Unifying Stabilization and Termination in Message-Passing Systems

Anish Arora
Dept. of Computer and Information Science,
Ohio State University,
Columbus, OH 43210,
anish@cis.ohio-state.edu

Mikhail Nesterenko
Mathematics and Computer Science,
Kent State University, Kent, OH 44242,
mikhail@mcs.kent.edu

Abstract

The paper dispels the myth that it is impossible for any stabilizing message-passing program to be terminating. We identify fixpoint-symmetry *as a necessary condition for a message-passing stabilizing program to be terminating. Our results do confirm that a number of well-known input-output problems (e.g., leader election and consensus) do not admit a terminating and stabilizing solution. On the flip side, they show that reactive problems such as mutual exclusion and reliable-transmission do admit such solutions. We go on to present stabilizing and terminating programs for both problems. Also, we describe a way to add termination to a stabilizing program, and demonstrate it in the context of our design of a solution to the reliable-transmission problem.*

1. Introduction

Asynchronous message-passing is a standard model for representing distributed systems. The model specifies no bound on the relative execution speed of processes, or the speed of message communication. While it captures a rich class of systems, it also allows many combinations of process states to be reachable during process execution. The set of combinations becomes even more complex in the presence of failures in processes and message communication. Calculating the set is often difficult. Moreover, in some cases it is found that arbitrary global states are reached in the presence of process crashes and communication failures [6]. Thus, there is motivation to consider recovery from arbitrary global states as a way of dealing with faults in asynchronous message-passing systems, i.e., to consider stabilization. *Stabilization* of a program means that with respect to a set of legitimate global states of the program, upon starting from an arbitrary initial global state, every computation of the program eventually reaches a state in that set and then remains in states in that set thereafter.

More often than not, *termination* is also a desirable property of message-passing programs. A program is terminating if, after its environment ceases to provide input, the program eventually stops its execution.

For some time now, there has been a general belief that stabilization and termination are not co-satisfiable in the asynchronous message-passing model. The intuition behind this belief stems from the fact that individual processes in a message-passing program cannot unilaterally determine whether the program is in a legitimate global state. Now, since a stabilizing program may start from an arbitrary state (including a terminating state), if there are no messages in the channels, no process can distinguish a legitimate terminating state from an illegitimate one. As a result of this belief, the stabilizing message-passing programs that have been reported hitherto in the literature have not been terminating: either their processes periodically exchange messages to verify the legitimacy of the global state [4] or they are assumed to receive input infinitely often [1]. Worse, it is suspected that making a program stabilizing incurs significant resource overhead. The property of stabilization is therefore questioned as a practical way of dealing with faults in asynchronous message-passing systems.

In a certain sense, the belief that stabilization and termination do not coexist is justified: Gouda and Multari [5] defined a notion of what they call "proper termination" and proved that stabilizing message-passing programs cannot be properly terminating. According to their definition, all final states in a properly terminating program have no messages in the channels and the actions of all processes are disabled; thus, each process is blocked from receiving any further messages. They then show that there exist illegitimate states where each process actions are disabled yet there are messages in the channels. The disabled process actions prevent processes from receiving the messages. Therefore, the program cannot be stabilizing.

The belief in the impossibility of termination in stabilizing message-passing systems has prompted researchers to explore behavior that is "close to" termination. Along these lines, Dolev et al [4] defined the concept of silency. A program is *silent* if it converges to a global silent state where the values of local process variables do not change. However, the processes still have to continually exchange messages to confirm the silency of the state. Mizuno and Nesterenko [8] and Nesterenko and Arora [9], for instance, have developed specific stabilizing silent programs.

The importance of the issue of termination of stabilizing programs has led us to clearly separate the impossible from the achievable when stabilization and termination are unified in message-passing systems.

Contribution of the paper. We say that a program is terminating if it reaches a fixpoint, i.e. a state where all program actions are disabled. In keeping with the spirit of the definition of termination used by Dijkstra and Scholten [3], we assume that a process always has an action enabled when there is a message in an incoming channel. Thus, this notion of termination is weaker than "proper" termination of Gouda and Multari [5].

We introduce the concept of *fixpoint-symmetry*: a program is fixpoint-symmetric if its reachable fixpoints have the following property. For every pair of fixpoints f_1 and f_2 and every pair of processes P_i and P_j there exists a fixpoint f_3 such that the state of P_i is the same in f_1 and f_3 and the state of P_j is the same in f_2 and f_3. We prove that for any stabilizing message-passing program to be terminating, it has to be fixpoint-symmetric.

We observe that most of the well-known input/output problems such as leader election, classic consensus, k-set consensus, spanning tree construction, renaming, routing, etc. admit only fixpoint-asymmetric solutions. For example, in a terminating state of a program solving the leader election problem no two processes can declare themselves leaders. We argue that in a message-passing system, if there is a fixpoint where one process is elected the leader and there is another fixpoint where a different process is elected the leader then there must be a fixpoint where both processes are leaders. Since a stabilizing program can start from an arbitrary state (including a fixpoint state), the possibility of having an incorrect fixpoint precludes the existence of a stabilizing and terminating solution.

By way of contrast, we show that there are numerous problems that admit fixpoint-symmetric solutions. We demonstrate that some reactive problems, such as mutual exclusion and reliable transmission, are fixpoint-symmetric. We present a stabilizing and terminating solution to the mutual exclusion problem. Our solution is loosely based on Ricart-Agrawala's [10] mutual exclusion algorithm. We also construct a stabilizing and terminating solution to the reliable transmission problem based on the alternating-bit protocol.

We demonstrate that in some cases termination can be added to a stabilizing program. The legitimate states of a system are the states where the program exhibits correct behavior. In particular all computations starting in a legitimate state satisfy the problem solved by the program. A program can stabilize from an illegitimate fixpoint only when the environment provides more input. Thus, every computation of a stabilizing program that has illegitimate fixpoints has to have infinitely many input actions (i.e. the computation has to be infinite.) Such program is non-terminating. However, by the definition of stabilization, all fixpoints of a stabilizing terminating program must be legitimate. We find that termination can be added to a stabilizing program by modifying it to eliminate the undesirable computations starting from illegitimate fixpoints. The modification is performed as follows. If a process does not have any actions enabled and it receives input from the environment to process, the process delays servicing this input until the program state is corrected. To correct the program state, the process launches a procedure that restores the program to a state from which every computation of the original program is allowed by the problem. We demonstrate the idea by adding termination to a stabilizing alternating-bit protocol.

Organization of the paper. In Section 2, we define the model, program syntax, and semantics. In Section 3, we discuss fixpoint-symmetry property and show that that only a fixpoint-symmetric program in a message-passing system can be both stabilizing and terminating. We present a stabilizing and terminating solution to the mutual exclusion problem in Section 4. In Section 5, we describe a way to make a stabilizing program terminating and illustrate it with an alternating-bit protocol. In Section 6, we discuss possible extensions and applications of our research.

2 Definitions

Syntax. A *program* consists of a set of *processes*, each of which consists of variables and actions. Variables range over a fixed domain, such as booleans, integers, or other common types. An action is of the form: $\langle guard \rangle \longrightarrow \langle command \rangle$. A *guard* is a boolean expression over the variables of the process. A *command* is a sequence of assignment and branching statements. A *parameter* is used to define a set of actions as one parameterized action. For example, let j be a parameter ranging over values 2, 5, and 9; then a parameterized action $AC.j$ defines the set of actions: $AC.(j := 2)\ []\ AC.(j := 5)\ []\ AC.(j := 9)$

A command may contain a boolean function **input** used to receive information from the environment. An action whose guard contains **input** is an *input action*. A command may contain a statement **output** used to convey information to environment. An action whose command contains **output** is an *output action*. Both **input** and **output** may specify the names of variables used for the information exchange.

In a program in a *message-passing system*, the only variables that may be shared by (i.e. are common to) processes are of type *channel*. A channel variable has a value chosen from the domain of (potentially infinite) sequences of messages. It is shared by exactly two processes, one of which is the sender and the other is the receiver; these two processes are *neighbors* of each other. Channel variables are accessed only in the following manner. The guard of a receiver action may be **receive** function, this function specifies the identity of the sender, the type of message to be received, and the receiver variables that will store the value contained in the message. The command of a sender action may contain a **send** statement, this statement specifies the identity of the receiver, the type of message to be sent, and the receiver variables whose values the message will carry. Actions whose guard contains **receive** and whose command contains **send** are a *receive action* and *send action* respectively.

Semantics. A process of the program interfaces with the environment as follows. A process receives input from the environment via **input**. The process computes output and conveys it back to the environment via **output**. In order to compute the output a process may want to communicate with other processes through messages. A process receives a message with **receive** and sends a message with **send**.

A *state* of a process consists of a value for each variable of the process. A state of a program consists of a state of each of its processes; for simplicity, we consider the value of a channel only in the receiver process. An action whose guard is **true** at some system state is *enabled* at that state. A *fixpoint* is a program state where no action is enabled.

An **input** can evaluate to either **true** or **false** in any state of the system. A receive function is **true** when a message of specified type is at the head of the corresponding channel.

A program action may be executed in a program state only if its guard is enabled at that state. The execution of an assignment statement changes the values of variables that are assigned to; the execution of **input** assigns the value supplied by the environment to the variable specified; the execution of **receive** removes the message from the corresponding channel and assigns the values contained in that message to the specified variables; the execution of **output** does not change the state of the system; and the execution of **send** appends the message being sent to the tail of the corresponding channel. Note the dual meaning of **input** and **receive**: these boolean functions have a side effect which influences the state of the system (input value received, message removed from the channel) when the action is executed.

A *computation* is a fair, maximal sequence of steps: each step consists of executing an enabled action in the preceding program state to obtain the next program state. By fairness, we mean that if a computation is infinite and an action is enabled in all but finitely many states of the computation then this action is executed infinitely often. That is, we assume *weak fairness* for action execution. By maximality, we mean that each computation is either infinite or ends in a fixpoint state. A program is *terminating* if every computation that has a finite number of input action executions is finite.

State predicates, problems. A *state predicate* (or just predicate) is a boolean expression over program variables. A state *conforms* to a predicate if the predicate evaluates to **true** in this state; otherwise, the state *violates* the predicate. By this definition every state conforms to predicate **true** and none conforms to **false**. A state is *reachable* if there exists a computation that starts in an initial state and contains this state.

Let P be a program and R and S be state predicates of P. R is *closed* in P if each state of every computation of P that starts in a state conforming to R also conforms to R. R *converges* to S in P if R is closed in P, S is closed in P, and any computation starting from a state conforming to R contains a state conforming to S. P *stabilizes* to R iff **true** converges to R in P. In the rest of the paper, we omit the name of the program whenever it is clear from the context.

A program *invariant* is a closed predicate defined as follows: if a computation starts in a state conforming to the invariant the computation is correct. A program is *stabilizing* if every computation starting from an arbitrary state contains a state belonging to the invariant. By this definition, every fixpoint of a stabilizing terminating program conforms to the invariant.

A program *solves* a certain problem. A problem is *reactive* if it requires a solution to have at least one correct computation containing an infinite number of input action executions. An extreme case of a non-reactive problem is an *input/output* problem. Input/output problem requires each process to have at most one input.

3 Fixpoint-symmetry

Definition 1 *A program is* fixpoint-symmetric *if its fixpoints have the following property. For every pair of conforming to the invariant fixpoints f_1 and f_2 and every pair of processes P_i and P_j there exists a conforming to the invariant fixpoint f_3 such that the state of P_i is the same in f_1 and f_3 and the state of P_j is the same in f_2 and f_3.*

If a program is not fixpoint-symmetric, it is *fixpoint-asymmetric.* Most well-known input/output problems admit only fixpoint-asymmetric solutions. Hence the following proposition.

Proposition 1 *The following input/output problems admit only fixpoint-asymmetric solutions: leader election, classic consensus, k-set consensus, spanning tree construction, renaming, and routing.*

We prove this proposition for leader election in the expanded version of the paper currently in preparation. We explain the intuition behind the proof below. The reader may want to skip the displayed material throughout the rest of the paper on the first reading.

> An *input variable* is a variable that holds the value transmitted to the program by the environment. An *output variable* is a variable that holds the value transmitted the environment by the program. An *input/output state* is an assignment of values to input and output variables for all processes in the system. Then a problem is a set of input/output state sequences. The last state in a finite sequence is a *terminal* state. In each sequence the environment provides an *input subsequence* and the problem stipulates what the possible *output subsequences* are. A *program state* is an input/output state with the values of internal program variables added. If the sequence of states of a program computation produces an input/output sequence (when the internal program variables are projected out and stuttering removed) then the computation *maps* to the input/output sequence. Note that a program fixpoint corresponds to a terminal state of an input/output sequence. A *symmetric set of terminal states* can be defined similar to Definition 1. A program *solves* a problem if for every input subsequence of the problem the program has a computation that maps to an input/output sequence with this input subsequence. Thus, if a terminating program solves a problem, the set of program fixpoints corresponds to a subset of the problem's terminal states such that there is at least one terminal state for every finite input subsequence.
>
> There is a fixpoint-symmetric solution to a problem only if the problem's terminal states have a symmetric subset where every input subsequence corresponds to at least one terminal state.
>
> Let us consider leader election problem. A non-trivial leader election problem has the following properties: (a) there are at least two terminal states t_1 and t_2 such that a different process is elected the leader in each state; (b) there are at least two input subsequences σ_1 and σ_2 such that no computation containing σ_1 ends in t_2 and no computation containing σ_2 ends in t_1. These conditions prohibit the existence of a symmetric subset of terminal states necessary for the existence of fixpoint-symmetric solution.

Theorem 1 *If a stabilizing program in a messages-passing system is terminating then it is fixpoint-symmetric.*

Proof: A terminating program has to have at least one reachable fixpoint. If a terminating program has only one fixpoint, it is trivially fixpoint-symmetric.

Let us consider a terminating program that has at least two fixpoints. We select an arbitrary pair of fixpoints f_1 and f_2 and an arbitrary pair of processes P_i and P_j, and we construct a state f_3 as follows. The state of P_i in f_3 is the same as in f_1, the state of P_j in f_3 is the same as in f_2, and the states of the other processes are taken from an arbitrary fixpoint. Note that since the states of each process (recall that we consider the channels to belong to receiver processes) are the same as in some fixpoint, in a message-passing system all actions must be disabled in f_3. That is, f_3 is a fixpoint of the program. Recall that every fixpoint

```
process P_i
const
    𝒫, set of process identifiers of the system
var
    myts, timestamp of CS request
    ℒ : ℒ ∈ 𝒫, set of locked processes
    needcs : boolean, true if CS needed
    ts, timestamp of received message
    needrep : boolean, true if message needs reply
param
    j : P_j ∈ (P)

    *[
(r1)    input ∧ ¬needcs  ⟶
            myts := newts(),
            ℒ := ∅,
            needcs := true
      ▯
(r2)    P_j ∉ ℒ ∧ needcs ⟶
            send(myts, true) to P_j
      ▯
(r3)    receive(ts, needrep) from P_j  ⟶
            if needcs then
                if ts ≥ myts then
                    ℒ := ℒ ∪ {P_j}
            else
                myts := newts(),
            if needrep then
                send(myts, false) to P_j
      ▯
(r4)    ℒ = 𝒫 ∧ needcs ⟶
            output, /* CS execution */
            needcs := false
     ]
```

Figure 1. Process of TRA

of a stabilizing program is reachable. Therefore, this terminating program is fixpoint-symmetric. ◻

Corollary 1 *The following problems have no stabilizing solution that is terminating: leader election, classic consensus, k-set consensus, spanning tree construction, renaming, and routing.*

4. Stabilizing and terminating programs in message-passing systems

In this section we describe a stabilizing and terminating solution to the mutual exclusion problem in message passing systems.

Mutual exclusion problem description. The mutual exclusion (MX) problem is stated as follows. During a computation each process in the system can request to execute *critical section* (CS) an arbitrary number of times. The solution of the problem has to satisfy the following properties. *Safety*: if an action executes CS it can be enabled in at most one process in one state of the system. *Liveness*: if process wishes to execute CS the process is eventually given a chance to do so. A process wishing to execute CS is *in CS contention.*

A more formal specification of the MX problem states that each process has one boolean input variable (indicating if the environment requests CS) and one boolean output variable (indicating if it is safe to enter CS.) Note that the problem specifies only one terminal state – the state where all processes are out of their critical sections and no process requests to enter CS. Thus, the problem allows a fixpoint-symmetric solution. In a straightforward manner the solution described below (TRA) can be converted to solve the problem as stated in this paragraph.

As an aside we would like to note that a slight modification in the problem statement prohibits the existence of a fixpoint-symmetric solution. Namely, if the environment is allowed to keep a process in CS indefinitely, there will be a number of terminating states – one for each process in CS. If an input/output sequence terminates in a state where a process executes CS, the input part of the terminal state indicates that the process is in CS. Thus, this input subsequence has only one matching terminal state and the set of terminal states does not have a symmetric subset.

TRA description. We call our terminating solution TRA because it is loosely based on Ricart-Agrawala's algorithm [10]. The code for a process P_i of TRA is shown in Figure 1. To make the idea of the program easy to understand we tried to make the code as simple as possible. We discuss how to make the code for TRA more realistic at the end of the section.

The idea of TRA is as follows. The program uses unbounded timestamps. A process with the smallest timestamp in the system is allowed to enter CS. When a process needs to enter CS it selects a new, greater than previous, timestamp and sends messages bearing this timestamp to all other processes. When process P_i

gets a message from another process P_j with a timestamp greater than P_i's timestamp, P_i *locks* P_j. When a process locks all the other processes in the system, the process enters CS.

We now describe the program actions and variables in greater detail. Function **newts**() returns a greater timestamp each time it is called. There is just one type of messages. A message carries the timestamp of the sender and a boolean value stating if the sender needs a reply.

The input action ($r1$ in Figure 1) models the request for CS entry. To enter CS the process obtains a new timestamp, stores it in *myts*, sets *needcs* to **true** to indicate that it is in CS contention, and empties the set of locked processes $\mathcal{L}$. If a process is not in $\mathcal{L}$ ($r2$) and P_i is in CS contention, P_i sends a message to this process requesting reply.

When P_i receives a message from another process P_j ($r3$), it compares its timestamp with P_j's. If P_i is in CS contention and P_j has a higher timestamp, P_i adds P_j's identifier to the set of locked processes. If P_i is not in CS contention it gets a newer timestamp in an attempt to exceed the timestamp of P_j and let P_j execute CS. If P_j requests reply, P_i sends it a message with its own timestamp.

When P_i locks all processes in the system ($r4$) it executes CS. After CS execution P_i sets *needcs* to **false**.

Theorem 2 *TRA is a stabilizing and terminating solution to the mutual exclusion problem.*

For reasons of space we relagate the proof to [2].

TRA implementation and efficiency improvements. TRA can be made more realistic in a straightforward manner. When joining CS contention the process may send messages to all the other processes in the system. In this case $r2$ is needed only as a timeout action to be periodically executed when a reply from a certain process is not received. Note, that this timeout need not be executed when the process is not requesting CS, thus termination is preserved.

Process P_i can maintain a set of processes that lost CS contention. After executing CS, P_i can send messages to these processes to let them know that they can proceed with CS contention.

Bounded overtaking specifies that each process can enter CS only a finite number of times before another process requesting CS enters it. Note that TRA does not guarantee bounded overtaking. To amend that the processes can execute Lamport's logical clock algorithm [7] to make sure that the timestamps at each process are synchronized.

5. Making stabilizing reactive programs terminating

All the fixpoints of the stabilizing program presented in the previous section belong to the invariant. Thus, the program is terminating. This, however, is not necessarily always the case.

Note that the some fixpoints of a stabilizing nonterminating program do not have to belong to the invariant. Stabilization from these fixpoints is dependent on the environment providing input. The program, however, does not guarantee correct behavior when it stabilizes from such a fixpoint.

We claim that termination can be added to such a program in the following manner. When a process does not have any actions enabled it assumes that the system can be in an illegitimate fixpoint. When the environment submits a new request, the process delays satisfying it and launches a procedure that ensures that the system is in a legitimate state. Thus, every computation from this fixpoint becomes legitimate which moves the fixpoint into the invariant.

As an example of this approach we present a nonterminating stabilizing version of alternating-bit protocol (ABP) and add termination to it.

Reliable transmission problem description. Sender process p and receiver process q are connected by two channels passing messages in the opposite directions. The channels are unreliable (may lose messages.) To model message loss we assume that **send** may either succeed or fail nondeterministically. If **send** succeeds, it inserts its message at the end of the channel's sequence of messages. If **send** fails, no message is inserted. To allow the possibility of satisfying any non-trivial liveness property we assume *transmission fairness*. That is, if there are infinitely many attempts to send a message over a channel in some computation then infinitely many attempts must succeed. The environment *submits* a packet to p to be transmitted to q. Process q *delivers* successfully transmitted packets to the environment. Any solution to the reliable transmission problem has to satisfy the following properties. *Safety*: q delivers packets in the order they are submitted by p. *Liveness*: q eventually delivers a packet submitted by p.

> A more formal specification of the reliable transmission problem states that q has an input variable – a queue of messages to be transmitted; and p an output variable – a queue of messages to be delivered. In the only terminal state of the problem the input and output

```
process p

*[
(p1)    input m ⟶
            enque(m, sbuf)
     []
(p2)    receive ack(i) ⟶
            if i = ns then
                deque(sbuf),
                ns := ns + 1
     []
(p3)    ¬empty(sbuf) ⟶
            send data(first(sbuf), ns)
 ]
```

Figure 2. Sender process of SABP

```
process q

*[
(q1)    ¬empty(rbuf) ⟶
            output deque(rbuf)
     []
(q2)    receive data(m, i) ⟶
            if i ≠ nr then
                enque(m, rbuf),
                nr := i,
            send ack(i)
 ]
```

Figure 3. Receiver process of SABP

queues are empty. Thus, the problem allows a fixpoint-symmetric solution.

SABP description. We first present a non-terminating version of stabilizing ABP (SABP), (see Figures 2 and 3.) Processes p and q maintain inifinite queues *sbuf* and *rbuf* respectively. The queues are called send buffer and receive buffer. Function *enque* adds a packet to the tail of a buffer, and *deque* removes the packet from the head of the buffer and returns the packet. Function *empty* returns **true** when there are no packets in the buffer. Function *first* returns the packet at the head of the buffer without removing it. Processes p and q also maintain integer counters ns and nr respectively to keep track of the packets transmitted. Counter ns keeps the sequence number (SN) of the packet last sent, nr – last received.

```
process p

*[
(p1)    input m ⟶
            enque(m, sbuf)
     []
(p2)    receive ack(i) ⟶
            if i = ns then
                deque(sbuf),
                ns := ns + 1
     []
(p3)    ¬empty(sbuf) ∧ go ⟶
            send data(first(sbuf), ns)
     []
(p4)    empty(sbuf) ∧ go ⟶
            go := false
     []
(p5)    ¬empty(sbuf) ∧ ¬go ⟶
            send dummy(ns)
     []
(p6)    receive dack(i) ⟶
            if i = ns then
                go := true,
                ns := ns + 1
 ]
```

Figure 4. Sender process of TABP

```
process q

*[
(q1)    ¬empty(rbuf) ⟶
            output deque(rbuf)
     []
(q2)    receive data(m, i) ⟶
            if i ≠ nr then
                enque(m, rbuf),
                nr := i,
            send ack(i)
     []
(q3)    receive dummy(i) ⟶
            if i ≠ nr then
                nr := i,
            send dack(i)
 ]
```

Figure 5. Receiver process of TABP

Input action ($p1$) submits a new packet to send. When the send buffer is not empty, p sends a *data* message carrying a packet and its SN ($p3$). When q receives a *data* carrying an SN different from SN last received ($q2$), the packet the message carries is appended to receive buffer. The receipt of a message is acknowledged by sending *ack* back to p. Correctly received message is delivered ($q1$). When the acknowledgement of the last message sent ($r2$) is received, ns is incremented and a new message can be sent.

Note that SABP is fixpoint-asymmetric and, by Theorem 1, non-terminating. In particular it has a fixpoint (where $ns = nr$) such that a computation starting from this fixpoint violates safety: the receiver does not deliver the first message submitted for transmission.

Theorem 3 *SABP is a non-terminating stabilizing solution to the reliable transmission problem.*

The proof is provided in [2].

Description of TABP. We add termination to SABP by incorporating a procedure that ensures $ns \neq nr$. Each time there is a possibility that the system is in a fixpoint ($empty(sbuf)$) the program launches this procedure. This ensures that any computation starting from this fixpoint is correct which brings the fixpoint into the invariant and makes the program fixpoint-symmetric.

Theorem 4 *TABP is a stabilizing and terminating solution to the reliable transmission problem.*

The proof is provided in [2].

6. Future research

In this paper we showed that, in spite of popular belief, useful programs in message-passing systems can be both terminating and stabilizing. We identified the condition of fixpoint-symmetry that every such program must satisfy. We are presently investigating useful sufficient conditions on problems for them to admit such solutions.

We described the addition of termination to a stabilizing ABP program. We are presently generalizing our approach to deal with any stabilizing non-terminating program, assuming that the problem solved by the program does not prohibit the existence of a terminating, stabilizing solution.

Lastly, we believe it would be desirable to investigate the relationship of the termination property with other properties of stabilizing programs such as time and space requirements for stabilization, locality, etc.

References

[1] Y. Afek and G. Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7:27–34, 1993.

[2] A. Arora and M. Nesterenko. Unifying stabilization and termination in message-passing systems. Technical Report OSU-CISRC-1/01-TR02, Ohio State University, 2001. ftp://ftp.cis.ohio-state.edu/pub/techreport/2001/TR02.ps.gz.

[3] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, Aug. 1980.

[4] S. Dolev, M. Gouda, and M. Schneider. Memory requirements for silent stabilization. In *PODC96 Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 27–34, 1996.

[5] M. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40:448–458, 1991.

[6] M. Jayaram and G. Varghese. The complexity of crash failures. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 179–188, Santa Barbara, California, 21–24 Aug. 1997.

[7] L. Lamport. Time, clocks and ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–564, 1978.

[8] M. Mizuno and M. Nesterenko. A transformation of self-stabilizing serial model programs for asynchronous parallel computing environments. *Information Processing Letters*, 66(6):285–290, 1998.

[9] M. Nesterenko and A. Arora. Stabilization-preserving atomicity refinement. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 254–268, 1999. http://www.mcs.kent.edu/~mikhail/refinement.ps.

[10] G. Ricart and A. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981.