To appear in Proc. 8th WDAG, October 1994

Self-Stabilization by Local Checking and Global Reset

Extended Abstract

Baruch Awerbuch[1 2], Boaz Patt-Shamir[2], George Varghese[3] and Shlomi Dolev[4 5]

[1] Dept. of Computer Science, Johns Hopkins University
[2] Lab. for Computer Science, MIT
[3] Dept. of Computer Science, Washington University
[4] Dept. of Computer Science, Texas A&M University
[5] School of Computer Science, Carleton University

Abstract. We describe a method for transforming asynchronous network protocols into protocols that can sustain any transient fault, i.e., become self-stabilizing. We combine the known notion of local checking with a new notion of *internal reset*, and prove that given any self-stabilizing internal reset protocol, any locally-checkable protocol can be made self-stabilizing. Our proof is constructive in the sense that we provide explicit code. The method applies to many practical network problems, including spanning tree construction, topology update, and virtual circuit setup.

1 Introduction

A network protocol is called *self-stabilizing* (or *stabilizing* for short) if when started from an arbitrary state, it eventually exhibits the desired behavior. In the context of computer networks, a self-stabilizing system may have an initial state with arbitrary messages at the links and arbitrary corruption of the state variables at the nodes. The practical appeal of stabilizing protocols is that they are *simpler* (i.e., they avoid a slew of mechanisms to deal with a catalog of anticipated faults), and they are *more robust* (e.g., they can recover from transient faults such as memory corruption as well as common faults such as link and node crashes).

Since the pioneering work of Dijkstra [11], the theory of self-stabilization has been extensively studied (e.g., [9, 16, 12, 2, 5]). While most of the work was directed at self-stabilization of specific tasks, some work was devoted to designing general algorithmic transformers that take a protocol as input, and produce as their output a self-stabilizing version of that protocol. These transformers typically exhibit trade-offs between their generality (i.e., the range of input protocols they can transform) and the efficiency of the resulting protocols. One such general transformation is given by Katz and Perry [16], where they show how to compile an arbitrary asynchronous protocol into a stabilizing equivalent. Briefly, the idea in [16] is that a leader node periodically takes "snapshots" of the global network state, and resets the system if some inconsistency is detected. We call this method *global checking and correction.* Due to its generality, this transformation is expensive in terms of space and communication; another drawback of this approach is that it requires an additional self-stabilizing mechanism that maintains routes that connect all nodes to some leader.

Afek, Kutten and Yung [5] suggested that global inconsistency could sometimes be detected by checking the states of neighbors — i.e., by local means. Using the idea of detecting faults locally and correcting them by a global operation, a stabilizing spanning-tree construction is developed in [5]. In [2], Arora and Gouda propose the use of distributed reset to maintain diffusing computations. Their reset protocol requires an underlying stabilizing spanning tree.

The idea of local detection of faults is formalized in [6, 7, 28] under the name of *local checking*. In [6, 28], the class of *locally correctable* protocols is also defined; these are protocols that can reach "good" global states by means of local correction actions. In [6, 28], a transformer that uses local checking and local correction is described. The transformer of [6] is efficient, but it can be applied only to protocols that are both locally checkable and locally correctable. Unfortunately, many interesting network protocols can be shown to be locally checkable, but not locally correctable.

In this paper, motivated on the one hand by the inefficiency of the transformer of [16], and by the narrowness of the transformer of [6] on the other hand, we introduce a new algorithmic transformer that can be used to make a wide class of protocols self-stabilizing. The idea is to combine local checking and global correction: bad states are detected by local checking mechanism, a global correction action (called "reset") is used to recover from faults. We contend that local checking and global reset is the right balance in many practical situations. First, we argue that global detection mechanisms such as the self-stabilizing snapshot [16] incur unnecessary large overhead (in terms of time, space and communication) practically always, since networks are fairly failure-free. Local checking detects faults quickly, and it can be done, as we show in this paper, with only a small increase in communication cost. Secondly, as mentioned above, there are many protocols that are locally checkable but not locally correctable (e.g., spanning tree construction and topology update [27, 20, 21, 22]). In these cases we are forced to use other techniques — e.g., reset.

Even though resetting an entire network may seem drastic and inefficient, there is evidence that this is not the case. For instance, consider routing protocols. The stabilization time of our method (using the best reset protocols) is proportional to a cross-network latency, which is the time that takes for many protocols to compute their results anyhow, *even after being started in a good state.* Empirical results also support the claim that resets perform quite well in practice. Specifically, DEC SRC's AN-1 network [25] employs a variant of global reset for dealing with topology changes (by making a reset request whenever a link fails or comes up).[6] The AN-1 designers found that the protocol recovered from link failures very fast [23]. The reason is that usually the routing protocol only operates for a small fraction of the time at a node; the remaining processing is devoted to forwarding data. During a reset, however, no data forwarding is done; all processing and bandwidth is devoted to the reset. The moral from the AN-1 experience is that reset schemes work well for small sized networks; for larger networks, the same approach should work if the routing protocol is hierarchical [15] and each level is reset independently.

The main result of this paper is a precise description and statement of the method of local checking and global reset. We provide formalization, analysis, and code. We believe that in doing so we contribute something that will help both theoreticians and practitioners. We remark that the ideas of local checking and global reset are not new; for instance, the stabilizing spanning tree protocol of [5] uses local detection, and Arora and Gouda [2] use

[6] The AN-1 reset is performed using a version of Finn's unbounded counter protocol [14].

reset to maintain diffusing computations. The contribution of this paper is in introducing a general transformer that can be used to stabilize any locally checkable protocol. The description of the transformer entails a description of a local checking mechanism, detailed requirements that the reset protocol being used must meet, and a description of the way to construct the resulting self-stabilizing protocol.

It is important to observe that the classical notion of reset [14, 1, 6] is insufficient for our purposes. In these papers, the task is specified in terms of an external entity that triggers the reset: for example, the reset can be triggered by a change in the topology (e.g., link crash). The important point is that this specification formalizes a reset that is invoked regardless of the way it affects the system. Below, we call such resets *external*. Notice that an external reset is inadequate for a general transformer: in our method there is no external entity. We have a protocol, which is checked by the local checking mechanism, that can trigger the reset, which in turn changes the state of the protocol being checked. If while resetting inconsistent states of the original protocol are created, the local checking mechanism might invoke the reset again, resulting in an endless vicious cycle of reset invocations. Therefore, another notion of reset is required in this setting. One of the contributions in this paper is an appropriate specification of a stronger reset, hereafter called *internal reset*. Intuitively, the requirement of external reset that there are only finitely many reset invocations is replaced in internal reset by a specification that guarantees that when used properly, the reasons for invoking reset eventually disappear.

Interestingly, some reset implementations [1, 6] are known to produce intermediate global inconsistencies [1, 28]. In this paper, however, we show that a certain "pairwise consistency" is sufficient; fortunately, it turns out that the above protocols (although designed as external resets) meet the requirements of internal reset.

The remainder of the paper is organized as follows. We start, in Section 2, with an overview of the network model and the definition of stabilization used in this paper. In Section 3 we define the notion of local checkability (this is a straightforward formulation of the ideas in [5, 6]). In Section 4 we give a definition of the requirements of internal reset protocols. Then, in Section 5, we give our main result, that connects the known notion of local checkability with the new notion of internal reset. Namely, we present a theorem that says that any locally checkable protocol can be made self-stabilizing using any self-stabilizing reset protocol. We sketch a specific implementation of the local checking process, and outline a proof of correctness for the combined protocol (explicit code is omitted from this extended abstract.) Some applications of our main result are mentioned in Section 6.

2 Model

In this section we describe our network model. We first review briefly the underlying formal model of Input/Output Automata (see [17, 18] for full definitions), and establish the notation we use throughout this paper. We also formalize the notion of self-stabilization in this framework. In the second part of this section, we specify the network model we are dealing with in this paper.

IO Automata, Stabilization, Time Complexity. An Input/Output Automaton (abbreviated IOA henceforth) is a state machine whose state transitions are given labels called *actions*.

There are three kinds of actions. The environment affects the automaton through *input actions* which must be responded to in any state. The automaton affects the environment through *output actions*; these actions are controlled by the automaton. *Internal actions* only change the state of the automaton without affecting the environment. Formally, an IOA $\mathcal{N}$ is defined by a *state* set $S(\mathcal{N})$, an *action* set $A(\mathcal{N})$, a *signature* $G(\mathcal{N})$ (that classifies the action set into input, output, and internal actions), a *transition relation* $R(\mathcal{N}) \subseteq S(\mathcal{N}) \times A(\mathcal{N}) \times S(\mathcal{N})$, and a non-empty set of initial states $I(\mathcal{N}) \subseteq S(\mathcal{N})$. We omit the automaton's name when it is clear from the context. An action a is said to be *enabled* in state s if there exist $s' \in S$ such that $(s, a, s') \in R$. Input actions are always enabled. For an automaton $\mathcal{N}$ and non-empty set $L \subseteq S(\mathcal{N})$, we define $\mathcal{N}|L$ to be the automaton that is obtained from $\mathcal{N}$ by setting the initial states to be L. In this paper we often deal with *uninitialized IOA* for which $I = S$ and S is finite. IOAs communicate by means of shared actions. More formally, IOAs can be composed (under certain compatibility conditions) to generate a composite state machine; an action which is output of one of the components and input of the other is performed simultaneously.

When an IOA "runs" it produces an execution. Formally, an *execution fragment* is an alternating sequence of states and actions $(s_0, a_1, s_1, \ldots)$, such that $(s_i, a_i, s_{i+1}) \in R$ for all $i \geq 0$. An execution fragment is *fair* if any internal or output action that is continuously enabled eventually occurs.[7] An *execution* is an execution fragment that begins with an initial state and is fair. A *schedule* is a subsequence of an execution consisting only of the actions. A *behavior* is a subsequence of a schedule consisting only of its input and output actions. Each IOA generates a set of behaviors. An IOA A *implements* another IOA B if the behaviors of A are a subset of the behaviors of B. For stabilization, we weaken this definition and require only that A eventually exhibit a behavior of B. Formally, we say that A *stabilizes* to B if every behavior β of A has a suffix which is a behavior of B. Note that this definition (based on a definition by Nancy Lynch) is formulated in terms of external behavior, as opposed to a (somewhat circular) state-based definition.

For time complexity, we use the timed IOA model of [18] (see [28] for formal details). Informally, we assume that every internal or output action that is continuously enabled occurs in 1 unit of time. We say that A stabilizes to B in time t if every behavior of A has a suffix that occurs within time t and is a behavior of B. The *stabilization time* from A to B is the smallest t such that A stabilizes to B in time t. For any automaton $\mathcal{N}$, define $\mathcal{N}(x)$ to be identical to $\mathcal{N}$ except that the time associated with each action is now x time units instead of 1 time unit.

Network Model. For the remainder of this paper we fix an underlying network topology, modeled by a directed symmetric graph $G = (V, E)$ with unique node identifiers.[8] Each node $v \in V$ represents a processor, and each directed edge represents a unidirectional communication link. We denote the number of network nodes by $n = |V|$, and the network diameter is denoted $d = diam(G)$. Each node and link is modeled by an IOA. Below, we describe verbally the links and node automata. Formal definitions are omitted from this abstract.

[7] The IOA model specifies fairness in terms of equivalence classes; here we assume each action is in a separate class.

[8] A graph $G = (V, E)$ is called symmetric if for all $u, v \in E$ we have that $(u, v) \in E$ implies $(v, u) \in E$.

In our model, links have bounded storage, i.e., only a bounded number of outstanding packets are stored on each link at any instant. The justification for this assumption is twofold: first, not much can be done with unbounded links in a stabilizing setting [13], and secondly, real links are inherently bounded anyway. In this paper we abstract this property by postulating that a link can store at any given instant at most one outstanding packet. Formally, a link from node u to node v is modeled as a queue $Q_{u,v}$ that can store at most one packet from some packet alphabet Σ at any instant. The external interface to the link (see Figure 1) includes an input action $\text{SEND}_{u,v}(p)$ (interpreted as "send packet p from u"), an output action $\text{RECEIVE}_{u,v}(p)$, (interpreted as "deliver packet p at v"), and an output action $\text{FREE}_{u,v}$ (interpreted as "the (u, v) link is currently free").[9] If a $\text{SEND}_{u,v}(p)$ occurs when $Q_{u,v} = \phi$, the effect is that $Q_{u,v} = \{p\}$; when $Q_{u,v} = \{p\}$, $\text{RECEIVE}_{u,v}(p)$ is enabled, and when it is taken, its effect is to set $Q_{u,v} \leftarrow \phi$; when $Q_{u,v} = \phi$, $\text{FREE}_{u,v}$ is enabled. If $\text{SEND}_{u,v}(p)$ occurs when $Q_{u,v} \neq \phi$, there is no change of state (intuitively, the incoming packet is just dropped). We note that by our timing assumptions, a packet stored in a link will be delivered in one unit of time.

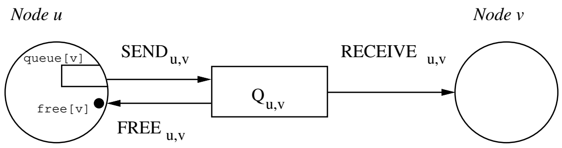

Fig. 1. *Schematic representation of a single link, connecting queued node automaton u to v. The link from v to u is symmetric and is not shown.*

A node automaton u has, for each neighbor v, output actions $\text{SEND}_{u,v}(p)$ to send packets to v, input actions $\text{RECEIVE}_{v,u}(p)$ to receive packets from v, and an input action $\text{FREE}_{u,v}$ to obtain indications of the (u, v)-link state. In this paper, we use a special node automaton which we call a *queued node automaton*. A queued node automaton u has the following discipline for sending packets. It has a bounded output queue called *queue*$[v]$ and a boolean flag *free*$[v]$ for each neighbor v of u (see Figure 1). Whenever a $\text{FREE}_{u,v}$ action is received from the link at u, the *free*$[v]$ flag is set. A $\text{SEND}_{u,v}(p)$ action is only performed when p is the head of *queue*$[v]$ and *free*$[v]$ is set; its effect is to clear *free*$[v]$.

Queued node automata allow us to easily superimpose a local checking process. The local checking process at each node needs access to the state of the node and also requires the sending of control packets. (The requirement of access to state rules out the possibility of formalizing the local checking process as a separate automaton). Queued node automata use a particular discipline for sending data packets on a link; this discipline makes it easy to multiplex data and control packets on each link. Any node automaton requires some discipline anyway to deal with bounded links. Thus the use of a particular discipline is not overly restrictive and makes it easy to add local checking.

For the given graph $G = (V, E)$, we define the automaton for G by the composition

[9] Our convention for action subscripts is that the first represents the sender and the second represents the receiver.

of node automata for each $u \in V$ and link automata for each edge $(u, v) \in E$. Our object of study in this paper, called hereafter *network automaton for* G, is an automaton for G in which all node automata are queued node automata.

3 Local Checkability

In this section we formalize the notion of local checkability. For the remainder of this section, fix a network automaton $\mathcal{N}$ for a given graph $G = (V, E)$. We start by defining the notion of *subsystems*.

Definition 1. Let $(u, v) \in E$. The (u, v)*-link subsystem* of $\mathcal{N}$ is the composition of the automata for nodes u and v, and edges (u, v) and (v, u).

For a state $s \in S(\mathcal{N})$ and node $u \in V$, let $s|u$ denote the projection of s onto the node automaton of u; similarly, for $(u, v) \in E$, let $s|(u, v)$ denote s projected onto the automaton for link (u, v). When $\mathcal{N}$ is in state s, the (u, v) subsystem state is characterized by the 4-tuple $(s|u, s|(u, v), s|(v, u), s|v)$. We now define the notions of predicates and local predicates.

Definition 2. A *predicate* of $\mathcal{N}$ is a subset of the states of $\mathcal{N}$. A *local predicate* $L_{u,v}$ of $\mathcal{N}$, where $(u, v) \in E$, is a subset of the states of the (u, v) subsystem. A state $s \in S(\mathcal{N})$ is said to *satisfy* a local predicate $L_{u,v}$ for (u, v) iff $(s|u, s|(u, v), s|(v, u), s|v) \in L_{u,v}$.

We shall also use the following standard definition.

Definition 3. A local predicate $L_{u,v}$ of $\mathcal{N}$ is called *stable* if for all transitions (s, a, s') of $\mathcal{N}$, we have that if s satisfies $L_{u,v}$ then so does s'.

We now arrive at our definition of local checkability.

Definition 4. Let $\mathcal{L} = \{L_{u,v}\}$ be a set of local predicates, and let Π be any predicate of $\mathcal{N}$. A network automaton $\mathcal{N}$ is *locally checkable for* Π *using* $\mathcal{L}$ if the following conditions hold.

(1) For all states $s \in S(\mathcal{N})$, if s satisfies $L_{u,v}$ for all $L_{u,v} \in \mathcal{L}$, then $s \in \Pi$
(2) There exists $s \in S(\mathcal{N})$ such that s satisfies $L_{u,v}$ for all $L_{u,v} \in \mathcal{L}$.
(3) Each $L_{u,v} \in \mathcal{L}$ is stable.

Certainly the most intriguing condition in Definition 4 above is (3). This condition is introduced so that periodic verification of local predicates can still be useful for fault detection. More specifically, it is aimed to rule out the case of an "evasive violation:" there are examples in which a (global) predicate can be expressed as a conjunction of local predicates, and such that under a certain schedule, whenever a local predicate is checked it turns out to be true; however, it may be the case that the global predicate never holds in this execution! Intuitively, the problem stems from the fact that a fault could "travel" through the network, escaping detection every time local predicates are verified. Imposing the stability condition guarantees that if a local predicate is known to hold, it will continue to hold through the rest of the execution.

Superficially, the stability requirement in Def. 4 may seem too strong a condition. We support our definition by the observation that many protocols, called *locally extensible*, can be made to have this property by means of a simple transformation. Informally, a protocol is said to be locally extensible if any correct state of any pair of overlapping subsystems can be extended to a globally correct state. Details can be found in [28].

4 Internal Reset

In this section we give a specification for an internal reset protocol in terms of observable behaviors. Intuitively, the goal of any reset protocol is to provide, upon request, a consistent *signal* common to all nodes in the network. The time point corresponding to the signal can be used to locally restart the protocol that we wish to reset. An internal reset protocol has to satisfy additional conditions regarding its output before termination. The basic idea in the specification below is an analogy to a pair of nodes connected by a data link protocol. Essentially, an internal reset protocol generalizes the guarantees of a data link protocol [3] to the *whole network*.[10]

4.1 Interface of Reset Protocols

We now define the interface of reset protocols (this interface is common to both external and internal resets). Let us start with some intuition. It is desired that a reset protocol be superimposed on any other network protocol, say P, which we think of as a "user." The reset may be invoked at any node, and its effect is to output signals at all the nodes in a consistent way. The notion of consistency is expressed in terms of the messages sent and received by the nodes. We therefore assume that the reset protocol has control over the messages sent and received by the user.

Motivated by this consideration, we define the external interface for a reset protocol at some node u as shown in Figure 2. The request action allows a local user at a node to invoke the reset protocol, and the signal action is used by the reset to provide a consistent time point to the user. The reset protocol also regulates all message traffic of P to and from the node. To avoid confusion, we call the messages generated by, and destined for the application P *messages*. We assume that these messages are drawn from some alphabet Σ. In addition, reset modules can communicate among themselves; the messages that reset protocols send and receive (including those it relays to and from P) are drawn from the links alphabet, which we denote by $\Sigma_{data} \supset \Sigma$. These message are called *packets*. Intuitively, messages output by users are relayed by the reset modules between network nodes using packets.

Formally, a $\text{SENDM}_{u,v}(m)$ action allows P at u to send a message to neighbor v, and a $\text{RECEIVEM}_{v,u}(m)$ action allows the reset service to deliver a message m from node v to the user at node u. The $\text{FREEM}_{u,v}$ action indicates that the reset service is ready to accept another message from u to node v. Thus the external interface between a reset service and its P-users mimics the link interface (cf. Figure 1) with packets replaced by messages. The reset service at node u, however, offers two additional actions: an input action REQUEST_u,

[10] Almost identical specifications are provided by virtual circuit protocols [26] and transport protocols [27].

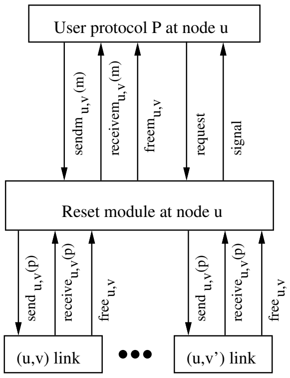

Fig. 2. *Interface specification for reset service*

used to enable the user to request a reset, and an output action SIGNAL$_u$, that informs the user that a reset has been completed at that node.

4.2 Behavior Specification for Reset Protocols

We now specify the requirements of an internal reset protocol. (The main difference between this specification and external reset [6] is in the consistency requirement below.) Our specification is parameterized by the *response time* of the protocol, denoted R in the sequel. This is convenient because different reset protocols have different response times.

Before describing the behaviors of reset, we impose a *well formedness* condition on the behaviors of the user P. Intuitively, we rule out cases where P injects messages when it is not sure that the link is free. Formally, a behavior is *well-formed* if between any two SENDM$_{u,v}$ events there is a FREEM$_{u,v}$ event.

To specify the requirements of an internal reset protocol, we define properties we call *timeliness, causality* and *consistency*. An internal reset protocol is required to be timely, causal, and consistent. We define these properties below.

Intuitively, a behavior is timely if, in the absence of reset requests, the reset protocol relays messages and "free" events to and from the node in constant time. Formally, we have the following definition.

Definition 5 (Timeliness). A behavior β is *timely* if for all $(u, v) \in E$ the following holds.

(1) At any point in β, either FREEM$_{u,v}$ occurs within constant time, or else SIGNAL$_u$ or SIGNAL$_v$ occur within $O(R)$ time.

(2) Every SENDM$_{u,v}(m)$ event in β is followed by RECEIVEM$_{u,v}(m)$ in constant time, or else SIGNAL$_u$ or SIGNAL$_v$ occur in $O(R)$ time.

Intuitively, a behavior is causal if reset signals are only caused by reset requests and reset requests result in reset signals. Formally, we have the following definition.

Definition 6 (Causality). A behavior β is *causal* if the following holds.

(1) For any signal event there is some request event that occurs within the preceding $O(R)$ time.
(2) For any REQUEST_u event, there is a SIGNAL_u event that occurs within the following $O(R)$ time.
(3) For any signal event, there are signal events that occur at all $v \in V$ within the preceding and following $O(R)$ time units.

Note that condition (1) of the causality definition guarantees termination: if reset requests stop, all signal events will stop in $O(R)$ time.

The intuition for the property of consistency of behaviors is harder to capture. As mentioned above, we would like to extend guarantees made by data link protocols to networks. Fix a behavior β. Our first step is to define correspondence between send and receive events: for any $\text{RECEIVEM}_{u,v}$ event a_i, we define its *corresponding send event* a_j to be the first $\text{SENDM}_{u,v}$ event before a_i in β, such that there is no other $\text{RECEIVEM}_{u,v}$ event between a_j and a_i. Next, we define the notion of *signal intervals* at a node u: these are the subsequences of β demarcated by SIGNAL_u events; if the number of SIGNAL_u events is finite, then u's *final interval* is the infinite interval that begins with the last SIGNAL_u event.

For a given signal interval I_u at a node u, we define, for each neighbor v of u, the sequence $s_v(I_u)$ which consists of the messages u sends to v in I_u, and the sequence $r_v(I_u)$, which consists of the messages u receives from v in I_u.

With these concepts, we arrive at the central definition of the *mating relation* among signal intervals.

Definition 7. Given a behavior, let $\mathcal{I}_u$ denote the set of signal intervals at $u \in V$, and let $\hat{\mathcal{I}}$ denote the set of all signal intervals. A *mating relation* "$\leftrightarrow$" is a reflexive, symmetric relation over $\hat{\mathcal{I}} \times \hat{\mathcal{I}}$ that satisfies the following conditions for any $(u, v) \in E$.
(1) If $\text{RECEIVEM}_{u,v}(m)$ occurs in I_v and the corresponding $\text{SENDM}_{u,v}(m)$ event occurs in I_u, then $I_u \leftrightarrow I_v$.
(2) For all $I_u \in \mathcal{I}_u$, there exists at most one $I_v \in \mathcal{I}_v$ such that $I_v \leftrightarrow I_u$.
(3) If $I_v \leftrightarrow I_u$, then $r_u(I_v)$ is a prefix of $s_v(I_u)$.
(4) For all $I_v \leftrightarrow I_u$, I_v is final iff I_u is final. If I_v and I_u are final, then $s_u(I_v) = r_v(I_u)$.
(5) If $I_v \leftrightarrow I_u$ and $I'_v \leftrightarrow I'_u$, then I_v precedes I'_v at v iff I_u precedes I'_u at u.

Note that only final intervals enjoy a "full guarantee" in the sense that the sequence of messages received is equal to the sequence of messages transmitted. Mating non-final intervals are allowed to "lose" the tail of the sequence.

We can now define the two flavors of consistency we consider.

Definition 8 (Consistency). A behavior β is called *weakly consistent* if every RECEIVEM action in β has a corresponding send event, and there exists a mating relation over the set of signal intervals of β. A behavior is called *strongly consistent* if the behavior is weakly consistent and, in addition, the mating relation is transitive.

Note that even for weak consistency, there is a transitive mating relation between *final* signal intervals. Since the mating relation is symmetric and reflexive, transitivity allows us to partition final signal intervals into equivalence classes, so that intervals "communicate" only with intervals in the same class: this seems to be the essence of network synchronization.

To specify an internal reset, we might require that any well-formed behavior be timely, consistent, and causal. Unfortunately, it appears that no implementation can stabilize to this set of behaviors! Messages stored in the initial state can result in executions in which all suffixes have some receive event that does not correspond to a send; this violates mating. Thus we settle for the following definition.

Definition 9 (Internal Reset). A protocol $\mathcal{R}$ is a *weak (strong) internal reset* if any well-formed behavior of $\mathcal{R}$ is a suffix of some timely, causal and weakly (resp., strongly) consistent behavior.

We remark that one difficulty with this definition is the hardness of proving that an IOA stabilizes to behaviors that are *suffixes* of behaviors of a second IOA.[11]

We note that the stabilizing reset protocols of [16] and [2] appear to be strong internal resets. The protocol of Katz and Perry [16] requires a stable set of paths to a fixed leader, and it stabilizes in $O(n^2)$ time. The protocol of Arora and Gouda [2] requires a stable directed spanning tree, and its stabilization time (given a tree) is $O(d)$. The reset protocol of Awerbuch, Patt-Shamir and Varghese [6] is designed as an external reset, but in fact it also satisfies the requirements of weak internal resets (see [28]). The stabilization time of [6] is $O(n)$, but enjoys the advantage that it does not require precomputed structures. This can be used, for example, to get a stabilizing spanning tree protocol with $O(n)$ stabilization time. Awerbuch and Ostrovsky [8] describe a modification of the protocol of [6] that reduces the space requirement to $O(\log^* n)$, at the expense of increasing the stabilization time to $O(n \log^2 n)$. Lastly, we remark that Awerbuch *et al.* describe in [4] a stabilizing spanning tree protocol that stabilizes in $O(d)$ time; it appears that that protocol can be combined with the reset protocol of [2] to yield an $O(d)$ strong internal reset protocol.

5 Self-Stabilization by Local Checking and Global Reset

In this section we state and sketch a proof of our main result. Basically, it says that any protocol that is locally checkable for some global property, can be transformed into an equivalent protocol, that stabilizes to a variant of the protocol in which the desired property holds in its initial state. This transformation increases the time complexity of the original protocol as follows. First, the stabilization time of the resulting protocol is $O(R)$, where R is the response time parameter of the internal reset protocol used (see Definition 6); and secondly, the behaviors of the transformed automaton are slowed down by a constant factor (due to the overhead of local checking).

We start this section with a statement of the main theorem.

Theorem 10. *Let $\mathcal{N}$ be any network automaton that is locally checkable for some predicate Π using a set of local predicates $\mathcal{L}$. Then there exists some constant c, and some uninitialized network automaton $\mathcal{N}^+$, such that $\mathcal{N}^+$ stabilizes to the behaviors of $\mathcal{N}(c)|\Pi$ in $O(R)$ time, where R is the response time of any stabilizing internal reset protocol.*

[11] In [28], relevant properties of suffixes are extracted and used to prove that the reset protocol of [6] is a weak reset in the above sense. This approach works but is messy; we prefer here to concentrate on what is needed for global correction.

Due to lack of space, we do not give a full proof here; below, we describe the transformation and outline its analysis. Details can be found in [28].

The transformation. Let $\mathcal{R}$ be an automaton for internal reset. Our first step is to compose the node automata with $\mathcal{R}$ (this requires us to first rename the packet sending actions of the node automata in Figure 1 to be message sending actions as in Figure 2).

Next, recall that by the assumption of local checkability of Π, there exists a set $\mathcal{L} = \{L_{u,v}\}$ of local predicates for Π. We now add a periodic checking process to each (u, v) subsystem to check whether the local predicate $L_{u,v}$ holds. This is implemented by having the lower ID node (say u) periodically initiate a *snapshot* of the (u, v) subsystem. This is a slight modification of the general Chandy-Lamport snapshot protocol [10]. More specifically, this is done as follows (see Figure 3). Node u sends a "SnapRequest" message, which is responded to by v with a "SnapResponse" message. When the response arrives, the snapshot is composed at u, and is then checked to see if $L_{u,v}$ holds; if not, u makes a reset request to the internal reset protocol $\mathcal{R}$.

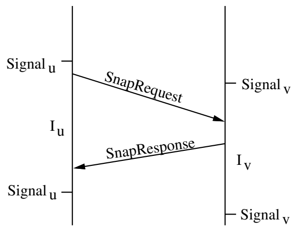

Fig. 3. *Correct Snapshots after nodes u and v have each performed a signal event. Time increases downwards.*

If there are user messages to be sent, one user message is sent between every two invocations of the snapshot; this slows down the communication of the user by a constant factor. The snapshot is made stabilizing by numbering SnapRequest and SnapResponse packets with a 4-valued counter, and by retransmitting SnapRequests until a SnapResponse with matching counter is received. The counter is incremented mod 4 on every invocation. When a SIGNAL$_u$ action is taken by the internal reset protocol, node u initializes the local state of $\mathcal{N}$ to some prespecified initial state $I_u$and in addition, initializes its snapshot variables.[12]

Finally, we rename messages to packets again since the transformation so far produces node automata that send messages.

Sketch of Analysis. First, the specific local checking process outlined above can be shown be stabilizing: in [28, 29] it is proven that after the fifth invocation, all snapshots produce accurate reports of the subsystem's state. We therefore have a correct local checking process, and we assume that we are given a correct internal reset protocol. The main difficulty in

[12] We choose the initial states I_u such that if $s|u = I_u$ for all nodes u and all links are empty in state s, then $s \in \Pi$. The definition of local checkability implies the existence of such a state s.

proving that the transformation works is to show termination. We need to prove that in $O(R)$ time, the local checking process will stop making reset requests. We start by arguing that eventually, the SnapRequest and SnapResponse are matched correctly. Define an ISI (for Initialized Signal Interval) at node u to be a signal interval that starts with a SIGNAL_u event. In any behavior, all but possibly the first signal interval at a node are ISIs. Suppose signal intervals keep recurring at node u. Then eventually any SnapResponse packet that crosses the (v, u) link (see Figure 3) is sent in an ISI at v (say I_v), and is received in an ISI at u (say I_u). If this SnapResponse is accepted as a matching response, the mating property can be used to deduce that the corresponding SnapRequest was sent in I_u and received in I_v: this follows from the fact that by the code, all snapshot variables and counters are initialized at the start of an ISI.

Thus any snapshot that completes after this point will have been completely executed within an ISI at u and an ISI at v. But all communication between two ISIs is exactly what might have occurred in some asynchronous execution of the (u, v) subsystem in which u is initialized at the start of I_u, v is initialized at the start of I_v and the two links are empty. In such an asynchronous execution, $L_{u,v}$ holds at the start; also, because $L_{u,v}$ is stable, it will continue to hold regardless of any messages received from other subsystems that may still be "incorrect," and the snapshot will not detect a violation. Thus if signal intervals keep occurring, local checking (of the (u, v) subsystem and all other local subsystems) will stop making reset requests. Hence, the causality property of reset implies that there will be a final signal interval at all nodes, which corresponds to a behavior of $\mathcal{N}$. This argument completes the proof of Theorem 10.

We remark that the proof hinges on two crucial points. First, it is important that each local predicate is stable: since a weak internal reset does not guarantee a transitive mating relation, it is possible for a subsystem to receive "inconsistent" messages from other adjacent subsystems during non-final intervals. The stability guarantees that such events will not trigger further inconsistencies. Note, however, that stability is required anyway in order to do local checking via snapshots: it is not an extra condition required for the global correction.

The second crucial point is the mating relation for signal intervals; without it, a snapshot at u could span two signal intervals at v, if there is another SIGNAL_v event between the receipt of the SnapRequest and the sending of the SnapResponse. This in turn could lead to persistent incorrect snapshots and the possibility of non-termination. Pairwise consistency is enough because we have taken "local" to mean a pair of neighbors; it would be insufficient for more general notions of locality (e.g., 3 node subsystems). A careful argument is more involved: we defer it to the final paper.

6 Conclusion

The main result of this paper is Theorem 10, which shows how can locally checkable protocols be made stabilizing automatically, using local checking and an internal reset. We have used this result to rigorously prove the correctness of a spanning tree protocol [28]. Our protocol computes the tree as a shortest paths tree rooted at the minimum ID node. This is the same idea used in the widely deployed IEEE 802.1 spanning tree protocol [21]. The main problem in the basic approach is that fictitious IDs may spoil the computation. The

802.1 protocol overcomes this problem by using timers. To get rid of fictitious IDs even in worst cases, the timeout periods are large always. By contrast, our protocol uses reset, and its stabilization time is proportional to actual network delays (which are in most cases significantly smaller the worst possible). In [28] we show that if the local predicates have a certain structure, then local checking can be done by having each node periodically send its state to its neighbors (without the need to implement local snapshots). The spanning tree algorithm has this structure and so the resultant protocol is quite simple.

Another application is topology update. Many existing networks [19, 20] use sequence numbers to broadcast topology information to all nodes. If the counter being used ever gets to the maximum value, a large timeout is used for recovery. We propose that instead of these large timeouts, global reset can be used. Similarly, the AN-1 network [23] uses a simple "large counter" to reset the network after topology changes [14]. This simple reset is more efficient than the stabilizing reset protocols but is vulnerable to counter errors. The AN-1 designers have suggested [24] that stabilizing reset could be used to reset the simple reset protocol when the local predicates of the simple reset protocol are violated.

We believe that all of the above provides strong indications that the idea of self-stabilization by local checking and global reset is a viable practical technique, as well as a convenient theoretical tool.

Acknowledgments

The first and the second authors are partially supported by Air Force Contract TNDGAFOSR-86-0078, ARPA/Army contract DABT63-93-C-0038, ARO contract DAAL03-86-K-0171, NSF contract 9114440-CCR, DARPA contract N00014-J-92-1799, and a special grant from IBM. The second author is also supported by NSF contract 9225124-CCR, AFOSR-ONR contract F49620-94-1-0199, and DARPA contract N00014-92-J-4033. The work of the third author was done while in MIT. The fourth author was partially supported by by NSF Presidential Young Investigator Award CCR-91-58478 and funds from Texas A&M University College of Engineering.

We would like to thank Nancy Lynch, Mark Tuttle and Shay Kutten for their crucial comments and suggestions.

References

1. Yehuda Afek, Baruch Awerbuch, and Eli Gafni. Applying static network protocols to dynamic networks. In *Proc. 28th IEEE Symp. on Foundations of Computer Science*, October 1987.
2. Anish Arora and Mohamed G. Gouda. Distributed reset. In *Proc. 10th Conf. on Foundations of Software Technology and Theoretical Computer Science*, pages 316–331. Spinger-Verlag (LNCS 472), 1990.
3. Baruch Awerbuch and Shimon Even. Reliable broadcast protocols in unreliable networks. *Networks*, 16(4):381–396, Winter 1986.
4. Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, and George Varghese. Time optimal self-stabilizing synchronization. In *Proc. 25th ACM Symp. on Theory of Computing*, May 1993.
5. Yehuda Afek, Shay Kutten, and Moti Yung. Memory-efficient self-stabilization on general networks. In *Proc. 4th Workshop on Distributed Algorithms*, pages 15–28, Italy, September 1990. Springer-Verlag (LNCS 486).

6. Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, October 1991.
7. Baruch Awerbuch and George Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, October 1991.
8. Baruch Awerbuch and Rafail Ostrovsky. Memory-efficient and self-stabilizing network RESET. In *Proc. 13th ACM Symp. on Principles of Distributed Computing*, August 1994.
9. J.E. Burns and J. Pachl. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11(2):330–344, 1989.
10. K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Comput. Syst.*, 3(1):63–75, February 1985.
11. Edsger W. Dijkstra. Self stabilization in spite of distributed control. *Comm. of the ACM*, 17:643–644, 1974.
12. Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. In *Proc. 10th ACM Symp. on Principles of Distributed Computing*, August 1990.
13. Shlomi Dolev, Amos Israeli, and Shlomo Moran. Resource bounds for self-stabilizing message driven protocols. In *Proc. 11th ACM Symp. on Principles of Distributed Computing*, Aug. 1991.
14. Steven G. Finn. Resynch procedures and a fail-safe network protocol. *IEEE Trans. on Commun.*, COM-27(6):840–845, June 1979.
15. L. Kleinrock and F. Kamoun. Hierarchical routing for large networks; performance evaluation and optimization. *Computer Networks*, 1:155–174, 1977.
16. Shmuel Katz and Kenneth Perry. Self-stabilizing extensions for message-passing systems. In *Proc. 10th ACM Symp. on Principles of Distributed Computing*, August 1990.
17. Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.
18. M. Merritt, F. Modugno, and M.R. Tuttle. Time constrained automata. In *CONCUR 91*, pages 408–423, 1991.
19. John McQuillan, Ira Richer, and Eric Rosen. The new routing algorithm for the arpanet. *IEEE Trans. on Commun.*, 28(5):711–719, May 1980.
20. Radia Perlman. Fault tolerant broadcast of routing information. *Computer Networks*, Dec. 1983.
21. Radia Perlman. An algorithm for distributed computation of a spanning tree in an extended LAN. In *Proceedings of the the 9th Data Communication Symposium*, pages 44–53, September 1985.
22. Radia Perlman, George Varghese, and Anthony Lauck. Reliable broadcast of information in a wide area network. *US Patent 5,085,428*, February 1992.
23. Thomas Rodeheffer and Michael Schroeder. Automatic reconfiguration in the Autonet. Proceedings of the 14th Symposium on Operating Systems Principles, November 1993.
24. Thomas Rodeheffer and Michael Schroeder. Personal communication.
25. M. Schroeder, A. Birrell, M. Burrows, H. Murray, R. Needham, T. Rodeheffer, E. Sattenthwaite, and C.Thacker. Autonet: a high-speed, self-configuring local area network using point-to-point links. Technical Report 59, Digital System Research Center, April 1990.
26. John M. Spinelli. Reliable communication. Ph.d. thesis, MIT, Lab. for Information and Decision Systems, December 1988.
27. A. Tanenbaum. *Computer Networks*. Prentice Hall, 2nd. edition, 1989.
28. George Varghese. Self-stabilization by local checking and correction. Ph.D. Thesis MIT/LCS/TR-583, Massachusetts Institute of Technology, 1992.
29. George Varghese. Self-stabilization by counter flushing. In *Proc. 13th ACM Symp. on Principles of Distributed Computing*, August 1994.