Finite-State Self-Stabilizing Protocols in Message-Passing Systems

Rodney R. Howell
Dept. of Computing and Information Sciences
Kansas State University
Manhattan, KS 66506, USA
howell@cis.ksu.edu

Mikhail Nesterenko
Mathematics and Computer Science Dept.
Kent State University
Kent, OH 44240, USA
mikhail@mcs.kent.edu

Masaaki Mizuno
Dept. of Computing and Information Sciences
Kansas State University
Manhattan, KS 66506, USA
masaaki@cis.ksu.edu

Abstract

We define a finite-state message-passing model using guarded commands. This model is particularly appropriate for defining and reasoning about self-stabilizing protocols, due to the well-known result that self-stabilizing protocols on unbounded-channel models must have infinitely many legitimate states. We argue that our model is more realistic than other models, and demonstrate its use with a simple example. We conclude by discussing how self-stabilizing protocols defined on this model might be implemented directly on actual networks.

1. Introduction

Dijkstra introduced the notion of *self-stabilization* in the context of distributed systems [3]. A system is defined to be *self-stabilizing* (SS) with respect to a set of legitimate states if regardless of its initial state, the system is guaranteed to arrive at a legitimate state in a finite number of steps and will never leave legitimate states after that. Thus, an SS system need not be initialized and is able to recover from transient failures by itself.

This paper addresses one of the major obstacles for theoretical and practical development of message-passing self-stabilizing systems; that is, Gouda and Multari[7] have shown that SS communication protocols with unbounded channel models cannot be finite-state. Several researchers have informally discussed that this proof is due to the model of computation used, rather than to the fundamental nature of distributed systems [1, 2] (cf. [6]). However, as we argue in Section 2, there is no formal model that adequately captures the notion of bounded channels.

In this paper, we formally define a finite-state message-passing model and demonstrate how to develop finite state protocols in the model. Since Gouda and Multari's guarded-command model is elegant and easy to program, we define our model based on their model, except that the channels are bounded and lossy. Like their model, we do not use a notion of time in this model. However, we place some restrictions on the timeout predicates of the Gouda and Multari's model, which were arbitrary predicates on the global state.

We then present a finite-state SS Alternate Bit Protocol (ABP) using this model. The purpose of this is two-fold. One is to demonstrate how to develop SS protocols and formally prove their correctness in our model. Another is to provide our model with reliable communication interface.

In order to strengthen our argument that SS protocols defined for our model can be directly implemented on actual networks, we conclude the paper with a discussion of how the remainder of the implementation might be done and with further research plans to investigate this possibility.

2. Related Work

Several different models are used in self-stabilizing message passing protocols. The following assumptions characterize the models.

1. Communication channels are bounded or unbounded: Almost all models assume the use of unbounded channels. However, Gouda and Multari proved that with a

very reasonable liveness assumption, any SS protocol defined using unbounded channels must have infinitely many legitimate states [7].

2. Communication channels are reliable or unreliable (lossy): Under the assumption that channels are reliable, message loss is considered to be a transient failure; that is, message loss is not considered while a system stabilizes.

   When bounded channels are assumed, either message loss or process blocking will occur when a process sends a message to a full channel. Therefore, either message loss or process blocking must be formally modeled.

3. A timeout mechanism is assumed or not: A protocol for a message-passing system is called message driven if any action of the processor is initiated by a message. A communication deadlock in a message driven system is a state in which all processors are waiting for messages but there are no messages on any link. A self-stabilizing system must assume arbitrary initial configurations, including a configuration with communication deadlock. For this reason, no non-trivial tasks can be performed by completely asynchronous, self-stabilizing, message driven systems [4, 7].

The following are models (characterized by the above categories) used in well-known distributed self-stabilizing protocols.

1. In [7], Gouda and Multari assume unbounded reliable channels with a timeout mechanism. The time-out mechanism preserves the message-driven structure of the protocol at the expense of compromising the complete asynchrony. Their timeout guard is defined by a predicate over global state. They presented an Alternate Bit Protocol (ABP) and a Sliding Window Protocol (SWP) in this model.

2. Katz and Perry [8] assume unbounded (reliable) channels without a timeout mechanism (their protocol structure is not message driven). Their protocol structure requires that at any configuration, there is at least one processor whose next operation is sending a message. In this model, they present an infinite protocol that transforms an arbitrary asynchronous protocol into an equivalent stabilizing protocol by centralized state checking and correction mechanism.

3. Afek and Brown's model in [1] assumes unbounded unreliable channels with a timeout mechanism. They present two finite ABPs in this model. However, they assume external devices (oracles) to generate an infinite aperiodic sequence and a random sequence, respectively. Therefore, the external devices are an infinite state machine and a probabilistic finite state machine, respectively.

   They claim without proof that a periodic sequence could be used if the channels were bounded. A proof of this claim would first require modifying their model so that the channels are bounded. A straightforward modification would be to simply assume that whenever the channel contents reach a certain bound, any additional message sent is lost. With such a model, it is possible to prove their claim.

4. In [4], Dolev, Israeli, and Moran presented three infinite-state token circulation algorithms using unbounded reliable channels. They assume either an abstract timeout device which detects communication deadlocks or that in every initial state, there is at least one message on some channel.

   The first algorithm uses unbounded counter values in messages. The second and third algorithms eliminate unbounded counters and instead use aperiodic sequence numbers generated by the XOR operation (thus, the size of messages is bounded). However, in the second algorithm, processor variables are unbounded, and in the third algorithm, the number of messages in channels grows without bound.

5. The model used by Awerbuch, Patt-Shamir, and Varghese in [2] assumes a reliable unit storage data link (UDL) which stores at most one message at any instant. The UDL uses blocking semantics. The UDL is implemented on top of Afek and Brown's stabilizing ABP [1] (see 3 above).

Having surveyed the models of computation used for describing message-passing protocols, we now consider what is typically being modeled by these abstractions. The underlying hardware includes not only the transmission line(s) between two nodes, but also buffers at both the sending and receiving end. Furthermore, a low-level protocol, such as ABP or SWP, is used to ensure reliable message delivery. Of course, it is necessary to design these low-level protocols using lower-level models, but for higher-level protocols, we would like to ignore such details as buffers and lost messages. However, their presence affects the behavior of channels in nontrivial ways, particularly when the system can be started in an arbitrary state (cf. [6]).

For example, suppose that initially a send buffer is full, but the corresponding receive buffer is not. Further suppose that a process attempts to send a message on the associated channel. Note that the channel — which abstracts the link and both buffers — is not holding its maximum number of messages. However, because there is no room for the new message in the send buffer, the channel must behave as if it were full; i.e., it must either block the sending process or drop a message. (See Section 4 for a more detailed example.) None of the models described above captures this kind of behavior, while at the same time providing some type of reliability.

In the remainder of this paper, we formally define a model that adequately captures the notion of bounded channels, and demonstrate development of finite state protocols in the model. Furthermore, we discuss how SS protocols developed for this model might be implemented on a real network so that self-stabilization is preserved.

3. Definitions

A protocol in our model is a set of two or more processes, each of which is described by the following syntax:

process ⟨process name⟩
  var ⟨variable name⟩ : ⟨variable type⟩; ...;
    ⟨variable name⟩ : ⟨variable type⟩;
  begin ⟨guarded command⟩ [] ... []
    ⟨guarded command⟩ **end**

such that the following hold:

- Each variable type is a finite domain of values. We allow simple subtyping; e.g., any message sent on any channel will be of type **message**, but different subtypes may be used to distinguish different kinds of messages.
- Each guarded command is of the form, ⟨guard⟩ $\rightarrow$ ⟨command⟩.
- Each guard is of one of the following forms:
  - **external**
  - ⟨boolean expression⟩
    or
  - ⟨local boolean expression⟩$\wedge$ **receive**(m, p)

  where each boolean expression is comprised of the variables of the system and predicates on the channel contents (see below), each local boolean expression is comprised only of local variables, *m* is a local variable of type **message**, and *p* is the name of another process. Guards that contain either nonlocal variables or predicates on the channel contents will be known as *timeout guards*.
- Each command is a composition of assignments to the variables of the process; **send** commands of the form **send**(*m, p*), where *m* is an expression of type **message** and *p* is the name of another process; sequencing; and **if**-statements.

The purpose of **external** guards is to give us the ability to compose protocols, or to otherwise abstract a reaction to an environment. The keyword **external** will evaluate to **true**, but fairness constraints will not apply to these guarded commands. We will call guarded commands with **external** guards *external* guarded commands. We will call all other guarded commands *internal* guarded commands.

Associated with each ordered pair of processes (p, q) is a *channel* C_{pq}. Associated with each channel are two integers: a positive *channel bound* B_{pq}, and a nonnegative *reliability threshold* T_{pq}, where $B_{pq} > T_{pq}$. (Typically, the channel bound will be much greater than the reliability threshold.) A *state* of the protocol is an assignment of a value to each of the variables in the protocol, and an assignment of a sequence of at most B_{pq} messages to each channels C_{pq}. A guarded command is *enabled* at a state s if its guard evaluates to **true** when the values from s are assigned to the variables and the channels (a **receive** evaluates to **true** if a message of the appropriate type is at the beginning of the sequence assigned to the specified channel). We assume a standard semantics of the execution of the guarded commands, except that a **send** may either *succeed* or *fail*, subject to the following conditions:

- If the number of messages in the channel is less than the reliability threshold, the **send** succeeds.
- If the number of messages in the channel is not less than the reliability threshold, but is less than the channel bound, the **send** may either succeed or fail (nondeterministically).
- If the number of messages in the channel is equal to the channel bound, the **send** fails.

If the **send** succeeds, it inserts its message at the end of the sequence assigned to the specified channel. If the **send** fails, it has no effect on the global state of the system. We call this model of communication *semi-reliable*.

The use of semi-reliable channels provides a great deal of expressive power and flexibility. First, consider the reliability threshold. The abstraction of reliable communication channels is implemented by low-level communications protocols such as the alternating-bit protocol or the sliding-window protocol. Such protocols provide reliable message delivery as long as the buffer of messages at the sending end does not overflow. Therefore, the reliability threshold corresponds to the sender's buffer size. For an unreliable channel, its realizability threshold is obviously 0 (0-reliable channel).

Next, consider the channel bound. In reliable communication protocols, some messages which have been successfully transmitted may be buffered at the receiving end. Such messages appear to be still in the channel. The receiving end cannot store more messages than the buffer will hold. Thus, there is a limit to the number of messages that a channel may hold while still guaranteeing faithful delivery. This limit is called the channel bound. For example, the channel bound of our SS implementation of ABP in the next section is at least two greater than the reliability threshold. Similarly, the channel bound of an unreliable communication channel is determined by buffer sizes and the number of messages actually being transmitted in a hardware wire.

When the number of messages in the channel is between the reliability threshold and channel bound, whether a next message can be sent reliably depends on where messages are located in the system. Suppose that the number of messages in the channel is exactly the reliability threshold. If all the messages are located in the sender's buffer, the protocol cannot accept any more message for reliable transmission. On the other hand, if some messages are in the receiver's buffer, the sender's buffer is not full and the protocol can accept more messages. In this way, if the number of messages in the channel is between the reliability threshold and channel bound, the send may either succeed or fail. Therefore, we claim that our model accurately reflects real networks.

Typically, bounds on both of these values are relevant to the correctness of an SS protocol: a protocol will usually require a minimum reliability threshold in order to guarantee that it works correctly after stabilizing, and it will usually require an upper bound on the capacity of the channels in order to guarantee stabilization. The way we have defined our model allows us to specify a reliability threshold that may be smaller than the reliability threshold actually guaranteed by the underlying system, and it allows us to specify a channel bound that may be larger that the actual channel capacity. Once a protocol designed for the channel is proved correct under such reliability threshold and channel bound, a correct implementation of the model will guarantee that the protocol runs correctly on the actual network.

To complete the definition of our model, we define a *computation* to be a (possibly infinite) string of the form $s_0 \overset{GC_1}{\to} s_1 \overset{GC_2}{\to} \cdots$ such that

- for each i, s_i is a state;
- for each i, GC_{i+1} is guarded command enabled at s_i, such that executing GC_{i+1} at s_i yields s_{i+1}; and
- if the string is finite, it ends in a state.

A computation σ is said to be *maximal* if either it is infinite or there is no other computation that has σ as a prefix. σ is said to be *fair* if each of the following hold:

- If the computation is infinite and an internal guarded command GC is enabled at all but finitely many s_i, then GC appears infinitely often in the computation (i.e., *weak fairness* holds for internal guarded commands).
- If infinitely many **send**s are attempted on the same empty channel, infinitely many must succeed. Note that this condition is nontrivial only for 0-reliable channels.

Let P be a set of states (or equivalently, a predicate on states). P is said to be *closed* if for every $s \in P$ and every guarded command GC, $s \overset{GC}{\to} s' \in P$. $R \subseteq S$ is said to *converge* to a closed subset P of S if every maximal fair computation from a state in R contains a state in P. A protocol $\mathcal{P}$ is said to be *P-stabilizing* if **true** converges to P in $\mathcal{P}$. When the predicate P is clear from the context, we will simply state that $\mathcal{P}$ is *self-stabilizing (SS)*.

The above definitions are sufficient for defining SS protocols and proving them to be SS; however, they allow us to define protocols that cannot be implemented in real network. In particular, the definition of a timeout is too powerful, because there are predicates on the global state of a network that a single node cannot evaluate. We therefore need to make some restrictions to our timeouts.

First, we want our protocols to be nonterminating. Gouda and Multari [7] have argued convincingly that SS systems should not have halting states (see also [6]). We therefore define a protocol to be *live* if in every state, at least one internal guarded command is enabled.

Gouda [5] has introduced a restricted form of timeout predicate called a *normal timeout*. He argues that such predicates may be implemented using real-time clocks. However, the form of these predicates is severely restricted. We choose instead to introduce two properties which more accurately capture the characteristics of a large class of timeout actions.

Let $G \to C$ be a guarded command in a protocol $\mathcal{P}$ such that G is a timeout guard. We define $\mathcal{P} \mid G$ to be the protocol obtained from $\mathcal{P}$ by replacing all timeout guards other than G with **external**. A guarded command $G' \to C'$ belonging to the same process as $G \to C$ is said to *refute* G if in every state s, $G' \Rightarrow \neg G$. G is said to be:

- *refutable* if in every weakly fair computation of $\mathcal{P} \mid G$ contains either $G \to C$ or some guarded command that refutes G; or
- *persistent* if G is closed under executions of guarded commands of $\mathcal{P} \mid G$ other than $G \to C$.

Intuitively, any timeout must have a mechanism for resetting it. Refutability ensures that such a mechanism is present. Also, timeouts are much more robust if they may be set to arbitrarily large values, above some minimum value. Persistence guarantees this property. Finally, it is much easier to compute timeout values if they are independent from each other. This is why we have used $\mathcal{P} \mid G$ in our definitions of refutability and persistence.

Finally, we define a live protocol to be *regular* if all of its timeout guards are persistent and refutable.

4. Implementing semi-reliable communication

In Section 2, we argued that for most protocols, it is most convenient to assume that channels provide reliable message transmission. We suggested that such an assumption is typically realized by implementing the channel using two buffers, a transmission link, and a transmission protocol. Such an implementation motivates the need for both the reliability threshold and the channel bound as defined in Section 3. In this section, we will fill in some of the details of such an implementation by defining a self-stabilizing alternating-bit protocol (SS-ABP) using two channels whose reliability threshold is 0 and whose combined channel bound is B. This example will provide the service of a single k-reliable, B'-bounded, unidirectional channel, where k and B' are independent of B, and $B' \geq k{+}2$. The main purpose of this example is to illustrate the model presented in Section 3.

The sender p in SS-ABP will maintain an SS queue *sbuffer* capable of storing up to k messages, and the receiver q will maintain an SS queue *rbuffer* capable of storing up to $B' - k - 1$ messages. In order to simplify our presentation, we will not give the code for the SS queue, but will simply assume the existence of the standard *enqueue* and *dequeue* operations, as well as boolean functions *empty* and *full* (we assume that an *enqueue* operation on a full queue will leave the queue unchanged).

In order to guarantee liveness and self-stabilization, the protocol needs to make progress even if there are no messages to deliver. For this purpose, we use a special message *IGNORE*, which may be sent from p to q, but which will never be delivered to the upper-level protocol.

SS-ABP will interface with other protocols through **external** guards. In particular, p will contain an **external** guard which will correspond to a **send** being executed by the other protocol, and q will contain an **external** guard which will correspond to a **receive** being executed by the other protocol. The **external** guarded command in p will use a function $get()$, which will return the message being sent by the other protocol, and the **external** guarded command in q will use a function $put(message)$, which will deliver the given message to the other protocol as the message received. For the purpose of reasoning about SS-ABP, we simply assume that $get()$ nondeterministically returns an arbitrary message, and $put(message)$ has no effect.

The code for SS-ABP is given in Figure 1. (For simplicity of presentation, we have deviated slightly from the formal definition of a guarded-command process, but it is a straightforward matter to translate our code to fit the definition.) This particular algorithm is not necessarily the most efficient SS version of ABP, but it is simple, and it illustrates the main points of our model. All of the variables in the protocol are bounded, and we assume that both ns and nr are large enough to contain the range $[0..B-1]$. The essential difference between this algorithm and the classical ABP is found in line p10. In the classical ABP, the addition is done modulo 2. Intuitively, this is sufficient because any message received by q is either a duplicate of a message that it has already received, but whose ack was lost, or the next message in the sequence. Thus, a single bit is enough to differentiate the two cases. However, addition modulo 2 is not sufficient for a self-stabilizing algorithm, because the channels may initially contain "garbage" messages.

We note here that although SS-ABP uses the same basic idea as the self-stabilizing ABP presented by Afek and Brown [1], there are some subtle differences in the algorithms. First, Afek and Brown's algorithm does not use a timeout; instead, their **send** action is always enabled. We chose to use a timeout so that we could illustrate the use of a persistent and refutable timeout. Second, their algorithm does not require an *IGNORE* message, because they have assumed that there is always a message available to send. Finally, their algorithm does not deal explicitly with the case in which there is no place to put a message when it is received, although by assuming the message is lost, they would end up treating this case as we have.

We first observe that SS-ABP is live, because if neither **receive** guard is enabled, then the **timeout** guard is enabled. Furthermore, the **timeout** guard is clearly persistent. We will now show that it is refutable.

Theorem 1 *The* **timeout** *guard in SS-ABP is refutable.*

Proof: If C_{pq} is empty or C_{qp} is nonempty, then either the **timeout** guard or the guard at line p4 will be continuously enabled until it is executed; hence, any weakly fair computation from such a state will include either the **timeout** command or the guarded command at line p4, which refutes the **timeout**. Suppose C_{pq} is nonempty and C_{qp} is empty. Then the guard at line q6 is continuously enabled until it is executed. When it is, if it successfully sends a message, the guard at line p4 will again be continuously enabled until it is executed. Otherwise, the number of messages in C_{pq} is reduced, and by induction on the number of messages in C_{pq}, either the **timeout** command or the command at line p4 will be continuously enabled until it is executed. Therefore, the **timeout** guard is refutable. □

We will now show that in this protocol, **true** converges to a certain predicate, R, which will denote the set of legitimate states. We use the following notation:

- $|C|$ - the number of the messages in channel C;
- C_{qpq} - the concatenation of the sequences of messages in the two channels, C_{qp} followed by C_{pq}.

```
p1   process p
p2   begin
p3       external → enqueue(get(), sbuffer)
p4       [] receive ack(i) →
p5           if i = ns then
p6               if empty(sbuffer) then
p7                   last := IGNORE
p8               else
p9                   last := dequeue(sbuffer)
p10              ns := ns + 1 (mod B)
p11              send data(last, ns)
p12      [] timeout (empty(C_pq) ∧ empty(C_qp)) →
p13          send data(last, ns)
p14  end

q1   process q
q2   begin
q3       external →
q4           if ¬empty(rbuffer) then
q5               put(dequeue(rbuffer))
q6       [] receive data(m, i) →
q7           if i = nr then
q8               send ack(i)
q9           else if ¬full(rbuffer) ∨ m = IGNORE then
q10              nr := i
q11              send ack(i)
q12              if m ≠ IGNORE then
q13                  enqueue(m, rbuffer)
q14  end
```

Figure 1. SS alternating bit protocol

Lemma 1 *In SS-ABP,* **true** *converges to the following closed predicate:*

$$|C_{qpq}| \leq 1 \tag{1}$$

Proof (sketch): Observe that there are at most B garbage messages initially in the two channels. Thus, immediately after p has received all initial garbage acks and all acks resulting from initial garbage messages, at most one message is tagged with the current value of ns. Furthermore, if such a message exists, it is the last one sent by p. Thus, all of the acks resulting from other messages in the channels at that point are ignored by p. Therefore, the channels will eventually contain a total of at most one message. The fact that the predicate is closed follows from the fact that the guarded commands that contain **receive**s each send at most one message, and the only other guarded command that sends a message is enabled only when both channels are empty. □

Given Lemma 1, we have the following Corollary.

Corollary 1 *In SS-ABP, predicate 1 converges to the conjunction of the following two closed predicates:*

$$data(m,i) \text{ in } C_{pq} \Rightarrow (m = last \wedge i = ns) \tag{2}$$

$$ack(i) \text{ in } C_{qp} \Rightarrow (i = nr = ns) \tag{3}$$

Let R be the conjunction of predicates 1, 2, and 3. Lemma 1 and Corollary 1 state that SS-ABP R-stabilizes.

We now wish to show that after SS-ABP converges to R, it simulates a B'-bounded k-reliable channel C. Given any R-state σ, we formally define the channel C simulated by σ as follows:

- if $nr = ns$ or $last = IGNORE$, then C is constructed by placing the contents of *sbuffer* at the back of *rbuffer*; or
- if $nr \neq ns$ and $last \neq IGNORE$, then C is constructed by enqueuing *last* to *rbuffer*, then placing the contents of *sbuffer* at the back of this queue.

The external guarded command in p will simulate a **send** on C. Recall that because C is k-reliable, any **send** that occurs when C contains fewer than k messages must succeed. Furthermore, because C is B'-bounded, any **send** that occurs when C contains B' messages must fail, but any **send** that occurs when C contains at least k messages but fewer than B' messages may either succeed or fail. Likewise, the external guarded command in q will represent a **receive** if the condition at line q4 is **true**. The internal guarded commands should not change the contents of C. This behavior is described formally by the following theorem, which is easily shown by inspection of SS-ABP.

Theorem 2 *The following safety properties hold at any R-state of SS-ABP:*

1. *No internal guarded command changes the contents C.*
2. *The external command at line p3 causes a message to be enqueued to C if the number of messages in the simulated channel is initially less than k.*
3. *The external command at line p3 either causes a message to be enqueued to C or has no effect on C if the number of messages in C is initially at least k but less than B'.*
4. *The external command at line p3 has no effect on C if the number of messages in C is B'.*
5. *The external command at line q3 dequeues a message from C if rbuffer is nonempty.*

In order to complete the correctness proof of SS-ABP, we need to show one liveness property.

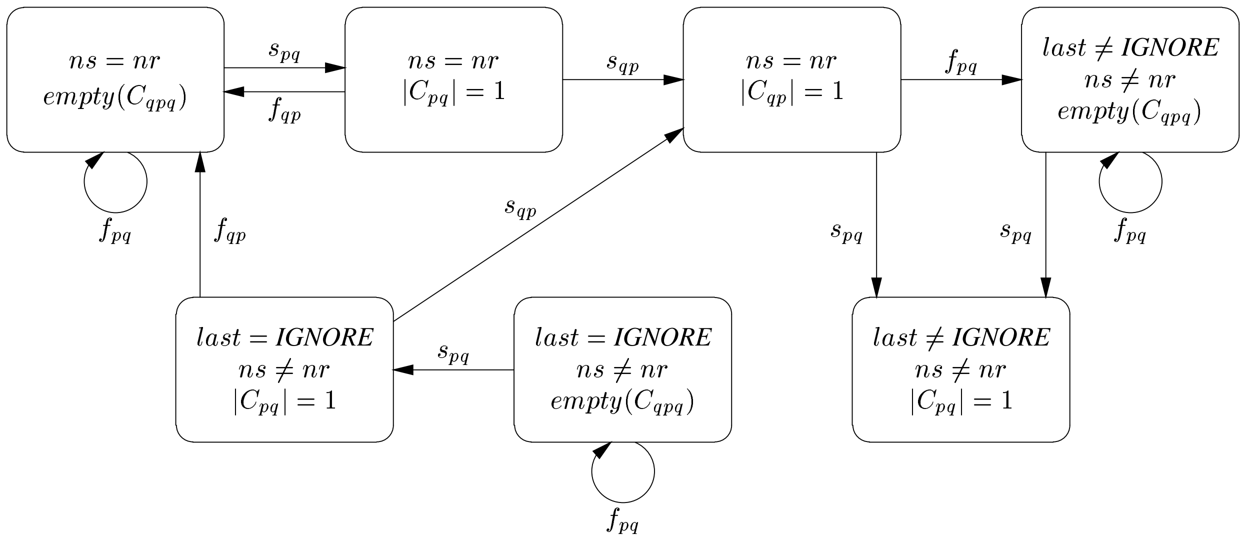

Figure 2. A finite automaton for Theorem 3

Theorem 3 *Any maximal fair computation from an R-state in which C is nonempty will eventually enter a state in which rbuffer is nonempty.*

Proof: Let S be the set of R-states in which C is nonempty and in which *rbuffer* is empty. Because SS-ABP is live, any maximal fair computation will contain infinitely many internal guarded commands. It therefore suffices to show that there is no fair computation with infinitely many internal guarded commands that stays entirely within S.

We first partition S into the seven sets shown as states in the finite automaton in Figure 2. This partitioning is done as follows. First, we partition S into two sets: those states in which $ns = nr$ and those states in which $ns \neq nr$. We then partition the set in which $ns = nr$ into three sets: those states in which C_{qpq} is empty, those states in which $|C_{pq}| = 1$, and those states in which $|C_{qp}| = 1$ (from Lemma 1, these three sets cover all cases). Finally, we partition the set in which $ns \neq nr$ into four sets depending on whether $last =$ *IGNORE* and whether C_{qpq} is empty (from Corollary 1, if C_{qpq} is nonempty, then $|C_{pq}| = 1$).

Note that the external commands, when executed from any state $\sigma \in S$, always yield a state σ' such that σ and σ' belong to the same set of the partition. The finite automaton in Figure 2 therefore shows how all of the internal guarded commands change state within S, modulo the partition described above. The labels on the arcs indicate whether a successful **send** takes place on a particular channel; i.e., s_{pq} and s_{qp} indicate that a successful **send** took place on C_{pq} and C_{qp}, respectively, and f_{pq} and f_{qp} indicate that an unsuccessful **send** took place on C_{pq} and C_{qp}, respectively (note that each internal guarded command attempts to send a message when executed on a state in S). It is now easily seen that any infinite path in this automaton must include infinitely many failed **send**s on some empty channel with no intervening successful **send**s on that channel. Therefore, any fair computation with infinitely many internal guarded commands must eventually leave S. □

5. Conclusion and future research

We have defined a guarded-command model with bounded channels and demonstrated that model by defining and verifying an SS alternating-bit protocol. We believe that SS regular protocols defined using this model can be implemented directly on actual hardware. In this section, we outline our plans for showing this fact.

Because semi-reliable channels can be implemented using SS-ABP, we only need to concern ourselves with implementing regular protocols that use 0-reliable channels. The biggest issue remaining is implementing refutable and persistent timeouts.

In order to enforce some upper bound on length of a timeout, we can implement guarded-command protocols by testing guards in a round-robin fashion, and executing those guarded commands whose guards are true. Clearly, such a scheduling mechanism will enforce weak fairness, but in addition, it will place an upper bound on the length of time any guard remains true before its command is executed. This upper bound will depend on the speed of the processor and the worst case message transmission time (without retransmission).

Consider now the proof of Theorem 1. In essence, it showed that either the timeout or the guarded command that refutes it will be executed after the internal guarded command in q executes at most n times, where n is the channel bound of C_{pq}. Thus, we can compute a maximum length of time before one of the internal guarded commands in p is executed. The guarded command that refutes the timeout can therefore set a timer to this maximum length whenever it executes, so that if the timer expires (after it has been correctly set at least once), the timeout guard is guaranteed to be true. In our ongoing work, we generalize this technique to apply to any regular protocol.

The remainder of the implementation seems to be a rather tedious but mechanical translation to a low-level model with fine-grained atomicity. The difference in atomicity is not a serious problem because each guarded command contains at most a single **receive**, which is not preceded by any **send**. This characteristic makes it a rather straightforward matter to show that a fine-grained implementation is serializable.

References

[1] Y. Afek and G. M. Brown. Self-stabilization of the alternating-bit protocol. *Distributed Computing*, 7:27–34, 1993.

[2] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, 1991.

[3] E. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, Nov. 1974.

[4] Dolev S., Israeli A., and Moran S. Resource bounds for self-stabilizing message-driven protocols. *SIAM Journal on Computing*, 26(1):273–290, 1997.

[5] M. Gouda. *Elements of Network Protocol Design*. Wiley, 1998.

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

[7] M. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Tran. on Computers*, 40(4):448–458, 1991.

[8] S. Katz and K. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7:17–26, 1993.