STABILIZATION AND PSEUDO-STABILIZATION

James E. Burns,[1] Mohamed G. Gouda,[2]
and Raymond E. Miller[3]

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712–1188

TR-90-13 May 1990
November 1990 (Revision)

Abstract

A *stabilizing* system is one which if started at any state is guaranteed to reach a state after which the system cannot deviate from its intended specification. In this paper, we propose a new variation of this notion, called pseudo-stabilization. A *pseudo-stabilizing* system is one which if started at any state is guaranteed to reach a state after which the system does not deviate from its intended specification. Thus, the difference between the two notions comes down to the difference between "cannot" and "does not"—a difference that hardly matters in many practical situations. As it happens, a number of well-known systems, for example the alternating-bit protocol, are pseudo-stabilizing but not stabilizing. We conclude that one should not try to make any such system stabilizing, especially if stabilization comes at a high price.

Keywords: alternating-bit protocol, communication protocols, convergence, self-stabilization, system specification.

[1]Georgia Institute of Technology.
[2]University of Texas at Austin.
[3]University of Maryland at College Park.

1 Introduction

We have been interested for some time now in classes of systems that "well-behave" irrespective of their initial states. Clearly, such systems are extremely robust in the face of transient faults which may yield them in arbitrary states; this should explain our interest in these systems. Our aim in this paper is to investigate the criteria for well-behaving in the context of arbitrary initialization.

The first criterion for the notion of well-behaving in spite of arbitrary initialization can be traced back to Dijkstra's seminal paper on self-stabilization [5, 6]. Dijkstra's criterion requires that if the system starts at an arbitrary state, then it is guaranteed to reach, within a finite number of transitions, a state after which the system *cannot* deviate from its intended specification. Many authors have adopted Dijkstra's criterion since then; see for example [2, 3, 4, 7, 9, 10, 11]. We refer to systems that satisfy this criterion as stabilizing systems.

In this paper, we propose a new criterion for the notion of well-behaving in spite of arbitrary initialization. Our criterion requires that if the system starts at an arbitrary state, then it is guaranteed to reach, within a finite number of transitions, a state after which the system *does not* deviate from its intended specification. We call systems that satisfy this criterion pseudo-stabilizing.

The "fragile" distinction between stabilization and pseudo-stabilization is better illustrated by an example. Consider a system with a state-transition diagram as shown in Figure 1. (In this diagram, circles represent system states and arcs represent transitions between the states in the usual way.) Starting from any state, this system is guaranteed to reach, within one transition, either state p or state q. From p, only one computation $(p, p, \ldots)$ can be executed, and from q, only one computation $(q, q, \ldots)$ can be executed. Thus, if the intended specification of this system is $F = \{(p, p, \ldots), (q, q, \ldots)\}$, then the system will reach within one transition a state (p or q) after which no deviation from F is possible. Hence, the system is stabilizing.

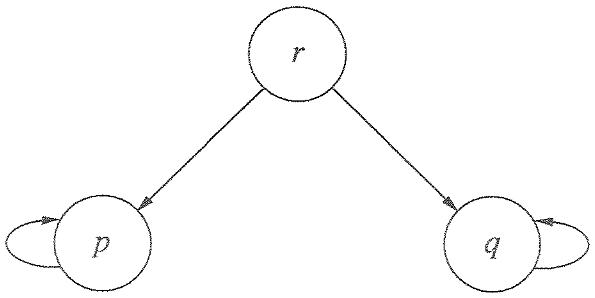

Figure 1. A Stabilizing System.

Now, consider a second system with a state-transition diagram as shown in Figure 2, and assume that the intended specification of this system is the same F as before. This system is not stabilizing because starting at state p, there is no guarantee that the system will ever leave p, yet at p the system can deviate from F by executing the computation $(p, q, q, \ldots)$ which is not in F. On the other hand, every computation of this system is in one of the following forms:

$$\begin{aligned}&(p, p, \ldots),\\ &(p, \ldots, p, q, q, \ldots)\\ &(q, q, \ldots).\end{aligned}$$

Thus, every computation has an infinite suffix in F. In other words, along every computation the system is guaranteed to reach a state after which the system does not deviate from F (although it may have an infinite number of chances to do so). Hence the system is pseudo-stabilizing.

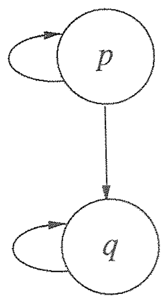

Figure 2. A Pseudo-Stabilizing System.

As we show later, pseudo-stabilization is strictly weaker than stabilization, with "does not deviate" replacing "cannot deviate". Thus, every stabilizing system is also pseudo-stabilizing but there are pseudo-stabilizing systems that are not stabilizing. We contend, however, that pseudo-stabilizing systems are adequate for most practical purposes, and there is no real need to make them stabilizing, especially if the price of stabilization is high.

The rest of this paper is organized as follows. In Section 2, we state our definitions of systems and their regions of execution, and identify two types of regions called "attractors" and "pseudo-attractors". In Section 3, we define what it means for a system to be stabilizing or pseudo-stabilizing to a given specification. In Section 4, we discuss an important relationship between (pseudo-) attraction and (pseudo-) stabilization. In particular, we prove that if a system has a (pseudo-) attractor where some specification is achieved, then the system (pseudo-) stabilizes to that specification. This result gives a sufficient condition for establishing that a given system stabilizes or pseudo-stabilizes to a given specification. We use this result, in Section 5, to show that the Alternating-Bit protocol pseudo-stabilizes to its specification. Concluding remarks are in Section 6. For convenience, our discussion of the Alternating-Bit protocol is divided into several subsections. An informal presentation of the protocol is given in 5.1, followed by a formal presentation in 5.2. A specification of the protocol is presented in 5.3, and a proof that the protocol pseudo-stabilizes to that specification is outlined in 5.4. Up to this point, our protocol can only tolerate message loss. Thus, we extend the protocol to tolerate message corruption in 5.5, and tolerate message reorder in 5.6, while preserving its property of pseudo-stabilization.

2 Attractors and Pseudo-attractors

Let S be a *system* defined by a set of *states*, and a set of *transitions*, where each transition is an ordered pair of states.

A *region* of system S is a subset of the system states. A region P is *closed* iff for every state p and q, if p is in P and (p, q) is a transition then q is in P.

A *computation* of S is a nonempty maximal sequence $(p_1, p_2, \ldots)$, where each p_i is a system state and each (p_i, p_{i+1}) is a system transition. The maximality condition implies that if the last state in a finite computation is p then for every system state q, (p, q) is not a system transition.

A computation *leads to* a region P iff it has a state in P.

Definition 1: A region P of S is an *attractor* iff it satisfies the following two conditions.

i. P is closed.

ii. Every computation of S leads to P. □

Thus, each system computation eventually reaches every system attractor. Moreover, once a computation reaches an attractor, it stays there indefinitely.

Definition 2: An infinite sequence $(P_1, P_2, \ldots)$ of regions of S is a *pseudo-attractor* iff it satisfies the following two conditions.

i. For every $k, k = 1, 2, \ldots$, the region $(\cup_{i=1}^{k} P_i)$ is closed.

ii. Every computation of S leads to the region $(\cup_{i=1}^{\infty} P_i)$. □

Let $(P_1, P_2, \ldots)$ be a pseudo-attractor of some system. Thus, for every system computation, there is $k, k = 1, 2, \ldots,$ such that the computation eventually reaches a region P_k but never reaches any of the preceding regions $P_1, \ldots, P_{k-1}$. In this case, the computation stays within P_k indefinitely (since $\cup_{i=1}^{k} P_i$ is closed).

The next theorem, which follows directly from the above definitions, states an interesting relationship between the attractors and pseudo-attractors of a system.

Theorem 1: Let P be a region of S and $(P_1, P_2, \ldots)$ be an infinite sequence of regions of S.

i. If P is an attractor of S,
then $(P, P, \ldots)$ is a pseudo-attractor of S.

ii. If $(P_1, P_2, \ldots)$ is a pseudo-attractor of S,
then $(\cup_{i=1}^{\infty} P_i)$ is an attractor of S. □

3 Stabilization and Pseudo-stabilization

A *specification* of a system S is a set of computations of S.

Next, we define what it means for a system to be stabilizing or pseudo-stabilizing to a given specification; but first we adopt the following notation.

Notation: Let c be any computation that has at least i states. Then, $c.i$ denotes the i^{th} state in c, and $c \uparrow i$ denotes the suffix of c starting with the i^{th} state $c.i$. □

Definition 3: A system S *stabilizes* to a specification F iff
($\forall$ computation c of S, $\exists$ a positive integer i:
every computation of S that starts with the state $c.i$ is in F) □

Definition 4: A system S *pseudo-stabilizes* to a specification F iff
($\forall$ computation c of S, $\exists$ a positive integer i:
the computation $c \uparrow i$ is in F) □

Informally, system S stabilizes to a specification F iff starting from an arbitrary state, S is guaranteed to reach a state after which F cannot be violated. By way of contrast, S pseudo-stabilizes to F iff starting from an arbitrary state, S is guaranteed to reach a state after which F is not violated. Thus, the distinction between the two definitions comes down to the difference between "cannot" and "is not". This distinction cannot be detected or observed by an external observer of the system. In particular, an external observer can observe only one computation of the system, namely the computation that the system executes. It can also observe that the intended specification *is not* violated after some point in the computation, but cannot observe that the intended specification *cannot* be violated after some point. In other words, an external observer can observe pseudo-stabilization but not stabilization. This suggests that the two concepts of stabilization and pseudo-stabilization are indistinguishable for all practical purposes. In principle, however, the two concepts are distinct; in particular, stabilization is a more strict property than pseudo-stabilization as shown by the next theorem.

Theorem 2: Every system that stabilizes to some specification F pseudo-stabilizes to the same F; the converse is not necessarily true.

Proof: From Definitions 3 and 4 and from the fact that each computation $c \uparrow i$ starts with the state $c.i$, it follows that stabilization to F implies pseudo-stabilization to F.

We show next that the converse is not necessarily true by exhibiting a system that pseudo-stabilizes, but does not stabilize, to some specification.

Consider a system S where
states $= \{p, q\}$, and
transitions $= \{(p,p), (p,q), (q,q)\}$

Thus, the state-transition diagram of S is as shown in Figure 2.

A possible specification of S is the following F that consists of two infinite computations

$$F = \{(p,p,\ldots), (q,q,\ldots)\}$$

System S does not stabilize to F because the infinite computation $(p,p,\ldots)$ of S has no state such that every computation that starts from that state is in F.

On the other hand, S pseudo-stabilizes to F because each computation of S is in one of the following forms:

$(p,p,\ldots)$,
$(p,p,\ldots,p,q,q,\ldots)$, or
$(q,q,\ldots)$.

Thus, each computation of S has a suffix in F. □

The next theorem shows that stabilization has an advantage over pseudo-stabilization in systems with a finite number of states. Specifically, the stabilization of a finite-state system guarantees an upper bound on the number of transitions that the system can execute starting from an arbitrary state until the system starts to behave according to its specifications. Pseudo-stabilization, on the other hand, does not guarantee such an upper bound in finite-state systems.

Theorem 3: Let S be a system with n states, and let F be a specification of S.

i. If S stabilizes to F, then
($\forall$ computation $c, \exists$ positive integer $k : k \leq n+1$ and $c \uparrow k$ is in F).

ii. The same consequence does not necessarily hold if S merely pseudo-stabilizes to F.

Proof: *Part i:* Let c be a computation of S. Because of the stabilization of S, every computation of S, including c, has a suffix in F, i.e., there is a positive integer k such that $c \uparrow k$ is in F. We now show that if $k > n+1$ then there is a positive integer i smaller than k such that $c \uparrow i$ is in F. Let k be larger than $n+1$. Thus, because S has only n states, the first $k-1$ states of c, namely $c.1, c.2, \ldots, c.(k-1)$, include at least two identical states. Therefore, there is an infinite computation d of S whose states are all taken from $\{c.1, c.2, \ldots, c.(k-1)\}$. Because S stabilizes to F, then one of the states in d, say $c.i$, is such that every computation that starts with $c.i$ is in F. Because $c \uparrow i$ is a computation that starts at $c.i$, then $c \uparrow i$ is in F.

Part ii: Consider the system S and its specification F in the proof of Theorem 2. Recall that S pseudo-stabilizes to F. We need to show that

($\exists$ computation c, $\forall$ positive integer k: $k > n+1$ or $c \uparrow k$ is not in F).

Note that $n = 2$ since S has two states. Consider the computation $c = (p, p, p, q, q, \ldots)$ of S; it is straightforward to check that ($\forall$ positive integer $k : k > 3$ or $c \uparrow k$ is not in F). □

4 Relationship Between Attraction and Stabilization

In this section, we present sufficient conditions for establishing that a given system S stabilizes or pseudo-stabilizes to a given specification F. In particular, we show that if S has a (pseudo-) attractor that satisfies certain conditions involving F, then S (pseudo-) stabilizes to F. An example of applying this method to establish that a given system pseudo-stabilizes to its specification is discussed in the next section.

Theorem 4: Let S be a system with a specification F. If S has an attractor P such that every computation of S whose states are all in P is in F, then S stabilizes to F.

Proof: Let c be any computation of S; we show that c has a state p such that every computation that starts with p is in F. Because P is an attractor, every computation of S, including c, has a state in P. Let p be any state of c that is also in P. Because p is in P and P is closed, then the states of a computation that starts with p are all in P. All such computations are in F. $\Box$

Theorem 5: Let S be a system with a specification F. If S has a pseudo-attractor $(P_1, P_2, \ldots)$ such that every computation of S whose states are all in the same $P_i, i = 1, 2, \ldots$, has a suffix in F, then S pseudo-stabilizes to F.

Proof: Let c be any computation of S; we show that c has a suffix in F. Because $(P_1, P_2, \ldots)$ is a pseudo-attractor, every computation of S, including c, has a state in $\cup_{i=1}^{\infty} P_i$. Let P_k be the region in the pseudo-attractor $(P_1, P_2, \ldots)$ such that c has a state in P_k but does not have a state in $\cup_{i=1}^{k-1} P_i$. Also, let p be any state of c that is also in P_k. Because p is in P_k, and $\cup_{i=1}^{k} P_i$ is closed, and because no state of c is in $\cup_{i=1}^{k-1} P_i$, then each of the states that follows p in c is also in P_k. In other words, c has a nonempty suffix whose states are all in P_k. Because every computation whose states are all in P_k has a suffix in F, c has a suffix in F. $\Box$

5 Case Study: The Alternating-bit Protocol

We show in this section that the well-known Alternating-Bit Protocol [13] pseudo-stabilizes to its intended specification. Our objective of this exercise is two-fold. First, we want to illustrate how Theorem 5 can be utilized in verifying that a given system pseudo-stabilizes to a given specification. Second, and more important, we wish to point out that the stabilization properties of this important protocol are not as bad as may have been suggested by recent results [8, 12] which showed that this protocol is not stabilizing, and so prompted "probabilistic versions" of the protocol in order to achieve stabilization [1].

5.1 Informal Presentation of the Protocol

Consider a system of two processes s and r that communicate by exchanging messages over two channels. Each channel is an unbounded first-in-first-out buffer that stores the messages sent by one process until each of them is either received by the other process or lost, i.e. disappeared from the channel. (Note that channel characteristics, e.g. unboundedness and the ability to lose messages, are not under the control of the system designer.) Despite the possibility of message loss, process s is required to transfer reliably an infinite stream of data messages to process r. The following protocol is adopted:

i. Process s sends the data messages one at a time. After sending each message, s waits to receive an acknowledgement (ack) message before sending the next data message.

ii. If after sending a data message process s does not receive an ack message for some time, it concludes correctly that either the data message or its corresponding ack message was lost. In this case, process s times out and re-sends the last data message.

iii. Each data message has two fields:
   data(t, b)
   where t is the message text and b is either 0 or 1. When s sends a data message for the first time, it assigns the message a different b from that of the last data message. If later s re-sends the data message, the re-sent message will have the same b (and same t) as the last message.

5.2 Formal Presentation of the Protocol

The *program* of each of the two processes s and r is a set of actions that has the form:

begin <action> [] ... [] <action> **end**

The symbol "[]" is a separator that separates the different actions in the program of a process. Each action has the syntax

<guard> $\rightarrow$ <sequence of statements>

Each *guard* is in one of the following three forms:

<local guard>,
rcv <message>, or
timeout <global guard>

where a *local guard* is a boolean expression over the local variables of its process, and a *global guard* is a boolean expression over the variables in the two processes s and r, and the contents of the two channels between them. The contents of a channel is a sequence of messages. Sending of a message consists of adding the message to the "tail" of the sequence, while receiving a message consists of removing the "head" message from the sequence. The content of the channel from s to r, denoted C_{sr}, is a sequence of data messages. The content of the channel from r to s, denoted C_{rs}, is a sequence of ack messages.

The program for process s is as follows.

```
process s

  const in : array [integer] of text     {* text of all messages to be sent *}

  var   ns : integer init 0              {* index of array "in" *}
        ; readys : boolean init true     {* ready to send next data message *}
        ; bs : (0,1) init 0              {* the alternating bit to be added *}
                                         {* to each data message *}

  begin    $readys \rightarrow readys := \mathbf{false}$; send data $(in[ns], bs)$
        [] rcv ack $\rightarrow ns, readys, bs := ns + 1, \mathbf{true}, 1 - bs$
        [] timeout $\neg readys \wedge \neg readyr \wedge C_{sr} =<> \wedge C_{rs} =<>$
                   $\rightarrow$ send data $(in[ns], bs)$

end s
```

Process s has three actions. In the first action, s sends the next data message to r and waits for an ack message. In the second action, s receives an ack message (presumably for the last data message it has sent) and gets ready to send the next data message. In the third action, when neither process can send and the two channels between them are empty (indicating a message loss), process s times out and re-sends the last data message.

The program for process r is as follows.

```
process r

  var   out : array [integer] of text      {* text of all received messages *}
        ; nr : integer                     {* index of array "out" *}
        ; readyr : boolean init false      {* ready to reply by an ack *}
        ; br : (0,1) init 0                {* expected alternating bit *}
        ; t : text                         {* received text *}
        ; b : (0,1)                        {* received alternating bit *}

  begin rcv data(t, b) → if b = br then out[nr], nr, br := t, nr + 1, 1 − br fi;
                         readyr := true
        [] readyr      → readyr := false; send ack

end r
```

Process r has two actions. In the first action, r receives a data message and checks whether it is an original message that should be stored or it is a repetition of the last received message and should be discarded. In either case, r gets ready to reply with an ack message which is then sent in the second action of r.

5.3 Specification of the Protocol

A *state* of the protocol is defined by a value for each variable in the two processes, and by a sequence of data messages for the forward channel from s to r, and by a sequence of ack messages for the backward channel from r to s.

An action in one of the two processes is *enabled* at a given state iff either its guard is a (local or global) predicate whose value is true at the given state or its guard is of the form **rcv** <message> and the input channel of its process has at least one message at the given state.

A state q of the protocol *follows* a state p iff at least one action in one of the two processes is enabled at p and executing the sequence of statements of an enabled action starting from p yields q.

A *computation* of the protocol is an infinite sequence of protocol states: $p_1, p_2, \ldots$ such that each p_{i+1} follows p_i. (Notice that the protocol has no terminating states; thus all its computations are infinite.)

Let p be a state of the protocol. For convenience, let in, ns, out, and nr denote respectively the values of variables in, ns, out, and nr at state p. Also, let ms and mr be two integers. State p is (ms, mr)-safe iff the following condition holds:

$$
\begin{aligned}
&(ms \leq ns) \wedge \\
&(mr \leq nr) \wedge \\
&(\forall i : 0 \leq i < nr - mr : in[ms + i] = out[mr + i]).
\end{aligned}
$$

A computation of the protocol is *successful* iff variable nr is incremented infinitely often along the computation, and there are two integers ms and mr such that each state in the computation is (ms, mr)-safe.

The protocol is specified by the set F of all successful computations.

5.4 Pseudo-stabilization of the Protocol

Consider a system that consists of the two processes s and r, and a "message loss" action which when executed discards one message, if any, from one of the two channels between s and r. We assume that along every (infinite) computation of the system, if the same message is sent over and over by one process, then it is eventually received by the other process.

In this section, we apply Theorem 5 to show that this system pseudo-stabilizes to specification F. In particular, we show that the system has a pseudo-attractor $(P_1, P_2, \ldots)$ such that every system computation whose states are all in the same $P_i, i = 1, 2, \ldots$, has a suffix in F.

Define a function "rank" that assigns to each state p of the system a natural number $rank.p$ as follows.

$$
\begin{aligned}
rank.p = \; & \text{the number of data messages in the channel from } s \text{ to } r \text{ at } p \\
& + \text{the number of ack messages in the channel from } r \text{ to } s \text{ at } p \\
& + X.p \\
& + Y.p
\end{aligned}
$$

$$
\text{where} \quad
\begin{aligned}
X.p &= 0 && \text{if } readys = \text{false at state } p \\
&\ 1 && \text{otherwise, and} \\
Y.p &= 0 && \text{if } readyr = \text{false at state } p \\
&\ 1 && \text{otherwise.}
\end{aligned}
$$

Notice that each action in the two processes of the protocol either reduces the value of rank by 1 or keeps it unchanged. The message loss action reduces the value of rank by 1.

Consider the infinite sequence $(P_1, P_2, \ldots)$, where

$P_1 = \{p|\ p$ is a system state where either $rank.p = 0$ or $rank.p = 1\}$,
and for $i = 2, 3, \ldots$

$P_i = \{p|\ p$ is a system state where $rank.p = i\}$

It is straightforward to show that $(P_1, P_2, \ldots)$ is a pseudo-attractor of the system. First, each action in the system, other than the timeout action, either reduces the value of *rank* or keeps it unchanged. The timeout action increases the value of *rank* from 0 to 1 but still keeps the system state within P_1. Therefore, the region $\cup_{i=1}^{k} P_i$ is closed for every $k, k = 1, 2, \ldots$. Second, since the value of $rank.p$ is a natural number for each system state p, each system state is in some P_i. Hence, each system computation leads (trivially) to the region $\cup_{i=1}^{\infty} P_i$. This completes the proof that $(P_1, P_2, \ldots)$ is a pseudo-attractor of the system.

It remains now to show that every system computation whose states are all in the same $P_i, i = 1, 2, \ldots$, has a suffix in F. Consider a computation whose states are all in the same P_i. We discuss the two cases of $i = 1$ and $i > 1$ separately. If $i = 1$, then at each state in the computation there is at most one message in the two channels between s and r. It is straightforward to show that such a computation has a suffix in F (provided that every message that is sent over and over is eventually received).

If $i > 1$, then every state in the computation has the same rank i which means that the message loss action is never executed (because it reduces the rank). Moreover, the timeout action in process s is never enabled, and so is never executed. Along the computation, process s merely keeps on receiving an ack message then sending the next data message (with a bit value different from that of the last data message), and process r keeps on receiving a data message with the expected alternating bit, storing it in array *out*, then sending back an ack message. It is straightforward to show that this computation has a suffix in F.

5.5 Dealing with Message Corruption

We have assumed so far that each sent message can either be lost or received correctly; hence the protocol is designed to deal only with message loss. We now discuss how to extend the protocol to deal with message corruption as well as message loss.

The corruption of a sent message, whether data or ack, consists of changing the message into a special message "error" before it is received. (Note that the form of corruption that changes a message into another message of the same type can already be countered by the pseudo-stabilization properties of the protocol. Hence, this form of corruption is not considered here any future.)

To deal with the "error" messages that may result from message corruption, the following action is added to each of the processes s and r:

rcv error $\rightarrow$ **skip**

With this action, each process discards all the corrupted messages it receives. In effect, each message corruption is eventually transformed into a message loss.

Now, consider a system that consists of the new processes s and r, and a message loss action (similar to that defined in Section 5.4), and a message corruption action which when executed changes a message in one of the two channels, if any, into an error message. We assume that along every (infinite) computation of the system, if the same message is sent over and over by one process, then it is eventually received correctly by the other process. It is straightforward to show that this system pseudo-stabilizes to specification F. (The proof is similar to the one given in Section 5.4; in fact, it is based on the same pseudo-attractor $(P_1, P_2, \ldots)$, and the same definition of rank as in the earlier proof.)

5.6 Dealing with Message Reorder

We now discuss how to extend the protocol to handle message reorder along with message loss and corruption. A message reorder consists of swapping two consecutive messages in the same channel. Because ack messages are all identical, the reordering of ack messages in the channel from r to s has no effect on the protocol. Thus, we need only to consider reordering of data messages in the channel from s to r.

We assume that along every (infinite) computation, infinitely many original data messages are not reordered, corrupted, or lost after they are sent. (Recall that each sent data message is either an *original* or a *repetition* of the last sent message.) This assumption guarantees that every message reorder is eventually followed by a message loss or corruption (which reduces the value of rank), or eventually causes process r to receive two consecutive data messages that are original and have the same alternating-bit value. In the latter case, if process r can detect the reception of these two messages, it can discard the second message causing a message loss (which reduces the value of rank). Thus in either case, each message reorder eventually reduces the value of rank which ensures pseudo-stabilization of the system as discussed in Section 5.4 and as utilized later in Section 5.5.

Note that without message reorder, process r can still receive two consecutive data messages with the same alternating-bit value. In this situation, however, the second of these two messages is a "repetition" of the first message. This fact differentiates between this situation and the earlier situation caused by message reorder. Hence process r should be able to detect whether a received data message is an original or a repetition.

This can be achieved by adding an extra bit c to each data message:

$$c = \begin{cases} 0 & \text{if the data message is an original} \\ 1 & \text{if the data message is a repetition} \end{cases}$$

Consequently, the first and third actions of process s are changed to become:

```
$readys$        $\rightarrow readys :=$ **false**; **send** data$(in[ns], bs,$ '0')

**timeout** $\neg readys \wedge \neg readyr \wedge C_{sr} = <> \wedge C_{rs} = <>$
                $\rightarrow$ **send** data$(in[ns], bs,$ '1')
```

Moreover, the first action of process r is changed to become:

```
**rcv** data$(t, b, c) \rightarrow$ **if** $b = br \vee c = 1$ **then** $readyr :=$ **true fi**;
                          **if** $b = br$ **then** $out[nr], nr, br := t, nr + 1, 1 - br$ **fi**
```

The first if-statement guarantees that every original data message with an unexpected alternating bit is not acknowledged reducing the value of *rank*.

It is straightforward to verify pseudo-stabilization of the system consisting of the modified processes s and r and the three actions of message loss, corruption, and reorder that satisfy the following two constraints along every (infinite) computation:

i. If the same message is sent over and over by one process, then it is eventually received correctly by the other process.

ii. Infinitely many original data messages are not lost, corrupted, or reordered after they are sent.

The proof of pseudo-stabilization is similar to the one given in Section 5.4 using the same pseudo-attractor $(P_1, P_2, \ldots)$ and the same definition of rank.

6 Concluding Remarks

Achieving stabilization can be an expensive proposition in some application domains. For example, it is shown in [8, 12] that stabilization of communication protocols can be achieved only by using unbounded sequence numbers—a feature that most system designers try to avoid. When it is clear that the cost of achieving stabilization is high, system designers should strive to meet weaker criteria. We believe that pseudo-stabilization is one such criterion.

On one hand, the notion of pseudo-stabilization is reasonably close to that of stabilization. On the other hand, pseudo-stabilization is clearly not as demanding as stabilization. After all, pseudo-stabilizing communication protocols can be achieved without unbounded sequence numbers, as evidenced by our case study of the alternating-bit protocols.

The alternating-bit protocol in this paper is new and intriguing. In this protocol, the sender process sends one additional bit per data message to identify whether the data message is an original or a repetition. This identification is utilized by the receiver process to ensure that each message reorder is eventually followed by a message loss, hence guaranteeing pseudo-stabilization in the presence of message reorder. It would be hard to motivate or explain this protocol without resorting to the notion of pseudo-stabilization.

Acknowledgements: We are thankful to Anish Arora for discussions that led to Theorem 3, to the referees for suggesting a number of improvements in the presentation, and to K. F. Carbone for carefully preparing the manuscript.

7 References

[1] Y. Afek and G. Brown, "Self-stabilization of the alternating-bit protocol," *Proceedings of the Eighth Symposium on Reliable Distributed Systems*, pp. 80–83, 1989.

[2] F. Bastani, I. Yen, and I. Chen, "Class of inherently fault-tolerant distributed programs," *IEEE Trans on Software Engineering*, Vol. 14, No. 10, pp. 1432–1442, 1988.

[3] G. Brown, M. Gouda, and C. Wu, "Token systems that self-stabilize," *IEEE Trans. on Computers*, Vol. 36, No. 6, pp. 845–852, 1989.

[4] J. Burns and J. Pachl, "Uniform self-stabilizing rings," the *ACM Trans. on Programming Languages and Systems*, Vol. 11, No. 2, pp. 330–344, 1989.

[5] E. Dijkstra, "EWD 391, Self-stabilization in spite of distributed control," 1973; reprinted in *Selected Writings on Computing: A Personal Perspective*, Springer Verlag, pp. 41–46, 1982.

[6] E. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Comm. of the ACM*, Vol. 17, pp. 643–644, 1974.

[7] M. Gouda and M. Evangelist, "Convergence/response tradeoffs in concurrent systems," Technical Report TR-88-39, Dept. of Computer Sciences, Univ. of Texas at Austin, 1988; also being revised for the *ACM Trans. on Computer Systems and Languages*, 1989.

[8] M. Gouda and N. Multari, "Stabilizing communication protocols," Technical Report TR-90-20, Dept. of Computer Sciences, Univ. of Texas at Austin, 1990; also *IEEE Trans. on Computers*, to appear in 1991.

[9] M. Gouda, R. Howell, and L. Rosier, "The instability of self-stabilization," *Acta Informatica*, Vol. 27, pp. 697–724, 1990.

[10] S. Katz and K. Perry, "Self-stabilizing extensions for message passing systems," presented at the MCC workshop on Self-Stabilization, August 1988.

[11] L. Lamport, "The mutual exclusion problem: Part II—statement and solutions," *Journal of the ACM*, Vol. 33, pp. 327–348, 1986.

[12] N. Multari, *Towards a theory for self-stabilizing protocols*, Ph.D. dissertation, Dept. of Computer Sciences, Univ. of Texas at Austin, 1989.

[13] W. Stalling, *Data and Computer Communications*, Macmillan Publishing Company, pp. 133–140, 1985.