' so that q can never be entered from the outside and can never be left, we have a Turing machine that simulates M and preserves (the absence of) self-stabilization. (Note that since it is decidable whether M is self-stabilizing, this construction is effective; in general, however, constructions need not be effective to show the existence of a simulation, which is simply a function.) □

The fact that there is no self-stabilization preserving simulation of finite-state machines by Turing machines may seem rather odd, since it is normally quite natural to consider a finite-state machine as a special case of a Turing machine in which the tapes are ignored. However, the fact that a Turing machine must have an infinite tape prohibits a finite-state machine from actually *being* a Turing machine. Self-stabilization makes this subtle distinction very important.

Besides Turing machines and finite-state machines, halting seems to affect self-stabilization to some degree on Boolean programs in which communication takes place exclusively via shared variables having exactly one reader and one writer, on communicating finite-state machines, and on Boolean CSP, although at this time we do not know the full extent of these effects. In particular, self-stabilization cannot be forced on either 1 reader/1 writer shared memory programs or communicating finite-state machines if halting is allowed. Furthermore, there is no self-stabilization preserving (forcing) simulation of either Boolean CSP or 1 reader/1 writer shared memory programs by communicating finite-state machines if halting is allowed. However, we do not know whether any of these results hold in the absence of halting. The proofs of all of these results involve isolation, which we discuss in more detail in the next section.

It can be seen from the proofs in this section that when halting affects self-stabilization, it tends to do so in a straightforward manner. In the next two sections, we examine factors interfering with self-stabilization in more subtle ways.

5. Problems involving isolation

In this section, we examine the effects of isolation of self-stabilization. The primary system class we discuss in this section is the class of CSP systems [Hoa78]. We first illustrate the effect of isolation by showing that self-stabilization cannot be forced on CSP systems. By using a similar strategy, we can also show that self-stabilization cannot be preserved by simulations of shared memory programs by CSP systems. We also show that when Boolean CSP systems are simulated by infinite-state CSP systems, self-stabilization can be preserved, but not forced. All of these results hold regardless of whether halting is allowed. By using halting, we can extend these techniques to obtain other results shown in Tables 1 and 2 concerning communicating finite-state machines [BZ83] and shared mem-

ory programs; however, at this time we do not know whether these results hold when halting is not allowed. It might also be noted that a special case of one of Dijkstra's proofs in [Dij73] may actually be viewed as a proof via isolation. (In particular, consider the proof of the impossibility of forcing self-stabilization on the token-passing system with identical machines, and restrict the proof to consider an even number n of machines, $n \geqq 6$, starting with two tokens; the details are left to the reader.)

We now present the class of CSP systems [Hoa78]. Since the formal definition of CSP is quite long, we will give only a short description of CSP processes; for a detailed definition, see [Hoa78]. CSP processes communicate with each other via message passing. The command "$P!a$" is the **send** command, interpreted as "**send** to process P message a". Likewise, the command "$P?x$" is the **receive** command, interpreted as "**receive** from process P a message to be stored in variable x". The communication takes place in a synchronous fashion; i.e., if M_1 sends a message to M_2, neither process may continue until the communication is complete. In terms of our formal definition of a system of concurrent processes, no transition $t_i \in \delta_i$ representing a **send** to M_j may take place unless it enables a transition $t_j \in \delta_j$ representing a **receive** from M_i. After t_i takes place, t_j cannot be disabled until it takes place, and no other transitions from $\delta_i \cup \delta_j$ can occur until t_j takes place. It is also possible to use **receive** commands in the guards of guarded commands. In this case, the value of the **receive** command is true when input is received and false when the other process (i.e., the one from which the message is to be received) has terminated. The guard remains unevaluated until one of these two events occurs. An alternative command in which none of the guards is evaluated is suspended until some guard is evaluated. The syntax of CSP is similar to Dijkstra's guarded command language [Dij75, Dij76]; in particular, "$\rightarrow$" separates guards from commands, "*" denotes repetition of an alternative command until all guards are false, ";" separates sequential statements, and "$\square$" separates nondeterministic choices. The variables in a CSP system are potentially unbounded. A *Boolean CSP system* is a CSP system in which Boolean variables are used instead of unbounded variables. We will now show that self-stabilization cannot be forced on CSP systems.

Theorem 5.1. *There is no self-stabilization forcing simulation of CSP systems by CSP systems.*

Proof. Let M be the following CSP system:

$$
\begin{array}{llll}
M_1 :: [\text{true} \rightarrow [M_2!0; & s_1 & M_2 :: [M_1?a; & t_1 \\
\qquad\qquad *[\text{true} \rightarrow \textbf{skip}]] & s_2 & \quad [a=0 \rightarrow *[\text{true} \rightarrow \textbf{skip}] & t_2 \\
\quad \square \text{true} \rightarrow [M_2!1; & s_3 & \quad \square a=1 \rightarrow *[\text{true} \rightarrow \textbf{skip}]]] & t_3 \\
\qquad\qquad *[\text{true} \rightarrow \textbf{skip}]]] & s_4 & &
\end{array}
$$

Suppose some CSP system $M' = (M'_1, M'_2)$ simulates M, and let h be the simulation homomorphism. Since M' can simulate each of the computations having a prefix $s_1 t_1 s_2^n t_2$, from König's Infinity Lemma [Kon36], M' has an infinite computation σ' such that $h(\sigma') = s_1 t_1 s_2^{\omega}$ in which M'_2 does not terminate. Since $\pi_2(h(\sigma')) = t_1$ is finite, $\pi_2(\sigma')$ must be finite (i.e., M'_2 executes only finitely many transitions, but never reaches termination). Thus, along σ', M'_1 reaches a local configuration C_1, after which M'_1 progresses indefinitely (to simulate s_2^{ω}) without any communication with M'_2. (Note that if M'_2 had terminated, M'_1 would have

been able to detect this via a **receive** command as a guard of a guarded command.) By similar reasoning, M'_2 can reach a local configuration C_2 after which M'_2 progresses indefinitely (to simulate $t_3^{(\omega)}$) without any communication with M'_1. Now consider the computation in M' where M'_1 and M'_2 start at C_1 and C_2, respectively, and each of them progresses infinitely often. This computation simulates a computation in M that has infinitely many s_2's and infinitely many t_3's. Since such a computation can never be executed in M, M' is not self-stabilizing. □

This proof is valid for both infinite-state CSP systems and Boolean CSP systems. Furthermore, the above proof shows that there is no self-stabilization forcing simulation of Boolean CSP systems by infinite-state CSP systems. However, it is easily seen that there is a self-stabilization preserving simulation of Boolean CSP systems by infinite-state CSP systems: given a Boolean CSP system M, we construct a system M' by interleaving the statements of M with statements that force a restart if any variable has a value other than 0 or 1. M' is clearly self-stabilizing iff M is.

The key feature of the above proof technique is that either process may execute arbitrarily many transitions while the other process is executing none. Such a situation is quite common in shared-memory programs – particularly if no **wait** statement is available. We can therefore extend this technique to show that there is no self-stabilization preserving (or forcing) simulation of Boolean programs communicating exclusively through shared variables by Boolean CSP programs; in fact, this result holds even when the shared variables are required to have at most one reader and one writer (see the Appendix, Theorem A.4).

Other results involving isolation include the fact that self-stabilization cannot be forced on 1 reader/1 writer shared memory programs. Also, there is no self-stabilization preserving (forcing) simulation of either Boolean CSP systems or 1 reader/1 writer shared memory programs by communicating finite-state machines. However, we do not know at this time whether any of these results hold in the absence of halting. See the Appendix, Theorems A.3, A.7, and A.8, for proofs of these results.

6. Problems involving look-alike configurations

A system class that illustrates very nicely the problems with look-alike configurations is that of Petri nets [Pet81, Rei85]. The set of configurations Q for a Petri net is the set of nonnegative integer vectors of a specified dimension k. Each transition in a Petri net may be defined by a k-dimensional nonnegative integer vector u and a k-dimensional integer vector v, $u+v \geqq 0$, in the following manner: $t_{u,v}(w) = w+v$ for all $w \geqq u$. This notation for a Petri net closely parallels the vector replacement system notation; see, e.g., [Kel72]. In terms of more conventional definitions of Petri nets (e.g., [Pet81, Rei85]) k is the number of places, the configuration vectors give the number of tokens on each place, the vector u above describes the number of incoming arcs from each place to the transition $t_{u,v}$, and the vector v above describes the net effect of firing $t_{u,v}$. It is easily seen that any infinite computation in the Petri net (N^k, w, δ) is also a computation in (N^k, w', δ) if $w < w'$; hence, if there is an infinite computation

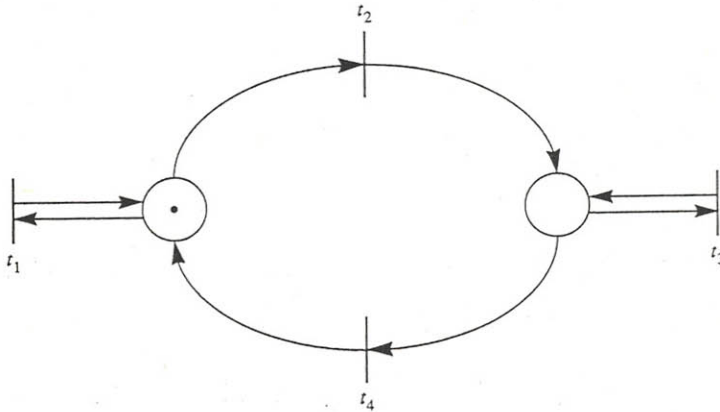

Fig. 2

from w, w and w' are look-alike configurations. The next lemma and its corollary illustrate how look-alike configurations affect self-stabilization in Petri nets.

Lemma 6.1. *If the self-stabilizing Petri net* (N^k, v_0, δ) *has an infinite computation, then for any* $w \in N^k$, *there is a* $w' \geqq w$ *such that* $w' \in R(N^k, v_0, \delta)$.

Proof. Suppose (N^k, v_0, δ) has an infinite computation σ and that there is a $w \in N^k$ such that for all $w' \geqq w$, $w' \notin R(N^k, v_0, \delta)$. Let $w' = v_0 + w$, and consider the computation σ in (N^k, w', δ). The set of vectors reached in the computation from w' is simply the set of vectors reached in the computation from v_0 with w added to each. Hence, each vector reached in the computation from w' is $\geqq w$, and is therefore not in $R(N^k, v_0, \delta)$. Therefore, (N^k, v_0, δ) is not self-stabilizing – a contradiction. □

Corollary 6.1. *If the self-stabilizing Petri net* (N^k, v_0, δ) *has an infinite computation, then there is a* $w \in R(N^k, v_0, \delta)$ *such that for all* $t \in \delta$, $t(w)$ *is defined.*

We can now show that self-stabilization cannot be forced in Petri nets.

Theorem 6.1. *There is no self-stabilization forcing simulation of Petri nets by Petri nets.*

Proof. Consider the Petri net P shown in Fig. 2. Suppose the self-stabilizing Petri net $P' = (N^k, v_0, \delta)$ simulates P. Since all computations of P are infinite, all computations of P' are infinite. Thus, from Corollary 6.1, there is a $w \in R(P')$ such that for all $t \in \delta$, $t(w)$ is defined. Since t_1^ω is a computation of P, there must be some transition $t'_1 \in \delta$ simulating a finite, nonempty sequence of t_1's. Likewise, since $t_2 t_3^\omega$ is a computation of P, there must be some transition $t'_3 \in \delta$ simulating a finite, nonempty sequence of t_3's. Thus, t'_1 and t'_3 are both enabled at $w \in R(P')$. However, there is no reachable configuration of P at which both t_1 and t_3 are enabled. Thus, P' does not simulate P – a contradiction. Therefore, no simulation forces self-stabilization. □

Under the above definition of Petri nets, self-stabilization cannot be forced on the Petri net in Fig. 2. On the other hand, if we allow explicit capacities to be given for the number of tokens on certain places, self-stabilization can be forced on this particular Petri net using the construction given in the proof

of Theorem A.11. However, by using more careful arguments, we can show that Theorem 6.1 holds even for this more general definition. Theorem 6.1 can also be extended in a very natural manner to show that there is no self-stabilization preserving (or forcing) simulation of vector addition systems with states (VASSs) [HP79] by Petri nets. This result is of particular interest because Petri nets and VASSs are often considered equivalent formalisms (see, e.g., [HP79, Pet81]). On the other hand, there is a self-stabilization preserving simulation of VASSs by Petri nets with capacities. Another extension of the techniques given in Theorem 6.1 yields the result that self-stabilization cannot be forced upon either general systems of communicating finite-state machines [BZ83] or systems of communicating finite-state machines whose channel contents never exceed some maximum number of messages. These results are formally shown in the Appendix, Theorem A.5, A.6, A.9, A.10, and A.11.

7. Conclusions

We begin this section by critically examining our definitions, suggesting possible alternatives, and exploring their ramifications on our results. We then consider possible conclusions that could be derived from our results. Finally, we suggest several areas for further research.

The concept of simulation is normally used in a very intuitive sense, its precise meaning depending upon its context. However, in order to formally state that there is no simulation of a class A by a class B satisfying certain requirements, we need to have a formal definition of simulation. There is always a danger associated with formally defining an "intuitive" concept: the definition chosen may not satisfy every usage of the concept. We have tried to make our definition of simulation as weak as possible, that it might capture most, if not all, usages of the concept. Still, there remain some valid criticisms of our definition.

The first criticism we consider concerns the requirement that a simulating computation is infinite iff the simulated computation is infinite. The basis for our decision here was simply a subjective opinion of what is "natural": we do not find it natural to consider that an infinite computation can simulate a finite computation. To others, this type of simulation might seem quite natural. Of course, whether or not this definition is considered natural may depend upon the domain in which it is used. An example of an instance in which dropping our restriction might be useful is in the simulation of cellular arrays by linear cellular arrays. Under our definition, if we try to simulate an arbitrary cellular array with halting states by a linear cellular array, it is not hard to see that an isolation can occur; thus it is not possible to simulate cellular arrays by linear cellular arrays, regardless of whether self-stabilization is to be preserved. By allowing infinite computations to simulate finite computations, this difficulty is avoided. It therefore seems worth considering what happens when we relax our definition in this way. At the very least, this modification invalidates several of our proofs (e.g., the proofs of Theorems 5.1, A.3, and A.4). Precisely how our results are affected, we do not know at this time.

A second criticism is that we do not impose any notion of fairness on the computations we consider. Indeed, it may not seem reasonable to allow the possibility, as in the proof of Theorem 5.1, of a process being continuously enabled, but never executing. Furthermore, a fairness assumption may provide

of Theorem A.11. However, by using more careful arguments, we can show that Theorem 6.1 holds even for this more general definition. Theorem 6.1 can also be extended in a very natural manner to show that there is no self-stabilization preserving (or forcing) simulation of vector addition systems with states (VASSs) [HP79] by Petri nets. This result is of particular interest because Petri nets and VASSs are often considered equivalent formalisms (see, e.g., [HP79, Pet81]). On the other hand, there is a self-stabilization preserving simulation of VASSs by Petri nets with capacities. Another extension of the techniques given in Theorem 6.1 yields the result that self-stabilization cannot be forced upon either general systems of communicating finite-state machines [BZ83] or systems of communicating finite-state machines whose channel contents never exceed some maximum number of messages. These results are formally shown in the Appendix, Theorem A.5, A.6, A.9, A.10, and A.11.

7. Conclusions

We begin this section by critically examining our definitions, suggesting possible alternatives, and exploring their ramifications on our results. We then consider possible conclusions that could be derived from our results. Finally, we suggest several areas for further research.

The concept of simulation is normally used in a very intuitive sense, its precise meaning depending upon its context. However, in order to formally state that there is no simulation of a class A by a class B satisfying certain requirements, we need to have a formal definition of simulation. There is always a danger associated with formally defining an "intuitive" concept: the definition chosen may not satisfy every usage of the concept. We have tried to make our definition of simulation as weak as possible, that it might capture most, if not all, usages of the concept. Still, there remain some valid criticisms of our definition.

The first criticism we consider concerns the requirement that a simulating computation is infinite iff the simulated computation is infinite. The basis for our decision here was simply a subjective opinion of what is "natural": we do not find it natural to consider that an infinite computation can simulate a finite computation. To others, this type of simulation might seem quite natural. Of course, whether or not this definition is considered natural may depend upon the domain in which it is used. An example of an instance in which dropping our restriction might be useful is in the simulation of cellular arrays by linear cellular arrays. Under our definition, if we try to simulate an arbitrary cellular array with halting states by a linear cellular array, it is not hard to see that an isolation can occur; thus it is not possible to simulate cellular arrays by linear cellular arrays, regardless of whether self-stabilization is to be preserved. By allowing infinite computations to simulate finite computations, this difficulty is avoided. It therefore seems worth considering what happens when we relax our definition in this way. At the very least, this modification invalidates several of our proofs (e.g., the proofs of Theorems 5.1, A.3, and A.4). Precisely how our results are affected, we do not know at this time.

A second criticism is that we do not impose any notion of fairness on the computations we consider. Indeed, it may not seem reasonable to allow the possibility, as in the proof of Theorem 5.1, of a process being continuously enabled, but never executing. Furthermore, a fairness assumption may provide

a more satisfying answer to the problem involving finite computations discussed above: by eliminating halting and assuming a strong enough fairness constraint, it might be possible to eliminate finite computations altogether (this would require eliminating all deadlocks). On the other hand, a consideration of fairness raises a number of issues. First of all, there is no consensus on the proper definition of fairness (see, e.g., [LPS81, QS83, Bes84, Fra86, Car87, HRY88]), or even whether fairness should be assumed. Second, assuming a fairness constraint adds a new difficulty to simulations; i.e., it may be difficult to use the fairness constraint in the simulating system to enforce a simulation of exactly the fair computations of the simulated system. Such a scenario occurs, for example, when nondeterministic choices resulting from the interleaving of process computations in the simulated system are simulated by nondeterministic choices within a process in the simulating system (see, e.g., the simulation preceding Theorem A.4). In such a case, ensuring that each enabled process eventually executes may not guarantee that a fair computation is simulated. In view of these difficulties, we felt it more appropriate to first consider what happens in the absence of fairness. There appears to be a great potential for research in this direction.

Perhaps the main conclusion to be derived from our results is that there are many seemingly minor features of the various system classes that interfere with self-stabilization. One might further conclude that self-stabilization is not a robust property. Alternatively, one might conclude that the computational models have oddities that lead to "unnatural" results. Whether either of these last two conclusions is made, we certainly need to be aware of the instability of self-stabilization in conjunction with the simulation paradigm.

As we have already mentioned, there is potential for further research concerning alternative definitions of simulation. Another area of potential research is in considering positive results in conjunction with stronger, more usable definitions of simulation. Finally, there are many computational models that we have not considered in this work that may yield some interesting results. For example, we have not considered I/O automata [LT87]. It is not hard to see that the technique used in the proof of Theorem 5.1 can also be used to show that self-stabilization cannot be forced upon I/O automata (provided fairness is not assumed). Although I/O automata communicate asynchronously in the sense that inputs are always enabled, it is the case that, like CSP, both automata involved in the communication must execute a transition in order for communication to take place. Thus, the same idea as used in Theorem 5.1 can be used to force an isolation that prevents self-stabilization.

A. Appendix

In this Appendix, we present the proofs omitted in the main body of the paper. The first proofs we give are those concerning cellular arrays. Since it is rather awkward to define cellular arrays in the terminology of our formal definition of a concurrent system, after first giving a standard formal definition, we will explicitly show how this standard definition may be translated into our terminology. A *cellular array* is a finite set $\{M_1, \ldots, M_n\}$ of finite-state machines. The machines operate in a synchronous fashion controlled by a clock; i.e., each time the clock fires, all machines that have not yet halted change states nondeterministically according to their respective *next-move relations*. The particular for-

mat of the next-move relation of a given machine depends upon the *topology* associated with the system. The topology is a mapping $g: \{1, \ldots, n\} \to 2^{\{1, \ldots, n\}}$ such that $i \in g(i)$ for $1 \leqq i \leqq n$. Informally, the topology specifies which processes the machine may reference in determining its next state. More formally, let $g(i) = \{i_1, \ldots, i_k\}$, and let the state set of each M_j be given by Q_j. Then the next-move relation δ_i of M_i is a subset of $Q_{i_1} \times \ldots \times Q_{i_k} \times Q_i$. The next-move relation is interpreted to mean that if M_{i_j} is in the state q_{i_j} for $1 \leqq j \leqq k$, M_i may move to state q_i iff $(q_{i_1}, \ldots, q_{i_k}, q_i) \in \delta_i$. There are two types of states in each machine: halting states and nonhalting states. If q_{i_j} is a halting state, then δ_i contains no tuples having q_{i_j} as a component; otherwise, for all $q_{i_1}, \ldots, q_{i_{j-1}}, \ldots, q_{i_k}$, there is at least one q_i such that $(q_{i_1}, \ldots, q_{i_k}, q_i) \in \delta_i$. Thus, in a nonhalting state, there is always some move available, regardless of the states of the other machines.

We will now show how the cellular array $(M_1, \ldots, M_n)$ defined above may be defined as a concurrent system (Q', q'_0, Δ'). The main problem is that in a cellular array, transitions from different machines execute simultaneously. Our concurrent system (Q', q'_0, Δ') will mimic this behavior by executing a "simultaneous" collection of transitions sequentially in the order of their machine subscripts. Since each machine in a cellular array executes a transition at each clock cycle until it halts, such a serial representation will give an unambiguous description of the actual computation. It will also be clear that (Q', q'_0, Δ') is self-stabilizing iff $(M_1, \ldots, M_n)$ is.

Let $(q_1, \ldots, q_n)$ be a configuration of $(M_1, \ldots, M_n)$ (i.e., each M_i is in state q_i). For $1 \leqq i \leqq n$, let $P_i(q_1, \ldots, q_n)$ be the set of machines M_j such that $j < i$ and q_j is not a halting state; i.e., $P_i(q_1, \ldots, q_n)$ will be the set of machines whose moves will be simulated prior to the move of M_i. Let $\delta_i(q_1, \ldots, q_n)$ be the set of transitions in δ_i enabled at $(q_1, \ldots, q_n)$. Let $D_i(q_1, \ldots, q_n) = \delta_{i_1}(q_1, \ldots, q_n) \times \ldots \times \delta_{i_k}(q_1, \ldots, q_n)$ if $P_i(q_1, \ldots, q_n) = \{M_{i_1}, \ldots, M_{i_k}\} \neq \emptyset$, and $D_i(q_1, \ldots, q_n) = \{\emptyset\}$ if $P_i(q_1, \ldots, q_n) = \emptyset$. The set of configurations Q' will be composed of a set NHC of nonhalting configurations and a set HC of halting configurations. We define $NHC = \{(q_1, \ldots, q_n, T) \mid q_j \in Q_j$ for $1 \leqq j \leqq n$, and for some i, q_i is a nonhalting state and $T \in D_i(q_1, \ldots, q_n)\}$. The components $q_1, \ldots, q_n$ give the current states of each of the machines, and T gives a prefix of some sequence of transitions simulating one step of the cellular array. We define $HC = \{(q_1, \ldots, q_n, \emptyset) \mid q_j$ is a halting state of M_j for $1 \leqq j \leqq n\}$, and $Q' = NHC \cup HC$. Let $t \in \delta_i$ for some $1 \leqq i \leqq n$. We define the transition $t' \in \delta'_i$ by

- $t'(q_1, \ldots, q_n, T) = (q_1, \ldots, q_n, T \cup \{t\})$ if $t \notin T$ and $(q_1, \ldots, q_n, T)$, $(q_1, \ldots, q_n, T \cup \{t\}) \in Q'$; and
- $t'(q_1, \ldots, q_n, T) = (p_1, \ldots, p_n, \emptyset)$ if $(q_1, \ldots, q_n, T) \in Q'$, $(q_1, \ldots, q_n, T \cup \{t\}) \notin Q'$, t is enabled in $(q_1, \ldots, q_n)$, and $(M_1, \ldots, M_n)$ reaches $(p_1, \ldots, p_n)$ upon simultaneously executing the transitions in $T \cup \{t\}$.

Thus, if from configuration $(q_1, \ldots, q_n)$ of $(M_1, \ldots, M_n)$ the set of transitions $\{t_{i_1}, \ldots, t_{i_k}\}$ simultaneously fire to produce $(p_1, \ldots, p_n)$, this action is simulated by

$$(q_1, \ldots, q_n, \emptyset) \xrightarrow{t'_{i_1}} (q_1, \ldots, q_n, \{t_{i_1}\}) \xrightarrow{t'_{i_2}} \ldots \xrightarrow{t'_{i_{k-1}}} (q_1, \ldots, q_n, \{t_{i_1}, \ldots, t_{i_{k-1}}\})$$

$$\xrightarrow{t'_{i_k}} (p_1, \ldots, p_n, \emptyset).$$

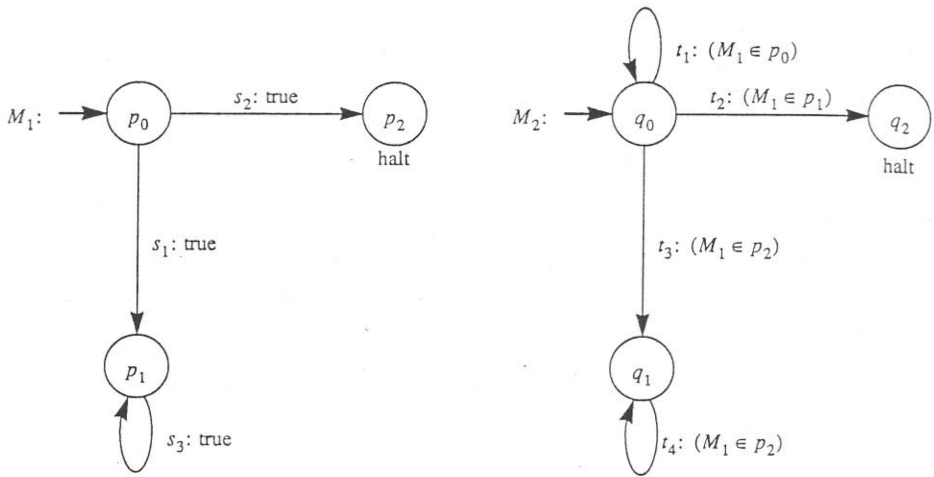

Fig. 3

We now define $\Delta' = \{\delta'_1, \ldots, \delta'_n\}$, where each $\delta'_i = \{t' \mid t \in \delta_i\}$. Finally, we let $q'_0 = (q_1, \ldots, q_n, \emptyset)$, where q_i is the initial state of M_i for $1 \leqq i \leqq n$. It is now a straightforward matter to verify that (Q', q'_0, Δ') exhibits the desired behavior.

A class of cellular arrays of particular interest is the class of linear cellular arrays. A *linear cellular array* is a cellular array $(M_1, \ldots, M_n)$ whose topology g is defined by

- $g(1) = \{1, 2\}$;
- $g(i) = \{i-1, i, i+1\}$ for $2 \leqq i \leqq n-1$; and
- $g(n) = \{n-1, n\}$.

We will first show that self-stabilization cannot, in general, be forced upon either cellular arrays or linear cellular arrays. In order to show the importance of halting to the proof of this theorem, we will then show that self-stabilization can be forced upon both cellular arrays with no halting states and linear cellular arrays with no halting states.

Theorem A.1. *There is no self-stabilization forcing simulation of cellular arrays (linear cellular arrays) by cellular arrays (linear cellular arrays).*

Proof. Let $M = (M_1, M_2)$ be the cellular array shown in Fig. 3. Two computations of M are possible: $\sigma_1 = s_1 t_1 s_3 t_2 s_3^\omega$, and $\sigma_2 = s_2 t_1 t_3 t_4^\omega$. (Recall that the transitions in each of the pairs $s_i t_j$ are executed simultaneously in the actual cellular array.) Let $M' = (M'_1, M'_2)$ be any cellular array that simulates M. Since M' must be able to simulate both σ_1 and σ_2, both M'_1 and M'_2 must have halting states. Thus, M' has a halting configuration, and is therefore not self-stabilizing. Since any cellular array of two machines can be viewed as a linear cellular array, the theorem follows. $\square$

Theorem A2. *There is a self-stabilization forcing simulation of cellular arrays with no halting states by linear cellular arrays.*

Proof. Let $M = (M_1, \ldots, M_n)$ be an arbitrary cellular array. Since each M_i is a finite-state machine, we can construct one finite-state machine A describing

the entire system M; i.e., each state of A is a tuple $(q_1, \ldots, q_n)$, where each q_i is a state of M_i. We may then remove all unreachable states from A. We will now describe a self-stabilizing linear cellular array $M' = (M'_1, \ldots, M'_n)$ that simulates M. All of the work will actually be done by the machine M'_1. This machine will select an infinite sequence of transitions from A representing a computation of M. After each transition t of A is selected, M'_1 will wait long enough for all the other machines to determine that t has been selected. At this time, all the machines simulate t. M'_1 then selects the next transition, and the simulation continues in the same manner.

More formally, let Q_A be the state set of A, and let δ_A be the transition relation of A. The state set of M_i, $1 \leqq i \leqq n$, will be $\{(q, i) \mid q \in Q_A\} \cup \{(t, j, i) \mid t \in \delta_A$ and $i \leqq j \leqq n\}$. The next-move relation of M'_1 is defined solely in terms of the current state of M'_1. If M'_1 is in state $(q, 1)$, it may move to any state $(t, 1, 1)$ such that t is enabled at q in A. If M'_1 is in any state $(t, j-1, 1)$, $2 \leqq j \leqq n$, it moves to state $(t, j, 1)$. If M'_1 is in any state $(t, n, 1)$, it moves to the state $(q', 1)$ such that transition t places A in state q'; moves of this last type will simulate of M_1. For $2 \leqq i \leqq n$, the next-move relation of M'_i is defined solely in terms of the state of M'_{i-1}. If M'_{i-1} is in some state $(q, i-1)$, M'_i moves to state (q, i). If M'_{i-1} is in some state $(t, j-1, i-1)$, $i \leqq j \leqq n$, M'_i moves to state (t, j, i). If M'_{i-1} is in some state $(t, n, i-1)$, M'_i moves to the state (q', i) such that transition t places A in state q'; moves of this last type will simulate moves of M_i. The initial state of M' has each M'_i in the state (q, i) such that q is the initial state of A. It is not hard to see that in any computation from the initial state, M' simulates the next move of M every $n+1$ moves; thus, M' simulates M. Furthermore, it is not too difficult to see that from any configuration, after at most $i-1$ moves, M'_i is in some state consistent with M'_1. Since all states of M'_1 are clearly reachable from the initial configuration, M' must be self-stabilizing. □

The above simulation is not very satisfying because the entire computation is actually being done by one machine, M'_1; the machines $M'_2, \ldots, M'_n$ simply execute the transitions that M'_1 tells them to execute.

The following corollaries follow immediately from Theorem A.2.

Corollary A.1. *There is a self-stabilization forcing simulation of cellular arrays without halting states by cellular arrays.*

Corollary A.2. *There is a self-stabilization forcing simulation of linear cellular arrays without halting states by linear cellular arrays.*

We now introduce another class of concurrent systems, the class of Boolean programs in which each variable may be read by at most one process and written by at most one process. All communication is therefore performed via shared variables to which one process writes and from which another process reads. The syntax we use to describe these systems is similar to CSP without the communication commands (see [Hoa78]). We will now show that self-stabilization cannot be forced on this type of system. The proof will use both halting and isolation.

Theorem A.3. *There is no self-stabilization forcing simulation of* 1 *reader*/1 *writer shared memory programs by* 1 *reader*/1 *writer shared memory programs.*

Proof. Consider the following system $M=(M_1, M_2)$, where all variables are initially zero:[4]

$$
\begin{array}{lll}
M_1 :: [\text{true} \rightarrow a:=1 & s_1 & M_2 :: [*[a=0 \wedge b=0 \rightarrow \textbf{skip}];\ t_1 \\
\quad \square \text{true} \rightarrow [b:=1; & s_2 & \quad a=1 \rightarrow *[\text{true} \rightarrow \textbf{skip}]] \quad t_2 \\
\qquad *[\text{true} \rightarrow \textbf{skip}]]] & s_3 &
\end{array}
$$

The only variables in M, a and b, are read by M_2 and written by M_1. M_1 first nondeterministically chooses either s_1 or s_2. If it chooses s_1, it writes 1 to a and halts. If it chooses s_2, it writes 1 to b and repeatedly executes s_3. Meanwhile, M_2 repeatedly executes t_1 until M_1 executes its first transition. If M_1's first transition is s_1, M_2 then repeatedly executes t_2. Otherwise, M_2 halts. Thus, all computations of M are infinite, but in any computation, one of the processes executes only finitely many times. Therefore, in any system $M'=(M'_1, M'_2)$ that simulates M, both M'_1 and M'_2 must have the ability to halt. Since all variables can be read by at most one process, there must be a configuration of M' in which both M'_1 and M'_2 have halted. Thus, M' is not self-stabilizing. □

We will now show that there is no self-stabilization preserving simulation of shared memory programs by Boolean CSP systems. It is not immediately clear, however, that there is *any* simulation between these system classes, whether or not self-stabilization is preserved. Hence, we first sketch an example of a simulation that does not preserve self-stabilization. Let $M=(M_1, \ldots, M_n)$ be an arbitrary system of 1 reader/1 writer shared memory programs. $M' =(M'_1, \ldots, M'_n)$ will be a Boolean CSP system that behaves as follows. M'_1 first creates a status vector containing the values of all shared variables of M and all processes which have not halted. It then nondeterministically decides which process M_i will execute first. If $i \neq 1$, M'_1 sends the status vector to M'_i. From this point on, there will be exactly one "active" process at any given time when no communication is taking place; the other processes will be waiting for messages from all other processes. The active process, say M'_i, will nondeterministically simulate a nonempty (but possibly infinite) sequence of transitions from M_i, updating the status vector accordingly. If it has chosen to simulate a finite sequence of transitions, it then nondeterministically selects some process from the list of processes which have not halted. It then sends the updated status vector to that process, which then becomes active. If at any time a process M'_i simulates the termination of M_i, it removes its own name from the list of processes which have not halted, sends the status vector to some process that has not halted, then halts. If at any time the list of processes that have not halted contains only one process, that process remains active until it simulates a termination. The details of the simulation are left to the reader. The following theorem, shown using isolation, now shows that neither this simulation nor any other simulation can be guaranteed to either preserve or force self-stabilization.

Theorem A.4. *There is no self-stabilization preserving (forcing) simulation of* 1 *reader*/1 *writer shared memory programs by Boolean CSP systems.*

[4] Throughout the remainder of the paper, we will always assume the variables to be initially zero.

Proof. Let $M=(M_1, M_2)$ be the following shared memory system:

$$M_1 :: \ *[a=0 \rightarrow b:=1 \quad s_1 \qquad M_2 :: \ *[b=0 \rightarrow a:=0 \quad t_1$$
$$\square a=1 \rightarrow b:=0] \quad s_2 \qquad \square b=1 \rightarrow a:=1] \quad t_2$$

M has no halting configurations, and M is clearly self-stabilizing, since all configurations are reachable. Suppose some CSP system $M'=(M'_1, M'_2)$ simulates M with simulation homomorphism h. Let $S=\{\sigma' \mid \sigma'$ is a finite prefix of some computation $\sigma'\sigma''$ of M', M'_2 has not terminated in σ', and $h(\sigma')\in s_1^*\}$. Since M can execute any computation beginning with $(s_1)^n t_2$ for any n, S is infinite. Hence, from König's Infinity Lemma [Kon36], there is an infinite string σ' such that any finite prefix of σ' is in S. Clearly, σ' must be a computation of M' in which M'_2 does not terminate. However, since $h(\sigma')$ contains no transitions from M_2, σ' must contain only finitely many transitions from M'_2. Thus, $\pi_1(\sigma')$ and $h(\pi_1(\sigma'))$ are both infinite. It must therefore be the case that $h(\sigma')=s_1^\omega$. Hence, there is some state of M'_1 from which there is an infinite computation simulating s_1^ω and containing no communication commands. By similar reasoning, there is a state of M'_2 from which there is an infinite computation simulating t_1^ω and containing no communication commands. There must therefore be a configuration of M' from which a computation containing infinitely many s_1's and infinitely many t_1's, but no s_2's nor t_2's, can be simulated. Since such a computation can never be executed in M, M' is not self-stabilizing. □

Note that the above theorem holds even when halting states are disallowed.

The next class we examine is that of communicating finite-state machines (CFSMs) [BZ83]. Informally, a system of CFSMs is a finite set of finite-state machines that communicate via unbounded FIFO channels. No empty channel detection is possible, and either a **read** or a **write** to some channel is performed by each transition. (See, e.g., [BZ83] for a formal definition of CFSMs.) We will first show that self-stabilization cannot be forced on systems of CFSMs. The proof uses both halting and look-alike configurations.

Theorem A.5. *There is no self-stabilization forcing simulation of CFSMs by CFSMs.*

Proof. Let $M=(M_1, M_2)$ be the system of two CFSMs defined as follows. M_1 contains only one state and no transitions. M_2 contains only one state and one transition, which writes some symbol to the output channel of M_2. Suppose there is a self-stabilizing system $M'=(M'_1, M'_2)$ of CFSMs that simulates M. Then for any computation σ' of M', $\pi_1(\sigma')$ is finite and $\pi_2(\sigma')$ is finite. Since M'_2 is finite-state and may only read finitely many symbols from its input channel, in any computation σ', M'_2 must enter the same state twice without executing any **reads** in between. Thus, M'_2 has a state q from which there is an infinite computation containing no **reads**. It follows from König's Infinity Lemma [Kon36] that the input channel to M'_2 is bounded, say, by m. If we therefore start M'_1 in some arbitrary state, M'_2 in q, and the input channel to M'_2 with some string longer than m symbols, there is an infinite computation in which the contents of the input channel to M'_2 are unchanged. Thus, M' is not self-stabilizing – a contradiction. □

Note that in the above proof both halting and isolation are involved. At this time, we are unable to show whether the elimination of halting states might allow self-stabilization to be forced.

A system of CFSMs is said to be *bounded* if its reachability set is finite. Consider any bounded system of CFSMs having an infinite computation σ. Let q be some system configuration that is reached more than once by σ, and let α be the string written to some channel c between the first two occurrences of q. Let q' be q modified by appending α to the contents of channel c enough times so that q' is not reachable. It is not hard to see that there is an infinite computation σ' from q' in which q' is reached infinitely often. Therefore, no bounded system of CFSMs having an infinite computation is self-stabilizing. This fact implies the following theorem.

Theorem A.6. *There is no self-stabilization forcing simulation of bounded CFSMs by bounded CFSMs.*

We now show that there is no self-stabilization preserving simulation of Boolean CSP systems by CFSMs.

Theorem A.7. *There is no self-stabilization preserving (forcing) simulation of Boolean CSP systems by CFSMs.*

Proof. Consider the following CSP system M:

$$M_1 :: [\textbf{skip}] \qquad M_2 :: *[\text{true} \rightarrow \textbf{skip}]$$

M is clearly self-stabilizing. Furthermore, M simulates the system of CFSMs given in the proof of Theorem A.5. Thus, if there were a simulation of M by a system of CFSMs, we would have a simulation of the system of CFSMs given in the proof of Theorem A.5 by CFSMs – a simulation we have already shown does not exist. □

The same technique may be used to show the following theorem.

Theorem A.8. *There is no self-stabilization preserving (forcing) simulation of* 1 *reader*/1 *writer shared memory programs by CFSMs.*

We now examine the class of vector addition systems with states (VASSs). The set of configurations of a VASS is of the form $Q \times N^k$, where Q is a finite set of machine states. The transitions are defined in terms of two machine states, q and q', and a vector $v \in N^k$ in the following manner: $t_{q,q',v}(q, w) = (q', w+v)$ for $w+v \geqq 0$. Thus, a VASS may be viewed as a Petri net augmented with a finite-state control (see, e.g., [HP79]). There is therefore a straightforward simulation of Petri nets by VASSs, and Hopcroft and Pansiot [HP79] have shown how to simulate a k-dimensional VASS by a $k+3$-dimensional Petri net. However, we show in the next theorem that in general there is no simulation of a VASS by a Petri net that preserves self-stabilization. This proof uses look-alike configurations.

Theorem A.9. *There is no self-stabilization forcing (preserving) simulation of VASSs by Petri nets.*

Proof. Consider the VASS M shown in Fig. 4. Since every configuration of M is reachable, M is self-stabilizing. Furthermore, M simulates the Petri net in Fig. 2, which we have already shown in Theorem 6.1 cannot be simulated by a self-stabilizing Petri net. □

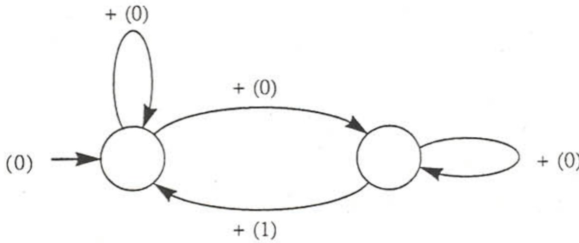

Fig. 4

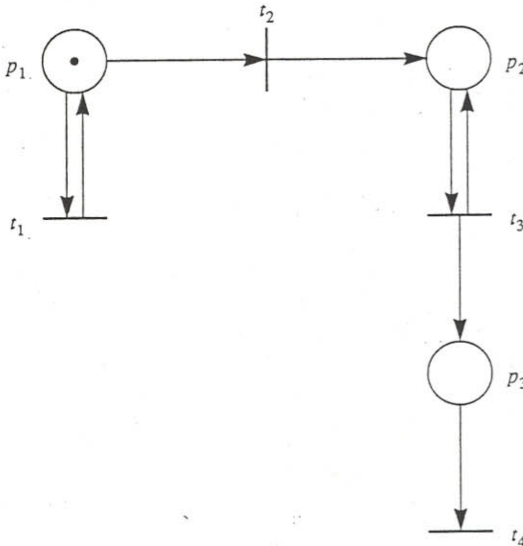

Fig. 5

Finally, we examine a slightly more general class of Petri nets. We define a *Petri net with capacities* in the same way as a Petri net with the exception that the set of configurations may be restricted so that certain vector coordinates (i.e., places) may not exceed specified bounds. In what follows, we first use look-alike configurations to show that self-stabilization cannot be forced upon Petri nets with capacities; we then give a self-stabilization-preserving simulation of VASSs by Petri nets with capacities.

Theorem A.10. *There is no self-stabilization forcing simulation of Petri nets with capacities by Petri nets with capacities.*

Proof. Let M be the Petri net shown in Fig. 5. M behaves as follows. First, t_1 may fire arbitrarily many times (possibly infinitely many times). At any time t_2 may fire once, permanently disabling t_1. After t_2 fires, M executes an infinite computation containing only t_3's and t_4's; however, t_4 may never have fired more times than t_3. Consider the marking v in which $p_1 = 1$, $p_2 = 0$, and $p_3 = 1$.

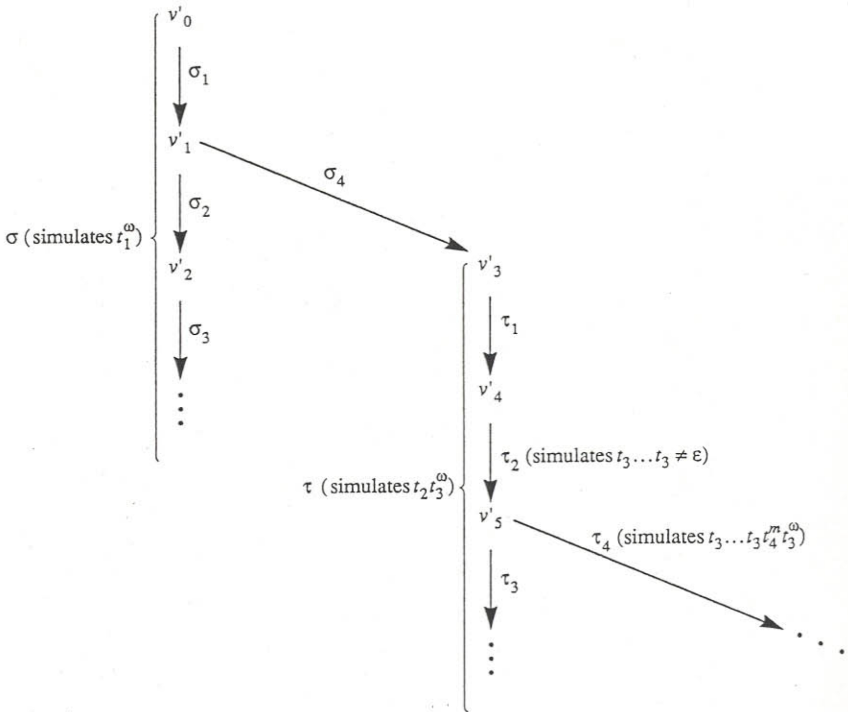

Fig. 6

(Note that v and v_0 are look-alike configurations). Clearly, v is not reachable. However, the computation $\sigma = t_1^\omega$ from v enters v infinitely often; therefore, M is not self-stabilizing. In the execution of σ from v, infinitely often some sequence of transitions $t_2\, t_3^k\, t_4^m$, $k<m$, is enabled. Since such a sequence is never enabled in any computation of M from its initial marking v_0, this fact is enough to show that M is not self-stabilizing. We will now show that such a phenomenon occurs in any Petri net with capacities that simulates M.

Let $M' = (Q', v'_0, \delta')$ be a Petri net with capacities that simulates M, and let h be the simulation homomorphism. Let S be the following set of suffixes of computations of M: $S = \{t_2\, t_3^i\, t_4^i\, t_3^\omega \mid i > 0\}$. For each $i \geqq 0$, there are only finitely many sequences σ of transitions from v'_0 such that $h(\sigma) = t_1^i$; otherwise, from König's Infinity Lemma, M' would have an infinite computation simulating t_1^j for some $j \geqq 0$. Hence, for each $i \geqq 0$, there is a σ_i such that

1. $v'_0 \xrightarrow{\sigma_i} w_i$ in M';
2. $h(\sigma_i) = t_1^i$; and
3. infinitely many computations in S can be simulated from w_i.

From König's Infinity Lemma, there is a computation σ of M' such that $h(\sigma) = t_1^\omega$ and for any finite prefix σ_1 of σ, there exist σ_2 and w such that $v'_0 \xrightarrow{\sigma_1 \sigma_2} w$ and infinitely many computations in S can be simulated from w. From [KM69], $\sigma = \sigma_1\, \sigma_2\, \sigma_3$ where $v'_0 \xrightarrow{\sigma_1} v'_1 \xrightarrow{\sigma_2} v'_2$, $\sigma_2 \neq \varepsilon$, $v'_1 \leqq v'_2$, and on all bounded coordinates, v'_1 and v'_2 are equal (see Fig. 6).

Let σ_4 be such that $v'_0 \xrightarrow{\sigma_1 \sigma_4} v'_3$ and infinitely many computations in S can be simulated from v'_3. Let S' be the infinite subset of S that can be simulated from v'_3. From König's Infinity Lemma, there is a τ such that $h(\tau) \in t_2 t_3^\omega$ and any finite prefix of τ simulates a prefix of some computation in S'. From [KM69], $\tau = \tau_1 \tau_2 \tau_3$ such that $v'_3 \xrightarrow{\tau_1} v'_4 \xrightarrow{\tau_2} v'_5$, $\tau_2 \neq \varepsilon$, $v'_4 \leqq v'_5$, and in all bounded coordinates, v'_4 and v'_5 are equal. Let m be such that $h(\tau_1 \tau_2 \tau_4) = t_2 t_3^m t_4^m t_3^\omega$ for some computation τ_4 from v'_5. Clearly, $h(\tau_2) \neq \varepsilon$ (otherwise $h(\sigma_1 \sigma_4 \tau_1 \tau_2^\omega)$ is finite) and $t_2 \notin h(\tau_2)$; thus, $h(\tau_2) \in t_3^+$.

Let $v' = v'_0 + v'_5 - v'_4$, $\sigma' = \sigma_1 \sigma_2^\omega$, and $\tau' = \sigma_4 \tau_1 \tau_4$. We claim that σ' is a computation from v' and that infinitely often in σ', τ' is enabled. To see this, first note that since $v'_5 \geqq v'_4$ and v'_5 and v'_4 are equal on all bounded coordinates, any infinite computation from v'_0 is also a computation from v' (v'_0 and v' are look-alike configurations). Clearly, $\sigma_1 \sigma_2^\omega$ is a computation from v'_0, and hence from v'. Let $v' \xrightarrow{\sigma_1 \sigma_2^i} w_i$, and let $x = v'_5 - v'_4$ and $y = v'_2 - v'_1$. (Note that both x and y are nonnegative and are zero in all bounded coordinates of M'.) Since $w_i = v'_1 + x + iy$, $w_i \xrightarrow{\sigma_4} v'_3 + x + iy \xrightarrow{\tau_1} v'_4 + x + iy = v'_5 + iy$, and τ_4 is a computation from $v'_5 + iy$. But $h(\tau') = t_1^j t_2 t_3^k t_4^m t_3^\omega$ where $k < m$ – a suffix that can never be executed in M. Hence, w_i is unreachable for all $i > 0$, so from v', σ' never reaches a reachable configuration. Therefore, M' is not self-stabilizing. □

Since Petri nets with capacities can clearly be simulated by VASSs, we have the following corollary.

Corollary A.3. *There is no self-stabilization forcing simulation of VASSs by Petri nets with capacities.*

We conclude by showing that there is a self-stabilization preserving simulation of VASSs by Petri nets with capacities. The obvious strategy to use in trying to prove this theorem is to use 1-bounded places to represent each of the states in the VASS. The difficulty in this approach is dealing with the extra configurations introduced while not allowing transitions to be enabled at the wrong times. However, we are able to overcome this difficulty in the proof that follows.

Theorem A.11. *There is a self-stabilization preserving simulation of VASSs by Petri nets with capacities.*

Proof. For ease of explanation, we will only give a self-stabilization preserving simulation of a finite-state machine by a Petri net with capacities. It should be clear how to extend this simulation to VASSs. Let M be an arbitrary finite-state machine. Without loss of generality, assume that each transition of M changes the state of M; otherwise, we can clearly add states as necessary to enforce this condition without affecting self-stabilization. Let the state set of M be $Q = \{q_1, \ldots, q_n\}$. We construct a Petri net M' with n 1-bounded coordinates as follows. We represent each state q_i of M by a vector v_i in which coordinate i is 0 and all other coordinates are 1. We simulate a transition t from state q_i to state q_j by transition t' shown in Fig. 7. (Note that t' is not enabled in any other configuration.) Clearly, the Petri net so constructed simulates M. We must now deal with the configurations of M' that do not correspond to

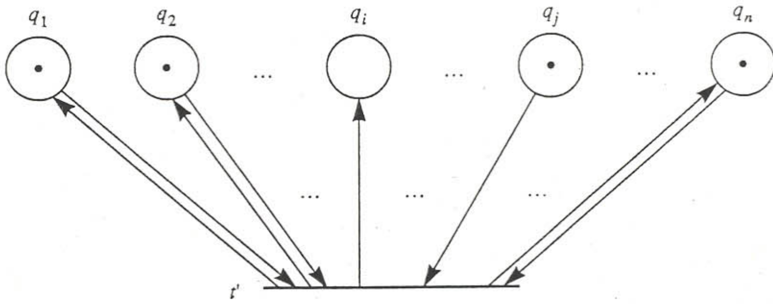

Fig. 7

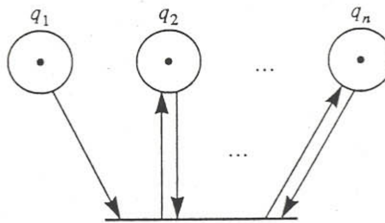

Fig. 8

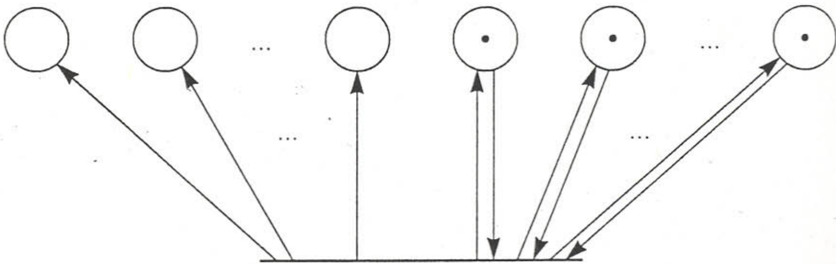

Fig. 9

any state of M; i.e., those configurations in which the number of coordinates having a value of 0 is not exactly one. The first of these configurations is the vector of all 1*s*. We introduce a new transition, only enabled at this vector, which brings M' to the configuration representing q_1 (see Fig. 8). Finally, we can bring all other vectors to the vector of all 1s as shown in Fig. 9 (note again that each of these transitions is only enabled at one vector). It should be clear that M' is self-stabilizing iff M is self-stabilizing. □

The simulation given in the above proof is much better than most of the other simulations we have given in this paper. First of all, it is a real-time simulation. Also, if we have some bound on the number of moves needed for

the VASS to stabilize, the bound for the Petri net with capacities is only two greater. Unfortunately, the size of the description is exponential in the size of the description of the VASS due to the large number of transitions from "bad" configurations.

Acknowledgment. We would like to thank the various referees, as well as several participants of *MFCS '89* and the *MCC Workshop on Self-Stabilizing Systems*, for many helpful comments which improved the presentation of these results.

References

[Bes84] Best, E.: Fairness and conspiracies. Inf. Process. Lett. **18**, 215–220 (1984); Addendum **19**, 162 (1984)

[BGW89] Brown, G., Gouda, M., Wu, C.: Token systems that self-stabilize. IEEE Trans. Comput. **38**, 845–852 (1989)

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

[BYC88] Bastani, F., Yen, I., Chen, I.: A class of inherently fault tolerant distributed programs. IEEE Trans. Software Eng. **14**, 1432–1442 (1988)

[BZ83] Brand, D., Zafiropulo, P.: On communicating finite-state machines. JACM **30**, 323–342 (1983)

[Car87] Carstensen, H.: Decidability questions for fairness in Petri nets. In: Proceedings of the 4th Symposium on Theoretical Aspects of Computer Science (Lect. Notes Comput. Sci., Vol. 247, pp. 396–407). Berlin Heidelberg New York: Springer 1987

[CK80] Cheney, W., Kincaid, D.: Numerical Mathematics and Computing. Monterey, CA: Brooks/Cole 1980

[Dij73] Dijkstra, E.: EWD391 Self-stabilization in spite of distributed control. 1973. In: Dijkstra, E. (ed.). Selected writings on computing: A personal perspective, pp. 41–46. Berlin Heidelberg New York: Springer 1982

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

[Dij75] Dijkstra, E.: Guarded commands, nondeterminacy and the formal derivation of programs. Commun. ACM **18**, 453–457 (1975)

[Dij76] Dijkstra, E.: A Discipline of Programming. Englewood Cliffs.: Prentice Hall 1976

[EL87] Emerson, E., Lei, C.: Modalities for model checking: Branching time logic strikes back. Sci. Comput. Programm. **8**, 275–306 (1987)

[Fra86] Francez, N.: Fairness. Berlin Heidelberg New York: Springer 1986

[Gou90] Gouda, M.: The Stabilizing Philosopher: Asymmetry by Memory and by Action. To appear in Science of Computer Programming, 1990

[Hoa78] Hoare, C.: Communicating sequential processes. Commun. ACM **21**, 666–677 (1978)

[HP79] Hopcroft, J., Pansiot, J.: On the reachability problem for 5-dimensional vector addition systems. Theoret. Comp. Sci. **8**, 135–159 (1979)

[HRY88] Howell, R., Rosier, L., Yen, H.: A taxonomy of fairness and temporal logic problems for Petri nets. In Proceedings of the 19th Symposium on Mathematical Foundations of Computer Science, pp. 351–359 (1988) (to appear in Theoret. Comp. Sci.)

[HU79] Hopcroft, J., Ullman, J.: Introduction to automata theory, languages, and computation. Reading, MA: Addison-Wesley 1979

[IKM85] Ibarra, O., Kim, S., Moran, S.: Sequential machine characterizations of trellis and cellular automata and applications. SIAM J. Comput. **14**, 426–447 (1985)

[Kel72] Keller, R.M.: Vector Replacement Systems: A Formalism for Modelling Asynchronous Systems. TR 117, Princeton University, CSL, 1972

[KM69] Karp, R., Miller, R.: Parallel program schemata. J. Comput. System Sci. **3**, 147–195 (1969)

[Kon36] König, D.: Theorie der Endlichen und Unendlichen Graphen. Leipzig: Akademische Verlagsgesellschaft 1936

[Kos74] Kosaraju, S.: On some open problems in the theory of cellular automata. IEEE Trans. Comput. **C-23**, 561–565 (1974)

[Lam86] Lamport, L.: The mutual exclusion problem: Part II – Statement and solutions. JACM **33**, 327–348 (1986)

[LPS81] Lehman, D., Pnueli, A., Stavi, J.: Impartiality, justice, and fairness: the ethics of concurrent termination. In Proceedings of the 8th International Colloquium on Automata, Languages, and Programming. (Lect. Notes Comput. Sci., vol. 115, pp. 264–277) Berlin Heidelberg New York: Springer 1981

[LR81] Lehman, D., Rabin, M.: 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, pp. 133–138, 1981

[LT87] Lynch, N., Tuttle, M.: Hierarchical correctness proofs for distributed algorithms. In: Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing, pp. 137–151, 1987

[Mul89] Multari, N.: Self-stabilizing Protocols. PhD thesis, Dept. of Computer Sciences, University of Texas at Austin 1989

[OL82] Owicki, S., Lamport, L.: Proving liveness properties of concurrent programs. ACM Trans. Programm. Languages Syst. **4**, 455–495 (1982)

[OWA89] Özveren, C., Willsky, A., Antsaklis, P.: Stability and stabilizability of discrete event dynamic systems. MIT LIDS Publication, LIDS-P-1853, 1989

[Pet81] Peterson, J.: Petri Net Theory and the Modeling of Systems. Englewood Cliffs, NJ: Prentice Hall 1981

[QS83] Queille, J., Sifakis, J.: Fairness and related properties in transition systems – a temporal logic to deal with fairness. Acta Inf. **19**, 195–220 (1983)

[Rei85] Reisig, W.: Petri Nets: An Introduction. Berlin Heidelberg New York: Springer 1985

[Smi71] Smith, A.: Cellular automata complexity tradeoffs. Inform. Control **18**, 466–482 (1971)