Distrib. Comput. (2004)
Digital Object Identifier (DOI) 10.1007/s00446-004-0111-6

DISTRIBUTED COMPUTING

Unifying stabilization and termination in message-passing systems*

Anish Arora[1,**], **Mikhail Nesterenko**[2,***]

[1] Department of Computer and Information Science, Ohio State University, Columbus, OH 43210, USA (e-mail: anish@cis.ohio-state.edu)
[2] Department of Computer Science, Kent State University, Kent, OH 44242, USA

Published online: November 15, 2004 –

Abstract. The paper dispels the myth that it is impossible for a message-passing program to be both terminating and stabilizing. We consider a rather general notion of termination: a terminating program eventually stops its execution after the environment ceases to provide input. We identify *termination-symmetry* to be a necessary condition for a problem to admit a solution with such properties. Our results do confirm that a number of well-known problems (e.g., consensus, leader election) do not allow a terminating and stabilizing solution. On the flip side, they show that other problems such as mutual exclusion and reliable-transmission allow such solutions. We present a message-passing solution to the mutual exclusion problem that is both stabilizing and terminating. We also describe an approach of adding termination to a stabilizing program. To illustrate this approach, we add termination to a stabilizing solution for the reliable transmission problem.

Keywords: Self-stabilization, termination, fault-tolerance

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 [10]. 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. A program is *stabilizing* if, regardless of its initial state, the program eventually reaches a state in the set of legitimate states and remains in this set thereafter. Another program property we address is termination. A program is *terminating* if it eventually stops its execution after its environment ceases to provide input. The inherent efficiency of this mode of program operation makes termination attractive.

For some time now, there has been a general belief that stabilization and termination are not co-satisfiable in the asynchronous message-passing model. Dolev, for example, gives an informal justification for this belief in his book [6, p. 10]. This belief stems from the fact that each individual process in a message-passing program cannot unilaterally determine whether the program is in a legitimate global state. Since a stabilizing program may start from an arbitrary state (including a terminal state), if there are no messages in the channels, no process can distinguish a legitimate terminal state from an illegitimate one. The work of Gouda and Multari [8] is cited as a partial justification of this belief. Gouda and Multari define "proper termination" and prove 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 the processes from receiving the messages. Therefore, the program cannot be stabilizing.

As a result of the belief in the lack of termination, the stabilizing message-passing programs that have been reported hitherto in the literature are not terminating: either their processes periodically exchange messages to verify the legitimacy of the global state [7] or they are assumed to receive input infinitely often [1]. This belief prompted the researchers to explore behavior that is "close to" termination. Along these lines, Dolev

* Some of the results in this paper were presented at the 21st International Conference on Distributed Computing Systems, Mesa, Arizona, April 2001, pp 99–106.

** Supported in part by DARPA contract OSU-RF #F33615-01-C-1901, NSF grant NSF-CCR-9972368, Ohio State University Fellowship, and 2002-2003,2003-2004 grants from Microsoft Research.
*** Supported in part by DARPA contract OSU-RF #F33615-01-C-1901 and by NSF CAREER Award 0347485

Correspondence to: Mikhail Nesterenko
(e-mail: mikhail@cs.kent.edu)

et al [7] 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 legitimacy of the state. Mizuno and Nesterenko [12] and Nesterenko and Arora [13], for instance, designed specific stabilizing silent programs. If termination and stabilization indeed cannot be jointly achieved, then making a program stabilizing has to incur significant resource overhead due to continuous operation. The property of stabilization is therefore questioned as a practical way of dealing with faults in asynchronous message-passing systems. 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 consider the program to be terminating if it eventually stops its execution after its environment ceases to provide input. In a terminal state the program is in a fixpoint – a state where all its internal actions are disabled. We assume (cf. [5]) that a process always has an enabled action if there is a message in an incoming channel. Thus, this notion of termination is weaker than "proper" termination of Gouda and Multari [8]. Our notion of termination is more general than *transformational termination* [15] used by some researchers. A transformationally terminating program assumes no inputs from the environment after the initial state. The results presented in this paper apply to this definition of termination as well. Observe that we assume that the fact that the program is in terminal state is known to the environment only and the processes are not aware of termination.

We introduce the concept of *state symmetry*. Intuitively, given some set of states of each individual process, the symmetric set of global program states is the Cartesian product of these process states. We define a problem specification to be *termination-symmetric* if it contains a symmetric set of global states where each such state is either (a) a prescribed terminal state or (b) a state where the environment is expected to provide additional input. A more formal definition of termination symmetry is given in the sequel. We prove that if a problem admits a stabilizing and terminating solution in a message-passing system, then this problem is termination-symmetric.

On the basis of this proof we show that the consensus problem is not termination-symmetric and thus does not admit a stabilizing and terminating solution. We observe that this negative result extends to a large number of well known problems: leader election, graph coloring, spanning tree construction, renaming and routing topology construction. By way of contrast, we show that there are termination-symmetric problems. We prove that the mutual exclusion and reliable transmission problems are in fact termination-symmetric. Furthermore, we present a stabilizing and terminating solution to the mutual exclusion problem. Our solution is loosely based on Ricart-Agrawala's [14] mutual exclusion algorithm.

We demonstrate how termination can be added to a stabilizing program. This addition eliminates undesirable computations starting from illegitimate fixpoints (i.e. such fixpoints are made legitimate). Recall that a stabilizing program has to eventually reach a legitimate state. Thus, if a stabilizing program has an illegitimate fixpoint then the program can stabilize from it by only receiving environmental input. Essentially, such a program relies on the environment to move it into a legitimate state. We eliminate such reliance as follows. If a process does not have any internal actions enabled and it receives input from the environment, the process delays servicing this input until the program state is corrected. The environment input enables an action starting a procedure that restores the program to a state from which every computation of the original program is allowed by the problem. We illustrate the idea by adding termination to a stabilizing alternating-bit protocol. We also discuss how this idea can be generalized.

Global state correctors are also used in snap-stabilization [4]. By definition, a snap-stabilizing program, regardless of its initial state, correctly services every request from the environment. To satisfy this requirement the servicing of each request is either done concurrently with state correction or suspended until the correction is completed. Our approach is less strict. During stabilization, a program does not have to produce correct output in response to the requests from the environment. Hence, the corrector is invoked less frequently in a stabilizing and terminating program. Namely, the corrector is run when there is both input from environment to process and a suspicion that the system is in an illegitimate terminal state.

Organization of the paper. In Sect. 2 we formalize the programming model to be considered. In Sect. 3 we define termination-symmetry and use it to state the necessary condition for a problem to allow a stabilizing and terminating solution in a message-passing system. We present such a solution to the mutual exclusion problem in Sect. 5. In Sect. 6 we describe a way to make a stabilizing program terminating and illustrate it with an alternating-bit protocol. We conclude this section by outlining how this idea can be generalized to an arbitrary program and how it can lead to a sufficient condition for the existence of a stabilizing and terminating solution to a problem. In Sect. 7, we discuss possible extensions and applications of our research.

2 Programming notation

2.1 Syntax

A *program* is a set of *processes* each of which consists of variables and actions. A variable ranges over a fixed domain, such as booleans, integers, or other common types. Variable v of process p is denoted $v.p$. Some variables of a process are *input* variables. Others are *output* variables. Input and output variables are *external* variables. 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: $ac.(j := 2) \; [\!] \; ac.(j := 5) \; [\!] \; ac.(j := 9)$. An action is *input* if it updates input variables, it is *internal* otherwise. A guard of an input action is always **true**. An input action does not mention internal variables.

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. A

channel is shared by exactly two processes, one of which is the sender and the other is the receiver. Channel variables are accessed only in the following manner. The receiver contains a *receive action* guard. The guard of a receive action starts with **receive**. This guard specifies the identity of the sender, the type of message to be received, and the receiver variables to store the values carried by the message. The sender contains a *send action*. This action contains **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 is to carry.

2.2 Semantics

States and state sequences. A *state* s of some set of variables V is an assignment of a value to every variable in V. This definition allows us to reason about states of a process and a program. For simplicity, we consider the value of a channel only in the receiver process. A state sequence σ, formed by a set of variables V, is a sequence of states of variables of V. The first and last states of a sequence are its *initial* and *terminal* states respectively.

A *projection* of a state sequence σ formed by a set of variables V onto a set of variables $W \subset V$ is obtained as follows. Every state in the sequence is projected onto W and consecutive identical states are eliminated. The projection of a state sequence is *external* if this state sequence is projected onto external variables.

An input *step* is a sequence of two states that differ in the value of the input variables. Given a set of sequences $\mathcal{A}$ a subset $\mathcal{B} \subset \mathcal{A}$ is *input-complete* if the sequences in $\mathcal{B}$ preserve both the results and the order of the input steps of $\mathcal{A}$. Namely, for every sequence $\alpha \in \mathcal{A}$ there exists a sequence $\beta \in \mathcal{B}$ such that: every input step s_1 of α is also in β and for every pair or input steps s_1 and s_2, their order α and β is the same.

Actions and computations. An action whose guard is **true** at some system state is *enabled* at that state. A receive statement is enabled if a message of the type specified in the guard of this statement is at the head of the corresponding channel.

A *fixpoint* is a program state where no internal action is enabled. A program action may be executed in a program state only if its guard is enabled in that state. Depending on the action type the effect of the execution of this action differs. The execution of an assignment statement changes the values of variables that are assigned to. The execution of a receive action removes the message from the specified channel and assigns the values contained in that message to the specified variables. The execution of **send** appends the message being sent to the tail of the corresponding channel. An input action is always enabled, its execution is prescribed by the particular problem that the program solves.

A *computation* is a fair, maximal sequence of steps: each step is produced 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 internal 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 internal action execution (weak fairness does not apply to input actions). By maximality, we mean that each computation is either infinite or ends in a fixpoint. A program is *terminating* if every computation of this program where the input is finite is itself finite.

Problems, state predicates. A *problem specification* $\mathcal{S}$ is a set of sequences formed by a set of external variables. We use the terms problem and problem specification interchangeably.

In problem sequences two consecutive states, where the values of input variables differ, signify input from the environment. If a state of a program projects to the first one of such external states the input program action may be executed. This execution takes the program to the state that projects to the second external state. This way the program receives input. An external state is *input-displaced* in a set of external sequences, if it only appears as a first state of an input step. An input-displaced state is significant because the only way for the system to move from this state is to receive extra input from the environment. Notice that, by definition, an input-displaced state cannot also be a terminal state. Notice also, that if a state is input-displaced in some set of external sequences, this state is also input-displaced in every subset of this set.

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

An *invariant* of program $\mathcal{P}$ for a specification $\mathcal{S}$ is a closed predicate on the states of $\mathcal{P}$ with the following property. For every computation of $\mathcal{P}$ that starts from a state conforming to the invariant, the projection of this computation on the external variables belongs to $\mathcal{S}$. $\mathcal{P}$ *satisfies* (or *solves*) $\mathcal{S}$ if there exists an invariant of $\mathcal{P}$ for $\mathcal{S}$ such that the projections of the computations of $\mathcal{P}$ starting from the invariant onto the external variables form an input-complete subset of $\mathcal{S}$. A program state conforming to an invariant is *legitimate*. $\mathcal{P}$ is *stabilizing* if every computation starting from an arbitrary state contains a legitimate state.

We conclude this section by discussing the relationship between some of the concepts we introduced. We define a problem to be a set of sequences of external variables. In essence these sequences prescribe the acceptable inputs and corresponding outputs that a solution to the problem must exhibit. A program that solves this problem, potentially adds some internal variables to the external state space. However, if the program starts its execution from a legitimate state and the program is given the inputs specified by one of the problem's sequences then the outputs produced by the program should also be the same as in this sequence. Observe, that a solution does not have to thus implement all sequences specified by the problem. Nonetheless, a solution has to accept all inputs. Hence our definition of input-completeness. Multiple computations of a solution may implement the same external sequence. These computations vary in the order of internal steps.

Some of the problem's sequences can be finite (i.e. end in a terminal state). The program may not implement such a sequence at all or implement it without termination. The latter case the program continuously updates the internal variables. However, if a computation of a program ends in a fixpoint, this fixpoint corresponds to a terminal state of the specification. A fixpoint does not necessarily terminate a computation. A fixpoint may appear in the middle of it. Since all the internal actions of a program are disabled in a fixpoint, the program may leave it only if an input action is executed. Thus, the external projection of a legitimate fixpoint is either a terminal or input-displaced state of the specification.

Stabilization is a particular program property. If a stabilizing program starts in an arbitrary state then its computation contains a suffix that implements one of the problem's sequences. Notice that by definition, if a computation of a stabilizing program terminates in a fixpoint, this fixpoint is legitimate.

3 Necessary condition for stabilization and termination

Definition 1 (State symmetry) *A set of states S is* symmetric *if for every pair of states $s_1 \in S$ and $s_2 \in S$ and every pair of processes p and q there exists a state $s_3 \in S$ such that the state of p is the same in s_1 and s_3 and the state of q is the same in s_2 and s_3.*

Figure 1 illustrates Definition 1. A program is *fixpoint-symmetric* if its set of fixpoints is symmetric.

Lemma 1 *If a program in a message-passing system is terminating then it is fixpoint-symmetric.*

Proof. A terminating program with less than two fixpoints 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 and q. We construct a state f_3 as follows. The state of p in f_3 is the same as in f_1, the state of q 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 internal process actions must be disabled in f_3. That is, f_3 is a fixpoint of the program. □

Lemma 2 *If a problem has a stabilizing solution, then each fixpoint of this solution projects onto either a terminal or input-displaced state of the problem.*

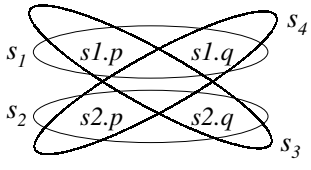

Fig. 1. Condition for system state symmetry. Let the respective states of a pair of processes p and q be $s1.p$ and $s1.q$ in system state s_1 and $s2.p$ and $s2.q$ in system state s_2. If a symmetric set of states contains both s_1 and s_2 then it also should contain s_3 (and, by symmetry, s_4)

Proof. Recall that a legitimate fixpoint indeed projects onto a terminal or input-displaced state. We now show that every illegitimate fixpoint of a stabilizing program projects onto an input-displaced external state.

Consider a computation whose initial state is an illegitimate fixpoint. All internal program actions are disabled in a fixpoint. Hence, the only way for a program to leave this state is to execute an input action. If no input is forthcoming the computation terminates in the fixpoint. By definition, every computation of a stabilizing program reaches a legitimate state. Since this fixpoint is illegitimate, the program has to leave it. Therefore, the fixpoint projects onto an input-displaced external state. □

Definition 2 (Termination-symmetry) *Problem specification $\mathcal{S}$ is* termination-symmetric *if there exists a symmetric set of states A whose states are either input-displaced or terminal in $\mathcal{S}$ and an input-complete subset $\mathcal{R}$ of $\mathcal{S}$, such that every terminal state of $\mathcal{R}$ belongs to A.*

Theorem 1 *If a problem has a stabilizing and terminating solution in a message-passing system, then the problem is termination-symmetric.*

Proof. Let $\mathcal{S}$ be a problem specification. Also, let $\mathcal{P}$ be a stabilizing and terminating solution to $\mathcal{S}$ in a message-passing system. By definition of solution, the external projections of the computations of $\mathcal{P}$ that start from a legitimate state form an input-complete subset $\mathcal{R}$ of $\mathcal{S}$.

Let F be the set of fixpoints of $\mathcal{P}$. Since $\mathcal{P}$ is terminating and its system model is message-passing, by Lemma 1, F is symmetric. Since $\mathcal{P}$ is stabilizing, according to Lemma 2, its every fixpoint projects onto either a terminal or input-displaced external state. Observe that state-symmetry is invariant under projection. Thus, the projections of F form a symmetric set A of terminal and input-displaced states.

To complete the proof we need to show that every terminal state of $\mathcal{R}$ belongs to A. Let us consider an arbitrary terminal state t of $\mathcal{R}$. By definition, it ends a finite sequence and, therefore, has finite input. This sequence is the projection of a computation of $\mathcal{P}$. Since $\mathcal{P}$ is terminating, a finite input computation ends in a fixpoint. Hence t is the projection of a fixpoint of $\mathcal{P}$. Which means that t belongs to A. The theorem follows. □

4 The necessary condition applied to example problems

Mutual exclusion. The problem of mutual exclusion $\mathcal{MX}$ is specified as follows. The processes request to execute the *critical section* (CS) of code. Throughout the computation a process may request the CS arbitrarily many (including zero) times. A process executing the CS eventually leaves it and starts executing non-critical code. The solution to the problem has to satisfy two properties: *safety* – no two processes execute the CS at the same time; *liveness* – each requesting process is eventually given a chance to execute the CS.

Let us give a more formal specification of $\mathcal{MX}$. Each process p has two external boolean variables: an input variable $require.p$ to indicate that the process requests the CS and an output variable $enter.p$ to indicate that it is safe to enter the CS. The specification contains state sequences conforming

to the following rules. In no two processes $enter$ is **true** in the same state. If for some process p, $request.p$ is **true** in some state of any sequence, then eventually either $request.p$ is **false** or $enter.p$ is **true**. If in some state of any sequence $enter.p$ is **true**, then $enter.p$ is **false** in one of the consequent states. Observe that if in some state $request.p$ is **false**, then the specification does not require $enter.p$ to eventually be **true**.

Observe that $\mathcal{MX}$ has only one terminal state: the state where the external variables of all processes are **false**. Hence, the set of terminal states of this specification is trivially symmetric. Any input-complete subset of state sequences of the problem specification which includes this terminal state satisfies the conditions of Theorem 1. This allows us to state the following theorem.

Theorem 2 *The mutual exclusion problem allows a solution in a message-passing system such that this solution is both stabilizing and terminating.*

Reliable transmission. The reliable transmission problem $\mathcal{RT}$ requires the sender process p to reliably pass a sequence of packets to the receiver process q over unreliable media. When we discuss $\mathcal{RT}$ we always use these names for the sender and receiver processes. The environment *submits* a sequence of packets to p to be transmitted to q. Process q *delivers* successfully transmitted packets to the environment. A solution to $\mathcal{RT}$ has to satisfy the following properties: *safety* – q delivers packets in the order they are submitted to p; *liveness* – q eventually delivers a packet submitted to p.

To simplify the presentation in our formal specification of $\mathcal{RT}$ we allow the program and the environment to jointly modify external variables. That is, the external variables are not separated into input and output ones. It is, however, straightforward to reconstruct the specification where the types of external variables are clearly defined. Infinite queues of packets $in.p$ and $out.q$ are the external variables of p and q respectively. The environment inserts a packet into $in.p$. Process p removes a packet from $in.p$ and transmits it to q. When q receives a packet it inserts it into $out.q$. The received packet is removed from $out.q$ by the environment. The specification of $\mathcal{RT}$ contains the state sequences with the following property: in every state sequence, the sequence of packets removed from $out.q$ is a prefix of the sequence of packets inserted into $in.p$.

Similar to $\mathcal{MX}$, $\mathcal{RT}$ has only one terminal state. This is the state where $in.p$ and $in.q$ are empty. The uniqueness of a terminal state ensures termination symmetry of $\mathcal{RT}$.

Theorem 3 *The reliable transmission problem allows a solution in a message-passing system such that this solution is both stabilizing and terminating.*

Consensus and similar problems. The problem of consensus $\mathcal{CN}$ is to make all processes of the program agree on one particular input value. The formal specification of $\mathcal{CN}$ is as follows. Each process p has a boolean input variable $in.p$ and a boolean output variable $out.p$. The set of sequences that $\mathcal{CN}$ contains, satisfies the following three properties: *termination* — in the initial state, in variable of each process is set to **true** or **false** and never changed, each sequence is finite; *agreement* – in a terminal state the out variable of each process contains the same value; *validity* – in a terminal state, for at least one process p, the values of $in.p$ and $out.p$ are the same.

Lemma 3 *The consensus problem is not termination-symmetric.*

Proof. We show that $\mathcal{CN}$ does not comply with the definition of termination-symmetry. Namely, we prove that an arbitrary subset $\mathcal{R}$ of $\mathcal{CN}$ is either not input-complete or does not contain a symmetric set A of terminal states augmented with input-displaced states.

According to the termination property of $\mathcal{CN}$, every sequence of $\mathcal{CN}$ is finite. Thus, if $\mathcal{R}$ is input-complete A is not empty. Observe that due to the agreement property, $\mathcal{CN}$ has only two terminal states. In one of these states all processes output **true**, and in the other – **false**. Let us denote these states t and f respectively.

Suppose that A includes only one of the terminal states. We assume without loss of generality that A includes t only. Let us consider an arbitrary external sequence γ where the input to each process is **false**. Due to the termination property of $\mathcal{CN}$, γ is finite. According to the validity property γ terminates in f. Hence γ does not belong to $\mathcal{R}$. Which means that $\mathcal{R}$ is not input-complete.

Let us now suppose that A includes both t and f. Let us consider two processes of the program: p and q. In t, $out.p$ is **true**. In f, $out.q$ is **false**. Let g be a state where $out.p =$ **true** and $out.q =$ **false** and the values for the variables of the other processes are taken from either t or f. According to Definition 1, any symmetric set of states containing t and f also contains g. Observe however, that in $\mathcal{CN}$ all input is given at the beginning of the sequence, hence $\mathcal{CN}$ does not contain input-displaced states. Also, by the validity property of $\mathcal{CN}$, g is not a terminal state. Thus, A does not contain g. However, this means that A is not symmetric.

That is, either R is not input-complete or A is not symmetric. The lemma follows. □

From Theorem 1 and Lemma 3 we obtain the following theorem.

Theorem 4 *The consensus problem does not allow a solution in a message-passing system such that this solution is both stabilizing and terminating.*

We observe that a number of problems share with $\mathcal{CN}$ the lack of terminating and stabilizing solutions. Certainly, the abundance of such problems caused the belief in the impossibility of such programs that we set out to disprove in this paper. We list some of the most well known problems in the proposition below.

Proposition 1 *The following problems do not allow solutions in message-passing systems that are both stabilizing and terminating: leader election, graph coloring, spanning tree construction, renaming and routing topology construction.*

To give an intuition as to how the theorem applies to these problems we consider leader election. Let the problem be stated such that the process with the smallest identifier is to be elected the leader. The input for the problem is a set of process identifiers and the output is the id of the leader. The system should terminate in a state where each process outputs this id. Observe that a non-trivial problem statement allows inputs where the identifier of the leader varies. Suppose there are two

potential leader identifiers: id_1 and id_2 and two corresponding outputs. To make the set of terminal states symmetric, there must exist a terminal state where some processes output id_1 and the others – id_2. This, however, would violate the specification of the problem of leader election. Further reasoning proceeds similar to the proofs of Lemma 3 and Theorem 4.

Notice that if the specification of $\mathcal{CN}$ is altered to allow input-displaced states then the modified problem may have a terminating and stabilizing solution. Consider a specification such that every output state, where processes do not agree on the output value, is displaced with extra input which sets all input variables to the same value. Such a specification allows a solution where each process simply copies from its in to its out. In this case, if the problem gives the same input to all processes, then the same output is produced. Otherwise, the problem gives extra uniform input to make processes agree on the output. Such problem specification, however, does not convey the nature of the original consensus problem, since all the agreement decisions are externalized.

The positive results for the mutual exclusion, reliable transmission and the above modification to the consensus problems may lead the reader to assume that for the problem to allow a stabilizing and terminating solution, the specification has to either have only one terminal state or have input-displaced states. It also seems that all transformational problems such as consensus, leader election, etc. do not allow such solution. Recall that a transformational problem does not have input from the environment after the initial state. This perception is false.

For a transformational problem to allow termination, the terminal states of the problem have to form a Cartesian product of the terminal states of the processes. Consider the following example specification. It has two processes p and q, each of which has one boolean input variable in and one boolean output variable out. Consider the following states of p:

$$\{(in.p = \textbf{true}, out.p = \textbf{true}),$$
$$(in.p = \textbf{false}, out.p = \textbf{false})\}$$

and q:

$$\{(in.q = \textbf{true}, out.q = \textbf{true}),$$
$$(in.q = \textbf{false}, out.q = \textbf{true})\}.$$

The set of terminal states of the specification is the Cartesian product of the above states of p and q. Therefore, there are four terminal states. Observe that this problem is transformational, hence there are no input-displaced states. Observe that this set is input-complete and this specification is termination-symmetric. Thus, due to Theorem 1, it allows a stabilizing and terminating solution. The trivial solution for the problem is where p copies the value from $in.p$ to $out.p$ and q ignores the input and assigns **true** to $out.q$.

5 Stabilizing and terminating programs

In this section we describe a stabilizing and terminating solution to the mutual exclusion problem ($\mathcal{MX}$) in message passing systems. Recall that the problem itself was specified in Sect. 4. We call our solution $\mathcal{TRA}$ because it is loosely based on Ricart-Agrawala's algorithm [14].

```
process p
const
    P, set of process identifiers of the system
var
    request : boolean, input, true if the CS needed
    myts, timestamp of CS request
    L : L ⊂ P, set of locked processes
    needcs : boolean, true if the CS needed
    ts, timestamp of received message
    needrep : boolean, true if message needs reply
param
    q : q ∈ P

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

Fig. 2. Process of $\mathcal{TRA}$

$\mathcal{TRA}$ description. The code for a process p of $\mathcal{TRA}$ is shown in Fig. 2. To make the program easy to understand we tried to present the code as simple as possible. Notice that we omit the environment actions that update the input. We discuss how to make $\mathcal{TRA}$ more realistic at the end of the section.

The idea of $\mathcal{TRA}$ is as follows. The program uses unbounded timestamps. A process with the smallest timestamp in the system is allowed to enter the CS. A process requesting the CS is in *CS contention*. When a process needs to enter the CS it selects a new, greater than previous, timestamp and sends messages bearing this timestamp to all other processes. When process p gets a message from another process q with a timestamp greater than p's timestamp, p *locks* q. When p locks all the other processes in the system, p enters the 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 variable $request$ is **true** when the environment needs to access the CS. To enter the CS the process obtains a new timestamp, stores it in $myts$, sets $needcs$ to **true** to indicate that it is in the CS contention, and empties the set of locked processes

L (see action $r1$). If a process is not in L ($r2$) and p is in CS contention, p sends a message to this process requesting reply.

When p receives a message from another process q ($r3$), it compares its timestamp with q's. If p is in CS contention and q has a greater timestamp, p adds q's identifier to the set of locked processes. If p is not in CS contention it gets a newer timestamp in an attempt to exceed the timestamp of q and let q execute the CS. If q requests reply, p sends it a message with its own timestamp. When p locks all processes in the system ($r4$), it executes the CS. After the CS execution p sets $needcs$ to **false**. Notice that to simplify the presentation we dispense with *enter* output variable described in $\mathcal{MX}$ specification and assume that a process completes the CS in one action ($r4$).

Correctness proof. Denote $C.p.q$ the channel from p to q.

Lemma 4 If a computation starts from a state where $needcs.p$ is **true** then the computation contains a state where $needcs.p$ is **false**.

Proof. To prove the lemma we need to show that if $needcs.p$ is **true** then p eventually executes $r4$. $r4$ is executed when $L.p = P$. Note that a process identifier is not removed from $L.p$ while p is in CS contention. Thus, we need to show that each process identifier is eventually added to $L.p$.

Observe that $myts.p$ does not increase if p is in CS contention. Also when p is in CS contention, p continuously sends messages to each process that is not in $L.p$ requesting reply. When another process q receives such a message, q replies with its own timestamp. When the reply is received, p adds q to $L.p$ if q has higher timestamp than p. Thus, p eventually adds to $L.p$ every process whose timestamp is greater than $myts.p$. If p has the smallest timestamp in the system, then eventually $L.p = P$ and $r4$ is executed.

Let us consider the case where p does not have the smallest timestamp in the system. Let q be the process with the smallest timestamp. According to the discussion above, if q is in CS contention it eventually enters the CS. When q exits the CS, q increases its timestamp. If q is not in CS contention and it receives a message from p requesting reply, q also increases its timestamp. Thus, in either case, when p is in CS contention the smallest timestamp in the system keeps increasing. Eventually p is going to have the smallest timestamp in the system which allows p to execute the CS. The lemma follows. □

We show that the following predicate (denoted I_{TRA}) is an invariant of $\mathcal{TRA}$. That is, every computation that starts from a state conforming to I_{TRA} satisfies $\mathcal{MX}$.

> The timestamp of the last message in $C.p.q$ is at most $myts.p$, and
> the messages in $C.p.q$ are in non-decreasing order of timestamps, and
> if $p \in L.q$ and $needcs.q$ is **true**
> then the timestamp of any message in $C.p.q$ and $myts.p$
> are greater than $myts.q$.

Lemma 5 *$\mathcal{TRA}$ stabilizes to I_{TRA}.*

Proof. We observe that if a state conforms to I_{TRA}, the execution of any action keeps the program in I_{TRA}. That is, I_{TRA} is closed.

To show convergence of the first two conjuncts of I_{TRA} we observe that $myts$ of any process can never decrease. The message always carries the value of sender's $myts$. Thus, the messages in channels will eventually be in non-decreasing order and the first two conjuncts of I_{TRA} become true. To show convergence of the last conjunct it suffices to demonstrate that $needs.q$ is eventually **false**. This is indeed the case due to Lemma 4. □

Theorem 5 *$\mathcal{TRA}$ is a stabilizing and terminating solution to the mutual exclusion problem.*

Proof. According to Lemma 5, $\mathcal{TRA}$ stabilizes to I_{TRA}. To prove the theorem we need to show that I_{TRA} is indeed the invariant for $\mathcal{MX}$ and that $\mathcal{TRA}$ is terminating. To prove the invariance of I_{TRA}, we show that any computation of $\mathcal{TRA}$ starting from a state conforming to I_{TRA} satisfies the safety and liveness properties of $\mathcal{MX}$. To show safety we observe that I_{TRA} prohibits states where more than one process have the CS action enabled. Liveness follows from Lemma 4.

Let us now focus on termination. To prove termination we need to show that every computation of $\mathcal{TRA}$ where the environment makes finitely many CS requests is itself finite. Due to liveness, $\mathcal{TRA}$ satisfies every CS request and every process eventually gets out of CS contention (both $request$ and $needcs$ are **false**).

We would like to show that the computation where every process is out of CS contention contains a state that has no messages in the channels. Indeed, if the process is not requesting the CS its actions $r1$, $r2$ and $r4$ are disabled. The execution of $r3$ may result in the message being sent. However, the execution of $r3$ results in a message being received. Hence, when none of the processes in the system are requesting the CS, the number of messages in the system may only decrease. Notice that when a process is out of CS contention, every message it sends carries **false** flag. This means that the receipt of such message does not result in another message being sent. Thus eventually, there will be no messages in the channels.

If the processes are out of CS contention and there are no messages in the channels, all the process actions are disabled, i.e. the system is in fixpoint. Thus, every computation with finitely many CS requests terminates. □

$\mathcal{TRA}$ implementation and efficiency improvements. Let us discuss ways to make $\mathcal{TRA}$ more realistic. When joining CS contention, the requesting 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 the CS, thus termination is preserved.

To speed up CS entry, each process p can maintain a set of processes that lost CS contention. After executing the CS, p can send messages to these processes to let them know that they can proceed with the CS.

Bounded overtaking property specifies that each process enters the CS only a finite number of times before another process requesting the CS enters it. Note that $\mathcal{TRA}$ does not guarantee bounded overtaking. To amend that the processes can execute Lamport's logical clock algorithm [11] to synchronize the timestamps of each process.

Observe that $\mathcal{TRA}$ uses unbounded timestamps. Unbounded variables present a problem for implementation of stabilizing programs. In a concrete computer architecture such variables have to be implemented using finite counters. This adds a constraint that the abstract program does not consider: the concrete program may have to stabilize from a state where the counter value reached its limit. A number of researchers [2,3,9] studied ways of dealing with this issue in the context of stabilization. We believe that the problems of bounding the variables and adding termination in the context of stabilization are orthogonal. Hence, bounded stabilizing and terminating programs are possible. For example, a bounded solution to $\mathcal{RT}$ can be achieved by using the techniques studied by Howell et al [9].

6 Making a stabilizing program terminating

All the fixpoints of the stabilizing program presented in the previous section are legitimate. This, however, is not necessarily always the case. As we discussed earlier, if a fixpoint of a stabilizing program is not legitimate, this fixpoint has to project to an input-displaced external state. We claim that termination can be added to such a program in the following manner. If a process does not have internal 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 non-terminating stabilizing version of alternating-bit protocol ($\mathcal{SABP}$) and add termination to it. $\mathcal{SABP}$ is a solution to a infinite version ($\mathcal{IRT}$) of the reliable transmission problem ($\mathcal{RT}$) presented in Sect. 4. $\mathcal{IRT}$ excludes the finite sequences of $\mathcal{RT}$. $\mathcal{SABP}$ has illegitimate fixpoints and relies, therefore, on environmental input to move it into legitimate states. Our termination addition results in stabilizing and terminating program – $\mathcal{TABP}$. $\mathcal{TABP}$ does not rely on finite environmental input and solves $\mathcal{RT}$.

6.1 Stabilizing alternating-bit protocol

Recall that the objective of a solution to $\mathcal{RT}$ is to pass packets from sender process p to the receiver process q over lossy channels. To model message loss in channels 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.

$\mathcal{SABP}$ description. The code for the sender and receiver of $\mathcal{SABP}$ is shown in Figs. 3 and 4 respectively. As with previous examples we omit the input actions in our presentation of the program.

Processes p and q maintain infinite queues *in* and *out* respectively. The queues are called send buffer and receive

```
process p

var
      in : queue, input, packets to transmit
      ns : integer, sequence number of the last message sent
      i : integer, sequence number of received message
*[
(p1)  receive ack(i) ⟶
            if i = ns then
                  dequeue(in),
                  ns := ns + 1
   []
(p2)  ¬empty(in) ⟶
            send data(first(in), ns)
]
```

Fig. 3. Sender process of $\mathcal{SABP}$

buffer. Function *enqueue* appends a packet to the end of the buffer. Function *dequeue* removes a packet from the head of the buffer and returns the removed 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. To keep track of processed packets, p and q also maintain (infinite) integer counters ns and nr respectively. Counter ns keeps the sequence number (SN) of the packet last sent, nr – last SN received.

When the send buffer is not empty, p sends a *data* message carrying a packet and its SN ($p2$). When q receives a *data* carrying an SN different from the SN last received ($q1$), the packet that the message carries is appended to receive buffer and delivered. The receipt of a message is acknowledged by sending *ack* back to p. When the acknowledgment of the last message sent ($p1$) is received, ns is incremented and a new message can be sent.

Why $\mathcal{SABP}$ is not terminating. Note that $\mathcal{SABP}$ has illegitimate fixpoints. Namely the fixpoints where $ns = nr$. A computation starting from any of these fixpoints violates safety: the receiver does not deliver the first message submitted for transmission. To stabilize from any of these fixpoints $\mathcal{SABP}$ needs extra input. Which means that illegitimate fixpoints have to project onto an input-displaced external states. Notice that all (legitimate and illegitimate) fixpoints of $\mathcal{SABP}$

```
process q
var
      out : queue, output, delivered packets
      i : integer, sequence number of received message
      m : packet, received packet
      nr : integer, sequence number
            of the last message received
*[
(q1)  receive data(m, i) ⟶
            if i ≠ nr then
                  enqueue(m, out),
                  nr := i,
            send ack(i)
]
```

Fig. 4. Receiver process of $\mathcal{SABP}$

project to the same external state. This is the state where the input and output queues are empty. Computations terminated by legitimate fixpoints project onto sequences terminated by this terminal state. By definition, an input-displaced external state cannot be a terminal state. Hence, $\mathcal{SABP}$ cannot be a stabilizing solution to a problem with this terminal state. Since this is the only terminal state of $\mathcal{RT}$, $\mathcal{SABP}$ is not terminating. We prove, however, that $\mathcal{SABP}$ is a stabilizing solution to an infinite version of the reliable transmission problem – $\mathcal{IRT}$.

$\mathcal{SABP}$ correctness proof. Denote by $C.qpq$, the concatenation of sequences of messages in the two channels: $C.q.p$ followed by $C.p.q$. We show that predicate I_{SABP} defined as $R \wedge (A \vee B)$ (where R, A and B are as defined below) is an invariant for $\mathcal{SABP}$. We call A and B *rotating predicates*.

$$\begin{array}{l}\text{The sequence number carried by a message is at most as large as } ns, \text{ and}\\ \text{messages in } C.qpq \text{ are in non-decreasing order of time-stamps, and}\\ nr \leq ns, \text{ and}\\ \text{the sequence number of any message in } C.p.q \text{ is at least as large as } nr, \text{ and}\\ \text{the sequence number of any message is } C.q.p \text{ is at most as large as } nr.\end{array} \tag{R}$$

$$\begin{array}{l} ns \neq nr \wedge \\ \big((data(m,i) \in C.p.q \wedge i = ns) \\ \quad \Rightarrow (\neg \mathit{empty}(in) \wedge m = \mathit{first}(in))\big) \wedge \\ \big((\forall ack(i) \in C.q.p) \Rightarrow (i \neq ns)\big) \end{array} \tag{A}$$

$$\begin{array}{l} ns = nr \wedge \\ \neg \mathit{empty}(in) \wedge \\ \big((data(m,i) \in C.p.q) \Rightarrow (i = ns \wedge m = \mathit{first}(in))\big) \end{array} \tag{B}$$

Lemma 6 *$\mathcal{SABP}$ converges to R.*

Proof. We observe that if a state conforms to R, the execution of any action keeps the system in R (i.e. R is closed). Let us focus on the convergence part. When p sends a *data*, the SN it carries is equal to ns. When q receives a *data* and sends an *ack* back, it copies the SN from the *data* to the *ack* and to nr. Thus, after p successfully sends a $data$ and q receives it, $nr \leq ns$ and the SN of any message in $C.p.q$ is no less than nr. q acknowledges every message it receives from p. After q acknowledges the first $data$, that p actually sent during the computation (it was not in $C.p.q$ initially), the SN of the message in $C.q.p$ is no greater than nr. That is $\mathcal{SABP}$ converges to R provided that eventually p succeeds in sending a $data$ and q in acknowledging it.

We assume that the environment requests to transmit infinitely many packets. This means that in is not empty in infinitely many states of a computation of $\mathcal{SABP}$. If in is not empty, $p2$ is enabled. Moreover, in can become empty only when $p2$ is executed. According to weak fairness assumption of action execution in a computation, $p2$ must be executed infinitely many times in a computation of $\mathcal{SABP}$. By message transmission fairness assumption the sends also have to succeed in entering $data$ into $C.p.q$ infinitely often. This leads to

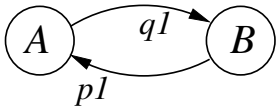

Fig. 5. Sets of legitimate states and inter-set transitions for I_{SABP}

q receiving $data$, sending ack infinitely often and eventually succeeding. □

Lemma 7 *$\mathcal{SABP}$ converges to I_{SABP}*

Proof. We observe that if a state conforms to R and one of the rotating predicates then the execution of any action either keeps the system in the same rotating predicate or moves the system to the other one. Thus, the I_{SABP} is closed. The diagram of transitions between rotating predicates is shown in Fig. 5. Loopback transitions are omitted.

Let us discuss convergence. According to Lemma 6, $\mathcal{SABP}$ converges to R. By argument similar to the one supplied in Lemma 6 we ascertain that $p1$ is executed infinitely often in any computation of $\mathcal{SABP}$. Note that if $p1$ is executed when R holds, the state after the execution conforms to A. Hence, $\mathcal{SABP}$ converges to I_{SABP}. □

Theorem 6 *$\mathcal{SABP}$ is a non-terminating stabilizing solution to the infinite reliable transmission problem.*

Proof. Lemma 7 demonstrates that $\mathcal{SABP}$ converges to I_{SABP}. To prove the theorem we show that a computation starting in a state conforming to I_{SABP} satisfies the properties of $\mathcal{IRT}$.

Safety: We construct a queue *Ch* as follows. If the system conforms to A, *Ch* contains *in* appended to the tail of *out*. If the system conforms to B, *Ch* contains all packets of *in* except the head appended to the contents of *out*. To demonstrate the safety property we show that if I_{SABP} holds, none of the actions change the contents of *Ch*.

Indeed, if A holds the only action that changes the state of the queues is $q1$. $q1$ appends the first packet of *in* to *out*. When *out* is updated the system moves from A to B and *Ch* is not affected. When B holds the only internal action that changes queues is $p1$. $p1$ deletes the first packet of *in*. However, in B the first packet of *in* is not in *Ch*. Thus, *Ch* remains unchanged.

Liveness: To prove liveness we show that every packet in *in* is eventually appended to *out*. To this end, it suffices to show that in every computation a packet at the head of *in* is eventually deleted.

Let us assume the opposite, suppose there is a computation σ starting in a state conforming to I_{SABP} such that a packet m is at the head of the queue in every state of the computation. Since *in* is not empty then $p2$ is enabled and must be executed infinitely many times during σ. $p2$ attempts to send a $data$ to q carrying the SN equal to ns. By transmission fairness assumption such attempts must succeed infinitely often. We apply similar reasoning to q to demonstrate that sending of an ack message to p carrying the SN equal to ns must also succeed infinitely often. When such ack is received the first packet in *in* is deleted. This contradicts our assumption.

Thus, $\mathcal{SABP}$ conforms to all properties of $\mathcal{IRT}$. □

process p

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

Fig. 6. Sender process of $\mathcal{TABP}$

process q

```
    *[
(q1)    receive data(m, i) ⟶
            if i ≠ nr then
                enqueue(m, out),
                nr := i,
            send ack(i)
      []
(q2)    receive dummy(i) ⟶
            if i ≠ nr then
                nr := i,
            send dack(i)
    ]
```

Fig. 7. Receiver process of $\mathcal{TABP}$

6.2 Stabilizing and terminating alternating-bit protocol

Program $\mathcal{SABP}$ is not terminating because it has illegitimate fixpoints (where $ns = nr$). To add termination to $\mathcal{SABP}$ we incorporate a procedure that ensures $ns \neq nr$ before the system starts processing additional input. Thus, the input is processed correctly which legitimizes the fixpoints and makes the program terminating.

$\mathcal{TABP}$ description. The sender and receiver process for the terminating and stabilizing version of the alternating-bit protocol ($\mathcal{TABP}$) are shown in Figs. 6 and 7 respectively. We skip the declaration of the variables in the figures since they are the same as in $\mathcal{SABP}$. The only new variable is a boolean go which indicates whether it is safe to start transmitting the packets.

The modification of the design of $\mathcal{TABP}$ compared to $\mathcal{SABP}$ is as follows. When p notices that the input buffer is empty, it sets go to **false** (see action $p3$) to indicate that $\mathcal{TABP}$ may potentially be in a fixpoint. When go is **false** and the input buffer is non-empty again, p sends a $dummy$ message ($p4$). The purpose of this message is to synchronize the values of ns and nr and avoid potential incorrect message delivery that $\mathcal{SABP}$ is prone to. When q receives such a message ($q2$), it updates the value of nr and acknowledges this message with $dack$. When p receives $dack$ ($p5$), p sets go to **true** and starts transmitting packets to q.

$\mathcal{TABP}$ correctness proof. We show that predicate I_{TABP} defined as $R' \wedge (A' \vee B' \vee C \vee D)$ (where R' and A' through D are as defined below) is an invariant of $\mathcal{TABP}$. Similar to $\mathcal{SABP}$ predicates A' through D are *rotating predicates*.

Sequence number carried by a message is no greater than ns, and
messages in $C.qpq$ are in non-decreasing order of time-stamps, and

if $ack(i)$ is in $C.q.p$ then $i \leq nr \leq ns$ and for every $data(m, j)$ in $C.p.q$ it follows that $j \geq nr$. (R')

$(ns \neq nr) \wedge go \wedge$
$\big((data(m, i) \in C.p.q : i = ns)$
$\quad \Rightarrow (\neg empty(in) \wedge m = first(in))\big) \wedge$
no *ack*(i), *dack*(i), or *dummy*(i) such that $i = ns$. (A')

$(ns = nr) \wedge go \wedge$
$\neg empty(in) \wedge$
$\big((data(m, i) \in C.p.q) \Rightarrow (i = ns \wedge m = first(in))\big) \wedge$
no *dack*(i), or *dummy*(i) such that $i = ns$. (B')

$(ns \neq nr) \wedge \neg go \wedge$
no *ack*(i), *dack*(i), or *data*(i) such that $i = ns$. (C)

$(ns = nr) \wedge \neg go \wedge$
$\big((dummy(i) \in C.p.q) \Rightarrow (i = ns)\big)$
no *ack*(i), or *data*(i) such that $i = ns$. (D)

Lemma 8 *$\mathcal{TABP}$ converges to I_{TABP}.*

Proof. Similar to the proof of Lemma 6 we observe that if the state of $\mathcal{TABP}$ conforms to R' then the execution of any action keeps the system in R'. That is, R' is closed. To show the closure of I_{TABP} we observe that if a state of the system conforms to R' and one of the rotating predicates then the execution of any action either keeps the system in the same

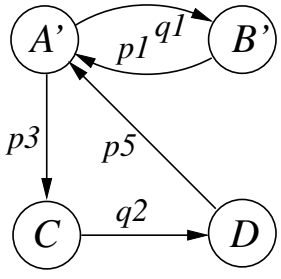

Fig. 8. Sets of legitimate states and inter-set transitions for I_{TABP}

rotating predicate or moves it to another rotating predicate. The diagram of transitions between rotating predicates is shown in Fig. 8.

To prove convergence we first consider finite computations. We prove that the last state of any finite computation conforms to the invariant. Since the last state of a computation does not have any actions enabled then $C.p.q$ and $C.q.p$ are empty, *in* and *out* are empty and $go =$ **false**. If $ns \neq nr$ – the state conforms to R' and C; if $ns = nr$ the state conforms to R' and D.

Let us now consider infinite computations. The convergence of $\mathcal{TABP}$ to R' in this case can be proven similar to the convergence of $\mathcal{SABP}$ to R (cf. Lemma 6). Note that if a computation starts in a state conforming to R' and either $p1$ or $p5$ are executed at least once, then the state after the execution conforms to A'. Suppose a computation that starts in a state conforming to R' but does not contain the execution of either $p1$ or $p5$. Similar to the proof of Lemma 7, it can be demonstrated that if the computation starts in a state where $go =$ **false** or contains the execution of $p3$, then the computation contains a state conforming to D. Again, similar to the proof of Lemma 7, it can be shown that in this case the computation contains a state conforming to B'. □

Theorem 7 $\mathcal{TABP}$ is a stabilizing and terminating solution to the reliable transmission problem.

Proof. According to Lemma 8, $\mathcal{TABP}$ stabilizes to I_{TABP}. To prove the theorem we need to show that any computation starting from a state conforming to I_{TABP} satisfies safety, liveness properties of $\mathcal{RT}$ and that $\mathcal{TABP}$ is terminating. The argument for safety satisfaction is similar to the one given in the proof of Theorem 6.

Liveness: As in the proof of Theorem 6 it is sufficient to show that in every computation the packet at the head of *in* is eventually deleted. If a computation starts in a state conforming to A' or B' the argument is similar to the non-terminating case. Suppose a computation starts in a state conforming to either C or D. If a computation starts in a state conforming to C and *in* is non-empty, $p4$ is enabled. Applying transmission fairness property we can show that the system eventually moves to D and then to A'.

Termination: To prove termination we show that any computation where in is finite, ends in a fixpoint. We start by observing that if the system starts from a fixpoint and in is empty then the system remains in either C or D.

If in is not empty in the beginning of the computation, due to liveness, the computation eventually contains a state where in is empty. Let us consider a computation starting from such a state. Since *in* is empty no messages are sent by p in this computation. After q receives all messages and p receives the resultant acknowledgments, the communication channels remain empty for the rest of the computation. If go is **true**, $p3$ is enabled. After $p3$ is executed go is **false**. Since *in* is empty, go cannot be changed to **true**. Thus, the system arrives at a fixpoint in C or D and remains there. □

Sufficient condition. The fixpoint cleaning technique used in $\mathcal{TABP}$ can be generalized to yield a sufficient condition for a problem to have a stabilizing and terminating solution in a message passing system. This condition will be based on a generic fixpoint cleaning program. Similar to $\mathcal{TABP}$, if a process receives input and the process suspects it is in an illegitimate fixpoint, this process initiates a global state corrective action. Observe that such action should not interfere with the performance of the main program and be resilient to launching from multiple places simultaneously. Designing such a program is an interesting further avenue of research.

7 Implications and impact of the unification

The results presented in this paper are of both basic and practical importance.

The formal study of the combination of stabilization and termination draws attention to termination as a fundamental aspect underlying efficient design of non-masking fault-tolerant programs. The negative results we obtain help demarcate the realm of possibility. On the other hand, the existence of non-trivial stabilizing and terminating programs increases the attractiveness of stabilization as an approach to handling faults in a distributed system. The existence of such programs should prompt the researchers to reevaluate their views on stabilization.

From a practical standpoint, the focus on termination leads to stabilizing programs that do not carry out communication when no input needs to be processed. Such a program performs illegitimate state detection and corrective actions only when additional input arrives. We illustrate this approach in the paper by adding termination to a stabilizing alternating-bit protocol. We also discuss how it can be generalized to lead to a sufficient condition for the existence of a terminating and stabilizing solution to a certain problem.

We would like to contrast this approach of achieving stabilization with a more traditional one where detection and correction is done periodically regardless of the input. Demand-driven stabilizing program consumes resources only when useful work is carried out. However, such program adds control overhead in direct proportion to the amount of input processed. Conversely, periodic checking and correction adds constant load to the system regardless of input. We consider both cases as extremes in the variety of scheduling policies for detection and correction. To this day, the selection of an appropriate policy is left up to the designer of a concrete system. We deem the study of stabilization-inducing scheduling and the relation between demand-based and time-based techniques of enabling stabilization to be a promising direction of future research.

References

1. Afek Y, Brown GM: Self-stabilization over unreliable communication media. Distributed Computing 7, 27–34 (1993)
2. Arora A, Gouda MG: Distributed reset. IEEE Transactions on Computers 43, 1026–1038 (1994)
3. Awerbuch B, Patt-Shamir B, Varghese G: Bounding the unbounded (distributed computing protocols). In: Proceedings IEEE INFOCOM 94 The Conference on Computer Communications, 1994, pp. 776–783
4. Bui A, Datta AK, Petit F, Villain V: State-optimal snap-stabilizing PIF in tree networks. In: Proceedings of the Fourth Workshop on Self-Stabilizing Systems (published in association with ICDCS99 The 19th IEEE International Conference on Distributed Computing Systems), 1999, pp. 78–85. IEEE Computer Society

5. Dijkstra EW, Scholten CS: Termination detection for diffusing computations. Information Processing Letters 11(1). 1–4 (1980)
6. Dolev S: Self-Stabilization. MIT Press, 2000
7. Dolev S, Gouda MG, Schneider M: Memory requirements for silent stabilization. In: PODC96 Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, 1996, pp. 27–34
8. Gouda MG, Multari N: Stabilizing communication protocols. IEEE Transactions on Computers 40, 448–458 (1991)
9. Howell RR, Nesterenko M, Mizuno M: Finite-state self-stabilizing protocols in message passing systems. Journal of Parallel and Distributed Computing 62(5), 792–817 (2002)
10. Jayaram M, Varghese G: The complexity of crash failures. In: Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing, pp. 179–188, Santa Barbara, California, 21–24 August 1997
11. Lamport L: Time, clocks and ordering of events in distributed systems. Communications of the ACM 21(7), 558–564 (1978)
12. Mizuno M, Nesterenko M: A transformation of self-stabilizing serial model programs for asynchronous parallel computing environments. Information Processing Letters 66(6), 285–290 (1998)
13. Nesterenko M, Arora A: Stabilization-preserving atomicity refinement. In: Proceedings of the 13th International Symposium on Distributed Computing, 1999, pp. 254–268
14. Ricart G, Agrawala AK: An optimal algorithm for mutual exclusion in computer networks. Communications of the ACM 24(1), 9–17 (1981)
15. Schneider FB: On Concurrent Programming. Springer, 1997