Finite-State Self-Stabilizing Protocols in Message-Passing Systems[1]

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

and

Mikhail Nesterenko[2]
Mathematics and Computer Science Dept.
Kent State University
Kent, OH 44240 USA
mikhail@mcs.kent.edu

and

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

Version: January 29, 2002

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 two simple examples. We then give a translation from this model to a lower-level model that uses a notion of time. We argue that this latter model is very close to a real network. We conclude by discussing how this translation might be used to implement self-stabilizing protocols on actual networks.

Key Words: self-stabilization, message-passing systems, computational models

1. INTRODUCTION

Dijkstra introduced the notion of *self-stabilization* in the context of distributed systems [4]. 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

[1]Some of the results in this paper were presented at the 19th ICDCS Workshop on Self-Stabilizing Systems, Austin, TX, June 1999 (pp. 62-69).

[2]Supported in part by DARPA contract OSU-RF #F33615-01-C-1901.

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[8] 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. [7]). 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. Our model consists of two levels of abstraction: the top level, which is called *the guarded-command level*, and the lower level, called *the timed guarded-command level*.

The guarded-command level provides essentially the same interface as the guarded-command model used by Gouda and Multari [8], which we consider elegant and easy to program. The main differences introduced in the guarded-command level are:

- Channels are bounded and semi-reliable.

  "Bounded" means that there is a maximum number of messages (called a *channel bound*) that each channel can hold at any time. This bound is essential to allow application protocols to be finite-state.

  "Semi-reliable" means that as long as the number of messages in the channel does not exceed a given threshold (called a *reliability threshold*), no messages will be lost.

  During protocol design, the channel bound of each channel may be specified by a variable, which at implementation time will be replaced by a realistic value that is guaranteed to be at least as great as the actual channel bound of the underlying network. On the other hand, the reliability threshold is typically derived from the protocol. In particular, many protocols require reliable message delivery in order to guarantee correctness. If such a protocol has finitely many legitimate states, there is a bound on the maximum number of messages in each channel over all legitimate states. Certainly, the channel reliability threshold must be chosen to be at least this bound in order to guarantee the correctness of the algorithm. In practice, this reliability is provided by a transport protocol such as the Alternating Bit Protocol or Sliding Window Protocol. Typically, the reliability threshold will be much smaller than the channel bound.

- Timeouts are guards defined from a restricted set of predicates over the global state. Gouda and Multari [8] modeled timeouts using arbitrary predicates on the global state. Though elegant, their approach suffers from the fact that many timeout predicates cannot be implemented in real systems. In [6], Gouda introduced certain restrictions on timeout predicates in order to alleviate this problem. However, we believe our restrictions are easier to use.

  We include techniques for showing that the protocol satisfies the restrictions. In the proof process, a value called a *refutability guarantee* for each timeout is derived. This value is then used in translating guarded-command protocols to timed guarded-command protocols.

In the timed guarded-command level, protocols may use only local variables in the boolean expressions in guards. In particular, a timeout guard from the guarded-command level will be replaced by a guard using a local counter, which behaves as a logical timer. A counter value is reset to a reset value by certain guards which are said to *refute* the timeout guard. In order to force these counters to simulate timeout guards, we assume a stronger notion of fairness (weak fairness is assumed for the guarded-command level). The stronger fairness assumption guarantees an upper bound on the maximum number of steps within which any guarded command may be continuously enabled without being executed. We refer to this bound as the *timing constant*. We are able to derive from the timing constant and the refutability guarantee appropriate reset values so as to guarantee faithful simulation of the timeout guards from the guarded-command level.

When implementing the protocol in practice, we can obtain a value for the timing constant from the message transmission times and relative processor speeds. Therefore, a timed guarded-command level protocol may be implemented on real hardware for which these bounds are known.

After we formally define these two levels, we give as an example a finite-state SS Alternating Bit Protocol (ABP) defined in the guarded-command level. This protocol uses channels with reliability threshold of zero. We prove that this protocol is correct and SS. Furthermore, we prove that the protocol satisfies the restrictions on timing guards, and we derive their refutability guarantees. We also define as a second example Dijkstra's k-state protocol using the guarded-command level. We use this example to illustrate the use of reliable channels with positive reliability thresholds.

We then formally present a general translation from a protocol in the guarded command level $\mathcal{P}_g$ to a protocol in the timed guarded command level $\mathcal{P}_t$. We show how to obtain the reset values for the logical timers from the refutability guarantee for the timeout guard and the timing constant for the execution environment. We then show that $\mathcal{P}_t$ implements $\mathcal{P}_g$. In particular, we show that when $\mathcal{P}_t$ executes a timeout, the corresponding timeout guard from $\mathcal{P}_g$ is guaranteed to be true. As an example, we show the translation of the SS ABP from the guarded command level to the timed guarded command level.

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 a protocol at the timed guarded-command level can be implemented on real hardware, 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 [8].

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 every 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 [5, 8].

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

1. In [8], Gouda and Multari assume unbounded reliable channels with a timeout mechanism. The timeout 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 the global state. They presented an Alternating Bit Protocol (ABP) and a Sliding Window Protocol (SWP) in this model.

2. Katz and Perry [9] 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 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 [5], 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 transport 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. [7]).

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. On the other hand, if a message is sent when the send buffer is not full — even if the total number of messages in the channel is more than the capacity of the send buffer — the message will be enqueued and the transport protocol successfully deliver the 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. The communication model resembles the unreliable-channel model of Afek and Brown [1] in the sense that messages can be lost. However, we restrict the channel capacities by assuming that initially the number of messages on any channel is no greater than that channel's bound and that when messages are sent on full channels, these messages are lost. Furthermore, in order to allow protocols to utilize reliable message delivery, we assume that if the number of messages in a channel does not exceed a certain threshold, the message will be successfully delivered. By introducing both a channel bound and a reliability threshold, we are able to account for such anomalies as described in the above paragraph while providing reliable message transmission under normal circumstances (e.g., when the system is in a legitimate state). Our timeout mechanism uses predicates on the global state as in [8], but with restrictions that make these predicates implementable. We then discuss how SS protocols developed for this model might be implemented on a real network so that self-stabilization is preserved.

3. DEFINITIONS

In this section, we present our computational models. For each model, we will describe protocols using a guarded-command style of notation. In each case, the protocol will describe a finite-state system. It is therefore convenient to view both of these models abstractly as finite-state machines (S, T), where S is a finite set of *states* and T is a finite set of *transitions*, each of which is a function mapping S into the power set of S. Given $s \in S$ and $t \in T$, if $t(s) \neq \varnothing$, we say t is *enabled* at s, and we write $s \xrightarrow{t} s'$ for each $s' \in t(s)$. A *computation* of (S, T) is any (possibly infinite) string σ of the form $s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \cdots$ such that for each i, $s_i \xrightarrow{t_{i+1}} s_{i+1}$, and such that σ ends in a state if it is finite. 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.

The specific computational models that we use are best viewed as two levels of abstraction, as follows:

1. The top level is the *guarded-command level.* It is essentially the same as the guarded-command model used by Gouda and Multari [8], except that the channels are bounded and semi-reliable (i.e., as long as the number of messages in the channel does not exceed a given threshold, no messages are lost), and the variables are finite. Furthermore, we place restrictions on the timeout predicates, which in [8] were arbitrary predicates on the global state.

2. In order to implement the guarded-command level, we assume the existence of a *timed guarded-command level.* This level uses a notion of time, and assumes an upper bound on the maximum time a guarded command may be continuously enabled without being executed.

In what follows, we formally define each level. We then conclude this section by formalizing the notion that a protocol at the timed guarded-command level implements some protocol at the guarded-command level. In subsequent sections, we will give a construction for implementing SS protocols defined in the guarded-command level with protocols in the timed guarded-command level. As we discuss in Section 7, we believe that it will be possible to implement the resulting protocols on real networks with bounded relative processor speeds and message transmission delays.

3.1. The guarded-command level

A protocol in the guarded-command level 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 contains 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⟩ → ⟨command⟩.

- Each guard is of one of the following forms:

  - **external**

- $\langle$boolean expression$\rangle$
  or
- $\langle$local boolean expression$\rangle \wedge$ **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; **if**-statements; and nondeterministic choice.

The keyword **external** will evaluate to **true**, but fairness constraints will not apply to these guarded commands. The purpose of **external** guards is to give us the ability to compose protocols, or to otherwise abstract a reaction to an environment. For example, suppose we are defining a communication protocol with a sender and a receiver. We can include in the sender the following guarded command to model a message becoming available to send:

$\textbf{external} \rightarrow msg := get(); avail := \textbf{true}$

where $get()$ nondeterministically returns some message. Thus, we can reason about the behavior of the protocol with no assumptions concerning the frequency or content of messages to be sent (although we are certainly free to add any appropriate assumptions). We will call guarded commands with **external** guards *external* guarded commands. We will call all other guarded commands *internal* guarded commands.

The primary purpose of a timeout guard is to avoid communication deadlocks, which often occur when all channels are empty. In order for any single process to be able to detect this condition, it must have more power than is given by boolean combinations of local variables and **receive** statements. Rather than incorporating an explicit notion of time into our model, we have chosen to follow the lead of Gouda and Multari [8], and model timeouts with arbitrary predicates on the global state. Thus, we can define a timeout predicate to enable an action only when all channels are empty. Clearly, however, we cannot hope to implement arbitrary predicates on the global state in practice. We will add appropriate restrictions to these predicates shortly.

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 channel with reliability threshold k is said to be a *k-reliable* channel. 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 channel 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 reliability threshold is obviously 0 (0-reliable channel).

Next, consider the channel bound. In any real network, there is clearly a bound on the number of messages a channel can contain at any given instant. However, this bound is typically greater than the capacity of the sender's buffer. For example, the receiver will typically have a buffer which may contain messages that are not in the sender's buffer, but have not yet been delivered to the application. Furthermore, there may be additional variables that can contain additional messages (such is the case in our SS ABP in the next section), or the links themselves may contain additional messages. The maximum number of messages that can be stored in all of these locations is the channel bound.

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 channel model accurately reflects the behavior observed in real network channels.

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 than 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.

Typically we will assume weak fairness holds for the internal guarded commands of a protocol. Whenever stronger versions of fairness are needed (e.g., to ensure progress is made when channels have reliability thresholds of 0), they will be explicitly assumed. Thus, we give the following definition:

DEFINITION 3.1. A computation σ is said to be *fair* if either σ is finite, or whenever some internal guarded command GC is enabled at all but finitely many states in σ, GC appears infinitely often in σ.

Note that Definition 3.1 says nothing concerning nondeterministic choice within a guarded command. If a protocol requires that these choices be made fairly, either a stronger version of fairness should be assumed, or the choices should be split into separate guarded commands.

We are now ready to formally define a self-stabilizing protocol for this model.

DEFINITION 3.2. Let P be a subset of the state set S (or equivalently, a predicate on S) of a protocol. P is said to be *closed* if for every $s \in P$ and every guarded command GC, $s \stackrel{GC}{\to} s' \in P$.

DEFINITION 3.3. Let S be the state set of a protocol. $R \subseteq S$ 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.

DEFINITION 3.4. A protocol with state set S is said to be *P-stabilizing* if S converges to the closed subset P (or equivalently, **true** converges to P).

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 [8] have argued convincingly that SS systems should not have halting states (see also [7]). We therefore define:

DEFINITION 3.5. A protocol is said to be *live* if in every state, at least one internal guarded command is enabled.

Gouda [6] 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. We first give some notation.

DEFINITION 3.6. 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**.

We now define the two restrictions that we will apply to our timeout predicates.

DEFINITION 3.7. A guarded command $G' \to C'$ belonging to the same process as the timeout guarded command $G \to C$ is said to *refute* G if in every state s, $G' \Rightarrow \neg G$. G is said to be *refutable* if every maximal fair computation of $\mathcal{P} \mid G$ contains either $G \to C$ or some guarded command that refutes G.

Definition 3.8. Let $G \to C$ be a timeout guarded command. G is said to be *persistent* if G is closed under executions of guarded commands of $\mathcal{P} \mid G$ other than $G \to C$.

In a real system, any timer implementing a **timeout** guard must have a mechanism for resetting it. This mechanism periodically detects that the **timeout** guard is false, and resets the timer. For example, an SS protocol typically will have a **timeout** to break a communication deadlock. An appropriate **timeout** guard is that all channels are empty. If a **receive** guard in the same process is executed, we know that its input channel was nonempty, so the timer should be reset. This guarded command has refuted the **timeout** guard. Our definition of refutability states that in any maximal fair computation, it is always the case that any **timeout** is eventually either executed or refuted. Also, protocols with **timeout**s are much more robust if in implementations timers can be set to arbitrarily large values, above some minimum value. Persistence guarantees that once a **timeout** guard becomes true, it remains true until that **timeout** is executed. Thus, it does not matter how long we wait to execute a **timeout**, provided we wait long enough for the **timeout** guard to become true. Finally, it is much easier to compute timer reset 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.

One of the consequences of using persistent timeouts is that there may be only one **timeout** guarded command in the protocol to handle a given condition. This seems to be a reasonable restriction. To see why, suppose two different processes have **timeout** guards that are enabled when all channels are empty. Presumably both of these guarded commands would result in the sending of a message. Thus, neither would be persistent, because each would disable the other. It would seem to be difficult in practice to prevent one of the corresponding timers from expiring after the other has expired, but before the resulting message is detected. Thus, it would seem to be impossible to implement such a protocol correctly. The restriction that **timeout** guards be persistent avoids this problem. On the other hand, the restriction to persistent timeouts does not prohibit a nondeterministic choice to take place as the result of a **timeout**: whereas including two **timeout** guarded commands $G \to C_1$ and $G \to C_2$ in the same process would likely violate persistence, this violation could be avoided by replacing them with the single guarded command $G \to \{C_1[]C_2\}$ (see, however, the note following Definition 3.1).

Later in the paper, we will give a technique for proving a timeout guard to be refutable. Proofs using this technique will give us all of the information we need to implement at the timed guarded-command level a live protocol whose timeout guards are both persistent and refutable. We therefore give the following definition:

Definition 3.9. A live protocol is said to be *regular* if all of its timeout guards are persistent and refutable.

3.2. The timed guarded-command level

A protocol in the timed guarded-command level is just like a protocol in the guarded-command level, except that only local variables may be used in the boolean expressions in guards. To allow for the implementation of timeout guards from the guarded-command level, we use a stronger notion of fairness in the timed guarded-command level. In particular, we associate with each protocol a positive integer *timing constant* τ. We then define:

DEFINITION 3.10. A computation of a protocol at the timed guarded-command level is *fair* if it satisfies the weak fairness of Definition 3.1 and in addition, for any subcomputation $\sigma = s_i \stackrel{GC_{i+1}}{\rightarrow} s_{i+1} \stackrel{GC_{i+2}}{\rightarrow} \cdots \stackrel{GC_{i+\tau}}{\rightarrow} s_{i+\tau}$ of τ guarded commands, if some internal guarded command GC is enabled at every state $s_i, ..., s_{i+\tau-1}$, then GC must be executed in σ.

Note that this condition is strictly stronger than the weak fairness condition of Definition 3.1. The added power in essence forces the message delivery times and relative speeds of all processes to be bounded. Thus, as we will show in Section 6 it is strong enough to allow us to implement persistent and refutable timeouts using local counters. Note, however, that the smaller τ is, the more difficult it becomes for an actual implementation to enforce fairness. From a practical standpoint, τ needs to be large enough so that over any subcomputation of length τ, any messages sent on empty channels prior to the subcomputation will have had time to have been delivered, and subsequently all guards in all processes will have been examined by their schedulers. (Note also that a reasonable scheduling algorithm, such as round-robin examination of the guards, is essential.) In Section 7 we will discuss in more detail how protocols at the timed guarded-command level can be implemented in practice.

4. EXAMPLES

In this section, we will give two simple examples of protocols defined at the guarded-command level. Both of these examples are based upon SS protocols that have previously been defined for other computational models. We will use these examples to illustrate the use of our model for defining, analyzing, and implementing protocols. These examples will also serve to motivate our main results.

4.1. Alternating Bit Protocol

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. The example presented in this subsection fills in some of the details of such an implementation. In particular, we present a self-stabilizing alternating-bit protocol (SS-ABP) using two channels whose reliability threshold is 0 and whose combined channel bound is B. This protocol 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 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.

In the code and discussion that follows, 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}.

The code for SS-ABP is given in Fig. 1. (For simplicity of presentation, we have omitted variable declarations.) 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 p8. 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 (Definition 3.5), because if neither **receive** guard is enabled, then the **timeout** guard is enabled. Furthermore, the **timeout** guard is clearly persistent (Definition 3.8). In order to conclude that the protocol is regular, we must show that its **timeout** guard is refutable. In the next section, we will introduce a proof technique for showing refutability, and use that technique to show that this **timeout** guard is refutable. For now, we will concentrate on the stabilization and correctness of this protocol.

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

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

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

FIG. 1 SS alternating bit protocol

LEMMA 4.1. *In SS-ABP,* **true** *converges to the following closed predicate:*

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

Proof. The predicate is clearly closed, as the only guarded command that increases the number of messages in C_{qpq} is the **timeout** at p9, and this guarded command cannot bring the number of messages greater than 1. According to Definition 3.3, we must show that every maximal fair computation contains a state in which $|C_{qpq}| \leq 1$. We proceed by induction on the initial value of $|C_{qpq}|$.

Base: $|C_{qpq}| \leq 1$. The Lemma holds trivially.

Induction Hypothesis: Assume that from any state in which $|C_{qpq}| = n \geq 1$, any maximal fair computation contains a state in which $|C_{qpq}| \leq 1$.

Induction Step: Let $|C_{qpq}| = n + 1$. The Lemma vacuously holds if $n \geq B$ because the channel bounds guarantee that there is no state in which $|C_{qpq}| > B$. Therefore, assume $n < B$. We will show that any maximal fair computation reaches a state with n messages. The Lemma will then follow from the Induction Hypothesis.

It is easily seen by inspection that if $|C_{qpq}| = n + 1$, any guarded command will either decrease the number of messages by 1 or leave the number of messages unchanged. We will focus on the latter. Furthermore, we will focus our attention on the sequence numbers of the messages in C_{qpq} (i.e., the value of i in $data(m, i)$ or $ack(i)$). Note that the only guarded commands that can change the sequence numbers in C_{qpq} (assuming $|C_{qpq}| = n+1$) are the ones at lines p4 and q5. Furthermore, because at least one channel is nonempty, one of these two must eventually execute in any maximal fair computation. If the guarded command at q5 executes without changing the number of messages in C_{qpq}, it also does not change the sequence numbers, and it guarantees that the guarded command at p4 is enabled. Thus, as long as $|C_{qpq}| = n + 1$, the guarded command at p4 must eventually be executed, and until it does, the sequence numbers in C_{qpq} remain the same.

Suppose the guarded command at line p4 executes, leaving $|C_{qpq}| = n + 1$. Suppose $ns = a$, $0 \leq a < B$, immediately prior to this execution. Because the **send** at line p8 must be successfully executed, ns is incremented to $(a+1) \mod B$, and a message with sequence number $(a+1) \mod B$ is placed in the $(n+1)$st position in C_{qpq} (counting from the front). A straightforward induction shows that in any computation, if the guarded command at line p4 executes $j \leq n$ additional times leaving $|C_{qpq}| = n+1$, then $ns = (a+j+1) \mod B$ and a message with sequence number $(a+1) \mod B$ is in the $(n+1-j)$th position in C_{qpq}. Letting $j = n$, we see that if $|C_{qpq}|$ remains $n+1$, eventually the message at the front of C_{qpq} will have a sequence number of $(a+1) \mod B$, and ns will have a value of $(a+n+1) \mod B$. Because $1 \leq n < B$, $ns \neq (a+1) \mod B$. Thus, when p eventually receives the next message, no message will be sent.

We therefore conclude that any maximal fair computation from any state in which $|C_{qpq}| = n + 1$ eventually reaches a state in which $|C_{qpq}| = n$. From the Induction Hypothesis, this computation must then reach a state in which $|C_{qpq}| \leq 1$. ∎

Given Lemma 4.1, we have the following Corollary.

COROLLARY 4.1. *In SS-ABP, predicate* (1) *converges to the conjunction of the following two closed predicates:*

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

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

Let R be the conjunction of predicates (1), (2), and (3). Lemma 4.1 and Corollary 4.1 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 s, we formally define the channel C simulated by s 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 4.1. *The following safety properties hold at any R-state of SS-ABP:*

1. *No internal guarded command changes the contents of C.*

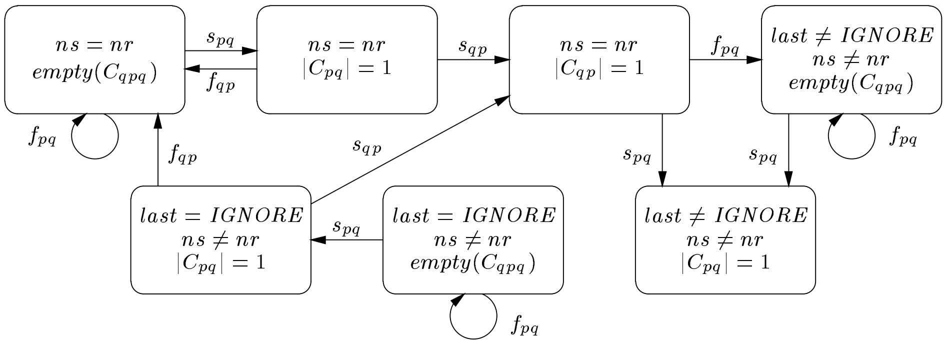

FIG. 2: A finite automaton for Theorem 4.2

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.

THEOREM 4.2. *Any maximal fair computation from an R-state in which C is nonempty will eventually enter a state in which rbuffer is nonempty, provided that if infinitely many* **send***s are attempted on the same empty channel, infinitely many of them succeed.*

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 Fig. 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 4.1, 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 4.1, if C_{qpq} is nonempty, then $|C_{pq}| = 1$).

Note that the external commands, when executed from any state $s \in S$, always yield a state s' such that s and s' belong to the same set of the partition. The finite automaton in Fig. 2 therefore shows how all of the internal guarded commands

process p_0
begin
 receive$(msg(j),\ p_{n-1}) \rightarrow$
  if $j = tok$ **then** $tok := tok + 1$ (**mod** k); **send**$(msg(tok),\ p_1)$
 [] **timeout** $(empty(C_{0,1}, ..., C_{n-1,0})) \rightarrow$ **send**$(msg(tok),\ p_1)$
end

process $p_i,\ 1 \leq i < n$
begin
 receive$(msg(j),\ p_{i-1}) \rightarrow$ **send**$(msg(j),\ p_{i+1\ (\mathbf{mod}\ n)})$
end

FIG. 3 Dijkstra's k-state protocol

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, provided that if infinitely many **send**s are attempted on the same empty channel, infinitely many of them succeed. ∎

4.2. Dijkstra's k-state protocol

In this section, we briefly present Dijkstra's k-state protocol [4] in our model. Our main purpose is to give an example of a protocol in which channels have nonzero reliability thresholds.

The protocol consists of n processes, $p_0, ..., p_{n-1}$. The processes are arranged as a unidirectional ring, so that the only channels used are $C_{p_i, p_{i+1(\mathrm{mod}\ n)}}$, $0 \leq i < n$. This protocol requires that each channel has reliability threshold 1, and the sum of the channel bounds is k. Process p_0 maintains a local integer variable, tok, whose range is $0..k-1$. The remaining processes have no local variables. The legitimate states are those in which there is exactly one message in the system, and this message contains a value equal to tok. This message represents a token, which is circulated around the ring. The protocol is given in Fig. 3.

Because the reliability threshold of each channel is 1, it is easily seen that the set of legitimate states is closed. Stabilization can easily be shown using techniques demonstrated in the previous subsection. The timeout guard is clearly persistent, and we will show that it is refutable in the next section.

5. PROVING REFUTABILITY

In the previous section, we demonstrated how protocols can be defined in the guarded-command level of our model, and how stabilization and correctness could be proven. One of the main purposes of this paper is to show that any regular protocol can be implemented in the timed guarded-command level. We will show this fact in the next section. Before we do that, we need to demonstrate how protocols in the guarded-command level can be shown to be regular (i.e., having only persistent and refutable timeouts). Showing persistence of a timeout simply involves showing a set of states to be closed under the execution of various guarded commands. Showing refutability, however, is more involved in general, due to the fact that refutability is a property of the maximal fair computations of a protocol (see Definition 3.7). Furthermore, even if we know that a protocol is regular, we still need to be able to find appropriate reset values for the timers in the timed guarded-command level.

In this section, we will introduce a general proof technique for showing timeouts to be refutable. This technique will yield a value, which we will call the refutability guarantee, that will be essential to the computation of timer reset values for the implementation at the timed guarded-command level. We will then illustrate the technique by applying it to the **timeout** guards in SS-ABP and Dijkstra's k-state protocol.

Suppose $G \to C$ is a **timeout** guarded command. The idea of our technique is to find a bound such that any computation exceeding this bound must include either $G \to C$ or a guarded command that refutes it. This bound is more abstract than simply a bound on the length of a computation. Rather, it bounds certain behavior in the computation. Furthermore, infinite fair computations will contain an infinite amount of this behavior. We will show that this bound is the refutability guarantee that we need to be able to compute timer reset values for the implementation at the timed guarded command level. The following definition describes the measure of the behavior that we wish to bound.

Definition 5.1. Let σ be a finite computation. We define the *rank* of σ, denoted $\rho(\sigma)$, recursively as follows:

1. If σ is empty (i.e., if σ consists of a single state with no transitions), or if there is some internal guarded command $G' \to C'$ such that G' is always **true** in σ, but $G' \to C'$ is never executed in σ, then $\rho(\sigma) = 0$.

2. Otherwise, let $\sigma = \sigma_1 \overset{GC}{\to} \sigma_2$, where σ_1 is the longest initial subcomputation with rank 0. Then $\rho(\sigma) = 1 + \rho(\sigma_2)$.

Thus, we have broken a finite computation into a series of subcomputations, each of which is a maximum-length computation over which some internal guarded command is always enabled but never executed. Clearly, we would not be able to break an infinite fair computation into finitely many such subcomputations. Furthermore, because our model is finite-state, if a computation has a high enough rank, some of these subcomputations must begin in the same state. Thus, we would have a loop which could be iterated to form an infinite fair computation. Therefore, if a timeout is refutable, either it or a guarded command that refutes it must occur somewhere in this loop. This is the key idea for the following theorem.

Theorem 5.1. *Let $\mathcal{P}$ be a live protocol at the guarded-command level, and let G be a timeout guard in $\mathcal{P}$. Then G is refutable in $\mathcal{P}$ iff there is a natural number k such that in every computation of $\mathcal{P} \mid G$ with rank k, either the guarded command containing G or a guarded command that refutes G is executed.*

The interesting part of this theorem is the only-if part, which says that if G is refutable, there must be a k as described in the theorem. We will show that this k

can be used to compute values for the counters to be used in implementing timeout guards. The if-part of Theorem 5.1 then tells us that if we find such a k, then G is refutable.

Before we prove Theorem 5.1, we need some additional notation. Let σ be a computation, and let σ_0 be a finite initial subcomputation of σ. Then we denote by $\sigma - \sigma_0$ the subcomputation of σ that follows σ_0; i.e., if $\sigma = \sigma_0$, then $\sigma - \sigma_0$ is the empty computation from the last state of σ, or if $\sigma = \sigma_0 \overset{GC}{\to} \sigma_1$, and s_0 is the last state of σ_0, then $\sigma - \sigma_0 = s_0 \overset{GC}{\to} \sigma_1$.

Proof of Theorem 5.1.

$\Rightarrow$: Suppose G is refutable, and let s_0 be any state of $\mathcal{P}$. Let k be the number of states in $\mathcal{P}$. Let σ be any computation of $\mathcal{P} \mid G$ from s_0 with rank k, and let σ_0 be the longest initial subcomputation of σ with rank 0. Furthermore (assuming $k > 0$), let $\sigma_0 \overset{GC_1}{\to} s_1$ be an initial subcomputation of σ. For $0 < i < k$, we recursively define:

- σ_i is the longest initial subcomputation of $\sigma - (\sigma_0 \overset{GC_1}{\to} \cdots \overset{GC_i}{\to} s_i)$ with rank 0 (note that s_i is the initial state of σ_i); and
- GC_{i+1} and s_{i+1} are such that $\sigma_0 \overset{GC_1}{\to} \cdots \overset{GC_{i+1}}{\to} s_{i+1}$ is an initial subcomputation of σ.

It follows by induction on i that for $0 < i < k$, $\rho(\sigma - (\sigma_0 \overset{GC_1}{\to} \cdots \overset{GC_i}{\to} s_i)) = k - i > 0$, so σ_i is well-defined, and $\sigma_0 \overset{GC_1}{\to} \sigma_1 \overset{GC_2}{\to} \cdots \overset{GC_i}{\to} s_i$ is an initial subcomputation of σ.

Because k is the number of states in $\mathcal{P}$, there must exist natural numbers i and j such that $0 \leq i < j \leq k$ and $s_i = s_j$. Note that the computation $\sigma_i \overset{GC_{i+1}}{\to} s_{i+1}$ has rank 1, so every internal guarded command that is always enabled in this subcomputation is executed at least once in this subcomputation. Thus, from s_i, there is an infinite computation $\sigma_i \overset{GC_{i+1}}{\to} \sigma_{i+1} \overset{GC_{i+2}}{\to} \cdots \overset{GC_{j-1}}{\to} \sigma_{j-1} \overset{GC_j}{\to} \sigma_i \overset{GC_{i+1}}{\to} \cdots$. Since G is refutable, this infinite computation must contain either the guarded command containing G or some guarded command that refutes G. Hence, σ must contain one of these guarded commands.

$\Leftarrow$: Suppose that there is a natural number k such that in every computation with rank k, either the guarded command that contains G or a guarded command that refutes G is executed. Let σ' be a maximal fair computation of $\mathcal{P} \mid G$. Because $\mathcal{P}$ is live, σ' must be infinite. Because σ' is fair, it must contain some initial subcomputation σ with rank k. The Theorem follows. ∎

The above proof is constructive, but the natural number k from the proof is the number of states in the protocol. This value is often too large to yield reasonable constants for use in timeout counters. However, as we demonstrate with the theorems below, much smaller values for k can often be found. Therefore, we will say that any k that satisfies Theorem 5.1 for a given refutable timeout guard G of a protocol $\mathcal{P}$ is a *refutability guarantee* for G in $\mathcal{P}$.

We now show that the **timeout** guards in our examples are refutable. In each case, we show a refutability guarantee. Recall that both Definition 3.7 and Theorem 5.1 concern computations from arbitrary states, not necessarily legitimate states. Thus, the refutability guarantees must apply even to computations that have not yet stabilized.

THEOREM 5.2. *The* **timeout** *guard in SS-ABP is refutable. Furthermore, a refutability guarantee for this* **timeout** *is $B_{pq}+1$, where B_{pq} is the channel bound for C_{pq}.*

Proof. Let G be the **timeout** guard. We first note that $\mathcal{P} \mid G$ is identical to $\mathcal{P}$. Thus, let σ be a computation of $\mathcal{P}$ with rank $B_{pq}+1$. Clearly, this computation must contain at least $B_{pq}+1$ internal guarded-command executions. Because C_{pq} is bounded by B_{pq}, these internal guarded-command executions cannot all be executions of the guarded command at line q5. Therefore, σ must contain either G or the guarded command at line p4. Because the guarded command at line p4 refutes G, it follows from Theorem 5.1 that G is refutable. ∎

In the above proof, the refutability guarantee is simply the maximum number of times a single guarded command must execute until either the **timeout** guard or the guard that refutes it is enabled. In general, the computation of a refutability guarantee is not that simple. A more interesting example is the k-state protocol of Fig. 3.

THEOREM 5.3. *The* **timeout** *in the k-state protocol is refutable. Furthermore, n is a refutability guarantee.*

Proof. For convenience of notation, we will rename p_0 to p_n for this proof. We will show by induction on m that in any computation σ with rank m or greater, if an internal guarded command from p_i is enabled at the initial state of σ, then internal guards from each of the processes p_j, $i \leq j \leq \min(i+m-1, n)$, are executed in σ. Because the protocol is live and the guard at a3 refutes the **timeout**, the Theorem will then follow from Theorem 5.1.

Base: $m=0$. The claim vacuously holds.

Induction Hypothesis: Assume the claim holds for $m \geq 0$.

Induction Step: Let σ be a computation with rank $m+1$, and suppose an internal guarded command GC from p_i is enabled at the initial state of σ. Clearly, any enabled guarded command in the protocol will remain enabled until it is executed. Because the rank of σ is greater than 0, GC must be executed in σ. Let $\sigma = \sigma_0 \stackrel{GC}{\rightarrow} \sigma_1$ so that σ_0 contains no executions of GC. Then σ_0 has rank 0, and σ_1 has rank at least m. If GC is in p_n, there is nothing more to show. Otherwise, GC must send a message to p_{i+1}. Since the reliability threshold of each channel is 1, there must be at least one message in $C_{p_i,p_{i+1}}$ in the initial state of σ_1. Hence, an internal guarded command from p_{i+1} is enabled at the initial state of σ_1. By the Induction Hypothesis, internal guards from each of the processes p_j, $j \leq i+1 \leq \min(i+m, n)$, are executed in σ_1. ∎

6. IMPLEMENTING REGULAR PROTOCOLS

In this section, we will show how any regular protocol in the guarded-command level can be implemented by a protocol in the timed guarded-command level. The key part of this construction will be to show how any persistent and refutable timeout guard can be implemented in the timed guarded-command level by using a local counter. However, before we can formally show this fact, we must first formalize the notion that one protocol implements another.

In what follows, let (S_h, T_h) and (S_l, T_l) be two finite-state machines. (We use the subscript h on identifiers that we will want to associate with the guarded-command level, and the subscript l on identifiers that we will want to associate with the timed guarded-command level.) Let f be a function mapping S_l into S_h and g be a partial function mapping T_l into T_h.

DEFINITION 6.1. If $f(s_l) = s_h$, we say that s_l *implements* s_h via f. Likewise, if $g(t_l) = t_h$, we say that t_l *implements* t_h via g.

We define a function $\mathcal{I}_{fg}$ from computations of (S_l, T_l) to finite or infinite strings as follows:

- If $\sigma = s_l$, where $s_l \in S_l$, then $\mathcal{I}_{fg}(\sigma) = f(s_l)$.
- If $\sigma = s_l \stackrel{t_l}{\to} \sigma_0$ where $s_l \in S_l$ and $g(t_l)$ is defined, then $\mathcal{I}_{fg}(\sigma) = f(s_l) \stackrel{g(t_l)}{\to} \mathcal{I}_{fg}(\sigma_0)$.
- If $\sigma = s_l \stackrel{t_l}{\to} \sigma_0$ where $s \in S_l$ and $g(t_l)$ is undefined, then $\mathcal{I}_{fg}(\sigma) = \mathcal{I}_{fg}(\sigma_0)$.

DEFINITION 6.2. If σ is an infinite computation of (S_l, T_l) and $\mathcal{I}_{fg}(\sigma)$ is an infinite computation of (S_h, T_h), or if σ is a finite maximal computation of (S_l, T_l) and $\mathcal{I}_{fg}(\sigma)$ is a maximal computation of (S_h, T_h), then we say σ *implements* $\mathcal{I}_{fg}(\sigma)$ via f and g.

DEFINITION 6.3. Let C_l be a set of maximal computations of (S_l, T_l) containing at least one computation from each state in S_l. If C_h is a set of maximal computations of (S_h, T_h) such that every computation in C_l has a suffix that implements some computation in C_h, then we say C_l *eventually implements* C_h via f and g.

Note that because both of our models may be viewed as finite-state machines, we can use the above definitions to express the proposition that a protocol in the timed guarded-command level implements some protocol in the guarded-command level. The sets C_l and C_h may be used to place certain restrictions, such as fairness, on the computations under consideration.

We will now show how to implement any regular protocol $\mathcal{P}_h$ from the guarded-command level as a protocol in the timed guarded-command level. The construction will involve using counters to implement timeouts. The guards that refute a given timeout will be responsible for reseting its counter. For each counter, an additional guarded command that decrements that counter will be added to the process using that counter. It is not hard to see that for such an implementation, a computation in which no timeout expires will eventually implement a computation of $\mathcal{P}_h$.

What is more subtle, and what we will need to be careful to prove, is that eventually, if a timeout expires in the implementation, it will be the case that the corresponding **timeout** guard of $\mathcal{P}_h$ must be true in the implemented computation of $\mathcal{P}_h$. Because the **timeout** guards are persistent, there is no danger of waiting too long to execute a timeout. Rather, it is important that the counter does not expire too quickly, so that the timeout is not executed in the implementation when the **timeout** guard of $\mathcal{P}_h$ is false. Thus, we need to make sure the counters are reset to large enough values. Note that the fairness assumption of the timed guarded-command level gives a bound the length of a computation during which some guarded command is always enabled but never executed. Thus, any implemented computation of the guarded command level having rank 0 is of a bounded length. Thus, we can use Theorem 5.1 to derive appropriate counter reset values from the refutability guarantees.

Let τ be some positive integer, which we will use as the timing constant for the timed guarded-command protocol. The value chosen for τ is unimportant as far as our results are concerned, but practically speaking, it should be large enough so that a reasonable scheduling mechanism can enforce the fairness assumption of the timed guarded-command level. (This value would depend on such factors as the relative processor speeds and maximum message transmission delays.) For each timeout guard G, let k_G be a refutability guarantee for G. We now show how to construct from $\mathcal{P}_h$ a protocol $\mathcal{P}_l$ in the timed guarded-command level. We begin by making an exact copy of $\mathcal{P}_h$, then modify it as follows:

1. To each process containing a timeout $G \to C$, we include a variable $count_G$, whose values range over the nonnegative integers no greater than some $n \geq \tau k_G$. A state s_l of $\mathcal{P}_l$ will implement a state s_h of $\mathcal{P}_h$ if all channels and all variables from $\mathcal{P}_h$ have the same values in s_h and s_l.

2. To each process containing a timeout $G \to C$, add the guarded command $count_G \neq 0 \to count_G := count_G - 1$. These guarded commands will not implement any guarded commands from $\mathcal{P}_h$.

3. Modify each guarded command $G \to C$ as follows:

   (a) If G refutes a timeout guard G', add to the end of C the statement $count_{G'} := \tau k_{G'}$.

   (b) If G is a timeout guard, replace G with $count_G = 0$, and add to the end of C the statement $count_G := \tau k_G$.

   (c) Otherwise, leave $G \to C$ unchanged.

   The resulting guarded command will implement the guarded command $G \to C$ from $\mathcal{P}_h$.

THEOREM 6.1. *Let $\mathcal{P}_h$ be a regular protocol of the guarded-command level, and let $\mathcal{P}_l$ be the protocol of the timed guarded-command level that is constructed from $\mathcal{P}_h$ by the above construction. Then the set of maximal fair computations of $\mathcal{P}_l$ eventually implements the set of maximal fair computations of $\mathcal{P}_h$.*

Proof. Let σ' be any maximal fair computation of $\mathcal{P}_l$. We first note that since $\mathcal{P}_h$ has no halting states, neither does $\mathcal{P}_l$. To see this, let s' be any state of $\mathcal{P}_l$, and let s be the state of $\mathcal{P}_h$ that s' implements. Since $\mathcal{P}_h$ is live, some internal guarded command $G \to C$ is enabled at s. If G is not a timeout guard, there is a guarded command $G \to C'$ in $\mathcal{P}_l$. Clearly, $G \to C'$ is enabled at s'. Otherwise, if G is a timeout guard, there are guarded commands $count_G = 0 \to C'$ and $count_G \neq 0 \to count_G := count_G - 1$ in the same process of $\mathcal{P}_l$. Clearly, one of these two guarded commands must be enabled at s'. Thus, $\mathcal{P}_l$ has no halting states, so σ' is infinite.

We will show that there is a suffix of σ' that implements some infinite fair computation of $\mathcal{P}_h$. Let σ'' be some suffix of σ' such that every guarded command that is executed in σ'' was executed prior to σ'' in σ'. Let f and g be the functions defining the implementation of the states and transitions, respectively, of $\mathcal{P}_h$. Let $\sigma = \mathcal{I}_{fg}(\sigma'') = s_0 \stackrel{GC_1}{\to} s_1 \stackrel{GC_2}{\to} \cdots$. We will show that σ is an infinite fair computation of $\mathcal{P}_h$.

First, let us show that σ is infinite. Because the new $count_G$ variables added to the protocol by the implementation may only contain natural numbers, it is clear that $\mathcal{P}_l$ can contain at most finitely many consecutive guarded commands of the form $count_G \neq 0 \to count_G := count_G - 1$. Since g maps all other guarded commands in σ' to some guarded command from $\mathcal{P}_h$, it follows that σ must be infinite.

We will now show that σ is a computation of $\mathcal{P}_h$, i.e., that for each i, $s_i \stackrel{GC_{i+1}}{\to} s_{i+1}$ in $\mathcal{P}_h$. We first note that in $\mathcal{P}_l$, there must be states s'_i and s'_{i+1} and a guarded command GC'_{i+1} such that

- $s'_i \stackrel{GC'_{i+1}}{\to} s'_{i+1}$ in $\mathcal{P}_l$;
- s'_i implements s_i;
- s'_{i+1} implements s_{i+1};
- and GC'_{i+1} implements GC_{i+1}.

Let $GC_{i+1} = G \to C$. If G is not a timeout guard, then $GC'_{i+1} = GC_{i+1}$, and clearly $s_i \stackrel{GC_{i+1}}{\to} s_{i+1}$ is in $\mathcal{P}_h$. Furthermore, if G is a timeout guard, and if G is **true** in s_i, then GC'_{i+1} is the guarded command

$$\mathit{count}_G = 0 \to C;\ \mathit{count}_G = \tau k_G \tag{4}$$

and it is easily seen that $s_i \stackrel{GC_{i+1}}{\to} s_{i+1}$ in $\mathcal{P}_h$. We must therefore show that if G is a timeout guard, then G is **true** in s_i.

Suppose G is a timeout guard. Then GC'_{i+1} is of the form of (4) above. Thus, $\mathit{count}_G = 0$ in s'_i. Since GC'_{i+1} is in σ'', it must have occurred previously in σ'; hence, at some state in σ' occurring before s'_i, $\mathit{count}_G = \tau k_G$. Let σ'_0 be the subcomputation of σ' beginning at the last state before s'_i in which $\mathit{count}_G = \tau k_G$ and ending at s'_i. Now let σ_0 be the result of replacing every timeout guarded command $G_0 \to C_0$ in $\mathcal{I}_{fg}(\sigma'_0)$ with **external** $\to C_0$. Clearly, σ_0 is a finite computation of $\mathcal{P}_h \mid G$.

We will now show that G is **true** in the last state of σ_0, which is s_i. In order to do this, we consider the rank of σ_0. By the selection of σ'_0, σ_0 cannot contain either $G \to C$ or a guard that refutes it in $\mathcal{P}_h \mid G$. Therefore, by Theorem 5.1 and the selection of k_G, σ_0 must have rank strictly less than k_G. In σ'_0, the guarded command $\mathit{count}_G \neq 0 \to \mathit{count}_G := \mathit{count}_G - 1$ must have been executed τk_G times. We can therefore divide σ'_0 into a sequence of k_G subcomputations, each containing τ occurrences of $\mathit{count}_G \neq 0 \to \mathit{count}_G := \mathit{count}_G - 1$. Because each of these subcomputations contains at least τ guarded commands, any guarded command that is enabled at the initial state of one of these subcomputations must either be executed or disabled at some point in the same subcomputation. Furthermore, because $\mathcal{P}_h$ is live and execution of $\mathit{count}_G \neq 0 \to \mathit{count}_G := \mathit{count}_G - 1$ has no effect on whether a guard from $\mathcal{P}_h$ is enabled, each of these subcomputations must contain at least one guarded command other than $\mathit{count}_G \neq 0 \to \mathit{count}_G := \mathit{count}_G - 1$. Thus, σ_0 can be divided into k_G nonempty subcomputations, each deriving from one of the k_G subcomputations of σ'_0. In each of these subcomputations, any internal guarded command, other than possibly $G \to C$, that is enabled at the beginning of the subcomputation must be either executed or disabled during the subcomputation. If in addition $G \to C$ were never enabled in σ_0, then σ_0 would have rank at least k_G; however, because σ_0 has rank strictly less than k_G, if follows that G must be **true** at some state in σ_0. Since G is persistent, and $G \to C$ does not occur in σ_0, G is **true** in s_i, the last state of σ_0.

```
process p
begin
    external → enqueue(get(), sbuffer)
    [] receive ack(i) →
        if i = ns then
            if empty(sbuffer) then last := IGNORE
            else last := dequeue(sbuffer)
            ns := ns + 1 (mod B); send data(last, ns)
        count := T * (B_pq + 1)
    [] count = 0 → send data(last, ns); count := T * (B_pq + 1)
    [] count ≠ 0 → count := count − 1
end
```

FIG. 4 The sender of SS-ABP at the timed guarded-command level.

In order to complete the proof, we must show that weak fairness holds for internal guarded commands in σ. This is straightforward. ∎

Fig. 4 shows the sender of SS-ABP after it has been transformed to the timed guarded-command level. Because the receiver contains no **timeout**s, it remains unchanged. Because the there is only one **timeout** in the protocol, we have omitted the subscript from *count*. Note that the refutability guarantee of $B_{pq} + 1$, obtained from Theorem 5.2, has been used in each counter reset.

7. DISCUSSION AND CONCLUSIONS

The main contributions of this paper are as follows:

1. A finite-state guarded-command model has been presented. The definition retains the elegance of the model used by Gouda and Multari [8], but allows us to define finite-state SS protocols. Furthermore, the model incorporates message loss in a flexible way, so that we can reason about protocols that must tolerate message loss, as well as protocols designed for reliable channels.

2. It has been shown that regular protocols defined for the guarded-command model can be implemented in a straightforward way on a lower-level model much closer to actual hardware. This result gives strong theoretical support for practical SS protocols.

3. A general proof technique has been given for showing timeouts in the guarded-command model to be refutable. Showing refutability is the more difficult step in showing a guarded-command protocol to be regular. The proof technique yields a value essential to the translation of guarded-command protocols to timed guarded-command protocols.

We illustrated how our model can be used by giving two examples: an SS version of the ABP, and Dijkstra's k-state protocol. Our examples illustrate how our definitions balance the elegance of a guarded-command message-passing model with the necessity of a finite set of states. We do not need to reason explicitly about what happens when a channel is full, because messages can be lost any time the number of messages on the channel exceeds the reliability threshold. However, we are able to use the channel bounds when reasoning about the number of messages in the channels. Furthermore, the second example illustrates the use of reliable channels in a protocol definition.

One aspect of our model that needs further refinement is the use of **external** guards. One motivation for having **external** guards is to facilitate protocol composition. However, the simple mechanism that we have presented lacks adequate facilities for exchanging information between two protocols. In order to be able to reason about composed protocols separately, we need a mechanism such as a parameter list that will allow us to separate the protocols more completely than shared variables will allow us to do. Adding this functionality does not appear to be hard, but we have chosen not to obscure the main results of this paper by including these additional details.

Finally, there is the issue of implementing protocols on an actual network. We believe the timed guarded-command level facilitates such an implementation. Each atomic action at the timed guarded-command level contains at most one **receive**, which must precede any **send**s. This structure makes it possible to implement the coarse-grained atomicity of the timed guarded-command level with much finer-grained atomic actions (e.g., individual **send** or **receive** operations) using no synchronization; i.e., the implementing computation consists of a serializable execution of fine-grained atomic actions (cf. [3]). Furthermore, round-robin scheduling can be used to enforce the fairness condition of the timed guarded-command level. The tedious part of this activity is computing the timing constant. We are currently working out the details of this implementation.

Once the above concerns are adequately addressed, the following procedure can be used to design and implement SS protocols:

1. Design the protocol at the guarded-command level, using convenient reliability thresholds and variables for channel bounds.

2. Prove the correctness of the protocol and that the timeouts are persistent and refutable.

3. Translate the protocol to the timed guarded-command level, using a variable for the timing constant.

4. Compose the translated protocol with SS ABP or some other SS transport protocol that has been translated to the timed guarded-command level. The transport protocol will provide the specified reliability threshold, so that the resulting protocol assumes unreliable channels.

5. Translate the resulting protocol to an actual network. At this point, the variables representing channel bounds and the timing constant will be replaced by platform-specific values (or safe approximations).

We believe that steps 3-5 above can be automated.

ACKNOWLEDGMENTS

We are grateful for the many comments provided by the anonymous reviewers. Their suggestions helped us greatly in improving the presentation of these results.

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] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Co., 1987.

[4] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

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

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

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

[8] M. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Trans. Comput.*, 40(4):448–458, 1991.

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

RODNEY R. HOWELL received a B.S. in Computer Science from Wichita State University in 1984 and a Ph.D. in Computer Science from The University of Texas at Austin in 1988. He is currently an Associate Professor at Kansas State University. His research interests include self-stabilization, theory of computation, and algorithm design and analysis.

MIKHAIL NESTERENKO received his Engineer-Mathematician degree from Moscow Engineering Physics Institute, Russia, in 1993 and a Ph.D. in Computer Science from Kansas State University in 1998. Mikhail is currently an Assistant Professor at Kent State University. His research interests are in fault-tolerance of distributed systems.

MASAAKI MIZUNO received B.S. and M.S. degrees in Electrical Engineering from Keio University, Japan, and an M.S. degree in Computer Science from Pennsylvania State University. In 1987, he received a Ph.D. degree in Computer Science from Iowa State University. Since then, he has been with the Department of Computing and Information Sciences at Kansas State University, where he is currently a Professor. His research interests are in operating systems and distributed systems.