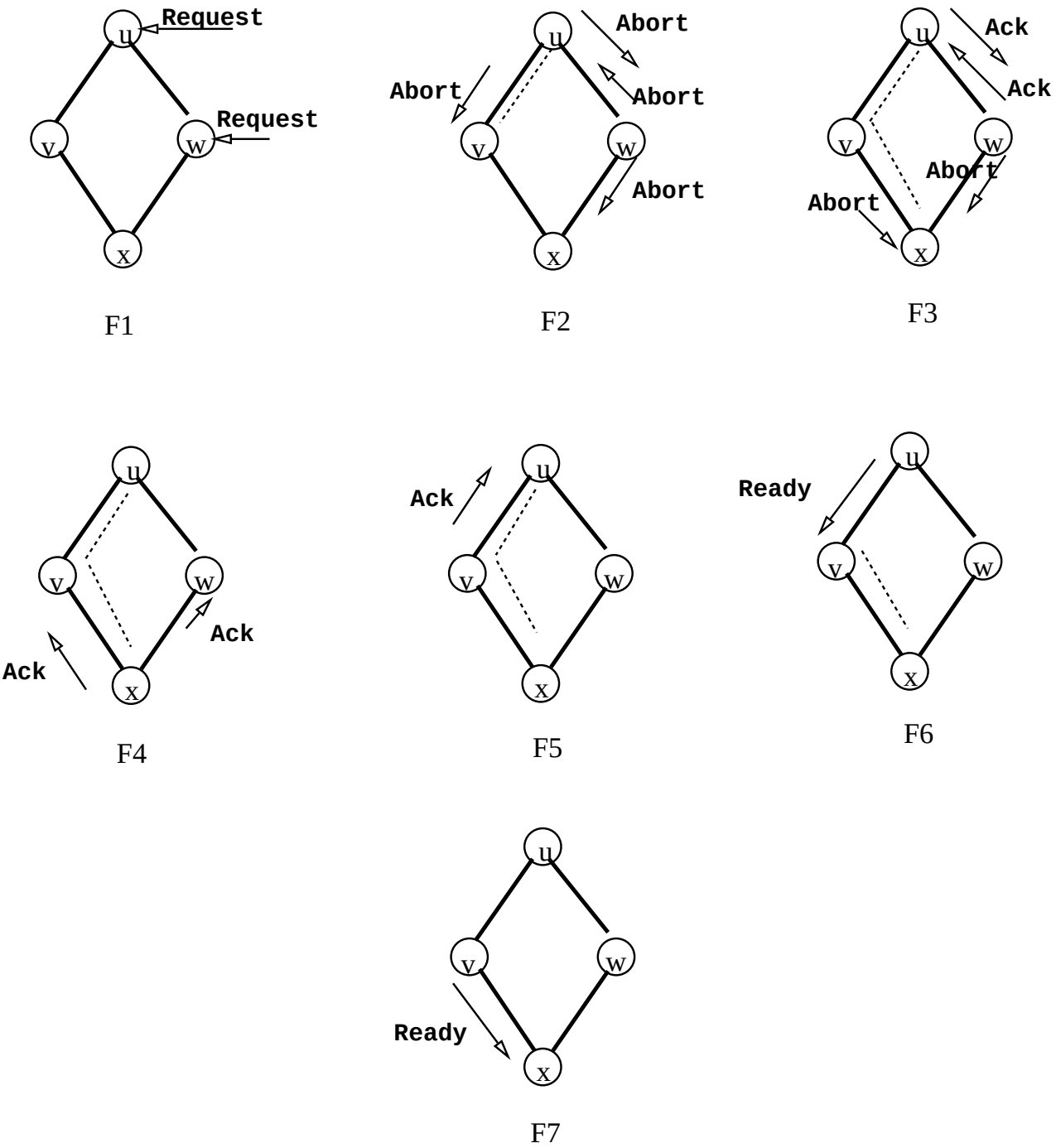

Figure 7.10: A sample execution for the reset protocol described in terms of seven snapshots.

uses the variables and macro specified in Figure 7.6. The piece of code that deals with implementing the main part of the reset protocol is described in Figure 7.7. The piece of code that deals with sending and receiving Σ-messages is described in Figure 7.8.

7.6 Reset Protocol is Locally Checkable and Correctable

Let G be the topology graph that models the topology of the network on which the reset protocol works.

Lemma 7.6.1 *$N = \{N_u, u \in G\}$ is a set of node automata for G with $P_{data} = \Sigma \cup \{(\text{ABORT}, *), \text{ACK}, \text{READY}\}$.*

Proof: Simple checking of the definitions given earlier. ∎

7.6.1 Overview of Predicates

Next, we prove that the protocol is locally checkable, by describing a set of predicates in Figure 7.14 and showing that these predicates are closed. The description of the predicates uses the shorthand notation shown in Figure 7.13. Please refer to both these figures during the following discussion.

Recall that $Q_{u,v}$ is the queue corresponding to the single packet that can be stored in channel $C_{u,v}$. For convenience, we define $xqueue_u[v]$ (i.e., the extended queue between u and v) as the queue formed by the concatenation of $Q_{u,v}$ and $queue_u[v]$. Thus in Figure 7.9, the extended queue between u and v is the queue formed by concatenating the outbound link queue and the link itself. Note that we do not include the inbound message buffer! If $Q_{u,v} \neq nil$, we assume that $Q_{u,v}$ is the head of $xqueue_u[v]$.

We informally describe the predicates. The first two predicates $\mathcal{A}$ and $\mathcal{B}$ deal with the ack flag $ack_u[v]$ at a node u. Intuitively, this bit is set if u expects an ack from v. $\mathcal{A}$ states that if u is expecting an ack from v then one of three possibilities must be true: either there is an abort packet in transit from u to v (Case 1 in Figure 7.11), OR v has received the abort packet and has chosen u as its parent (Case 2 in Figure 7.11),

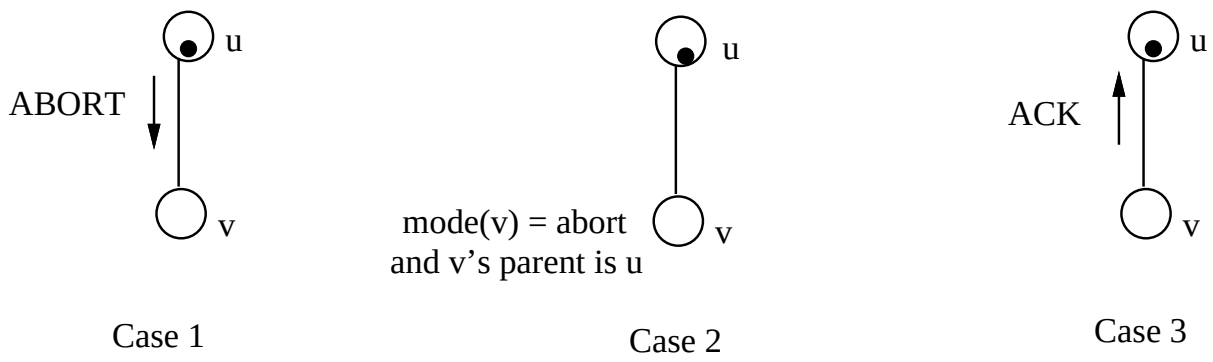

Figure 7.11: The Three Cases for the first and second predicates. The black dot before an edge indicates that a node is waiting for an Ack on that edge.

OR there is an ack packet in transit from v to u (Case 3 in Figure 7.11). Predicate $\mathcal{B}$ says that at most one of these three possibilities can be true at the same time. In some sense, $\mathcal{A}$ and $\mathcal{B}$ govern the first two phases of the reset protocol

Consider Figure 7.10. In that example execution, the first case is illustrated in F2, the second case in F3 and F4, and the third in F5.

The next two predicates $\mathcal{C}$ and $\mathcal{D}$ govern the second and third phases of the reset protocol. They deal with the $parent_u$ variable at a node u. Intuitively, $parent_u = v$ if v is the parent of u in the abort tree. $\mathcal{C}$ states that if v is the parent of u and the the mode of u is *Converge* then one of three possibilities must be true: either there is an ack packet in transit from u to v (Case 1 in Figure 7.12), OR v has received the ack packet and has cleared its ack bit for u but has not changed its mode to *Ready* (Case 2 in Figure 7.12), OR there is a ready packet in transit from v to u (Case 3 in Figure 7.12). Predicate $\mathcal{D}$ says that if v is u's parent, then at most one of these three possibilities can be true at the same time.

As an example, consider nodes x and v in Figure 7.10, where v is the parent of x. In that example execution, the first case is illustrated in F4, the second in F5 and F6, and the third in F7.

The next predicate $\mathcal{E}$ is crucial to the proof of termination of the protocol. It states that the distance of a child in the abort tree is one more than the distance of its parent. The only exception to this is if the parent has "abdicated" by sending a ready packet that is currently in transit to the child. Essentially, this predicate shows that abort trees are acyclic and have a maximum height of n, the number of nodes; the proof of

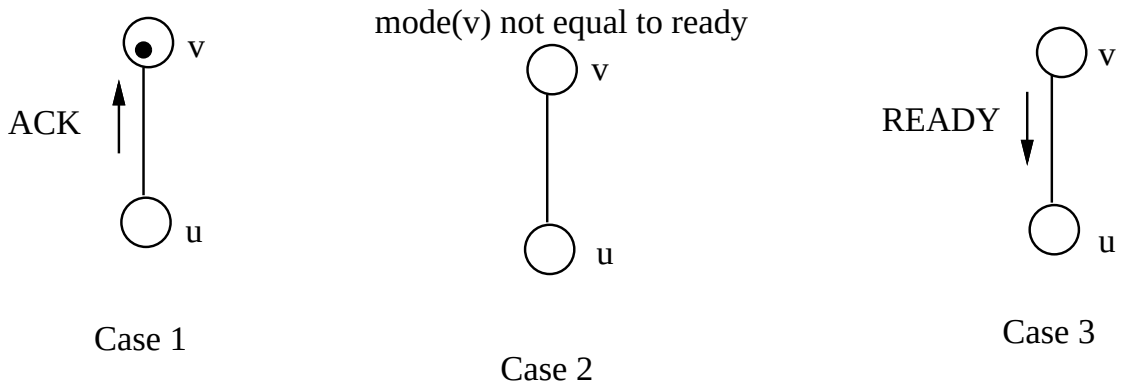

Figure 7.12: The Three Cases for the third and fourth predicates. Node v is the parent of Node u in all three cases.

termination will essentially consist of induction on the height of the abort trees. The predicate $\mathcal{F}$ is a supporting predicate required to prove that $\mathcal{E}$ is closed. It states that if u sends an abort to v then the distance in the abort packet is one more than u's current distance.

Predicate $\mathcal{G}$ is is again a supporting predicate required to prove that some of the other predicates are closed. Suppose there is a packet p, that is either a ready packet or a user message or a Σ-Ack, in transit from u to v. Then p must have been sent when u was in the *Ready* mode. Thus, it must be that either u is still in the *Ready* mode or that u has gone into *Abort* mode since the time p was sent. But in the latter case there must be an abort packet behind p in transit from u to v.

Predicate $\mathcal{H}$ governs the flow control scheme that ensures at most one message in transit from u to v. If $freem_u[v]$ is *true* then the $\text{FREEM}_{u,v}$ action is enabled and so there must be no message in transit from u to v and also no message acks (Σ-Ack's) in the reverse direction. On the other hand, if $freem_u[v]$ is *false* then either a message is in transit from u to v, or a message ack is in transit in the reverse direction, but not both.

Predicate $\mathcal{Q}$ is the reason why we can get away with an outbound queue size for links (e.g., the size of $queue_u[v]$) of 5. It states that the outbound queue (even after concatenation with the channel queue) can contain at most one packet of each type: abort, ack, and ready. Since $\mathcal{H}$ tells us we can have at most one Σ-message and at most one Σ-Ack, it means that a queue size of 5 is sufficient.

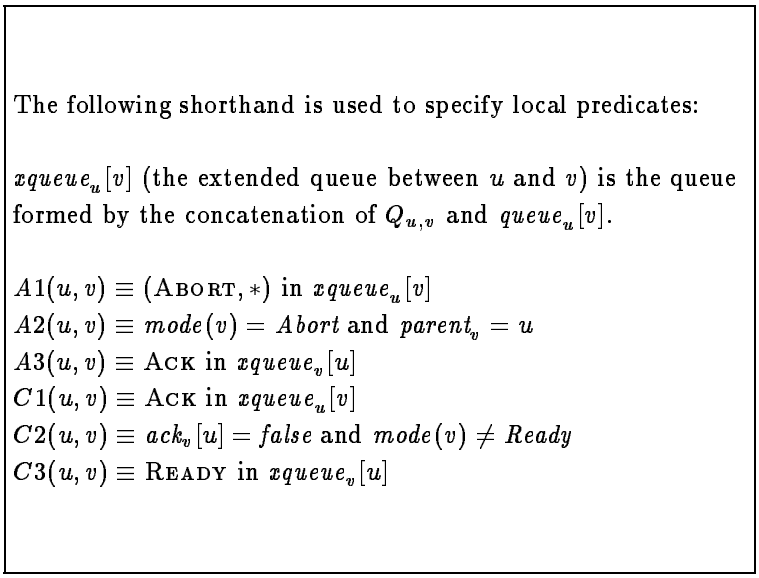

Figure 7.13: Shorthand Used to Define Local Predicates

We now show that the conjunction of these predicates is a closed predicate. The proofs, which we relegate to the appendix, consist basically of rigorous (and somewhat tedious) case-analysis.

7.6.2 Proving that the Local Predicates of the Reset Protocol are Closed

First, the the local predicates of Figure 7.14 are specific to a directed link (u,v). Recall that for local checkability we need exactly one local predicate for each edge. Hence:

Definition 7.6.2 *We let $G_{u,v}$ be the intersection of the local predicates in Figure 7.14 for any edge (u,v). For any edge (u,v), we let $L_{u,v}$ be the intersection of $G_{u,v}$ and $G_{v,u}$.*

Definition 7.6.3 *Let $L_{u,v}$ be the local predicate defined in Definition 7.6.2. Let $\mathcal{L}$ be the link predicate set containing $L_{u,v}$ for every (u,v) in G and let $L = Conj(\mathcal{L})$.*

$\mathcal{A}$: $ack_u[v] = true$ iff
one of $A1(u,v)$, $A2(u,v)$, or $A3(u,v)$ holds.

$\mathcal{B}$: At most one of $A1(u,v)$, $A2(u,v)$, or $A3(u,v)$ holds.

$\mathcal{C}$: $parent_u = v$ implies
$mode(u) = Converge$ iff
one of $C1(u,v)$, $C2(u,v)$, or $C3(u,v)$ holds.

$\mathcal{D}$: $parent_u = v$ implies
at most one of $C1(u,v)$, $C2(u,v)$, or $C3(u,v)$ holds.

$\mathcal{E}$: $parent_u = v$ implies
one of the following holds:
$dist_u = dist_v + 1$ and $mode(v) \neq Ready$ OR
$C3(u,v)$.

$\mathcal{F}$: If $xqueue_u[v]$ contains a (ABORT, d) packet then $d = dist_u + 1$.

$\mathcal{G}$: If p in $xqueue_u[v]$ and $p =$ READY or p is a Σ-message or $p = \Sigma$-Ack then
Either $mode(u) = Ready$
Or there is an ABORT in $xqueue_u[v]$ after p.

$\mathcal{H}$: Let $M_{u,v}$ denotes the concatenation of $xqueue_u[v]$, and $buffer_v[u]$. Then:
If $freem_u[v] = true$ then
There is no Σ-message in $M_{u,v}$ and no Σ-Ack in $xqueue_v[u]$
Else one of the following holds:
There is exactly one Σ-message in $M_{u,v}$ OR
There is exactly one Σ-Ack in $xqueue_v[u]$

$\mathcal{Q}$: $xqueue_u[v]$ contains at most one (ABORT, $*$), ACK, or READY packet.

Figure 7.14: Reset Protocol: Local Predicates for edge (u,v). Refer to the code given in Figure 7.13 for an explanation of the shorthand used.

The proof, which is in the appendix, consists of showing that each $L_{u,v}$ is a closed local predicate.

Lemma 7.6.4 *For a leader edge (u,v) and transition (s,a,s') of $\mathcal{R}$, if s satisfies $L_{u,v}$, then s' satisfies $L_{u,v}$.*

Proof: By lemmas, D.1.4, D.1.5, D.1.6, D.1.7, D.1.8, and D.1.3 in Section D.1 of the appendix. The predicates are closed because of the code in [AAG87] and the heuristic of removing unexpected packet transitions.

We quickly sketch what is involved in such a proof. Consider predicate $\mathcal{A}$ as sketched in Figure 7.11. We essentially consider all states that satisfy this local predicate and show that no transition can cause this predicate not to hold in the next state. For example, consider Figure 7.11. If u is not expecting an ack from v in a state, then the only transitions that can cause this to happen is if u gets a reset request or an ABORT packet while in *Ready* mode. But this causes u to send an an ABORT packet to v which leaves us in Case 1 of the predicate.

Rather than consider all possible transitions, we save some effort by first identifying the transitions that can affect key variables. Then we focus our attention on such transitions. This method is described in the appendix. ∎

The following theorem is immediate from the last lemma.

Theorem 7.6.5 *The reset automaton $\mathcal{R}$ can be locally checked for L using $\mathcal{L}$.*

7.6.3 Reset Protocol is Locally Correctable

Consider the function f defined in Figure 7.15. Let $<$ be the trivial partial order such for all u,v,w,x in G, $\{u,v\} \not< \{w,x\}$. (i.e., no pair of neighbors is less than any other pair of neighbors.) We claim that f is a local reset function for $\mathcal{R}$ with respect to $\mathcal{L}$ and partial order $<$.

Lemma 7.6.6 *The function f defined in Figure 7.15 is a local reset function for network automaton $\mathcal{R}$ with respect to link predicate set $\mathcal{L} = \{L_{u,v}\}$ and partial order $<$.*

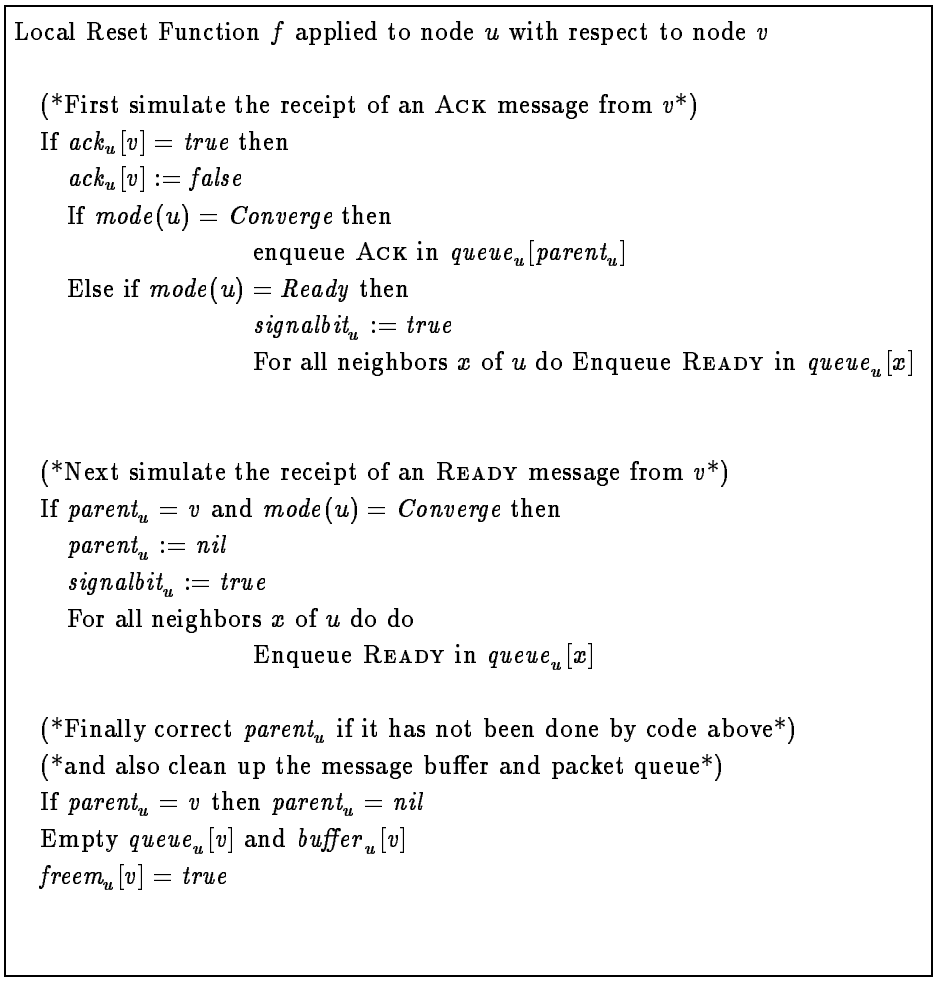

Figure 7.15: Reset Protocol. Local Reset Function

Proof: Consider any state s of $\mathcal{R}$ and any leader edge (u,v) of G: We show each of the properties required by a local correction function.

- **Correction:** In the state $f(s|u,v)$, it is easy to check that $parent_u \neq v$, $ack_u[v] = false$ and $queue_u[v]$ and $buffer_u[v]$ are empty. Also $freem_u[v] = true$. Similarly, in the state $(f(s|v,u)$, it is easy to check that $parent_v \neq u$, $ack_v[u] = false$ and $queue_v[u]$ and $buffer_v[u]$ are empty and $freem_v[u] = true$. Thus it follows that $(f(s|u,v), nil, nil, f(s|v,u)) \in L_{u,v}$.

- **Stability:** We need to show the following fact for any neighbor w of v:
  If $(s|u, s|(u,v), s|(v,u), s|v) \in L_{u,v}$ then $(s|u, s|(u,v), s|(v,u), f(s|v,w)) \in L_{u,v}$.

  But if we look at the code for $f(s|v,w)$ we see that $f(s|v,w)$ is the same state that would occur at v in an execution in which the reset protocol starts in state s and consisting of the following sequence of actions:

  - A $\textsc{Receive}_{w,v}(\textsc{Ack})$ action (see Figure 7.7) followed by
  - A $\textsc{Receive}_{w,v}(\textsc{Ready})$ action (see Figure 7.7) followed by
  - A hypothetical internal action that results in the execution of the last three lines of Figure 7.15 applied to node v with respect to node w.

  Now we know that the state at v resulting after the $\textsc{Receive}_{w,x}(\textsc{Ack})$ and $\textsc{Receive}_{w,x}(\textsc{Ready})$ events must satisfy the stability condition because we have already proved this in Lemma 7.6.5.

  Now consider the hypothetical internal action applied to v. Note that this internal action can only change $parent_v$ from w to nil. Also a careful look will show that this internal action will never change the value of $mode(v)$ since it is only applied after the simulated processing of a $\textsc{Ready}$ packet from w. Now the predicates described in $L_{u,v}$ only depend on the value of $mode(v)$, $ack_v[u]$ and the predicate $parent_v = u$. But none of these values are affected by the hypothetical internal action. Thus $L_{u,v}$ is unaffected by the hypothetical internal action.

  Thus the result of executing all three actions must satisfy the stability condition.

∎

Lemma 7.6.7 *Let f be the reset function described in Figure 7.15. Then $\mathcal{R}$ is* locally correctable *to L using link predicate set $\mathcal{L}$, reset function f, and partial order $<$.*

Proof: Follows from Lemma 7.6.6 and Lemma 7.6.5. ∎

Theorem 7.6.8 *$\mathcal{R}^+$ stabilizes to the behaviors of $\mathcal{R}(t_p)|L$ in time t_q, where t_q and t_p are constants.*

Proof: Follows directly from the Local Correction theorem, Theorem 5.4.3 and Lemma 7.6.7. Notice that $height(<) = 1$ since $<$ is the trivial partial order. ∎

7.7 The behaviors of a reset protocol after it stabilizes

We have already shown that $\mathcal{R}^+$ stabilizes to the behavior of $\mathcal{R}(t_p)|L$ in constant time. Recall that $\mathcal{R}(t_p)|L$ is identical to the automaton $\mathcal{R}$ except that the node and link delays are increased by a constant factor, and all link predicates hold.

We now show that $\mathcal{R}(t_p)|L$ solves the reset problem RP. However, because the set of behaviors specified by RP remains unchanged after scaling by constant factors, it suffices to show that $\mathcal{R}|L$ solves the reset problem RP. Thus in the remainder of this section and in the appendix, we show that every behavior β of $\mathcal{R}|L$ is in RP.

We relegate the proof of this theorem to the appendix. However, in this section, we intuitively explain why any behavior β of $\mathcal{R}|L$ is timely, consistent, and causal. The intuition provided in this section should help the reader understand the proof in the appendix.

In the proofs when we talk of a state s or an execution α, we mean a state or execution of $\mathcal{R}|L$. Recall that we defined a derived variable called $status(u)$ which is *on* if $mode(u) = Ready$ and $signalbit(u) = false$ and *off* otherwise. From the code, it is easy to see that u will only send and receive messages when $status(u) = on$.

The first important lemma is what we call the *Termination Lemma*. This states the following. Consider any node u. Assume that $mode(u) \neq Ready$ in some state s_i of any execution α. The lemma states that $mode(u)$ will change to *Ready* in $O(n)$ time after s_i.

7.7.1 Why the Termination Lemma Works

Let c be the worst-case time for a packet queued at node u to reach neighbor v. It's not hard to see that c is a constant because the outbound queue size at each node is at most 5 and because of the properties of a UDL. In the following, we will say that a node u is a root in some state s of the reset protocol if $mode(u) = Abort$ and $parent_u = nil$ in state s.

By assumption, $mode(u) \neq Ready$ in state s_i. Since $mode(u) \neq Ready$, u must either be a root or have a parent (say v) in state s_i. By using invariants $\mathcal{A}$ and $\mathcal{C}$ repeatedly, we can obtain a chain of nodes starting with u such that each node is the parent of the previous node. Also the chain must either end with a READY packet or must end with a root node r such that $mode(r) = Abort$ in s_i. This is shown in Figure 7.16. We also know from invariant $\mathcal{E}$ that the distance of each node in the chain is one more than that of its parent. Thus no node can occur more than once in this chain and hence this chain can consist of at most n nodes.

(The following is a more detailed argument that explains why each chain must end in a READY packet or a root r. If $mode(u) = Abort$, then either u is a root or u has a parent, say v. In the latter case by $\mathcal{A}$, $ack_v[u] = true$ and so $mode(v) = Abort$. If $mode(u) = Converge$ then u has a parent, say v. By $\mathcal{C}$, either there is an ACK in transit from u to v (in which case by $\mathcal{A}$, $mode(v) = Abort$), or $mode(v) \neq Ready$, or there is a READY packet in transit from v to u. Thus either u is a root, or u has a parent v such that $mode(v) \neq Ready$ or there is a READY packet in transit from v to u. We now repeat this argument until we either arrive at a root node or a READY packet.)

Consider the case where the chain ends with a READY packet. In this case, it is not hard to see that within $O(n)$ time after s_i, a READY packet will reach u which will cause u to go into *Ready* mode.

So consider the latter case where the chain ends with a root r. Suppose we can show that in $O(n)$ time after s_i, there is an action a_j in which r changes its mode to *Ready* and sends a READY packet to all its children. But if we can show this we are done in $O(n)$ time after a_j using the arguments in the previous paragraph. So all we have to do is to show that in $O(n)$ time after s_i, r will change its mode to *Ready*

If the mode of r is *Abort*, then by definition r must have some neighbor v that it expects an ack from (i.e., $ack_r[v] = true$). Now by invariant $\mathcal{A}$, this means that (see

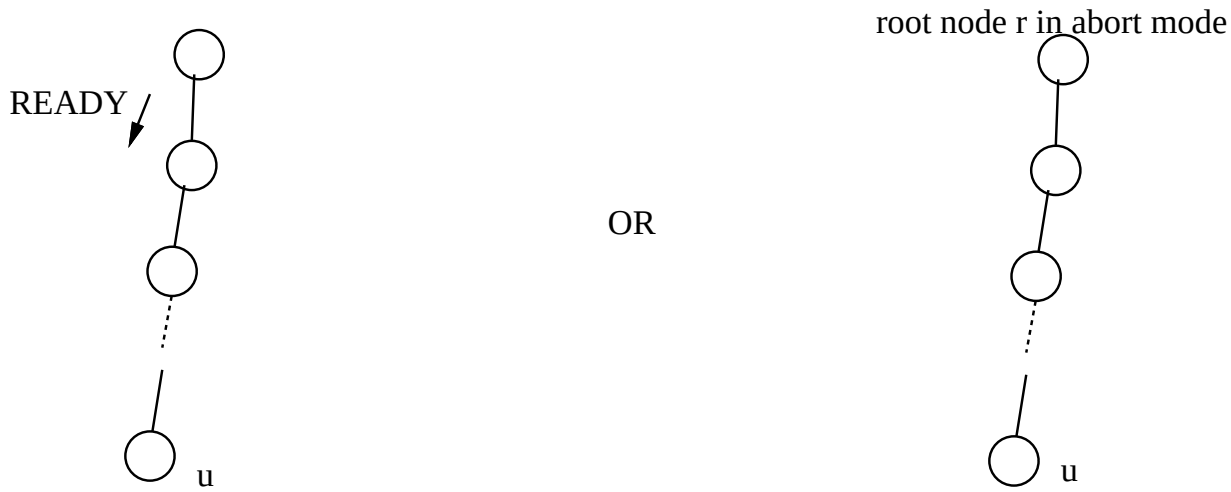

Figure 7.16: The two cases that can occur if u's mode is not *Ready*. Each node in a chain is the parent of the node below it.

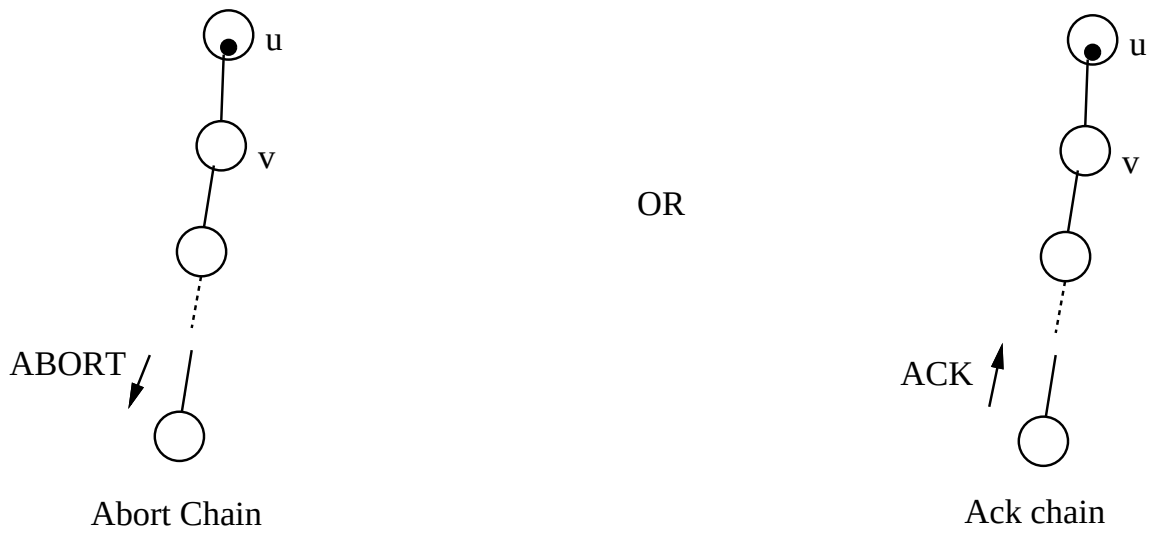

Figure 7.17: The two cases that can occur if node u is expecting an ack on the link to v.

Figure 7.11) either there is an abort in transit from r to v, or an ack in transit from v to r, or v is also in *Abort* mode and r is the parent of v. But if v is in in *Abort* mode, we can continue this argument inductively to produce a chain of nodes starting with r such that each node is the parent of the next node. We also know from invariant $\mathcal{E}$ that this chain can consist of at most n nodes. At the end of this chain (see Figure 7.17, there must be either an abort or an ack. We call a chain that ends with an abort packet an *abort chain* and we call a chain that ends with an ack packet an *ack chain*

Recall that c is the constant that reflects the worst-case time for a packet queued at node to reach a neighbor. Now, observe that in c units of time any abort chain must either increase in size by 1 or be converted into an ack chain. But since the size of an abort chain cannot increase beyond n (the number of nodes), within $O(n)$ time any abort chain must have converted into an ack chain. Similarly in c units of time, any ack chain must decrease in size by 1. Thus in $O(n)$ time any ack chain will disappear.

The upshot is that within $O(n)$ time, $ack_r[v]$ will become *false*. Since this happens for any child v of r, within $O(n)$ time after s_i, r will change its mode from *Abort* to *Ready* and we are done.

A formal proof can be made based on these arguments. The appendix contains a formal statement of the lemma (Lemma D.2.1) but omits a detailed formal proof.

7.7.2 Why behaviors of the reset automaton are timely, consistent, and causal

Recall that we defined a derived variable called $status(u)$ which is *on* if $mode(u) =$ *Ready* and $signalbit(u) =$ *false* and *off* otherwise. From the code, it is easy to see that u will only send and receive messages when $status(u) = on$.

The first important tool is what we call the *Signal Lemma.* Consider any execution α of $\mathcal{R}|L$ and any node u and any state s_i in α. The signal lemma basically says that if $status(u) =$ *off* in state s_i, then a SIGNAL_u event occurs within $O(n)$ time after s_i in α. This follows because if $signalbit_u =$ *true* in s_i, then a SIGNAL_u event must occur within constant time of s_i by the timing conditions. On the other hand, if $mode(u) \neq$ *Ready* in s_i, then the Termination Lemma tells us that in $O(n)$ time after s_i, we reach a state s_j in which $mode(u) =$ *Ready*. If s_j is the first such state after s_i, then by the code, we

know that u cannot change its mode to *Ready* without also setting $signalbit_u = true$. Then, as before, a SIGNAL$_u$ event must occur within constant time of s_j in α.

Now we consider an execution α of $\mathcal{R}|L$ and sketch why the behavior corresponding to α satisfies the timeliness, consistency, and causality properties.

Proving the Timeliness Property

Consider first the timeliness property:

1. **Normal Receipt of Messages**: We need to show that there is some constant c such that every every receive event that occurs at time greater than $\beta.start + c \cdot n$ in any execution α is normal. Also if a_j is any normal receive event and a_i is the send corresponding to a_j, then a_j occurs within $O(n)$ time after a_i.

   This follows because that any message m in transit from say v to u (i.e, stored either in the queue at v, the link from v to u, or the buffer at u) cannot remain in transit for more than $O(n)$ time. If a message m is stored either in the queue at v or in the link from v to u, then (by the properties of the link and the fact that the queue holds at most 5 packets), the message m will be stored in the buffer at u in constant time. Next, we argue that in $O(n)$ time after a state s_i in which a a message m is in the buffer at u, m will either (see Figure 7.18) be delivered or be "flushed" by an ABORT packet sent from v.

   If $status(v)$ remains *on* for a constant amount of time after s_i, then the message m will be delivered. Thus the only other possibility is that message m remains in the buffer because $status(v)$ is *off* in constant time after s_i. But in that case by the Signal Lemma, there will be a SIGNAL$_u$ event in $O(n)$ time after s_i. With a little work (see appendix) we can show that such SIGNAL$_u$ events cannot keep occurring at node u for a linear amount of time without causing v to send an ABORT packet to u; this will flush the buffer at u.

2. **Periodic Free Events**: Consider any t-suffix γ of execution α. Then in γ either a FREEM$_{u,v}$ occurs within constant time or a signal event occurs at u within $O(n)$ time or a signal event occurs at v within $O(n)$ time.

   This follows because of the following observation. Let c be a sufficiently large constant time. Suppose either $status(u)$ or $status(v)$ is *off* in c time after the

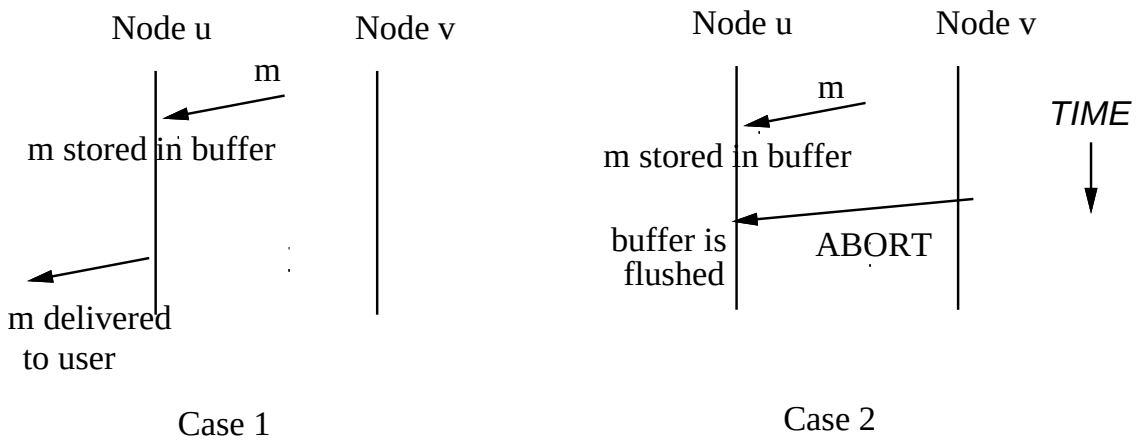

Figure 7.18: Two Cases for how a message is removed from a link

start of γ. Then we are done by the Signal Lemma. But if this is not true, and c is sufficiently large, then any message m in transit from u to v will be delivered in constant time; also any Σ-Ack message in transit from v to u will be delivered in constant time; this will result in $\mathit{freem}_u[v] = \mathit{true}$ in constant time. In constant time after $\mathit{freem}_u[v]$ is *true* (and assuming that c is large enough so that $\mathit{status}(u)$ remains *on* in this interval), a $\text{FREEM}_{u,v}$ event will occur.

3. **Timely Message Delivery**: Suppose a_j is a safe $\text{SENDM}_{u,v}(m)$ event in γ. Then either a $\text{RECEIVEM}_{u,v}(m)$ occurs within constant time after a_j or a signal event occurs at u within $O(n)$ time or or a signal event occurs at v within $O(n)$ time.

   This follows because of a similar observation to the one used to show periodic free events. Let c be a sufficiently large constant time. Suppose either $\mathit{status}(u)$ or $\mathit{status}(v)$ is *off* in c time after a_j. Then we are done by the Signal Lemma. If not, by arguments similar to the ones above, we show that message m will be accepted and stored at u and then sent to v where it will be delivered in constant time. We assume that c is large enough so that $\mathit{status}(u)$ and $\mathit{status}(v)$ remain *on* in this interval.

4. **Signals at a Node induce Signals at Neighbors**: There is some constant c such that for every SIGNAL_u event a_j that occurs at time greater than $\beta.\mathit{start}+c\cdot n$ there is a SIGNAL_v event that occurs in linear time before or after a_j.

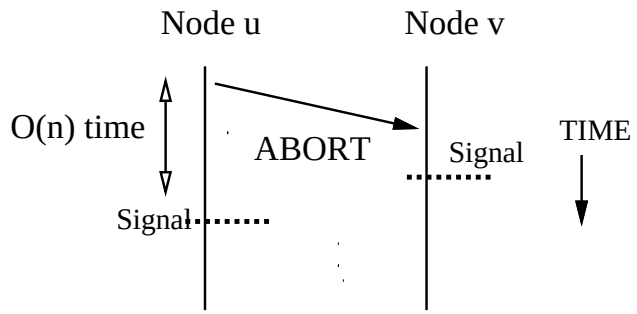

Figure 7.19: A signal event at u that occurs sufficiently "late" must be preceded in linear time by the sending of an ABORT packet to v. This in turn causes a signal event to occur within linear time at v.

First, within linear time of the start of the execution α corresponding to β, there must be some state s_h in which $mode = Ready$ (by the termination lemma). In constant time after s_h, there must be some state s_i in which $signalbit_u = false$. This follows because $signalbit_u$ cannot remain *true* for constant time without a SIGNAL_u action occurring, which sets $signalbit_u = false$. Thus any SIGNAL_u action a_j that occurs after state s_i must have been "caused" by u receiving a reset request or an ABORT packet while in *Ready* mode. Let this action be $a_{j'}$; by the termination lemma, $a_{j'}$ occurs in linear time before a_j. But as part of action $a_{j'}$, the code will also send an ABORT packet to neighbor v as sketched in Figure 7.19. Thus in constant time after $a_{j'}$, this ABORT packet will arrive at v and result in a state in which $status(v) = off$. Thus, by the Signal Lemma, in $O(n)$ time after $a_{j'}$, a SIGNAL_v event occurs. Since $a_{j'}$ occurs within linear time before a_j, the SIGNAL_v event occurs within linear time before or after a_j.

Proving the Consistency Property

Consider now the consistency property. As in the Simple Reset Protocol, the consistency conditions follow from the sending of ABORT packets between signal intervals. *We define a signal interval S_u at u and a signal interval S_v at v to be mates iff a normal message sent in S_u is received in S_v or vice versa.* We can show that the mating relation is well-defined and symmetrical because of the following two properties.

The first property which we call *send consistency* states that messages sent in a signal interval at v can be received in at most one signal interval at u; conversely

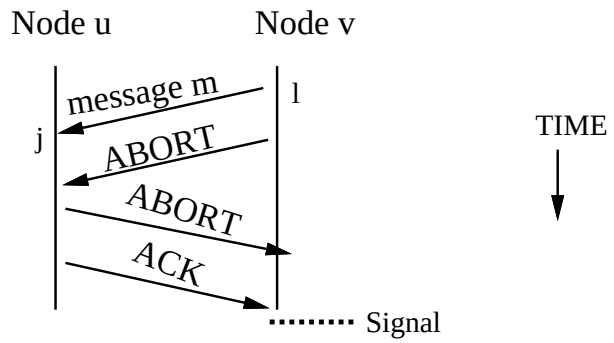

Figure 7.20: What happens between the sending of a message and a later Signal event.

messages received in a signal interval at u could have been sent in at most one signal interval at v. This will help establish that each signal interval at u can have at most one mated signal interval at v.

Send Consistency: Let a_j and a_k be any two normal receive events at u from v in β. Let a_l and a_m be the send events corresponding to a_j and a_k respectively. Then there is a SIGNAL$_v$ event between a_l and a_m iff there is a SIGNAL$_u$ event between a_j and a_k.

Consider Figure 7.20. This figure shows that if there is a SIGNAL$_v$ event after a_l then there must be an ABORT packet sent in between from v to u. On receipt of this packet, all packets in the buffer at u will be flushed and *status*(u) must become *off*. Next, u will not deliver any messages until it has performed a SIGNAL$_u$ and *status*(u) is *on* again. But any messages sent by v after the SIGNAL$_v$ event will arrive (by the FIFO property of the queue and link) after the ABORT packet arrives at u. This guarantees that such messages will not be delivered until u performs its SIGNAL$_u$ event. Similar (but more complicated) arguments can be used to show the other side of this claim.

Next, we show a second property which states (in essence) that a signal interval at u cannot send messages to and receive messages from different signal intervals at v. This will help show that the mating relation is symmetric.

Send-Receive Consistency Let a_j be a normal receive event at u from v and let a_m be a normal receive event at v from u. Let a_l and a_k be the send events corresponding to a_j and a_m respectively. Then there is a SIGNAL$_v$ event between a_l and a_m iff there is a SIGNAL$_u$ event between a_j and a_k.

Consider Figure 7.20. This figure shows that if there is a SIGNAL$_v$ event after a_l then there must be an ABORT packet sent from v to u before v performs its SIGNAL$_v$ event. When this packet arrives at v there are two possibilities. Suppose the mode of u is not equal to *Ready* at this point. If u has sent any message m to u in the past, then u must have sent an ABORT packet to v after message m. On the other hand, if the mode of u is *Ready* when the ABORT packet from v arrives, then u will immediately send an ABORT packet to u. In either case, the ABORT packet from u to v will arrive at v before the ACK (see Figure 7.20 which shows the second case) from u to v. But when the ABORT packet arrives at v, it will flush out any messages in transit from u to v; it is only later that the SIGNAL$_v$ event can be performed.

This effectively means that any messages sent by u after the ABORT is received at u can only be delivered at v after the SIGNAL$_v$ event. Similar arguments can be used to complete the proof of this claim.

Now we show the third property of a consistent behavior listed in Definition 7.3.5.

Successful Sending of Messages: Between any safe SENDM$_{v,u}(m)$ event and a later FREEM$_{v,u}$ event, there is either a RECEIVEM$_{v,u}(m)$ event or a SIGNAL$_v$ event.

Call the first SENDM$_{v,u}(m)$ event a_i. If in state s_i, $status(v) = off$, message m will be dropped; but in this case by the Signal Lemma a SIGNAL$_v$ will occur after s_i. But in the period between s_i and the SIGNAL$_v$ event, since $status(v) = off$, the code ensures that no FREEM$_{v,u}$ event can occur. So suppose m is placed in $queue_v[u]$ in s_i. Now we return to Figure 7.18. We know that either (case 1) m is delivered to u or m is destroyed by a later ABORT packet. In either case, we know from predicate $\mathcal{H}$, that $freem_v[u]$ will be *false* until m is no longer in transit and so no FREEM$_{v,u}$ can occur in the interim. In Case 2, after the ABORT is sent by v, $status(v)$ will remain *off* until a SIGNAL$_v$ occurs. Thus in this period as well no FREEM$_{v,u}$ event can occur.

Next, we show the fourth property of a consistent behavior listed in Definition 7.3.5.

Mating of Final Signal Intervals: Let a_j be a normal receive event at u from v in α and a_l be the send corresponding to a_j. Then there is no SIGNAL$_u$ event after a_j iff there there is no SIGNAL$_v$ event after a_l.

This follows from Figure 7.20. If there is a SIGNAL$_v$ after a_l there must be an ABORT packet that will be received at u after a_j. This will result in $status(u)$ becoming

off after a_j. The Signal Lemma now tells us that there will be a SIGNAL$_u$ event after a_j. The reverse argument is similar.

Finally, the fifth property of a consistent behavior (i.e., the mating relation preserves temporal ordering) follows in essence from the fact that the UDLs are FIFO links and the fact that ABORT packets sent between signal intervals flush the links and buffers of previously sent messages.

Proving the Causality Property

To show that the behavior corresponding to execution α is causal, we prove the two properties of a causal behavior:

1. There is some constant c such that every signal event a_k in α that occurs at time greater than $\beta.start + c \cdot n$ is preceded by a request event a_j that occurs in linear time before a_k.

   It is sufficient to show that there is some constant c such that the following is true: if we consider any interval $[s_i, s_k]$ in which no reset request occurs and such that $s_k.time - s_i.time \leq cn$, then a_k cannot be a signal event. The required property follows from this because it implies that if a_k is a signal event, then a reset request must have occurred in linear time before a_k.

   If we choose c large enough, then we can break the interval $[s_i, s_k]$ into four subintervals $[s_i, s_{i'}]$, $[s_{i'}, s_j]$, $[s_j, s'_j]$ and $[s_{j'}, s_k]$ such that:

   - Every node u has had $mode(u) = \mathit{Ready}$ in the interval $[s_i, s_{i'}]$. (More precisely, for every node u there is some state s_l in the interval $[s_i, s_{i'}]$ such that $s_l.mode(u) = \mathit{Ready}$.)
   - Every node u has had $mode(u) = \mathit{Ready}$ in the interval $[s_{i'}, s_j]$.
   - Every node u has had $mode(u) = \mathit{Ready}$ in the interval $[s_j, s_{j'}]$.
   - For any u, any signal event enabled in $s_{j'}$ is guaranteed to occur in the interval $[s_{j'}, s_{k-1}]$.

   Intuitively, we choose the first three subintervals to be long enough, so that every node will have had a chance to go to *Ready* once in each subinterval. The termination lemma tells us that this can be done using subintervals whose duration

is $O(n)$. Finally, the fourth subinterval needs to be sufficiently long so that any signal events enabled at the start will occur before the end of the subinterval. By the timing conditions (see code), this can be done using a subinterval whose duration is a constant.

First, note that a node u can become a root (i.e. $mode_u = Abort$ and $parent_u = nil$ only through two events: first, by a reset request at node u; and second by receiving an (ABORT, n') packet with the distance variable at the maximum value. We call the second transition, a *spurious reset request*. We first claim that a spurious reset cannot occur in any state in which the following *bounded distance* predicate holds: for all nodes u, if $parent_u = nil$ then $dist_u = 0$. This follows because if there is an (ABORT, n') in transit from say v to u we can apply $\mathcal{A}$ and $\mathcal{B}$ repeatedly to obtain a chain of nodes ending with a root node r. If $dist_r = 0$ then by $\mathcal{E}$ and $\mathcal{F}$ we can arrive at a contradiction in that any ABORT packet must carry a distance less than $n - 1$.

Next, we claim that all spurious reset requests disappear after the first subinterval. This is because the bounded distance predicate must hold after the first interval. If any node u becomes a root after the first interval, since it was *Ready* at some time during the first interval, it must have become a root through either a reset request or a spurious reset request; but either of these transitions will also set $dist_u = 0$.

Next, we claim that for every node u, at the end of the third subinterval that $mode(u) = Ready$. This follows because of the following intuitive argument.

Recall that a root of an abort tree is a node u with $parent_u = nil$ and $mode(u) = Abort$. Second, we claim that any roots of abort trees that existed at the start of the second subinterval can no longer exist by the end of the second subinterval. This is because (by choice of subinterval) any such root node must have changed its mode to *Ready* by the end of the second subinterval. But since no reset requests occur in the entire interval, and no spurious requests occur after the first subinterval, no new roots can be created after the first subinterval. Thus there are no roots by the end of the second subinterval.

Now consider any node u. But if there are no roots at the end of the second subinterval, u cannot enter *Abort* mode in the third subinterval. This is because in any state s in which $s.mode(u) =$ ABORT there must be a root node. (This in

turn follows by applying predicate $\mathcal{B}$ repeatedly.) But we know that u changed its mode to *Ready* somewehere in the third subinterval. Also, u cannot change its mode from *Ready* to *Converge* or *Abort* without first going into *Abort* mode. Thus, u must be in *Ready* mode by the end of the third subinterval.

Finally, if no real or spurious reset requests occur in the fourth subinterval and every node is ready by the end of the third subinterval, then no signal event can occur at the end of the fourth subinterval. This is because we choose the fourth subinterval such that any signal actions enabled at the start of this subinterval would occur *before* the end of this subinterval.

2. A SIGNAL_u event occurs within $c \cdot n$ time after any REQUEST_u event in α.

   This follows easily because immediately after the REQUEST_u event, $status(u) = off$. The claim now follows from the Signal Lemma.

Thus, as we show more formally in Theorem D.2.28, every behavior of $\mathcal{R}|L$ is in RP. This can be used to show:

Theorem 7.7.1 *$\mathcal{R}^+$ stabilizes to the behaviors in problem RP in constant time.*

Proof: Follows directly from Theorem 7.6.8 and Theorem D.2.28. ∎

7.8 Local Correctability and Dynamic Network Protocols

A *dynamic network* is a network in which faults are limited to link failures: links can crash and recover in arbitrary fashion. A dynamic protocol is a protocol that works correctly in dynamic networks. If we assume that the topology (and the list of neighbors of each node) eventually stabilizes, then any stabilizing protocol P will eventually work correctly in a dynamic network. This is because any finite sequence of link failures can only leave P in some arbitrary state.

A large number of protocols have been designed for dynamic networks. Dynamic protocols are useful because the most common faults in real networks are link and node crashes. Many of these existing protocols have not explicitly been designed to be

stabilizing. However, in this section, we conjecture that a number of dynamic protocols can be made locally correctable.

We start with the reset protocol described in [AAG87] on which the reset protocol described in this chapter is based. This protocol was originally designed for dynamic networks. Thus besides the actions described in this chapter, the protocol in [AAG87] had actions for link failure and recovery. Thus for every node u and every neighbor v of u, the protocol had an input action LINK_UP$_{u,v}$ (corresponding to the link to v coming up at node u) and an input action LINK_DOWN$_{u,v}$ (corresponding to the link to v coming down at node u).

Next, consider the reset function (Figure 7.15) used in this chapter for the reset protocol. The reset function applied to a node v with respect to a neighbor u is exactly:

- The code performed in [AAG87] for a LINK_DOWN$_{u,v}$ event, immediately followed by
- The code performed in [AAG87] for a LINK_UP$_{u,v}$ event.

In other words, we can obtain a local reset function by simulating a link failure immediately followed by a link recovery. Is this a coincidence?

We present a rough (but incomplete) argument as to why this might work. First, consider the stability condition for local correctability. Consider any neighbor w of u. Suppose that the (u, w) subsystem is in a good state – i.e., in a state that belongs to $L_{u,w}$. Now when the link to v comes up or goes down, the original protocol had to preserve the stability of $L_{u,w}$. Thus the code for the LINK_UP$_{u,v}$ and LINK_DOWN$_{u,v}$ events preserves the stability of $L_{u.w}$.

Now consider the correction condition. Clearly it is possible in the dynamic protocol to have the link at u go down simultaneously at both ends and then come up simultaneously at both ends. When the link comes up it should come up with no messages in the links and with both u and v in states that satisfy $L_{u,v}$.

Both arguments given above are incomplete. For example, consider our "proof" of the stability condition. We claimed that if the (u, w) subsystem was in a good state, the LINK_DOWN$_{u,v}$ events would preserve the stability of $L_{u.w}$. To make a more careful argument we have to add the following local extensibility condition. For every

state $a \in L_{u,w}$, there is some valid global state s of the network, in which the (u,w) subsystem is in state a and the link from node u to node v is considered to be up at node u. Since s is a valid state of the original protocol, and the LINK_DOWN$_{u,v}$ can occur in state s, the result of taking the LINK_DOWN$_{u,v}$ action must result in a new valid state s'. But since s' is a valid state of the original protocol it must satisfy all local predicates, including $L_{u,w}$.

It is possible to formalize the intuitive arguments by adding similar local extensibility conditions, and showing that the protocol in [AAG87] satisfies these conditions. We will not do so here.

7.9 Summary

The three main ideas in this chapter are as follows:

First, we have given a new definition of the correctness of a reset protocol in terms of its external behaviors. We have seen that the resulting definition is, in some sense, a generalization of the synchronization guarantees offered by a Data Link protocol. However, while a Data Link protocol synchronizes two nodes, a reset protocol synchronizes multiple nodes. Notice that our definition specifies the behavior of the reset protocol when reset requests are continuously being made and not just in the event that there is a last reset request. The definition we used in our original paper ([APV91b]) only specified the behaviors in the event that there is a final reset request. However, in trying to apply the reset protocol (for instance, in Chapter 8) we soon found a need for the present specification.

Second. we have applied the Local Correction theorem to stabilize a version of the reset protocol described in [AAG87]. We had to make some subtle changes to the original protocol to make it locally checkable and correctable.

Third, we have conjectured that many locally checkable protocols that work in dynamic networks can be made locally correctable. To obtain a reset function, we concatenate the code that the original protocol used for a link down event with the code used for a link up event. As another example, Spinelli [Spi88a] describes a *virtual circuit* protocol that works in dynamic networks. This protocol appears to be locally checkable and it appears that we can use the link up and link down code to create a reset function. Interestingly, Spinelli [Spi88a] makes his protocol stabilizing by sending

a message periodically from the source of the virtual circuit to the destination of the virtual circuit and back. He also uses a timer whose value is proportional to the maximum end-to-end delay in the network. If, as we conjecture, local checking and correction is applicable to Spinelli's protocol, the resulting protocol will be simpler and faster than the one presented in [Spi88a].

Chapter 8

Global Correction Theorem

The previous three chapters have been concerned with local checking and *local* correction. This chapter marks an important transition as we move to local checking and *global* correction. For the next two chapters, we will study the use of the stabilizing reset protocol of Chapter 7 for global correction. In this chapter, we will prove a Global Correction Theorem. This theorem states that any *locally checkable* protocol can be stabilized in time proportional to the number of network nodes using global correction. Thus global correction removes the need for the original protocol to be locally correctable but pays a price in terms of stabilization time. In the next chapter, we will apply global correction to a simple synchronizer protocol [Awe85].

The focus of this chapter is the Global Correction Theorem. The Global Correction theorem should be contrasted with the Local Correction Theorem (Theorem 5.4.3) of Chapter 5. The extra price paid for Global Correction is a stabilization time that is proportional to the *number of network nodes* instead of the *height of some underlying partial order.*

A second important idea contained in this chapter is the concept of one-way checkability. One-way checking is a special case of local checking that results in simpler and more efficient stabilizing protocols. We illustrate the two ideas in this chapter by using a simple spanning tree protocol. The spanning tree protocol is locally checkable and hence can be stabilized using the Global Correction Theorem. However, because the Spanning Tree protocol is also one-way checkable, we can create a simpler (and more efficient) stabilizing spanning tree protocol.

This chapter is organized as follows. In the first two sections, we state and prove the *Global Correction Theorem*. In Section 8.3 we describe a locally checkable protocol to compute a spanning tree. After all local predicates hold, this protocol computes a spanning tree in time proportional to the diameter of the network. Because it does not appear that the spanning tree protocol is locally correctable, the methods of Chapter 5 do not seem applicable. However, the Global Correction Theorem applies to this spanning tree protocol.

Next, in Section 8.4 we explain the concept of one-way checkability, and show that the spanning tree protocol is one-way checkable. We combine this observation with with the Global Correction Theorem to yield a simple stabilizing spanning tree protocol. Finally, in Section 8.6 we quickly sketch how Global Correction can also be applied to the design of a stabilizing protocol for topology maintenance in dynamic networks.

8.1 Statement of Global Correction Theorem

In this section and the next, we show that *any* locally checkable protocol $\mathcal{N}$ can be efficiently stabilized using a reset protocol. The basic idea in the proof of the Global Correction theorem is extremely similar to the transformation used in Chapter 5 (see Figure 5.6 and Figure 5.7) for the proof of the Local Correction theorem. As before, we augment the original automaton with actions to perform local snapshots on every (u, v) subsystem. However, there are two crucial differences in the proof of Global Correction:

- First, we replace all links (UDL's) in the original protocol by a stabilizing reset protocol for graph G as described in Chapter 7. In other words, each node u now communicates with its neighbors using the interfaces provided by the reset subsystem. Now, the reset protocol offers an interface that is identical to a UDL except that packets are replaced by messages. Thus we also have to replace the actions that send and receive packets in the node automata with actions to send and receive messages.

- Second, when a violation is detected in the (u, v) subsystem, a global reset request is made using the REQUEST_u action offered by the reset protocol interface. Then

when a SIGNAL_u action occurs, node u will essentially do a "local restart" by initializing its state. This will guarantee that no more violations of the (u, v) subsystem will be detected after both u and v have initialized their state.

Suppose we are given a locally checkable automaton $\mathcal{N}$. Then in the transformation outlined above, we need to replace the events to send and receive packets by events to send and receive messages. Thus we cannot guarantee that the transformed automaton stabilizes to the behaviors of $\mathcal{N}|L$ but to a renamed version of $\mathcal{N}|L$ in which action names have been suitably renamed. This motivates:

Definition 8.1.1 *Consider a network automaton $\mathcal{N}$. We define $R(\mathcal{N})$, the message-renamed version of $\mathcal{N}$, to be the automaton that is identical to $\mathcal{N}$ except that for all u, v:*

- *Every* $\text{SEND}_{u,v}(*)$ *(i.e., packet send) event in $\mathcal{N}$ is renamed as a* $\text{SENDM}_{u,v}(*)$ *(i.e., message send) event.*
- *Every* $\text{RECEIVE}_{u,v}(*)$ *(i.e., packet receive) event in $\mathcal{N}$ is renamed as a* $\text{RECEIVEM}_{u,v}(*)$ *(i.e., message receive) event.*
- *Every* $\text{FREE}_{u,v}$ *(i.e., packet free notification) event in $\mathcal{N}$ is renamed as a* $\text{FREEM}_{u,v}$ *(i.e., message free notification) event.*

The message-renamed version of a set of behaviors of a network automaton is defined similarly.

Note that renaming does not really affect the services offered by an automaton because the users of the automaton can change their interface to accommodate the renaming.

With this definition, we can state the main theorem of this chapter. It states that any locally checkable network automaton $\mathcal{N}$ can be transformed into another automaton $\mathcal{N}^+$ such that $\mathcal{N}^+$ stabilizes to a version of $\mathcal{N}$ in which i) All local predicates hold ii) The node and link delays are increased by a constant factor iii) The packet events are renamed as message events.

Theorem 8.1.2 Global Correction: *Consider any network automaton* $\mathcal{N} = Net(G, N)$ *that is* locally checkable *for some predicate* L *using link predicate set* $\mathcal{L}$. *Then there exists some* $\mathcal{N}^+$ *that is a message-renamed UIOA for graph* G *and a constant* c *such that* $\mathcal{N}^+$ *stabilizes to the behaviors of* $R(\mathcal{N}(c)|L)$ *in* $O(n)$ *time.*

8.2 Proof of the Global Correction Theorem

We present the main ideas in the proof of the Global Correction Theorem in the following subsections. We first sketch the construction of the augmented automaton $\mathcal{N}^+$. Then, we show that after linear time of the start of any execution of $\mathcal{N}^+$ all reset requests disappear. We use this to conclude that $\mathcal{N}^+$ stabilizes to a message-renamed version of $\mathcal{N}$ in linear time. Finally, we make some observations about modularity in the proof.

8.2.1 Construction of $\mathcal{N}^+$

Let $L_{u,v}$ be the local predicate for each leader edge (u, v). We construct $\mathcal{N}^+$ as follows. As in Chapter 5, for every leader edge (u, v) we add actions to nodes u and v to perform the local snapshot protocol on the (u, v) subsystem. The local snapshot protocol is used to detect a violation of predicate $L_{u,v}$. When the leader u detects a violation, u *makes a reset request.* The code is derived from the code used for the proof of the Local Correction theorem in Chapter 5 (Figures 5.6, 5.7, and 5.5) by making the following changes:

- All sending and receiving of packets in the modified node automata N_u^+ is replaced by sending and receiving of messages so that N_u^+ can be a user of the reset subsystem. Thus instead of sending request and response packets, we send request and response messages. Similarly we have to replace the free packet notification events with free message notification events.
- There is no longer a need for a *mode* variable in N_u^+ because each phase is implicitly a snapshot phase. Whenever the mode is checked in the original code, we only follow the code path corresponding to $mode = snapshot$.
- We add a $requestbit_u$ variable to N_u^+ to remember to do a reset request.

- When a response is received at the leader and a local predicate violation is detected, $requestbit_u$ is set to *true* (instead of changing $mode_u[v]$ to *reset*).

- We compose the modified node automata N_u^+ with the reset protocol $\mathcal{R}|L$ of Chapter 7. Thus in the final automaton, N_u^+ has input action SIGNAL_u and output action REQUEST_u.

- N_u^+ makes a reset request REQUEST_u whenever $requestbit_u$ is *true*; on taking this action $requestbit_u$ is set to *false*.

- On receiving a SIGNAL_u event, N_u^+ does a local restart. It first resets the basic state of the original automaton N_u to some initial value I_u (we will discuss how I_u is determined below). Also for each neighbor v of u, u will locally reset all the phase and free variables. More precisely, the $free_u[v]$, $freeq_u[v]$ and $phase_u[v]$ variables are all set to *false* for all neighbors v. The counter variable $count_u[v]$ is initialized to 1 (if u is the leader) and to 0 (if u is not the leader). This will ensure that after a signal event at u and v, all (u, v) phases are clean and all message send events at u are safe.

- The definition of local checkability in Chapter 5 ensures that there is always at least one global state s which satisfies all local predicates. Now if there are any messages in transit in global state s, we consider the state s' that results when all these messages are received by their destinations. Because the local predicates are all closed predicates, s' satisfies all local predicates as well. But in addition s' has no messages in transit. Then we choose the initial state I_u for each node u to be equal to $s'|u$.

The modified code is described in Figure 8.1, Figure 8.2, and Figure 8.3. The message formats and timing partitions are as in the proof of the Local Correction Theorem. As before, we hide all actions of $\mathcal{N}^+$ that are not actions of $\mathcal{N}$.

8.2.2 All reset triggers disappear in Linear Time

In the proof we will make heavy use of the reset protocol specification given in Chapter 7 and some aspects of the proof of the Local Correction Theorem in Chapter 5. Refer to Chapter 7 for definitions of normal messages, the send corresponding to a receive,

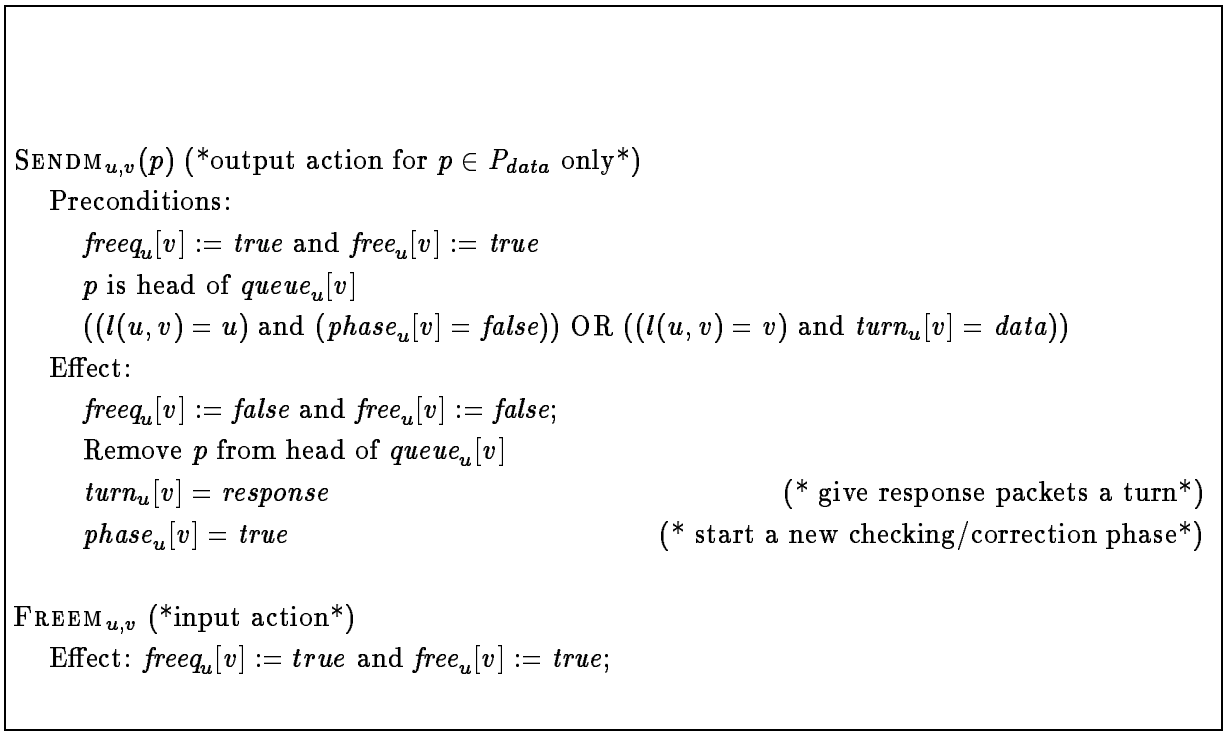

Figure 8.1: Code for the modified SEND$_{u,v}(p)$ actions at a modified node N_u^+ in order to do Global Correction.

SENDM$_{u,v}(p'_{req})$ (*output action: u repeatedly sends a request till it gets a response*)

Precondition:

$l(u,v) = u$ (*u is the leader of link subsystem*)

($phase_u[v] = true$) or ($queue_u[v]$ is empty) (*phase in progress or no data waiting*)

$freeq_u[v] = true$ (* no message in transit on link to v *)

$p'_{req}.count = count_u[v]$; (* counter correct*)

Effect:

$freeq_u[v] = false$ (* set to false until link says it is free*)

$phase_u[v] := true$; (*remains true until matching response returns*)

RECEIVEM$_{v,u}(p'_{req})$ (*input action, receive request at u from v*)

Effect:

If $p'_{req}.count \neq count_u[v]$ and $l(u,v) = v$ then (*not a duplicate or invalid message*)

$count_u[v] := p'_{req}.count$; (*remember count*)

Figure 8.2: Code to send and receive request messages at node u in order to do Global Correction.

SENDM$_{u,v}(p'_{resp})$ (*output action: u repeatedly sends a response to last request*)
  Preconditions:
    $l(u,v) = v$ (*u is not the leader of link subsystem*)
    ($turn_u[v] = response$) or ($queue_u[v]$ is empty) (*response's turn or no data messages waiting*)
    $freeq_u[v] = true$ (* no message in transit on link to v *)
    $p'_{resp}.count = count_u[v]$;
    $p'_{resp}.node_state = s|u$
  Effect:
    $turn_u[v] := data$ (*give data messages a turn*)
    $freeq_u[v] := false$ (* set to false until link says its free*)

RECEIVEM$_{v,u}(p'_{resp})$ (*input action to receive response at u from v*)
  Effect:
    If ($count_u[v] = p'_{resp}.count$) and ($phase_u[v] := true$) and ($l(u,v) = u$) then
      If $(s|u, nil, nil, p'_{resp}.node_state) \notin L_{u,v}$ then $requestbit_u := true$
      $phase_u[v] := false$; (*end of phase*)
      $count_u[v] := (count_u[v] + 1) \bmod 4$;

REQUEST$_u$ (*input action to make a reset request*)
  Preconditions:
    $requestbit_u := true$;
  Effect:
    $requestbit_u := false$;

SIGNAL$_u$ (*input action to receive a Signal at u*)
  Effect:
    For all neighbors v (*make links to all neighbors clean*)
      $phase_u[v] := false$;
      If $l(u,v) = u$ then $count_u[v] := 1$ else $count_u[v] := 0$
      Empty $queue_u[v]$ and set $freeq_u[v]$ and $free_u[v]$ to $false$
    $s|u = I_u$ (*initialize node state*)

Figure 8.3: Code to send and receive response messages and to make reset requests and respond to signals at node u.

causality, timeliness, and the mating relation. Refer to Chapter 5 for definitions of a phase and a clean link.

The proof consists of showing that all reset requests will eventually stop and that in linear time after this L holds. Define a (u,v) *reset trigger* to be the receipt of a snapshot response from v at u that causes u to set $requestbit_u = true$. The main part of the proof consists of showing that all reset triggers disappear in linear time (which by causality implies that all signals disappear). We start with some preliminary definitions.

Consider an execution α of $\mathcal{N}^+$. Define a *quiescent state* in α to be a state s_i in α such that:

- All messages received after state s_i in α are normal.
- All signal events that occur after state s_i are preceded by a request event that occurs in linear time before the signal event.
- For any pair of neighbors u and v, a signal event a_j at u that occurs after state s_i is accompanied by a signal event at node v that occurs within linear time before or after a_j.

We have the following lemma:

Lemma 8.2.1 Quiescent Lemma: *A quiescent state occurs within linear time of any execution of* $\mathcal{N}^+$.

Proof: $\mathcal{N}^+$ is the composition of the augmented node automata with $\mathcal{R}|L$. We know the behaviors of $\mathcal{R}|L$ are timely and causal. The lemma follows from the first timeliness property, the first causality property, and the fourth timeliness property. ∎

Suppose that u is the leader of some (u,v) subsystem. We show that in any execution α of $\mathcal{N}^+$, there exists some linear time t such that all (u,v) reset triggers disappear in time t after a quiescent state in α. Since a quiescent state occurs in linear time after the start of α, it follows that all reset triggers disappear in linear time after the start of α.

The proof is in two parts. First, we show that one of two cases must occur in linear time after a quiescent state. Next, we show that no (u,v) reset triggers can occur after the occurrence of *either* of the two cases.

One of Two Cases must Occur in Linear Time after a Quiescent State

To define the two cases, we need two simple preliminary definitions. Recall from Chapter 7 the definition of a signal interval and the definition of a send corresponding to a normal message. Intuitively, an *initialized signal interval* at a node is a signal interval that begins with a signal event at the node; we give it this name because each node *initializes* its local state immediately after a signal event. Next, a *regular* message receipt is a message that was received in an initialized signal interval and was sent in an initialized signal interval. Formally:

Definition 8.2.2 *An* initialized signal interval *at node u is a signal interval at node u that begins with a* SIGNAL_u *event.*

Definition 8.2.3 *A message receive event is* regular *if*

- *The message receive event is normal*
- *The message is received in an initialized signal interval at the receiving node.*
- *The send corresponding to the receive event occurs in an initialized signal interval at the sender.*

Next, we state the main lemma of this subsection.

Lemma 8.2.4 Quiescent Cases: *Consider any execution α of $\mathcal{N}^+$ and any leader edge (u,v). Within linear time after any quiescent state s_i of α there is a state s_j such that one of the following two cases is true:*

- **Case 1:** *Any message received by u from v (and vice versa) after s_j is* regular *OR*
- **Case 2:** *In state s_j, $L_{u,v}$ holds and (u,v) is clean. Also, the interval $[s_i, s_j]$ is contained in some signal interval at u as well as some signal interval at v, and these two signal intervals are mates.*

Proof: (Sketch) Recall the definition of a (u,v) phase from Chapter 5. Roughly speaking, a (u,v) phase is an interval that begins with the leader u sending a request and ends with u receiving matching response. Recall also the definition of a clean phase from Chapter 5. The proof also makes considerable use of the timeliness, consistency, and causality properties (see Definitions 7.3.3, 7.3.5, and 7.3.6) of the reset automaton $\mathcal{R}|L$.

We choose state s_j as some state that occurs after state s_i and satisfies either one of two properties:

a) **Property 1**: Six (u,v) phases occur in the interval $[s_i, s_j]$ AND there are no signal events at u or v in $[s_i, s_j]$ AND $L_{u,v}$ holds in state s_j.

OR

b) **Property 2**: There is some state s_k in the interval $[s_i, s_j]$ such that at least one signal event occurs at *both* u and v before s_k. Also, for any message receive event that occurs after s_j, the corresponding send event occurs after s_k (i.e., all messages in transit in state s_k are delivered before state s_j).

We now show that we can find such a state s_j that occurs within linear time after s_i.

From the second and third timeliness conditions, we see that message delivery between u and v behaves exactly like a UDL in the absence of signal events at u or v. In other words, the reset subsystem delivers messages and free notifications in constant time in the absence of signal events. Thus from the proof of the Phase Rate lemma in Chapter 5 (Lemma 5.6.5), we see that in the absence of signal events at either u or v, a (u,v) phase will complete in constant time. Similarly, in the absence of signal events at either u or v, six (u,v) phases will complete in constant time. By Lemma 5.6.12, the sixth (u,v) phase is a *clean* phase – i.e., a phase in which the matching response is sent by v after the receipt of the request. Thus the sixth phase will produce a correct snapshot of the (u,v) subsystem. Thus by the end of the sixth phase, either $L_{u,v}$ holds or node u will make a reset request. But if node u makes a a reset request, then by the second causality property, a signal event will occur at u in linear time after s_i.

We conclude from the last paragraph that in linear time after s_i either we reach a state s_j satisfying Property 1 or a signal event occurs at either u or v. But if a signal event occurs at *either* u or v in linear time after s_i, then we know (from our choice of s_i and the fourth timeliness property), that a signal event occurs at *both* u and v by

some state s_k that occurs within linear time after s_i. But in the latter case, the first timeliness property tells us that there is some state s_j that occurs within linear time of s_k and such that all messages "in transit" in state s_k have been delivered before state s_j. Thus in linear time after s_i we reach a state s_j satisfying either Property 1 or Property 2.

We are now ready to prove the lemma. If s_j satisfies Property 2 we show that Case 1 must be true; and if s_j satisfies Property 1 we show that Case 2 must be true.

If s_j satisfies Property 2 then we know that all messages received at u from v (and vice versa) after state s_j was sent after some state s_k. We also know that at least one signal event has occurred at both u and v before state s_k. Thus all such message receive events are regular, and we have Case 1 in the lemma statement.

If s_j satisfies Property 1 then we know by Lemma 5.6.12 that (u,v) is clean in state s_j. We also know that $L_{u,v}$ holds in s_j and that no signals occur at either u or v in $[s_i, s_j]$. Thus there must be some signal interval (say S_u) at u that contains the interval $[s_i, s_j]$. Similarly, there must be some signal interval (say S_v) at v that contains the interval $[s_i, s_j]$. Now the sixth (u,v) phase in $[s_i, s_j]$ is a clean phase. But in a clean phase, v sends a response to u after v receives a request from u. Thus there is at least one message sent after s_i by u that was received before s_j at v. Hence, from the consistency property, the intervals S_u and S_v must be mates. Thus we have Case 2 in the lemma statement. ∎

All (u,v) triggers disappear after either Case 1 or Case 2

Next we show that for any leader edge (u,v), all (u,v) reset triggers stop after one of these two cases listed above occur. Since we have just shown that that one of these two cases must occur in linear time, we conclude that all reset triggers disappear in linear time.

We first show all (u,v) triggers disappear after Case 1 occurs:

Lemma 8.2.5 Case 1 Trigger Termination: *Consider any execution α of $\mathcal{N}^+$ and any leader edge (u,v). Suppose that after any quiescent state s_i of α there is a state s_j such that any message received by u from v (and vice versa) after s_j is* regular. *Then no (u,v) triggers occur after state s_j in α.*

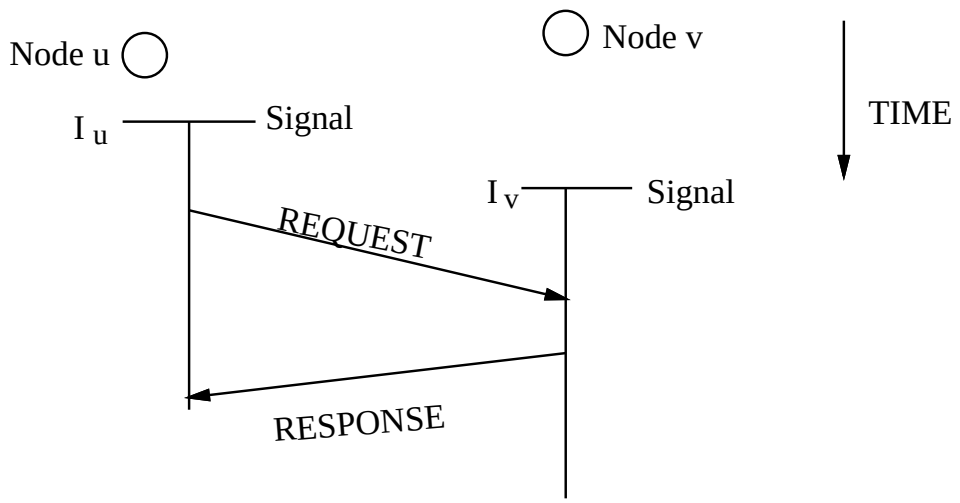

Figure 8.4: Correct Snapshots after nodes u and v have each performed a signal event.

Proof: (Sketch) Recall that a (u,v) trigger is the receipt of a response at u which causes u to set $requestbit_u = true$. Consider the receipt of any response at u after s_j. By construction, this response must be received in an initialized signal interval (say S_u at u) and must correspond to a response sent in an initialized signal interval (say S_v at v). This is shown in Figure 8.4.

We claim that the sequence of messages received by u in S_u is a prefix of the sequence of messages sent by v in S_v, and vice versa. We will refer to this as the *prefix* property. The prefix property follows from the consistency property because:

- S_u and S_v are mates,
- All messages received after s_i are normal and
- All messages sent in an initialized signal interval are safe.

We now claim that that the matching response cannot be a (u,v) trigger because of the following argument. First, at the start of S_u, u sets its basic state to I_u and at the start of S_v, node v sets its basic state to I_v. But $(I_u, nil, nil, I_v) \in L_{u,v}$. Next, at the start of S_u, $count_u[v]$ is initialized to 1; also at the start of S_v, $count_v[u]$ is initialized to 0. Thus by the prefix property, the sequence of states and messages sent and received in S_u and S_v *could have occurred in some execution* γ of the (u,v) subsystem, such

that (u,v) is clean and $L_{u,v}$ holds in the first state of γ. Thus, as in Chapter 5, any matching responses will carry accurate snapshot information, and will not result in a (u,v) reset trigger. ∎

Note that what makes this and the result in Chapter 5 work is that $L_{u,v}$ *is a closed local predicate* — once $L_{u,v}$ holds, it continues to hold, regardless of the behavior of other subsystems. Next, we show all (u,v) triggers disappear after Case 2 occurs:

Lemma 8.2.6 Case 2 Trigger Termination: *Consider any execution α of $\mathcal{N}^+$ and any leader edge (u,v). Suppose that after any quiescent state s_i of α there is a state s_j such that:*

- *$L_{u,v}$ holds and (u,v) is clean in s_j.*
- *The interval $[s_i, s_j]$ is contained in some signal interval at u as well as in some signal interval at v, and these two signal intervals are mates.*

Then no (u,v) triggers occur after state s_j in α.

Proof: (Sketch) The argument is extremely similar to that for Case 1 except in the signal intervals at both u and v that contain the interval $[s_i, s_j]$. This is shown in Figure 8.5. Let the signal interval at u that includes state s_j be called S_u. Let the signal interval at v that includes state s_j at be called S_v. We know that S_u and S_v are mates by assumption, and that in s_j, $L_{u,v}$ holds and (u,v) is clean. This is shown in Figure 8.5.

Thus the sequence of states and messages sent and received in S_u and S_v after state s_j *could* have occurred in some execution γ of the (u,v) subsystem, such that (u,v) is clean and $L_{u,v}$ holds in the first state of γ. But then, any matching responses will carry accurate snapshot information, and will not result in a (u,v) reset trigger. Thus in the first signal intervals for Case 2 we rely on an explicit state s_j (in which the local predicate holds and the link is clean). By contrast, in Case 1 we relied on the first signal intervals being initialized signal intervals.

The argument for signal intervals after S_u and S_v in Case 2 is identical to the argument for Case 1 because such signal intervals are initialized signal intervals. ∎

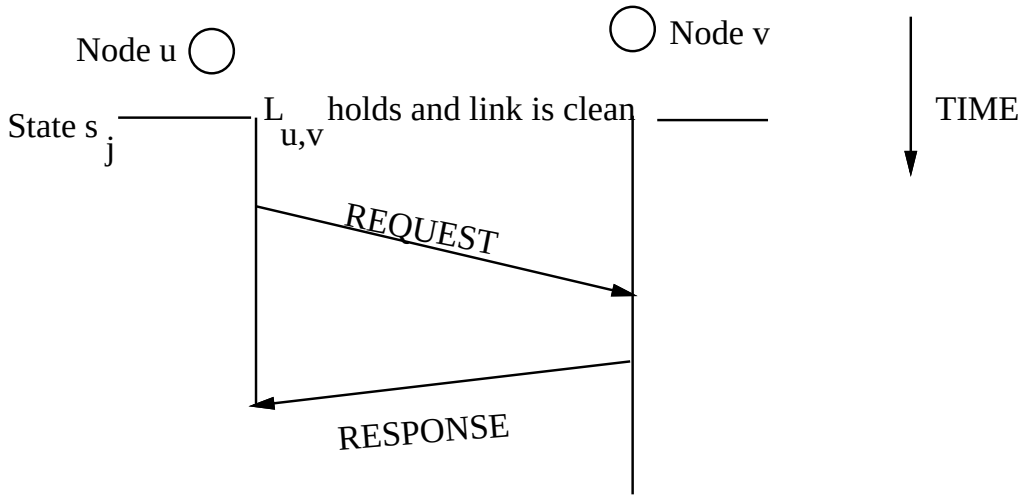

Figure 8.5: Correct Snapshots after $L_{u,v}$ holds and link (u, v) is clean.

8.2.3 All Local Predicates Hold and All Signals Stop in Linear Time

From the Quiescent Lemma, the Quiescent Cases Lemma, and the Case 1 and Case 2 Trigger Termination Lemmas, it follows that all reset triggers disappear in linear time after the start of any execution of $\mathcal{N}^+$. Hence, by causality, all signal events also disappear in linear time. More precisely, any execution α of $\mathcal{N}^+$ has some $O(n)$-suffix γ in which there are no signal events.

But now it is easy to see that in constant time in γ, all local predicates hold. We know from the second and third timeliness conditions (and the fact that there are no signals in γ) that six (u, v) phases will complete in constant time after the start of γ'. But after these six phases are complete, $L_{u,v}$ must hold. If not, u would detect a violation in the sixth phase (which is guaranteed to be a clean phase) and u would have made a reset request. But this will result in a signal event in γ, a contradiction.

Lastly any execution of $\mathcal{N}^+$ which contains no signal events and in which $L_{u,v}$ holds for all links can be shown to be an execution of $\mathcal{N}(c)|L$ for some constant c. As in Chapter 5, the c represents a constant slowdown due to the overhead of periodic checking.

8.2.4 Proof of Global Correction Theorem is Modular

We note that in the transformation used to prove the Global Correction Theorem we used the reset protocol $\mathcal{R}|L$ from Chapter 7. However, the proof only uses properties of the behaviors of $\mathcal{R}|L$ and thus $\mathcal{R}|L$ could have been replaced by any automaton that has the same interface as $\mathcal{R}|L$ and has the same behaviors. Further, note that the modified node automata N_u^+ (that we create by the transformation) are UIOA and $\mathcal{R}|L$ is a CIOA (since every reachable state is also a start state). Thus, the modularity theorem assures us that we can replace $\mathcal{R}|L$ by its stabilizing implementation (the automaton $\mathcal{R}^+$ as described in Chapter 7) without affecting the Global Correction result.

8.3 A Locally Checkable Spanning Tree protocol

We begin by describing earlier stabilizing spanning tree protocols and their disadvantages. Then we describe a locally checkable spanning tree protocol. This protocol will be used in Section 8.5 to construct a stabilizing spanning tree protocol

8.3.1 Previous Work on Spanning Tree Protocols

The basic idea in virtually all spanning tree algorithms is that nodes report the smallest node ID seen so far (and the shortest distance to this smallest ID node) to their neighbors. Each node then picks as its parent the neighbor that knows of the smallest ID. If more than one neighbor reports the smallest ID, the node picks from among these the neighbor that reports the smallest distance. If two neighbors report the same distance and root ID, then an arbitrary tie-breaker is used to select the parent. A node sets its estimate of the root ID to equal its parent's root ID, and updates its distance from the root to be one plus the parent's distance. Because of the way the distance from the root is calculated, this approach is basically an adaptation of the Bellman-Ford Algorithm.

However, in a dynamic network (and in a stabilizing setting), this approach encounters an obstacle known as "ghost roots". This phenomena occurs whenever the root crashes: its ID, which was the smallest in the system, is still the smallest from

the point of view of nodes that do not know of its crash. Even in a static network, the same effect can be caused by initial errors that introduce a root ID lower than the ID of any network node. This ID can potentially remain forever in the system! Even if the nodes maintain a counter to reflect their distance from the alleged root, this counter will simply grow unboundedly. Since the ghost root phenomenon is not a rare event, several ways to overcome this difficulty have been suggested.

One solution, known as the "hop counter" approach, is to have some pre-determined bound on the diameter of the network at each node, and to discard the old root whenever the associated counter reaches some limit (see, e.g.,[AG90]). Unfortunately, this pre-determined bound must be quite high, and hence, the stabilization time of such counting up schemes is poor in practice [CRKG89a, CRKG89b, Gar89, RF89, Awe90].

Another widely used stabilizing Spanning tree protocol is the IEEE 802.1 bridge routing protocol which is based on the design in [Per85]. This solution uses an approach that we call timer flushing. The basic idea [Per85] is that each node "times out" information received from its neighbors unless the information is refreshed periodically. Any node that thinks it is the root is responsible for periodically sending updates to all its neighbors in order to refresh their state information. Any other node X sends estimates to its neighbors only after receiving a message from the node X thinks is its parent. The upshot is that if there is an old root in the system, the information about this old root will eventually "time out", after which the system will stabilize.

However, timer flushing suffers from several drawbacks. First, the node clocks (by which packet lifetimes are enforced) can have different rates. Second, the message delivery time over links typically has large variance; since the topology of the network is not known in advance, the variance of the message delivery time across the whole network is even higher. A conservative timeout bound must take into account high message latencies and the worst-case topology, even though the latencies of an actual execution may be considerably (e.g., an order of magnitude) smaller. This leads to an order of magnitude slowdown of the stabilization process. Lastly, the parameters of such "global" timeout protocols often have complicated dependencies. Tuning these parameters is typically quite difficult.

Th stabilizing spanning tree protocol we will describe in Section 8.5 is extremely similar to the two schemes we have described above. However, it uses a different mechanism for detecting and recovering from states with ghost roots that speeds up the stabilization time of the resulting protocol.

The detection mechanism is based on the following observation. Consider an execution of the simple spanning tree protocol that starts with a state in which all nodes are correctly initialized and there are no messages in transit on the links. Now focus on some node. Throughout the execution the node maintains a current estimate of the root ID, and another estimate for its distance from this alleged root. It can be shown that in the course of legal executions, the node's estimate of the root ID never goes up; also while such a root estimate is fixed, the distance estimate never goes up. This property can be cast in the form of a local predicate for each link. If the predicate holds, then the algorithm will produce a spanning tree. This immediately suggests the stabilizing algorithm: whenever the predicate is violated, the node that detects the violation makes a reset request. In the execution that follows the last reset signal, all the information will be correct.

Rather than directly describe the final spanning tree protocol of Section 8.5, we will derive it in the following way. In the next subsection, we will describe a locally checkable (but non-stabilizing) spanning tree protocol on which the final spanning tree protocol is based. Of course, the Global Correction Theorem is applicable to this protocol. However, we will show in the following section that the spanning tree protocol is also *one-way checkable*. We then combine these observations to produce the final stabilizing spanning tree protocol in Section 8.5. The resulting protocol is simpler and more efficient than a spanning tree protocol based only on Global Correction.

8.3.2 Code for Locally Checkable Spanning Tree Protocol

We now describe the code for our locally checkable (but non-stabilizing!) spanning tree protocol. After all local predicates hold, this protocol computes a spanning tree in time proportional to the diameter of the network.

Before we delve into the code of the protocol, we define what it means for a spanning tree protocol to be stabilizing. We assume that the network topology is described by some topology graph G and that there is an output action $\text{REPORT}_u(p)$ at every node u. This action is used to report the parent p of node u in the spanning tree, where p is some neighbor of u in the network graph. We say that a spanning tree protocol is stabilizing if it stabilizes to the *stable tree behaviors* for graph G. The stable tree behaviors for graph G are the behavior β in which for every node u:

- For every t-suffix γ of β and for every node u, a REPORT_u event occurs in constant time. (i.e., nodes report their parent values at constant intervals of time.)
- For every node u, if there is $\text{REPORT}_u(p_1)$ event and a later $\text{REPORT}_u(p_2)$ event in β, then $p_1 = p_2$. (i.e., the parent values reported by nodes never change).
- Consider any set of $\text{REPORT}_*(*)$ actions in β containing exactly one $\text{REPORT}_u(p_u)$ action for every node u. Then the graph induced by the values of the p_u variables is a spanning tree of G. (i.e., the parent values reported by nodes forms a spanning tree of G)

Our locally checkable spanning tree protocol is described by the automaton T_u shown in in Figure 8.6. The protocol is based on the simple idea alluded to earlier. Each node keeps track of the smallest node ID seen so far and the shortest distance to this smallest ID node. Each node uses this information to compute its parent in the tree.

Thus each node u maintains a parent pointer $parent_u$, current estimate of the root's identity r_u, and a current distance estimate d_u. We denote by (r_u, d_u) the ordered pair at node u. We will use lexicographic ordering for 2-tuples and 3-tuples. For example, $(r_v, d_v) < (r_u, d_u)$ means that either $r_v < r_u$, or that $r_v = r_u$ and $d_v < d_u$. Similarly, $(r_v, d_v, parent_v) < (r_u, d_u, parent_u)$ means that that either $(r_v, d_v) < (r_u, d_u)$, or that $(r_v, d_v) = (r_u, d_u)$ and $parent_v < parent_u$. Each node also maintains a copy (possibly outdated) of the root and distance estimates of each its neighbors. Thus the estimates of neighbor v are stored at u in the variables $r_u[v]$ and $d_u[v]$.

For compactness, we let both arrays have an entry for the node itself, in which the default values are "hardwired": thus $r_u[u] = u$, and $d_u[u] = -1$. Now a node always chooses its own estimate based on the minimum of the estimates of all its neighbors and its own default estimate. Node u chooses its own estimate of the root from this minimum estimate; u also chooses its own estimate of the distance as one plus the distance from the minimum estimate. Thus if u itself is the smallest root, the distance u chooses for itself will be $-1 + 1 = 0$, which is as it should be. Thus $d_u[u]$ is set to -1 simply as a sentinel value to avoid an extra case.

Every node u periodically sends its own estimate of root and distance to all its neighbors using an "Announce" packet. To make the T_u a node automaton, we have followed the convention (see Chapter 5) of first enqueuing the "Announce" packet on an

outbound queue for the link called $queue_u[v]$; then the packet is sent out from the queue when the link is free. The Announce packet is encoded as a tuple $(Announce, r, d)$. When a node u receives an Announce packet from v, u first checks whether the estimate in the packet is greater than the previous estimate stored from v and the distance in the estimate is not already at the maximum possible value. If this is not the case, u stores the latest estimate it has received from v; u also updates its own estimate as the minimum of all neighbor estimates.

Let T_u be the automaton shown in in Figure 8.6. We assume that for any u, v, all actions of the form $\text{SEND}_{u,v}(*)$ are in a separate class, and any $\text{ENQUEUE}_{u,v}$ action is in a separate class. Similarly for any u, all actions of the form $\text{REPORT}_u(*)$ are in a separate class.

Let $\mathcal{T}$ be the composition of the automata node T_u for every node in G with a UDL for every edge (u, v) in G. In the next subsection we show that $\mathcal{T}$ is locally checkable for a predicate L that we define. In the following subsection, we show that $\mathcal{T}|L$ stabilizes to the stable tree behaviors in time proportional to the diameter of graph G. This sets the stage for applying the global correction theorem to $\mathcal{T}$.

8.3.3 Local Predicates for $\mathcal{T}$

We state a set of local predicates for the spanning tree protocol. Let us denote by $O_{u,v}$ the intersection of the local predicates shown in Figure 8.7 for any edge (u, v). Note that $O_{u,v}$ consists of two predicates, $O1(u, v)$ and $O2(u, v)$. $O1(u, v)$ says that the sentinel values that node u uses (as a neighbor estimate from itself) are correct. It also says that node u's estimate is the minimum of all its neighbors.

$O2(u, v)$ shows that the sequence of estimates sent from u to a neighbor v are non-increasing. Suppose there is an $(Announce, r, d)$ packet in transit from u to v, then the estimates that v already has stored from u can be no less than the estimates in the packet (i.e., $(r_v[u], d_v[u]) \geq (r, d)$). Also the estimates in the packet can be no less than the estimates at u (i.e., $(r, d) \geq (r_u, d_u)$). If there is no $(Announce, *, *)$ packet in transit from u to v, then the estimates that v already has stored from u can be no less than than the current estimates at u (i.e., $(r_v[u], d_v[u]) \geq (r_u, d_u)$). Recall that a Unit Capacity Link only allows one outstanding packet on each link, just as in a UDL.

State

$neighborSet_u$	set containing all neighbors of u and u itself
$parent_u$	parent pointer, in $neighborSet_u$
d_u:	distance estimate, integer in the range $0..n'$, n' is an upper bound on number of nodes.
r_u:	root estimate, all root estimates are in the range of node identifiers
$r_u[v]$:	estimate of r_v for each v in $neighborSet_u$ except $r_u[u] = u$
$d_u[v]$:	estimate of d_v for each v in $neighborSet_u$ except $d_u[u] = -1$
$free_u[v]$	boolean, true if link to v is free for each v in $neighborSet_u$ except u itself.
$queue_u[v]$	a queue containing at most one $(Announce, *)$ packet to be sent to v.

Actions

FREE$_{u,v}$ (*input action to receive free notification from link to neighbor v*)
Effects: $free_u[v] = true$

SEND$_{u,v}(p)$ (*output action to send estimate to neighbor v*)
Preconditions: p is the head of $queue_u[v]$ and $free_u[v] = true$
Effects: $free_u[v] = false$; Remove p from $queue_u[v]$

ENQUEUE$_{u,v}$ (*output action to enqueue estimate to neighbor v*)
Preconditions: $queue_u[v]$ is empty
Effects: Add $(Announce, r_u, d_u)$ to $queue_u[v]$

RECEIVE$_{v,u}(Announce, r, d)$ (*input action to receive estimate from neighbor v*)
Effects:
If $(r, d) \leq (r_u[v], d_u[v])$ then (*received estimate is no greater than previous estimate*)
If $(d < n')$ (*received estimate not at max value *)
$(r_u[v], d_u[v]) := (r, d)$ (*update estimate from v*)
$(r_u, d_u, parent_u) := min\{(r_u[v], d_u[v] + 1, v), v \in neighborSet_u\}$

REPORT$_u(q)$ (*output action to report parent estimate*)
Preconditions: $q = parent_u$

Figure 8.6: Spanning tree protocol. Code for a node u.

Definition 8.3.1 *For any edge (u,v), we let $L_{u,v}$ be the intersection of $O_{u,v}$ and $O_{v,u}$. Let $\mathcal{L}$ be the link predicate set containing $L_{u,v}$ for every (u,v) in G and let $L = Conj(\mathcal{L})$.*

It is easy to show that each $L_{u,v}$ is a closed local predicate.

Lemma 8.3.2 *For a leader edge (u,v) and transition (s,a,s') of $\mathcal{R}$, if s satisfies $L_{u,v}$, then s' satisfies $L_{u,v}$.*

Proof: First, $O1(u,v)$ is clearly closed because the code never changes the value of $r_u[u]$ and $d_u[u]$. Also, if the code changes $(r_u, d_u, parent_u)$, it always sets this 3-tuple equal to the minimum of $(r_u[v], d_u[v], parent_u[v])$ for all v in $neighborSet_u$ as required by $O1(u,v)$.

Next, $O2(u,v)$ is closed because of a simple observation: *the (r,d) estimates sent in $(Announce, r, d)$ packets by node u are always non-increasing.* This follows because node u changes its own estimate after receiving an *Announce* packet from some neighbor w. But (see code), u will not accept the packet from w unless the new estimate from w is no greater than the previous value that u has stored from w and if the distance from w is not already at the maximum value n'. But if the new estimate from w is no greater than the previous value that u has stored from w, then the new estimate that u calculates will be no greater than before. But if $O2(u,v)$ is true initially and all estimates sent by u are non-increasing, then it is easy to see that $O2(u,v)$ remains true. Note that if u does not accept a packet from neighbor v (e.g., because the distance estimate in the packet is equal to n') this does not affect $O2(v,u)$

Finally, since the above argument can be applied symmetrically for the link (v,u), it follows that $L_{u,v}$ is closed. ∎

Note that the check in the code that prevents v from accepting an estimate larger than a previously stored estimate is really an application of the heuristic of removing unexpected packet transitions (see Chapter 6). Clearly if u receives an estimate from w larger than the previous estimate stored from w, then $O2(w,u)$ is not true in the state before the estimate was received. Thus this is an unexpected packet transition; by removing such a transition, we ensure that $L_{u,v}$ is a closed predicate.

The following theorem is immediate from the last lemma.

Theorem 8.3.3 *The network automaton $\mathcal{T}$ can be locally checked for L using $\mathcal{L}$.*

O1(u,v):

$r_u[u] = u$

$d_u[u] = -1$

$(r_u, d_u, parent_u) := min\{(r_u[v], d_u[v] + 1, v), v \in neighborSet_u\}$

O2(u,v):

If there is an $(Announce, r, d)$ packet in transit from u to v then

$(r_v[u], d_v[u]) \geq (r, d) \geq (r_u, d_u)$

Else (*there is no Announce packet in transit*)

$(r_v[u], d_v[u]) \geq (r_u, d_u)$

Figure 8.7: Spanning Tree Protocol: Local Predicates for edge (u, v).

8.3.4 Fast Computation of Spanning Tree after all Link Predicates Hold

We now show that $\mathcal{T}|L$ stabilizes to the stable tree behaviors in time proportional to the diameter of G. Recall that $\mathcal{T}|L$ is the spanning tree automaton described in Figure 8.6 such that all local predicates hold in the initial state.

The basic idea is to show first that if all local predicates hold then there can be no ghost roots; next, we show that if the spanning tree protocol begins in a state that does not contain a ghost root, then the protocol quickly converges to a stable spanning tree.

We assume that each node has a unique ID that is modelled by the name of the node in the topology graph. Thus the unique ID of node u is u. Our spanning tree protocol depends heavily on the fact that each node has a unique ID that cannot be corrupted. Thus we are assuming that the node ID is part of the code at a node.

We first define what it means to have a ghost root in a state of $\mathcal{T}$. Informally, consider the set of all the IDs present (in all the root estimates at nodes and in all the *Announce* packets in the links) in state s. If the minimal ID of all nodes in the network is not the the minimal ID in this set, then we have a ghost root. Formally:

Definition 8.3.4 *Consider a state s of $\mathcal{T}$. We say that s has a ghost root r if $r < u$ for all nodes u in G and either:*

- *There is an $(Announce, r, *)$ packet in transit from u to v in s for some u, v, OR*
- *$r_u[v] = r$ in s for some u, v, OR*
- *$r_u = r$ in s for some u.*

The only way we can have ghost roots in any execution is because of bad data in links and nodes when the protocol starts up.

Lemma 8.3.5 *Consider any execution α of $\mathcal{T}$. Suppose there is a state s_i in α such that there is no ghost root in s_i. Then then there is no ghost root in the state following s_i in α.*

Proof: From the code it is easy to see that the only way a new value is added to set of existing roots is if a node u adds its own ID to the set (e.g., by changing its root estimate to the default). Thus ghost roots cannot be created after state s_i. ∎

Next, we show that any state in which all local predicates hold cannot contain a ghost root.

Lemma 8.3.6 *There is no ghost root in any state of $\mathcal{T}|L$.*

Proof: Suppose we did have a ghost root in some state s of $\mathcal{T}|L$. Then there must be a ghost root r with the lowest ID among ghost roots in s. If it is in an *Announce* packet in transit from say v to u, then we know from $O2(v, u)$ that $r_v \leq r$ and hence $r_v = r$. On the other hand, if there is some w such that $r_w[x] = r$ or $r_w = r$, then we know from $O1(w, *)$ that $r_w \leq r$ and hence $r_w = r$. Thus, we have some node, say w, such that $r_w = r$.

By $O1(u, v)$ if $parent_w \neq w$ we can continue inductively following parent pointers. Each time we move from say w to $x = parent_w$, we know from $O1(w, *)$ that $(r_w, d_w) = (r_w[x], d_w[x] + 1)$. But since $r_w = r$, $(r_w[x], d_w[x]) < (r, d_w)$. But we know from $O2(x, w)$ that that $(r_x, d_x) \leq (r_w[x], d_w[x])$. Thus we conclude that $(r_x, d_x) < (r, d_w)$.

But the distance estimates are non-negative and the root estimates in this chain are all equal to r since r is the lowest ID ghost root. Hence the process of following *parent* pointers must terminate at some node say z such that $parent_z = z$ and $r_z = r$. Thus by $O1$ applied to z, we know that $r = z$. But that contradicts the fact that r is a ghost root. ∎

The last important fact to observe is that if there are no ghost roots and all local predicates hold, then the protocol converges in time proportional to the diameter of the network. As usual, we will say that $t = O(d)$ to mean $t \leq cd$ where c is a constant. We denote the diameter of G by D.

Lemma 8.3.7 *Consider an execution α of $\mathcal{T}|L$. Then there is some t-suffix of execution α (where $t = O(D)$) whose behavior is a stable tree behavior for graph G.*

Proof: We know from from Lemma 8.3.6 that there is no ghost root in the initial state of α and hence by Lemma 8.3.5 there is no ghost root in any state of α. From the fact that $O2(u,v)$ is closed, we know that $O2(u,v)$ holds for all links (u,v) in all states of α. Let q be the node with the smallest ID in graph G. We prove the lemma using induction on the distance d of a node u from q.

Inductive Hypothesis: There exists some constant c such that within $c \cdot d$ time after the start of α, there is some state in which $(r_u, d_u, parent_u) = (q, d, v)$, where v is some neighbor of u at distance $d-1$ from q. Also u will not change r_u,d_u or $parent_u$ from this state onwards in α.

The inductive hypothesis implies the lemma because it implies that within time proportional to the diameter, every node has a parent pointer that points to a node that is one hop closer to the root q. It also implies that the parent pointers do not change after this point. Also, since for each u, all $\text{REPORT}_u(*)$ actions are in a separate class, such a $\text{REPORT}_u(*)$ action must occur in constant time in any suffix of an execution.

Basis, $d = 0$: Node q is the only node at distance 0 from itself. In all states s_i of α, $(r_q, d_q, p_q) = (q, 0, q)$, or there would be a ghost root in state s_i which we have already ruled out.

Inductive Step: Suppose it is true for all nodes at a distance d from s and we wish to show that it is true for a node v at a distance $d+1$ from s. Thus there is some state s_i such that all nodes u within distance d from q have $(r_u, d_u, parent_u) = (q, d, *)$.

Also, s_i occurs within $c \cdot d$ time after the start of α. We first claim that in all states of α, $(r_v, d_v) \geq (q, d+1)$. Suppose this were not true in some state s_k of α. Then by following parent pointers as we did in the proof of Lemma 8.3.6 and by using predicates O1 and O2 iteratively, we can show that we must end in a ghost root. Since we have ruled out ghost roots, $(r_v, d_v) \geq (q, d+1)$ in all states of α.

Next, v must have some neighbor at a distance d from q. Let u be the neighbor with the lowest ID among the neighbors of v at distance d from q. Thus by the properties of a UDL, within constant time after state s_i, an $(Announce, q, d)$ packet from u will arrive at v which will cause $(r_v, d_v) = (q, d+1)$. (Note that this packet will be accepted at v because in the previous state $(r_v, d_v) \geq (q, d+1)$ as we have just shown and $d < n'$.) Also since u is the lowest ID neighbor at distance d from q, v will set $parent_u = u$ and this will remain unchanged for the rest of the execution. ∎

Theorem 8.3.8 *$\mathcal{T}|L$ stabilizes to the stable tree behaviors for graph G in $O(D)$ time, where D is the diameter of G.*

Proof: From Lemma 8.3.7 ∎

At this stage, we could apply the Global Correction Theorem to obtain a stabilizing version of $\mathcal{T}$. However, there is an even simpler transformation because $\mathcal{T}$ is *one-way checkable*. Before we present the final version of the stabilizing spanning tree protocol, we first define what we mean by a one-way checkable protocol.

8.4 One Way Predicates and Local Checking

In Section 8.1, we claimed that every locally checkable protocol could be stabilized using the global reset protocol of Chapter 7. The proof sketch suggested that this could be achieved by periodically doing a local snapshot of each local subsystem and making a reset request if a violation is detected.

However, in this section we will show that if a locally checkable protocol P is also what we call *one-way checkable*, then we can locally check P using a simpler and faster method than doing a local snapshot. In this method, each node periodically sends its entire state to each of its neighbors. We can call this *local checking by periodic sending of state* or simply periodic sending. The question that remains is: when is periodic

sending adequate to detect local violations in lieu of a local snapshot? The answer is that periodic checking is sufficient if each local predicate is what we call a *separable* local predicate.

Intuitively, a separable local predicate can be separated into two one-way predicates, one for each direction of a link subsystem. Intuitively a *one-way predicate* for link (u,v) is a predicate that only involves the state of u, the state of the link (u,v), and the state of node v. *Note that it does not depend on the the state of the link* (v,u). Formally:

Definition 8.4.1 *Consider any network automaton $\mathcal{N}$ with graph G. Let (u,v) be any edge in G. We say that $O_{u,v}$ is a one-way predicate for edge (u,v) of $\mathcal{N}$ if:*

- *$O_{u,v}$ is a local predicate of $\mathcal{N}$ for edge (u,v).*
- *For any two states s and s', suppose s satisfies $O_{u,v}$ and $s|u = s'|u$ and $s|v = s'|v$ and $s|(u,v) = s'|(u,v)$. Then s' satisfies $O_{u,v}$.*

For example, consider the second predicate of the spanning tree protocol listed in 8.7. Recall that it essentially says that the estimate values in transit from v to u are always non-increasing. Clearly this is a one-way predicate. It is not hard to see that the first predicate (which involves only the state of one node) is also a one-way predicate.

Definition 8.4.2 *Consider any network automaton $\mathcal{N}$ with graph G. Let (u,v) be any edge in G. We say that $L_{u,v}$ is a separable local predicate for edge (u,v) of of $\mathcal{N}$ if there exist two one-way predicates $O_{u,v}$ and $O_{v,u}$ (one for edge (u,v) and edge (v,u) respectively) such that:*

Any state s of $\mathcal{N}$ satisfies $L_{u,v}$ iff s satisfies both $O_{u,v}$ and $O_{v,u}$

Once again it is easy to see that the spanning tree protocol has a link predicate set that consists of separable predicates. In this case, we say that the spanning tree protocol is *one-way checkable.*

Definition 8.4.3 *A network automaton $\mathcal{N}$ is one-way checkable for predicate L using link predicate set $\mathcal{L}$ if:*

- *$\mathcal{N}$ is locally checkable for predicate L using link predicate set $\mathcal{L}$.*
- *Each $L_{u,v} \in \mathcal{L}$ is a separable local predicate.*

We now claim the following fact informally. *Any one-way checkable automaton can be locally checked by periodic sending.* We simply add actions to send the state of a node periodically to each of its neighbors and actions to receive such packets.

Suppose that in state s, node u receives such a periodic packet from node v containing the value x. Then u checks whether $(x, nil, nil, s|u) \in O_{v,u}$; if this is not true u detects a local violation.

Notice that local checking by periodic sending is not just simpler than doing a local snapshot but is also *faster*. The proof of Lemma 5.6.12 in Chapter 5 tells us that it takes 6 checking/correction phases for the local snapshot protocol to stabilize. Roughly, this means that the local snapshot protocol takes 12 link delays (where a link delay is the time to send a control packet from a node to its neighbor) to stabilize. By contrast, the periodic sending protocol takes 1 link delay to stabilize. Thus *for one-way checkable protocols it always pays to use periodic sending for local checking.*

8.5 Simple Stabilizing Spanning Tree Protocol

We have shown that the spanning tree protocol described in Figure 8.6 is locally checkable. Thus we can apply the Global Correction theorem to this protocol. This requires checking link predicates using local snapshots. However, since the spanning tree protocol described in Figure 8.6 is also one-way checkable, we can can replace the local snapshot by periodic sending of state information. The resulting protocol is simpler and faster than a spanning tree protocol that uses local snapshots.

If we look at the protocol in Figure 8.6, we see that each node sends $(Announce, r, d)$ packets periodically containing the root and distance estimates at the node. But if we look at the predicates in Figure 8.7, $O1(u, v)$ involves only variables at u, and $O2(u, v)$ only involves root and distance estimates at v. *Thus we can rely on the* $(Announce, r, d)$ *message for periodic checking.* This is shown in the code of Figure 8.8 which only describes the changes to the code of Figure 8.6 to convert it into a stabilizing protocol.

The first change is that we have applied message renaming to the original protocol and replaced all packet events with message events.

The rule for one-way checking is as follows. If in state s, node u receives a periodic message from node v containing the value x, then u checks whether $(x, nil, nil, s|u) \in O_{v,u}$; if this is not true u detects a local violation. In our case, the $(Announce, r, d)$ message does not contain the complete state x of node v, but only a projection of the state of node v that is sufficient to do checking. But $(x, nil, nil, s|u) \in O2(v, u)$, iff $(r, d) \leq (r_u[v], d_u[v])$. Thus to check whether $O2(v, u)$ holds it is sufficient to check whether $(r, d) \leq (r_u[v], d_u[v])$ and make a reset request if a violation is detected. Also notice that we do not need to check for the $O1(v, u)$ predicate because we have added a few lines of code that ensure that $O1(v, u)$ will hold after the first *Announce* message received by v.

Let T'_u be the modified node automaton shown in Figure 8.8. Let $\mathcal{R}|L$ be the reset automaton for graph G as described in Chapter 7. Let $\mathcal{T}'$ be the automaton formed by composing T'_u for each node u with $\mathcal{R}|L$.

Then we have the following theorem:

Theorem 8.5.1 *$\mathcal{T}'$ stabilizes to the stable tree behaviors for graph G in $O(n)$ time.*

Proof: (Sketch) We first show that $\mathcal{T}'$ stabilizes to the behaviors of $R(\mathcal{T}(c)|L)$ in $O(n)$ time. The operator R denotes message-renaming, as in the Global Correction Theorem. This part of the proof uses arguments similar to the ones used in the proof of the Global Correction Theorem except that the arguments are simpler because of the use of one-way predicates and one-way checking.

Next, we know from Theorem 8.3.8 that $\mathcal{T}|L$ stabilizes to the stable tree behaviors of graph G in $O(D)$ time. D is the diameter of G and is no greater than n, the number of nodes. But if $\mathcal{T}|L$ stabilizes to the behaviors in P in $O(D)$ time, then so does $R(\mathcal{T}(c)|L)$. This is because the set of stable tree behaviors is invariant under the operations of message-renaming and scaling by constant factors. Finally, since behavior stabilization is transitive, we conclude that $R(\mathcal{T}(c)|L)$ stabilizes to the behaviors in P in $O(D + n) = O(n)$ time. ∎

Finally, we note that because T'_u is a UIOA and $\mathcal{R}|L$ is a CIOA the modularity theorem assures us that we can replace $\mathcal{R}|L$ by its stabilizing implementation ($\mathcal{R}^+$ as

described in Chapter 7) without affecting the result. This is an elegant application of the modularity theorem because it means that we can begin with a stabilizing version of a UDL; next, as shown in Chapter 7 we can use the stabilizing UDL implementations to construct a stabilizing version of a reset protocol; and finally we can use the stabilizing version of the reset protocol to construct a stabilizing spanning tree protocol. In the process, we have two applications of the modularity theorem. We can go even further and change $\mathcal{T}'$ so that it offers a suitable interface to the token passing example of Chapter 6. If we do so, we could construct a stabilizing token passing protocol using the stabilizing spanning tree, once again relying on the modularity theorem.

In any of these modular constructions, we can replace major pieces of the construction by other modules that offer the same external behavior.

8.6 Stabilizing topology update protocol

The topology update task is to broadcast the list of incident operating edges at a node to all other network nodes. Thus the goal of the topology update task is to produce at each node u a database listing the operating edges of each node that is reachable from u.

The following simple strategy is often used for topology maintenance. Each node maintains a sequence number. Whenever a change occurs, the incident nodes increment their sequence number and broadcast the new status of all their incident links in a link state packet ([Per83]). Any link state packet sent by node u contains node u's sequence number. Whenever a node v receives a message purporting to be from u, v checks whether the sequence number on the message is larger (i.e., newer) than the sequence number of the last update v has stored from u. If so, v stores this new update from u (after discarding any previous update it has stored from u) and broadcasts the update to all neighbors. Otherwise, the message is simply discarded as an outdated update.

Now the sequence number field is finite. Even if we allocate 64 bits for sequence numbers, it is always possible (due to errors) for a link state packet with a maximum sequence number value to be present in the initial state of the network.

In the ARPANET [MRR80] and DECNET [Per83] protocols, erroneous updates with the maximum number are removed by what we had earlier called "timer flushing".

State

$requestbit_u[v]$ boolean, true if a request is to be done later.
All other variables remain unchanged.

Modified Actions

$\text{FREEM}_{u,v}$ (*message-renamed free action, replaces $\text{FREE}_{u,v}$ action*)
Effects: $free_u[v] = true$

$\text{SENDM}_{u,v}(p)$ (*message-renamed output action to send estimate to neighbor v*)
Preconditions: p is head of $queue_u[v]$ and $free_u[v] = true$
Effects: $free_u[v] = false$; Remove p from $queue_u[v]$

$\text{RECEIVEM}_{v,u}(Announce, r, d)$ (*input action to receive estimate from neighbor v*)
Effects:
If $(r, d) \leq (r_u[v], d_u[v])$ then (*received estimate is no greater than previous estimate*)
If $d < n'$ (*distance estimate not already at max value*)
$(r_u[v], d_u[v]) := (r, d)$ (*update estimate from v*)
Else $requestbit_u = true$ (*if received estimate is greater make reset request later*)
$(r_u[u], d_u[u]) := (u, -1)$; (*next two lines make $O1(u, v)$ hold*)
$(r_u, d_u, parent_u) := min\{(r_u[v], d_u[v] + 1, v), v \in neighborSet_u\}$

REQUEST_u (*output action to request a reset*)
Preconditions: $requestbit_u = true$
Effects: $requestbit_u = false$

SIGNAL_u (*input action to receive a reset signal*)
Effects:
For all $v \neq u \in neighborSet_u$ do
$(r_u[v], d_u[v]) := (\infty, \infty)$ (*reset all estimates*)
$free_u[v] := false$
$(r_u[u], d_u[u]) := (u, -1)$;
$(r_u, d_u, parent_u) := (u, 0, u)$

Figure 8.8: Spanning tree protocol. Modifications to the Code for a node u.

Each update (and hence the associated counter value) is assumed to remain in the network only for some limited "lifetime," after which it is discarded. This prevents the problem because after its lifetime expires, an erroneous counter value is no longer present in the network. Once again, the disadvantage of timer flushing is that the timeout periods have to be high, which results in high stabilization times.

Instead of using timer flushing, we can use global correction to create a faster stabilizing topology update protocol. This is because the protocol is easily seen to be locally checkable for the property L which states that the maximum value of any sequence number is not present in the network. Thus we can apply the Global Correction theorem to ensure that maximum size sequence numbers are removed from the network. On receiving a signal, each node deletes the link state packet of all nodes other than itself, and resets its own sequence number to 0. A node that does not have a link state packet from u will accept any update sent by u as being newer.

Thus, the number of bits allocated for the sequence number affects only the performance of the algorithm: errors that cause the sequence numbers to reach the maximum value incur a performance penalty — a network latency of $O(n)$.

A complete design of a stabilizing topology update protocol would also have to add a number of other actions as suggested in [Per83]. For instance, each node must periodically increment its sequence number and send its latest Link State Packets to all its neighbors. Basically, we suggest keeping the essential simplicity of [Per83] and only replace the need for global timers in [Per83] with global correction.

8.7 Summary

The major point of this chapter is to show that any locally checkable protocol can be stabilized using the global reset protocol developed in the last chapter. In general, this is done by having the leader of every every link subsystem do a periodic snapshot; if the leader detects a local violation, it makes a reset request. However, in the special case that the local predicates can be separated into one-way predicates, we can dispense with a snapshot and use periodic sending of state. Local Checking by periodic sending is simpler and faster than using a local snapshot.

An important example of both techniques is furnished by the spanning tree protocol described in Section 8.3. Both the spanning tree and topology update protocols

described are quite practical and could be used in real networks.

The topology update protocol demonstrates a useful paradigm for designing stabilizing protocols. The essence of the idea is to design a stabilizing protocol assuming the use of unbounded counters. Next, we use global correction to convert this protocol into a more practical protocol that uses bounded counters.

This chapter also demonstrates why the specification of a reset protocol must specify the behavior in non-final signal intervals and not just in the final signal intervals. The proof of the Global Correction relies strongly on the mating relation between non-final intervals. The intuition is that the augmented automaton is doing local checking of the system during non-final intervals; if local checking does not eventually observe consistent behavior, the augmented automaton may keep making reset requests and the protocol may never stabilize.

Chapter 9

Compiling Synchronous Protocols

In the previous chapters we have shown how we can stabilize certain protocols by local checking and/or correction of the state of a node and its neighbors. We applied this technique to obtain stabilizing solutions to some important tasks such as mutual exclusion, network resets, and computing spanning trees. By contrast, this chapter provides a general technique for a special class of problems (known as *non-interactive* tasks), for many of which (e.g., Minimal Spanning Tree, Min Cost Flows, etc.) *no locally-checkable implementation is known.* In fact, for many of the problems solved in this chapter, no efficient stabilizing solution was known previously.

In this chapter, we describe two compilers (Sections 9.5 and 9.6) that convert a deterministic synchronous protocol π into a stabilizing, asynchronous version of π. In essence, we efficiently transform a solution for the most restrictive model (synchronous, fault-free networks) to a solution that works in a very permissive model (asynchronous networks with catastrophic faults that stop).

Let T_π be the time complexity of π and S_π be the space complexity of π. Let n be the number of nodes in the network. The first compiler produces a version of π that stabilizes in time $O(n + T_\pi)$ and has the same space complexity as π. Thus the first compiler is extremely efficient if the time complexity of the original protocol π is at least $O(n)$. The second compiler produces a version of π that stabilizes in time $O(T_\pi)$ and has a space complexity of $T_\pi \cdot S_\pi$. Thus the second compiler is extremely efficient if the time complexity of the original protocol π is very small – i.e., $T_\pi = O(log(n))$.

These two compilers allow us to provide efficient stabilizing solutions for many

problems including minimum spanning trees, colorings, maximum flows, and maximal independent sets.

Despite the apparent change of direction, the ideas in this chapter are closely related to the ideas in the previous chapters. First, as we have said earlier, the ideas in this chapter extend the range of our general techniques. Our previous general techniques (Local, Tree and Global Correction) only apply to protocols that are locally checkable. Second, the compilers in this chapter are implemented using the the techniques developed earlier. The first compiler is based on a simplified form of local checking and correction that we call one-way checking and correction (see Chapter 8). The second compiler is based on local checking and global correction.

When we first presented these results in [AV91], the second compiler, the *Resynchronizer*, was based on a complicated construction. In this chapter, we provide a simplified construction using the reset protocol described in Chapter 7. We do not have a complete proof of the simplified construction, but we will outline why we believe our simplified construction is correct. Thus while our confidence in the *Resynchronizer* result is based on the original result in [AV91], we believe that the construction in this chapter offers the potential for a much simpler solution.

This chapter is organized as follows. First, we describe how we model interactive protocols and synchronous protocols. Next, we summarize the major results of the chapter. Then we contrast the notion of *distributed checking* (that we use in this Chapter) with the independent local checking that we have used in previous chapter. Next we briefly describe the first *Rollback* compiler and then describe the simplified version of the *Resynchronizer* compiler. Finally, we outline extensions for randomized protocols.

9.1 Non-interactive Protocols

Non-interactive protocols form an important subclass of distributed algorithms. These are protocols whose correctness can specified by a relation (the I/O relation) between its input and output. For example, if the protocol has to compute a spanning tree then the output (the tree) should be a spanning tree of the input (the graph G). By contrast, mutual exclusion is an interactive task because (see Chapter 6) its correctness must be expressed in terms of sets of valid behaviors or sets of valid executions.

We define the notion of a non-interactive protocol more formally using a slight variant of the network model introduced in Chapter 5. Recall that in Chapter 5, we modelled a network using a topology graph $G = (V, E, l)$, where V is a set of nodes, E is a set of directed edges and l is a leader function. For this chapter we will augment the notion of a topology graph to have an *extra component* I that represents the input. We will define an *augmented topology graph* $G = (V, E, l, I)$ where V,E and l are as before. I, however, is a vector of inputs such that for any node u, I_u represents the *local input* at node u. (In previous chapters the only input to a node automaton for node u was the list of neighbors of u and, in the case of a tree topology, the parent of u.)

We start by fixing an input domain $\mathcal{I}$ and an output domain $\mathcal{O}$.

Definition 9.1.1 *An augmented topology graph $G = (V, E, l, I)$ is a topology graph (V, E, l) together with an input specifier I, where I is vector of local inputs (I_u) and $I_u \in \mathcal{I}, \forall u \in V$.*

Informally, this amounts to modelling the input to the non-interactive protocol as part of the local code of each node (which cannot be corrupted) and not as part of the state of each node (which can be corrupted). Clearly no non-interactive protocol can hope to produce correct output from corrupted input! However, more generally, the input to the protocol could be the output of another stabilizing protocol. Thus in Chapter 5, we assumed the list of neighbors of each node u was part of the code at node u. But in many real networks, the list of neighbors of a node would be provided as the output of a stabilizing Data Link protocol. In this chapter, we will often refer to an augmented topology graph as an augmented graph.

Once again, we will model the protocol using node automata N_u (one for each node u in G), and unit capacity data links $C_{u,v}$ (one for each edge (u, v) in G).

Definition 9.1.2 *An automaton for augmented graph $G = (V, E, l, I)$ is an automaton consisting of an IOA for each $u \in V$ together with a UDL $C_{u,v}$ for each $(u, v) \in E$.*

Refer to Chapter 5 for a definition of the UDL $C_{u,v}$. Recall that for any state s of the automaton we let $s|u$ denote the state of the IOA for node u. For an augmented

graph, we also let (X_u) denote a vector consisting of an element X_u for each node u in the graph.

For non-interactive protocols we will also assume that there is some *local output function* O_u of the state of each N_u, such that $O_u(s)$ provides the *local output* of N_u in state s. Let $\mathcal{N}$ be the network automaton formed by the composition of the node and channel automata as described in Chapter 5. Using the O_u we can define the *output function* O to be a function of the state s of the network automaton $\mathcal{N}$ such that $O(s)$ produces a vector of local outputs that is identical to the outputs produced by each O_u when applied to $s|u$. Thus:

Definition 9.1.3 *Let $\mathcal{N}$ be an automaton for augmented graph $G = (V, E, l, I)$. We say that O is an output function for $\mathcal{N}$ if O is a vector of functions (O_u), such that for every state s of $\mathcal{N}$ and for every node $u \in V$, $O_u(s|u) \in \mathcal{O}$. We will also abuse notation by defining $O(s) = (O_u(s|u)), \forall u \in V$.*

Traditionally, the correctness of a non-interactive protocol $\mathcal{N}$ is described using an input-output relation R. In the traditional definitions, $\mathcal{N}$ is allowed to be an ordinary IOA (i.e., we are allowed to specify initial states). Next, the correct executions of N are those executions α of $\mathcal{N}$ in which there is some t-suffix γ of α such that for all states s in γ, $(I, O(s)) \in R$. In other words, in all executions of the protocol the output must eventually "settle" to a value that satisfies the I/O relation. Thus:

Definition 9.1.4 *Let $\mathcal{N}$ be an automaton for augmented graph $G = (V, E, l, I)$. An I/O relation for $\mathcal{N}$ is a set of tuples (I', O') where each I' is a vector $(I'_u), I'_u \in \mathcal{I}$ and each O' is a vector $(O'_u), O'_u \in \mathcal{O}$.*

Finally we define a non-interactive protocol to consist of two components: an automaton and an output function.

Definition 9.1.5 *A non-interactive protocol for augmented graph G is a tuple $(\mathcal{N}, O)$ where $\mathcal{N}$ is an automaton for augmented graph G and O is an output function for $\mathcal{N}$.*

For stabilization, we will keep the the correctness definition exactly as in the traditional definitions *except that $\mathcal{N}$ will typically be a UIOA.*

Definition 9.1.6 *Let $\mathcal{P} = (\mathcal{N}, O)$ be a non-interactive protocol for augmented graph $G = (V, E, l, I)$ and let R be an I/O relation for $\mathcal{N}$. Let C be the set of executions α of $\mathcal{N}$ such that for all states s in α, $(I, O(s)) \in R$. We say that non-interactive protocol $\mathcal{P}$ stabilizes to I/O relation R in time t if $\mathcal{N}$ stabilizes to the executions in C in time t.*

Thus this definition is a special case of the general definition of "stabilizes to the executions of" given in Chapter 3.

We now formally define two complexity measures that we use to evaluate stabilizing non-interactive protocols: the first measures the worst-case time for the protocol to stabilize, and the second measures the worst-case amount of space used by any node.

We define the *stabilization time* of a non-interactive protocol $\mathcal{P}$ with respect to I/O relation R to be the infimum of all t such that $\mathcal{P}$ stabilizes to R in time t.

The *space complexity* of $\mathcal{P} = (\mathcal{N}, O)$ is the worst case across all nodes u of the size of the set $\{s|u : s \in states(\mathcal{N})\}$

9.2 Synchronous Protocols

We model a deterministic synchronous protocol π as follows. The network topology is again specified by an augmented graph $G = (V, E, l, I)$, where I_u once again represents the input to node u. The protocol is synchronous because it works in rounds numbered from 0 to T_π, where T_π is some finite, known bound on the running time. The channels are no longer UDL's but simply obey the property that any packet sent on a channel at the start of a round is delivered by the end of the round. Similarly, the node protocol at each node u is no longer modelled by a node automaton but instead by a 4-tuple (s_u^0, F_u, M_u, O_u), where s_u^0 is an *initial state*, F_u is a *state transition function*, M_u is a *message generation function*, and O_u is the *output function*. All these correspond to the standard notions for such protocols but we explain them briefly below.

The state s of the synchronous protocol consists of the state $s|u$ of each node in G. An *execution* of the synchronous protocol is generated as follows. At the start of the first round, each node u is placed in the initial state s_u^0 and all channels are empty. At the start of each round, a node sends a packet to each neighbor. The contents of

the packet are determined by the message generation function M_u that takes as input the state of u at the start of the round. At the end of each round, a node changes its state using the state transition function F_u that takes as input the messages received by u in the round as well as the state at the start of the round.

The output of π in any execution is determined by the value of $O_u(s|u)$ for every node u, where $s|u$ is the state of node u at the end of round T_π. Exactly as for non-interactive protocols, we define an output function O to be a function of the state s of π such that $O(s)$ produces a vector of local outputs that is identical to the outputs produced by each O_u when applied to $s|u$.

Exactly as for non-interactive protocols, we define the correctness of synchronous protocols using an I/O relation R. The synchronous protocol π is said to be correct if all runs of π end in a state s such that $(I, O(s)) \in R$.

The *time complexity* of a synchronous protocol π is the number of rounds, T_π. The space complexity of π is the worst case across all nodes u of the number of states of u.

Let π be a synchronous protocol with time complexity T_π and Input-Output relation R. We also introduce the notion of a *checker* for π. A synchronous protocol χ is said to be a checker for π if it satisfies the following property.

Suppose each node u in χ is given as input both the input of π (i.e., I) as well as a value O' that purports to be the output of π on input I. Informally, χ must detect (at some node) if the purported output O' of π could have been produced in some run of π on input I. Thus χ has a boolean output at every node. If any node outputs the value *false*, then it must be that O' could never have been produced by π; if all all nodes output the value *true*, then it must be that O' could have been produced by π.

Note that our definition only allows deterministic synchronous protocols whose output is a function of its input. Thus for such protocols π, π has a trivial checker that simply runs π again and compares the output $O_u(s)$ at each node with the purported output O'_u. The checker outputs *true* at node u iff $O_u(s) = O'_u$.

9.3 Results

Recall the definitions of stabilization time and space complexity for a non-interactive protocol $\mathcal{P}$ and the definitions of time and space complexity for a synchronous protocol

Method	Stab. Time	Space
Rollback	T_π	$T_\pi \cdot S_\pi$
Resynchronizer	$T_\pi + n$	S_π

Figure 9.1: Comparison of the complexity of our compilers

π. Recall that n is the number of nodes in the network.

Our solutions are in the form of compilers that can compile synchronous protocols into stabilizing versions that have the same I/O relation when run on an asynchronous network. The performance of our compilers is summarized in Figure 9.1

Our simplest compiler is the *Rollback* compiler that takes a synchronous protocol π as input and produces a stabilizing, non-interactive version of π. This is expressed as the following theorem:

Theorem 9.3.1 Rollback Compiler: *Consider any synchronous protocol π for augmented graph G with I/O relation R, time complexity T_π and space complexity S_π. Then there is a corresponding non-interactive protocol $\mathcal{P} = (\mathcal{N}, O)$ such that:*

- *$\mathcal{N}$ is a UIOA.*
- *The space complexity of $\mathcal{P}$ is $O(T_\pi \cdot S_\pi)$.*
- *The stabilization time of $\mathcal{P}$ with respect to R is $O(T_\pi)$.*

The main contribution of this chapter is a second compiler called a *Resynchronizer*. In its simplest form, the *Resynchronizer* takes a *deterministic* synchronous protocol π as input and produces a stabilizing non-interactive version of π.

Theorem 9.3.2 Resynchronizer Compiler: *Consider any synchronous protocol π for augmented graph G with with I/O relation R, time complexity T_π and space complexity S_π. Then there is a corresponding non-interactive protocol $\mathcal{P} = (\mathcal{N}, O)$ such that:*

- *$\mathcal{N}$ is a UIOA.*
- *The space complexity of $\mathcal{P}$ is $O(S_\pi)$.*
- *The stabilization time of $\mathcal{P}$ with respect to R is $O(T_\pi + n)$.*

Our original proof of the Resynchronizer Compiler Theorem was based on a complicated construction given in [AV91]. We conjecture that the simplified construction in Section 9.6 also provides a proof of the same theorem.

We will also see in Section 9.7.2 how to use the *Resynchronizer* to compile randomized, synchronous protocols (of course, this would entail generalizing our model of a synchronous protocol to allow coin-tosses at each node.)

Note that, when compared to the *Resynchronizer*, the *Rollback* construction removes the additive factor of n in the stabilization time but increases the space by a factor of T_π. Thus *Rollback* is useful only for "fast" protocols that have $T_\pi \ll n$. The two compilers can be used to efficiently stabilize several non-interactive tasks.

Some sample results obtained by applying the *Resynchronizer* are as follows. For the problems of computing a spanning tree and single source shortest paths we achieve $O(n)$ stabilization time and $O(\log n)$ space. This is comparable to the results achieved in Chapter 7 and the best previous results. For the problem of computing a minimal spanning tree [GHS83] we achieve $O(n)$ stabilization time (which is as good as the time of the best non-stabilizing synchronous protocol) using $O(\log n)$ space. For the problem of computing a maximum flow [Gol85, GT88] we achieve $O(n^2)$ stabilization time, which is as good as the time of the best non-stabilizing synchronous protocol.

The *Rollback* compiler gives good results when applied to symmetry breaking problems such as the problems of computing a Maximum Independent Set [AGLP89], $\Delta+1$ Coloring in sparse networks [GPS87], and Δ^2 Coloring in general networks [Lin87]. For instance, for $\Delta + 1$ Coloring in sparse networks we achieve $\log^* n$ for both measures. This is much better than any previous results.

9.4 Distributed Program Checking

A deterministic sequential algorithm can make itself stabilizing by periodically saving its output and running itself again; when it finishes it can check its output. For

sequential algorithms, this is ugly and unnatural – after all, we want the computer to move on and run other programs!

However, as we have said before, periodic checking is quite natural for distributed computing. Once we have accepted the inevitable periodic cost of stabilization we can ask: *why not run a checking process to check the algorithm periodically?*. If the check reveals a problem, we restart the main algorithm. In the previous chapters we have already seen how we can do this in some cases if each link subsystem independently does local checking. However, that method (which we can call independent checking of link subsystems) is limited to locally checkable protocols.

One approach to check a distributed program introduced by Katz and Perry [KP90] is to collect all information at a single "leader" node, thus reducing distributed checking to centralized checking. However, the space and time complexities of this method are quite large because of the bottleneck at the "leader" node.

In this chapter, we will introduce a form of distributed checking that is neither the independent local checking of the previous chapters nor the centralized checking introduced by Katz and Perry. In many cases, we can improve performance by doing distributed program checking.

Such distributed program checking clearly requires coordination of all the network nodes. For example, we need to ensure that a node not move to a new phase (whether it be checking or executing) before every other node in the network has completed this phase. The main difficulty is to implement this coordination in a stabilizing fashion. We will describe two such implementations – the *Rollback* protocol in Section 9.5 and the *Resynchronizer* in Section 9.6.

9.5 Rollback Compiler

There is a naive implementation of distributed checking that requires a large amount of storage and bandwidth and works only for deterministic protocols. In the naive method, every node keeps a log of every state transition it has taken to reach its current state. If each node constantly sends its current log to all neighbors, every node can check and correct every transition it has made in the past. Since the inputs are always correct, eventually all transitions are corrected. This method works only because it is possible to check transitions in a stabilizing fashion.

Clearly, for an arbitrary asynchronous protocol the logs can grow very large. However, if the asynchronous protocol is simulating an underlying synchronous protocol π then the size of the log can be reduced to the time complexity T_π of π. This idea is implemented in the *dynamic synchronizer* of [AS88]. However, in [AS88] the logs are only used to avoid unnecessary recomputation after an input change, and hence are not periodically checked. By adding the periodic checking of logs, we obtain a *Modified* Dynamic Synchronizer that we call *Rollback*.

The disadvantage of the *Rollback* method is that it blows up the space utilization and periodic bandwidth by a factor of T_π. This is not a problem for protocols with small time complexity – i.e., those for which $T_\pi \ll n$. Thus the naive method leads to efficient solutions for such problems as coloring and MIS. However, for protocols in which $T_\pi \geq n$, *Rollback* is a poor choice.

Notice that the *Rollback* compiler combines the use of logs with a simplified form of local checking and correction. The *Rollback* compiler does local checking by periodic sending of state (as opposed to doing a local snapshot). This is because the *Rollback* compiler is one-way checkable (see Chapter 8 for a definition of a one-way checkable protocol). Also, the *Rollback* compiler does local correction by simply correcting the local state of a node when periodic sending detects a problem (as opposed to doing a local reset). If local correction can be performed in this simple manner, we will call the protocol one-way correctable. Chapter 10 discusses one-way correctability in a little more detail.

9.6 Simplified Resynchronizer Compiler

In the previous section, we saw how *Rollback* did distributed checking by maintaining a log of its computation. Consider a deterministic protocol π. Clearly, we can avoid the need for a log if we could simply re-execute π. However, this constant re-execution requires coordination among the network nodes which must be implemented in a stabilizing fashion. In general, the *Resynchronizer* avoids a log by constantly re-executing a checker χ for π. We introduce the basic idea by assuming that π is deterministic and that π is its own checker. We return to separate checking later.

The *Resynchronizer* can be thought of as a stabilizing version of a *synchronizer* [Awe85]. Any synchronous protocol can be simulated asynchronously by using a pulse

number at each network node. Let us call a node *synchronized* if its pulse number differs by at most 1 from any of its neighbors. In ordinary synchronizers, every node is initially synchronized by setting the pulse numbers of all nodes to 0. Thereafter, a node increments its pulse number from p to $p+1$ only after all its neighbors have reached pulse p, thereby maintaining synchronization. If each node executes the corresponding step of the synchronous protocol at pulse p just before incrementing to $p+1$, then the asynchronous protocol has the same I/O relation as the underlying synchronous protocol.

Since a stabilizing synchronizer cannot rely on correct initialization, we introduce a *Termination_Detection* phase. This phase will two do things. First, this phase ensures that each node has finished executing the synchronous protocol. Second, once each node has finished executing the synchronous protocol, a reset (see Chapter 7) is initiated. Once all nodes get a signal (see Chapter 7) all nodes reset their pulse numbers to 0 and the cycle continues.

Thus the *Resynchronizer* can be considered to be an application of global correction to the simplest synchronizer protocol described in [Awe85]. Our original construction and proof [AV91] relied on a special-purpose reset protocol that was specially crafted to work with the synchronizer protocol. In this chapter, we will describe a simplified version of the construction that uses the general purpose network reset protocol of Chapter 7. While we do not have a complete proof of the simplified construction, we believe the construction in this chapter provides the basis for a simpler and more elegant solution. *In what follows, we only describe the simplified construction.*

When $pulse_u \in [0, T_\pi]$ we say that node u is in the *Execute* phase. In the *Execute* phase node u simply executes the normal synchronizer algorithm described earlier. It also implements the underlying synchronous protocol, starting from initialization at pulse 0 followed by writing its output at pulse T_π. Let D be an upper bound on the diameter of the network.[1] We assume that $D \leq n$.

We denote by $Max = T_\pi + D$ the maximum value of $pulse_u$. When $pulse_u \in [T_\pi + 1, Max]$ we say that node u is in the *Termination_Detection* phase. Suppose that all nodes are synchronized when some node exits the *Execute* phase. Then the *Termination_Detection* phase ensures that *every* node has had a chance to correct its

[1] Unfortunately our protocol needs an upper bound on the diameter of the network. This is a liability of the protocol.

output before the pulse number of some node wraps around to 0. If the nodes are synchronized by the start of this phase, a node need only wait D dummy pulses to make sure that all other nodes have reached the *Termination_Detection* phase.

If $pulse_u$ reaches Max, node u makes a reset request and waits to get a signal. This potentially destroys synchronization. However, we can rely on the properties of the reset protocol to deliver a signal at all nodes in a consistent way. When node u receives a signal it sets its pulse number to 0 and the cycle continues. Notice that all nodes constantly reexecute the underlying synchronous protocol.

The *Resynch* compiler must deal with arbitrary pulse numbers and arbitrary messages on links in the initial state. Second, it must cope with the fact that the pulse numbers are finite and hence have to wrap around. Recall that one of our motivations for studying stabilization was to make real network protocols more fault-tolerant. Any real counter implementation is bounded.

In our description and in the code below we only describe the *Resynchronizer* as a tool that can be used to compile a deterministic protocol by re-executing it. It is easy to extend these ideas slightly to add a separate checking phase to the *Resynchronizer*.

9.6.1 Resynchronizer Code

We described how to reduce the problem of stabilizing the output of a synchronous protocol π to the problem of building a stabilizing synchronizer that constantly re-executes pulse numbers in the execute region. Thus when presenting the code, for simplicity we ignore the details of executing π; instead we concentrate on the major task of synchronizing pulse numbers. To actually execute π, we need to supplement the code as follows:

- Additional state variables are added at every node u to keep track of the state of π. Also, we assume that every pulse message with pulse p carries the state of π (at the sending node) on pulse number p and $p-1$.
- Whenever $pulse_u$ reaches p and p is in the *Execute* phase, the synchronous protocol π is executed at node u. (Sometimes the code will cause the pulse number at node u to jump from say p to $p+2$. In that case, the synchronous protocol must be executed at pulse $p+1$ and pulse $p+2$.)

- Whenever $pulse_u$ reaches T_π, node u corrects its output to be the local output of π.

The protocol is formally presented below in Figures 9.2 and 9.3. Every node u keeps the variables described in Figure 9.2.

We assume that each node is a user of the reset protocol $\mathcal{R}|L$ described in Chapter 7. As in Chapter 7, we assume that $\mathcal{R}|L$ has an interface to make reset requests and accept reset signals as shown in Figure 7.2 in Chapter 7. Recall also that the reset subsystem also has an output action FREEM that tells the user (in this case the spanning tree protocol) when it can send a new user message. Of course, the similarity to the FREE action of a UDL is no coincidence. Just as the user of a UDL needs to keep a *free* variable to record whether the UDL is free, each node (exactly as in the spanning tree protocol of Chapter 8) will keep a variable *free* for each neighbor. In fact, the entire protocol bears a strong resemblance to the spanning tree protocol of Chapter 8 although it performs an entirely different function.

Every node u periodically sends its own pulse number to all neighbors. using a "Pulse" message. The Pulse message is encoded as a tuple $(Pulse, p)$. When a node v receives an Pulse message from u, v compares the pulse in the message (say p) to the previous pulse estimate stored from u ($pulse_v[u]$). During normal synchronizer operation the following local predicate always holds: $pulse_v[u] \leq p$ and $p \leq pulse_v + 1$. (Intuitively, the pulse estimates sent by u to v are non-decreasing and are never more than one higher than the current pulse number of v) If v realizes that the local predicate has been violated, v makes a reset request. If this is not the case, v stores the latest estimate it has received from u; v also updates its own estimate as the minimum of all neighbor estimates.

Once node u finds that $Pulse_u = Max$, it makes a reset request. Once node u gets a signal, node u resumes normal synchronizer operation.

We assume that the topology of the network is specified by an augmented graph G. Let $\mathcal{R}|L$ be the reset protocol described in Chapter 7. Let SS_u be the automaton shown in in Figure 9.3. Let $\mathcal{S}$ be the composition of the automata node SS_u for every node in G with $\mathcal{R}|L$.

Variables

- $pulse_u$: Highest pulse which was performed correctly until now. Its domain is restricted to lie in $\{0..Max\}$. Recall that $Max = T_\pi + D$, where D is an upper bound on the network diameter.
- $pulse_u[v]$: Node u's estimate of the current pulse number at node v. In normal synchronizer, the estimate is always a lower bound on the true value. Its domain is also restricted to lie in $\{0..Max\}$.
- $free_u[v]$: Boolean, which if true indicates that link to v is free to accept new messages.
- $requestbit_u[v]$: Boolean, which is set to true to remember that node u should make a reset request in the future.

Figure 9.2: Declarations of variables at node u used by Resynchronizer code.

9.6.2 Proof Sketch for the Simplified *Resynchronizer* Construction.

We outline an argument that explains why we believe the simplified construction is correct. We emphasize that this is an intuitive argument that needs further polishing.

Just as in the spanning tree protocol of Chapter 8, the normal operation of the synchronizer can be defined by two one-way predicates $S1(u,v)$ and $S2(u,v)$ shown in Figure 9.4. The first states that a node's pulse number is one higher than the smallest estimate it has from any neighbor). The next states that the pulse estimates sent by node u to neighbor v are strictly non-decreasing. However, any estimate sent by u can never be more than one higher than the current pulse number of v. If we compare the local predicates of Figure 8.7 with the local predicates of our spanning tree protocol (Figure 9.4) we notice a remarkable similarity between two different protocols.

If these two local predicates hold for all edges (u,v) we can show that all nodes are *synchronized* - i.e., the pulse number of each node differs by at most 1 from any of its neighbors. This is sufficient to ensure that the pulse numbers will eventually grow until some node reaches the maximum value.

Unlike the spanning tree protocol, however, the *Resynchronizer* protocol keeps making reset requests and there will never be final signal intervals in any execution

RECEIVEM$_{v,u}(Pulse, p)$ (*receive pulse message from neighbor v*)

If $(pulse_u[v] \leq p)$ and $(p \leq pulse_u + 1)$ then

$pulse_u[v] := p$

If $min\{pulse_u[v], v \in neighborSet_u\} + 1 \leq Max$ then

$pulse_u := min\{pulse_u[v], v \in neighborSet_u\} + 1$

Else $pulse_u := Max$

Else $requestbit_u = true$

SENDM$_{u,v}(Pulse, p)$ (*output action to send pulse to neighbor v*)

Preconditions: $p = pulse_u$ and $free_u[v] = true$

Effects: $free_u[v] = false$

REQUEST$_u$ (*output action to request a reset*)

Preconditions: $requestbit_u = true$ or $pulse_u = Max$

Effects: None

SIGNAL$_u$ (*input action to receive a reset signal*)

Effects:

$pulse_u := 0$

For all neighbors v of u do

$pulse_u[v] := 0$

$free_u[v] := false$

FREEM$_{u,v}$ (*receive free signal from link to neighbor v*)

$free_u[v] := true$

Figure 9.3: Resynchronizer Code for any node u

of the protocol. Thus we need to make heavy use of the causality property of reset behaviors. For example, we can show that within $O(n)$ time of any suffix of an execution, some node must make a reset request. This is proved by contradiction. If not, in linear time all signals must stop and in constant time after this all local predicates must hold. (The second part follows because the code makes a reset request if it detects a violation of $S2(u,v)$. Also, the receipt of the first pulse message at u will make $S1(u,*)$ hold.) Thus by normal synchronizer operation, the node with the lowest pulse number will increase its pulse number by 1 in constant time; thus in $O(\mathit{Max}) = O(n)$ time, some node will have reached the maximum pulse number and hence will make a reset request.

The overall argument consists of showing that there is some t-suffix suffix γ of an execution of the Resynchronizer protocol, where $t = O(n)$ and such that:

1. All nodes receive a signal event in linear time after the start of γ.
2. In linear time after all nodes reach pulse number 0, some node reaches pulse number Max and all nodes reach pulse number $T_\pi = \mathit{Max} - D$

As in the proof of the Global Correction Theorem, we can argue that in linear time, all reset requests that are caused by local predicate violations will disappear. Thus we choose γ such that reset requests in γ are only caused by nodes reaching the maximum pulse number Max.

The intuition behind the first part is that since some node makes a reset request within $O(n)$ time of γ, within $O(n)$ time in γ all nodes will receive a signal. Notice that when a node gets a signal, the node resets its pulse number to 0.

The intuition behind the second part is as follows. After all nodes have reached pulse number 0, we can again argue that some node makes a reset request in linear time. By our choice of γ, a reset request is only caused by some node, say u, reaching pulse number Max. But since u was previously at pulse number 0 in γ, u's pulse number must have grown from 0 to Max in γ. We then argue that all neighbors of u have grown from 0 to $\mathit{Max} - 1$ in γ, and finally that all nodes have grown from 0 to $\mathit{Max} - D$ in γ.

Finally, if all nodes grow from 0 to $\mathit{Max} - D$ in γ, we argue that by the end of γ all nodes have correct output. Intuitively, this is because each node begins executing the synchronous protocol at pulse 0 and corrects its output at pulse $T_\pi = \mathit{Max} - D$.

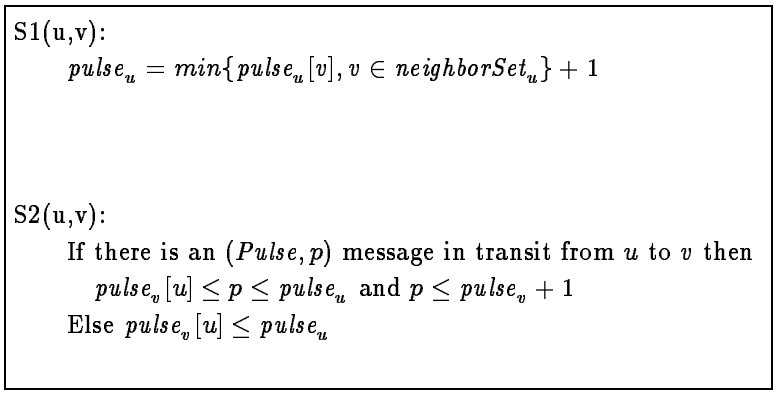

Figure 9.4: Normal Synchronizer: Local Predicates for edge (u, v).

9.7 Extensions

9.7.1 Better Synchronous Checkers for Deterministic Protocols

Suppose we limit ourselves to stabilizing protocols π that execute a distinct checking protocol χ even after reaching a legal state. Let us informally define the stabilization bandwidth of π to be the worst case message complexity per link of the checking protocol. Clearly the checking protocol must be executed at least once every T time units – where T is the stabilization time – even after the protocol has stabilized. Hence this is really a bandwidth cost. For example, the stabilization bandwidth of the protocol in [KP90] is the worst case message complexity per link to do a snapshot, which is $\Omega(m)$, where m is the number of links.

If we check a deterministic protocol π by re-executing π, we have to pay a high price (T_π) in stabilization bandwidth. Suppose instead, we can find a checker χ for π that has $T_\chi = 1$ – i.e., can check π in a single pulse. Then after the execution phase we can add a single check pulse number T_{cp}. When a node reaches pulse T_{cp} it stays at T_{cp} executing the checking protocol until it detects a problem; if it does detect a problem it drops back to pulse 0. By avoiding multiple pulses for checking, we remove the need for costly a termination detection and reset operation in the checking phase. The stabilization bandwidth drops to $O(S_\pi)$.

There are a number of tasks that we can check in a single pulse. These include the classical problems of shortest paths, topology update, leader election, computing a spanning tree, and computing a maximum flow. Because the first four tasks are commonly used in real networks, it is important to improve their stabilization bandwidth.

We do so by what by essentially introducing a new form of local checking which we will call *synchronous local checking.* The idea of doing local checking was already introduced in previous chapters. In the previous chapters, we described asynchronous protocols which continuously checked the state of every link subsystem. Note that state of of every link subsystem includes the state of the channels as well as that of the nodes. However, to build an efficient synchronous checker, all we have to do is to locally check a *synchronous* protocol that has *terminated.* Hence the checking procedures are much simpler. Note that for a synchronous protocol, the state of every link subsystem consists only of the state of two nodes.

Consider the problem of checking the shortest cost paths from a given source to every other node. The key is the ability to check, in a single pulse, the shortest cost distances from a given source to every other node. Let s denote the source and D_i denote the distance node i has recorded to s; let $B_{i,j}$ denote the cost of a link from Node i to Node j, and $N(i)$ the set of neighbors of Node i. Then in a checking pulse a) s checks that $D_s = 0$ b) Nodes other than s check that $D_i > 0$ and $D_i = Min_{j \in N(i)}(D_j + B_{i,j})$. It is quite simple to prove that if any of the distances are wrong some node will detect this after checking.

By adding an extra distance label to the other tasks, the other tasks can also be checked in one pulse, using the same trick. We can do this, for instance, in leader election by adding the distance to the leader, and in maximum flow by adding the distance to the source in the residual graph.

In general, given our application, the design of faster checkers for synchronous protocols becomes an interesting and practical problem. For instance, we can we check a minimum spanning tree in fewer pulses than it would take to compute the tree from scratch while using only small storage? There are a number of similar open problems that arise from our work.

9.7.2 Randomized Protocols

In all our models, both of asynchronous and synchronous protocols we have disallowed the possibility of nodes tossing coins. Thus our formal models preclude the description of *randomized protocols.* In a randomized protocol, nodes are given the ability to toss coins. The definitions of correctness now become probabilistic and it is much harder to give careful definitions of stabilization. However, we will assume that the reader has an intuitive understanding of randomized protocols to understand the following informal discussion.

First, for randomized protocols, we will assume that any supposedly random bits in the initial state of a node can be arbitrarily corrupted and hence non-random. However, subsequent coin-tosses will produce truly random bits.

Next, note that we need a separate checker to compile a randomized protocol. This is because re-executing the original protocol can lead to a different output, and cause the checker to detect an error when there was none. Saving the original random bits in the state does not help either as these bits could be corrupted (see previous assumption). Further, this checker must be *oblivious*: it must not depend on the correctness of the supposedly random bits currently in the state. It appears that a stabilizing algorithm that uses a randomized checker needs an infinite supply of random bits since it cannot rely on the old random bits at any stage.

A simple example of compiling a randomized protocol is furnished by the problem of electing a leader in an anonymous network - i.e., a network in which nodes do not have any unique IDs.[2] Clearly we need randomization to break symmetry. To construct a stabilizing protocol for this task, we demonstrate a synchronous protocol for execution together with an oblivious synchronous checker.

Recall that we use D to denote an upper bound on the diameter of the network. In the execution protocol, each node picks a random ID uniformly and independently in a space of $1..X$. In the next D pulses, a node considers itself as the leader if it finds out that its ID is the largest in the network. In the checking protocol, each node i that considers itself a leader picks a new random value t_i in the space $1..X$ and broadcasts t_i during the next D pulses. At the end of D pulses, a node detects an error if it has received either no values or it has received more than one value. While both checking

[2] If nodes have unique IDs we must assume that the IDs are protected; dropping this assumption makes the system more fault tolerant.

and execution can fail, by picking X to be a polynomial (of sufficiently high degree) in the number of nodes, we can ensure that a correct output will be produced in constant expected number of phases. Since each phase takes $O(D)$ the *expected* stabilization time is $O(D)$.

A more efficient protocol for this purpose (that works in time proportional to the actual diameter as opposed to a bound on the diameter) is given in [DIM91b]. However, our solution seems to be simpler. As in the case of deterministic protocols, checking randomized synchronous protocols seems an interesting research area.

9.8 Summary and open problems

The main result of this chapter the *Resynchronizer*, is a compiler that transforms any synchronous protocol into a stabilizing version for dynamic asynchronous networks. The transformation adds $O(n + D)$ overhead to the time complexity of the protocol, where D is a bound on the diameter of the final network. Clearly D and n can be much larger than the actual diameter of the final network. A natural open problem is to obtain a compiler whose time overhead only depends on the actual diameter of the final network.

When the results in this chapter were first presented [AV91], the *Resynchronizer* compiler used a much more complicated construction (which could be regarded as a special reset protocol optimized for the case of synchronizer operation). The transformation in [AV91] added only $O(D)$ overhead to the time complexity of the protocol, removing the factor of n which arises from the use of the reset protocol described in Chapter 7. However, for many real networks the only meaningful bound on the diameter of the network is the number of nodes. Thus this does not seem to be a meaningful distinction in practice. Also, the construction in this chapter is much simpler than the original. However, a careful proof of the simplified construction is needed.

Chapter 10

Conclusions and Open Questions

This thesis investigates the power and applicability of local checking and correction for the design of stabilizing network protocols. The thesis provides a rigorous theoretical foundation for these concepts. However, the emphasis is on using these concepts to devise novel fault-tolerant protocols for real networks.

We have confined the notion of "local" to link subsystems consisting of a pair of neighboring nodes and the two unidirectional links between them.[1] We have defined a protocol to be *locally checkable* for a good property L if two conditions hold. First, there is a local property for each link subsystem (formalized by a predicate of the link subsystem) such that property L holds if the local properties of all link subsystems hold. Second, each local property is *closed* - i.e, once the local property is true, it remains true.

We have also defined a protocol to be *locally correctable* to property L if the protocol is locally checkable for L and there is a way to locally correct each node with respect to a link subsystem (formalized by the existence of a *local reset function*) such that the global property L becomes true. The index contains pointers to the formal definitions of these concepts.

This chapter summarizes and evaluates the major contributions of the thesis. The chapter ends with a list of open questions.

[1] However, in the list of open questions we suggest that this notion of *locality* can be usefully generalized.

10.1 Contributions

The major contribution of this thesis is the construction of general techniques for designing stabilizing protocols. All our techniques revolve around the concepts of local checking and correction. We use the general techniques to design new stabilizing protocols (many of which are practical) and to understand existing stabilizing protocols. In the process of formalizing these techniques, we were also led to invent new definitions of stabilization, a simple modularity theorem, and a new Data Link model. We believe this modelling effort is useful in its own right. Thus we divide the contributions of this thesis into five categories:

- General techniques for constructing stabilizing protocols.
- New or improved stabilizing solutions to specific problems.
- The Modularity Theorem
- Modelling of self-stabilization.
- Better understanding of existing work in self-stabilization.

We describe these contributions in more detail in the following subsections.

10.1.1 General Techniques

The thesis contains four general techniques and a powerful heuristic. The four general methods are all organized around the theme of local checking and form a natural progression of ideas. The methods are:

- **Local Correction**: The Local Correction Theorem (Theorem 5.4.3) of Chapter 5 shows that every locally checkable and correctable protocol can be stabilized in time proportional to the height of the underlying partial order. This is achieved by adding actions to do *independent* checking and resetting of each link subsystem

  The Local Correction Theorem formalizes a useful design technique for building stabilizing protocols. First, we add some (hopefully small) state to the protocol

to make it locally checkable. Next, we look for a local reset function that can be used for local correction. In some cases, it is possible to construct a local reset function by combining several actions of the original protocol. In Chapter 7 we conjectured that this (i.e., the construction of a local reset function) is possible for many existing protocols that work in dynamic networks. The stabilizing reset protocol of Chapter 7 is an example of this design technique.

- **Tree Correction**: The Tree Correction Theorem (Theorem 6.5.1) of Chapter 5 shows that every locally checkable protocol that works on a tree topology can be stabilized in time proportional to the height of the tree. Thus we can dispense with the need for local correctability if the underlying topology is a tree. This is achieved by adding actions to do *independent* checking and resetting of each link subsystem in such a way that the correction of a link (u, v) does not invalidate the correctness of links above link (u, v) in the tree.

  This theorem also formalizes a useful design technique. First, we construct a spanning tree of the network using a stabilizing tree protocol. Next, we design a locally checkable protocol to solve the desired problem that works over a tree. Finally, we apply the transformation underlying the Tree Correction theorem. The stabilizing token passing protocol of Chapter 6 is an example on which this design technique could be applied.[2]

- **Global Correction**: The Global Correction Theorem (Theorem 8.1.2) of Chapter 8 shows that every locally checkable protocol can be stabilized in time proportional to the number of network nodes. That is we can dispense with the need for local correctability or the restriction to a tree topology if we are willing to incur a stabilization time that is proportional to the number of network nodes. This is achieved by adding actions to do *independent* checking of each link subsystem as well as actions to do a *coordinated global reset* of the entire network if a local violation is detected.

  This theorem also formalizes a useful design technique. We construct a locally checkable protocol and then apply Global Correction. However, since the stabilization time of this method is comparatively high, it pays to use the Local Correction Theorem whenever a locally checkable protocol can also shown to be

[2] However, in Chapter 6 this example is derived using Local Correction.

locally correctable. The stabilizing spanning tree protocol of Chapter 6 is an example of the Global Correction technique.

- **Stabilizing Compiler for Synchronous Protocols**: The Resynchronizer and Rollback Compiler ideas in Chapter 9 show that any synchronous protocol π with time complexity T_π can be converted to an asynchronous, stabilizing version of π, with either an additive cost of $O(n)$ in stabilization time or a multiplicative factor of T_π in storage. Thus for such protocols *we can dispense with even the need for local checkability.* This is achieved by applying global correction to a simple synchronizer protocol. While Chapter 9 sketches the main ideas for both compilers, the method still requires further development, especially to complete and simplify the proofs.

  The synchronous compilers also suggest a useful design technique. Suppose the correctness of a task can be specified by an Input/Output relation. Then we can construct a synchronous protocol for a given task and then apply one of the two compilers. This technique provides stabilizing solutions for many tasks for which locally checkable solutions are not known to exists. For instance, this can be used to produce efficient and stabilizing minimal spanning tree, min-cost flow and coloring protocols.

Besides the four general methods, we have a useful heuristic.

Heuristic of Removing Unexpected Packet Transitions: In many cases we can make a protocol weakly locally checkable for a good property L (i.e., L holds if the local properties of all link subsystems hold) by adding a small amount of state. To make the protocol locally checkable for L we also have to ensure that each local property is closed - i.e., if the local property is ever true, it remains true. This can often be done by the method of removing *unexpected packet transitions* described in Chapter 6. The basic idea is that when a node u receives a packet p from a neighbor v, u only accepts the packet if this transition could have occurred in some good state of the (u,v) subsystem. If the (u,v) subsystem is in a bad state but the (u,w) subsystem is not, the only way the (u,v) subsystem can affect the (u,w) subsystem is if v sends "unexpected" packets to u. Weeding out such unexpected packet transitions is often sufficient to ensure that each local predicate remains closed.

The heuristic of removing enexpected packet transitions is used throughout the thesis. It is used in the Token Passing Protocol of Chapter 6, the Reset Protocol of

Chapter 7, the Spanning Tree Protocol of Chapter 8, and the Resynchronizer Protocol of Chapter 9.

Comparison with Only Previously Known General Technique

The only previously known general method for stabilization is the elegant result of Katz and Perry [KP90]. How do our methods compare with the general method of [KP90]? Recall that in [KP90], checking is done by a single leader node periodically taking a snapshot of the entire network.

- *Message Congestion*: From a practical standpoint, the most important difference is that *all* our methods have considerably smaller stabilization bandwidth than that of [KP90]. Recall from Chapter 9 that stabilization bandwidth is the periodic overhead of checking that must be paid *even when the protocol is in a good state and behaving correctly.* Suppose the protocol is to stabilize in time T. Roughly, [KP90] has to pay the price of a snapshot of the entire network every T units of time. Now a snapshot requires at least $O(m)$ state, where m is the number of network links. Thus links *leading to the leader node* must carry $O(m)$ message bits every T units of time. Thus the worst case bandwidth per link is very high. By contrast, in our local correction, tree correction, and global correction methods, each link only carries a *constant* number of message bits every T units of time. Even the naive synchronous compilers of Chapter 9 have better stablization bandwidth than the method of [KP90] because the communication overhead of checking is *spread out among all the links* rather than being concentrated on links leading to the leader. In real networks, each link has a limited bandwidth and the worst case bandwidth requirement is an important measure of link congestion.

- *Speed*: If all links are UDL's, the method of [KP90] requires $O(n)$ time to stabilize, where n is the number of network nodes. The local correction method, tree correction method, and Rollback compiler can all provide much faster stabilization times.

- *Storage*: The method of [KP90] requires $O(m)$ storage at the leader to store the snapshot information, where m is the number of links. For fault-tolerance,

every node must be prepared to be the leader of the network and be able to store $O(m)$ information bits. By contrast, except for the Rollback compiler, the storage required by our methods is negligible.

- *Generality*: The method of [KP90] is clearly more general than our methods. Our methods require protocols to be either locally checkable or to have a correctness specification that can be expressed (see Chapter 9) in terms of an I/O relation. However, there is at least one case where the method of local checking and correction is applicable where the method of [KP90] is not. This is the stabilizing end-to-end protocol that we describe in [APV91b]. The problem here is that some unknown set of network links may have infinite delay. Thus the global snapshot of [KP90] may never terminate. However, it is sufficient ([APV91b]) to do local checking and correction on the so-called *viable* links that have bounded delay.

To summarize, our methods are less general but more efficient than the method of [KP90]. Despite the loss of generality, our methods can be used to stabilize many useful tasks, as summarized below.

10.1.2 New or Improved Stabilizing Solutions for Specific Problems

Our general techniques provide new or improved solutions for Mutual Exclusion, Network Resets, Computing Spanning Trees, Topology Update, Minimal Spanning Trees and other tasks. We have also applied local correction to an important theoretical problem – the problem of end-to-end message delivery in unreliable networks. In this problem links can fail *continuously* – the only guarantee is that there is no cut of permanently failed links that separate the sender and receiver. We have not described our solution to this problem in this thesis but more details can be found in [APV91b].

There are a number of other protocols to which we believe local checking and correction can be applied. These include the efficient resource allocation algorithm of Awerbuch and Saks ([AS90]), and the elegant virtual circuit protocol due to Spinelli ([Spi88b]). We hope to produce stabilizing versions of these protocols. We believe our general methods provide solutions to many important networking tasks.

We emphasize that *many of our solutions are practical and can be applied in real networks.* Messages required for local checking can easily be piggybacked on the "keep-alive" traffic sent between neighbors in real networks. Thus solutions based on the first three of our general methods (all of which are based on local checking) can be added to real networks without appreciable loss of efficiency. Solutions based on the synchronous compilers of Chapter 8 can be practical if either the time complexity of the underlying synchronous protocol is low (for the Rollback compiler) or if the underlying synchronous protocol has an efficient synchronous checker (for the Resynchronizer compiler). A disadvantage of the synchronizer methodology is that the method tends to slow down the network to the speed of the slowest link. Thus the synchronous compiler methodology is best suited for use in networks where all links have roughly the same speed.

Among the most useful practical protocols described in this thesis are the stabilizing reset protocol of Chapter 7 (which was briefly tested in a trial implementation on the AUTONET) and the spanning tree and topology update protocols of Chapter 8. The topology update protocol also illustrates another general paradigm that may be useful in practice. We start with a simple protocol P that uses unbounded sequence numbers and use global correction to convert P into a stabilizing version P' that uses bounded sequence numbers. In the absence of catastrophic errors, P' is as efficient (except for the small overhead of local checking) and simple as P.

10.1.3 Modularity Theorem

The Modularity Theorem (Theorem 3.5.7) is simple but extremely useful. It helps us to prove facts about the stabilization of a big system by proving facts about the stabilization of each of its parts, as long as each part is suffix-closed. The modularity theorem gives us a formal basis for a building-block approach. For example, we can start with a stabilizing implementation of a UDL; use this to build a stabilizing reset protocol as shown in Chapter 7; and then use the reset protocol as a building block to construct a spanning tree protocol (Chapter 8) and a compiler for synchronous protocols (Chapter 9). As we have argued at the end of Chapter 4, the requirement that each of the parts be suffix-closed is not very restrictive. Essentially, this is because our methods tend to produce CIOAs (i.e., automata in which every reachable state is a start state) that are suffix-closed.

As we build up a complex stabilizing protocol in several layers, the stabilization time of the system can be calculated by applying the Modularity Theorem and the Transitivity Lemma for behaviors (Lemma 3.2.6). For example, suppose the overall system is described by an automaton P. Suppose also that P is identical to the stabilizing reset protocol $\mathcal{R}^+$ of Chapter 7 *except* that every UDL $C_{u,v}$ in $\mathcal{R}^+$ is replaced by a stabilizing implementation of a UDL called $C'_{u,v}$. Suppose that $C'_{u,v}$ stabilizes to the behaviors of $C_{u,v}$ in time t_1. Then by applying the Modularity Theorem (which is possible because all the constituent automata in $\mathcal{R}^+$ are UIOA) we conclude that P stabilizes to the behaviors $\mathcal{R}^+$ in time t_1. But we know from Chapter 7, that $\mathcal{R}^+$ stabilizes to the behaviors of $\mathcal{R}|L$ in some constant time, say t_2. (Recall that $\mathcal{R}|L$ is a "correctly working" reset protocol in which all local predicates are true in the initial state.) Thus we conclude from the Transitivity Lemma that P stabilizes to $\mathcal{R}|L$ in time $t_1 + t_2$. Notice that the stabilization times of each "layer" add up due to the Transitivity Lemma. Proceeding similarly, we can calculate the stabilization time of a version of the stabilizing tree protocol of Chapter 8 in which the reset protocol $\mathcal{R}|L$ is replaced by P.[3]

Our theorem can be viewed as a generalization of previous modularity results [DIM90]. Previous results only applied to the case when a lower layer protocol computed values of a shared variable that were read by a higher layer protocol. By contrast, our theorem applies to more dynamic interaction between the various components of a system.

10.1.4 Modelling

The models and proof techniques we use in this thesis are based on the large body of existing work that has been done in the area of protocol verification. Our model of computation is based on the timed I/O automaton [MMT91] which is in turn based on the I/O automaton of [LT89]. We have even described the shared memory network model in Chapter 4 as an I/O automaton that meets certain restrictions.

We also use standard proof techniques. We use mapping techniques (Refinement Mapping Theorem, Theorem 3.4.3) to show that one automaton has the same behaviors as another automaton. We prove that a local predicate is closed using the standard

[3]This time, however, we can apply the Modularity Theorem because $\mathcal{R}|L$ is a CIOA and the other nodes implementing the spanning tree protocol are UIOA.

inductive techniques used to prove program invariants. Perhaps the only unusual technique is the Execution Convergence Theorem (Theorem 3.4.5) which is a key tool for proving stabilization results. To apply this theorem we have to prove a stability condition and a "liveness" condition. We prove the stability condition using standard inductive arguments. We prove the "liveness" condition by proving time bounds on the occurence of events. We have proved time bounds in a fairly ad hoc, operational way. However, there is no reason why the time bounds could not be established by more rigorous inductive arguments (as in [LA90]). Thus while the Execution Convergence Theorem may appear slightly unusual, it is really a combination of existing verification techniques.

We believe this continuity (with the body of existing work in verification) is an advantage of our work. By contrast, previous papers on stabilization have sometimes tended to invent new models and proof techniques. Despite our strong linkage with existing work, we do have two interesting modelling contributions in this thesis. These are the use of a behavior specification for stabilization and the use of unit storage Data Links.

Behavior Specification

The definitions of stabilization in terms of external behaviors are different from previous definitions that are in terms of the states and executions of the underlying automaton. The external behavior definition allow us to define that automaton A stabilizes to another automaton B even though A and B have different state sets. This is most useful when A is a low level model (e.g., an implementation) and B is a high level model (e.g., a specification). This can be done using a definition of stabilization in terms of executions if we are prepared to introduce an abstraction function into the definitions as in Lamport's work [Lam83]. However, we prefer the behavior definitions as they seem to be more natural; we prefer to use the equivalent of an abstraction function to *prove* behavior stabilization results using the Refinement Mapping Theorem.

Unit Storage Data Links

In a stabilizing setting it is necessary to define Data Links that have bounded storage. First, as shown in [DIM91a], almost any non-trivial network task is impossible in

a stabilizing setting in which the links have unbounded storage and the nodes are restricted to be finite state machines. Second, bounded storage models correspond to physical reality.

Thus we use the standard asynchronous message passing model of a computer network except that each link is what we call a Unit Storage Data Link (UDL) that can store at most one packet. We have chosen unit storage links (UDLs) because they are practical (see Section 5.2) and they can be modelled elegantly. We have also defined a stabilizing interface to a UDL. This is done by having the link periodically deliver a free signal (to avoid deadlock) and by having the sender keep a variable that indicates whether the link is free. We hope the UDL model will be used by others. The UDL model can easily be generalized to Bounded Storage Data Links by changing the free signal to report the number of packets currently stored in the link.

Earlier papers in stabilization (e.g., [DIM90]) seem to have used shared memory models for communication in order to avoid the problems caused by unbounded storage links. It does seem very likely (see Open Problems) that protocols that work in the low atomicity shared memory model of [DIM90] can be transformed to work correctly in our network model. However, we believe that our protocol descriptions are more accessible to designers of "real" network protocols because most "real protocol" specifications assume the use of message passing primitives.

10.1.5 Understanding Existing Work

The concept of local checking helps in crisply understanding many existing stabilizing protocols. Chapter 4 shows that some existing work in the shared memory model can be understood crisply in terms of local checking and correction. We believe that many existing stabilizing protocols can be understood using three general ideas – one way checking and correction, counter flushing, and timer flushing.

One Way Checking and Correction

In Chapter 8, we said that a protocol P was one-way checkable if it was locally checkable using what we called one-way predicates. Intuitively, a one-way predicate is a predicate that involves the state of a node u, the state of any neighbor v of u, and the state of the link between u and v. Unlike a general local predicate, a one-way

predicate only depends on the state of one of the links between the two nodes. We saw that one-way checkable protocols could be checked without the need for a local snapshot – it is sufficient for each node to periodically send its state to its neighbors. The protocols in [Per83] and [Per85] do checking in this way.

In some cases, the protocol is also one-way correctable – i.e., we can apply local correction to a one-way checkable protocol without the need for a local reset of the link subsystem. For example, when v receives a copy of u's state and detects a violation of the one-way predicate from u to v, it may be possible for v to apply a local reset function to its own state so as to make the one-way predicate true. Of course, this could falsify other "adjacent" one-way predicates. For the protocol to be one-way correctable, the dependency relation among the one-way predicates must be acyclic, as in the definition of local correctability. For example, the *Rollback* protocol of Chapter 9 is one-way correctable. We speculate that the stabilizing topology update protocol of Spinelli-Gallager [SG89] is also one-way correctable

Counter Flushing

Suppose a sender wishes to periodically send a REQUEST packet to a set of network nodes. The responders must each send back a RESPONSE packet before the sender sends its next request. In Chapter 5, for example, we implemented local snapshots and resets using such a request-response protocol initiated by the leader of each link subsystem. In order to properly match responses to requests, the sender numbers each request with a counter. Let m be the number of packets that can be in transit between the sender and responder and let n be the number of responders. Then the sender uses a counter that has $Max = m + n + 1$ distinct values. For example, in Chapter 5 we used a counter in the range 0...3 because there can be at most two packets in transit in a link subsystem and there is only one responder.

Responders only accept REQUEST packets with a number different from the last REQUEST accepted. After accepting a REQUEST the responder sends back a RESPONSE with the same number as the REQUEST. The sender retransmits the current REQUEST till it receives each matching RESPONSE with the same number. After all matching RESPONSE packets arrive, the sender increments its counter.

The size of *Max* ensures that within *Max* increments of the counter, the sender will reach what we call a "fresh" counter value – i.e. a counter value that is not

currently stored in either the links or the responders. We call the method counter flushing because the request-response protocol must guarantee the following "flushing" property. *Suppose the sender sends a request numbered c, where c is a fresh value. Then after all matching responses to this request arrive, there must no counter values other than c that are stored in the links or at the responders.* In other words, the sending of a freshly numbered request and the receipt of all matching responses, should "flush" the links and responders of "old" counter values.

In Chapter 5, the flushing condition is guaranteed because the sender and receiver are connected by two FIFO links in either direction. Similar forms of counter flushing can be used to implement Data Links ([AB89]) and token passing [DIM91a]) in link subsystems with bounded storage. Counter flushing is, however, not limited to link subsystems. The first example in [Dij74] can be simply understood as counter flushing in a unidirectional ring (see Appendix E for more details). Katz and Perry [KP90] extend the use of counter flushing to arbitrary networks in an ingenious way. Our stabilizing end-to-end protocol ([APV91b]) is obtained by first applying local correction to the Slide protocol [AGR92] and then applying a variant of counter flushing to the Majority protocol of [AGR92].

Timer Flushing

The main idea is to bound the lifetime of "old" state information in the network. This is done by using node clocks that run at approximately the same rate and by enforcing a maximum packet lifetime over every link. State information that is not periodically refreshed is "timed out" by the nodes. In Perlman's [Per83] topology update protocol, timer flushing is used to get rid of erroneous updates that are numbered with the maximum possible sequence number. In Perlman's [Per85] spanning tree protocol, timer flushing is used to get rid of "ghost" roots (see Chapter 8 for details.) Spinelli [Spi88b] uses timer flushing to build stabilizing Data Link and virtual circuit protocols.

10.2 Open Questions and Further Problems

The following is a list of further problems. They are arranged under four categories: modelling, increased understanding of local checking and correction, new algorithms, and new directions.

10.2.1 Modelling

The following is a list of what we believe are the further problems that are motivated by our work in modelling stabilizing protocols.

- **Extend model and theorems of Chapter 3 to allow the use of randomization and lower bounds on the time between actions**: The model of Chapter 3 makes no provision for randomization or for lower bounds on the time between actions. While the model in Chapter 3 is sufficient for modelling asynchronous protocols it cannot be used to model many interesting randomized and timing based algorithms. This seems to be an important problem. The hard part is extending the model to obtain corresponding versions of the modularity and transitivity results.

- **Find a Proof technique for Generalized Stabilization Definitions**: The definitions of a stabilizing reset protocol and other protocols can be made more elegant if we modify the stabilization definitions as follows. We only require that the t-suffix of a behavior (execution) be a t-suffix of a behavior (execution) of the target set. One problem with this modified definition is that we know of no good proof technique to prove that the behaviors (executions) of an automaton are *suffixes* of a specified set of behaviors (executions). While a general proof technique for this purpose may be infeasible, it may be possible to find a proof technique that works for a large number of cases.

- **Discover stronger variants of the modularity theorem.** The modularity theorem allows us to infer the stabilization properties of a large system from the stabilization properties of its pieces. However, we required that each piece be suffix-closed. It is natural to look for weaker or alternate conditions. This may be somewhat technical as the suffix-closed requirement does not appear to be very restrictive.

- **Find a General Definition of Stabilization Bandwidth**: In Chapter 9, we intuitively described an important measure for a stabilizing protocol – the amount of periodic bandwidth the protocol consumes. The definition we gave in Chapter 9 only applies to protocols that have a certain structure. A precise, general definition would be very useful.