LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TM-341

A MODULAR PROOF OF CORRECTNESS FOR A NETWORK SYNCHRONIZER

A. Fekete
N. Lynch
L. Shira

September 1987

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

A Modular Proof of Correctness for a Network Synchronizer

A. Fekete

N. Lynch

L. Shrira

Laboratory for Computer Science,

Massachusetts Institute of Technology

Abstract: In this paper we offer a formal, rigorous proof of the correctness of Awerbuch's algorithm for network synchronization. We specify both the algorithm and the correctness condition using the I/O automaton model, which has previously been used to describe and verify algorithms for concurrency control and resource allocation. We show that the model is also a powerful tool for reasoning about distributed graph algorithms. Our proof of correctness follows closely the intuitive arguments made by the designer of the algorithm by exploiting the model's natural support for such important design techniques as stepwise refinement and modularity. In particular, since the algorithm uses simpler algorithms for synchronization within and between 'clusters' of nodes, our proof can import as lemmas the correctness of these simpler algorithms.

Keywords: verification, modularity, network protocols, synchronization.

September 1987

A Modular Proof of Correctness for a Network Synchronizer[1]

1 Overview

1.1 Verification methods and models

As computer science has matured as a discipline, its activity has broadened from writing programs to include reasoning about those programs: proving their correctness and efficiency, and proving bounds on the performance of any program that accomplishes the same task. Recently distributed computing has begun to broaden in this way (albeit a decade or two later than the part of computer science concerned with sequential, uniprocessor algorithms). There are several reasons why particular care is necessary to prove the correctness of algorithms when the algorithms are distributed. First, human thought tends to operate sequentially, that is, we usually focus our attention on one aspect of a problem at a time. This leaves us vulnerable when examining distributed protocols, where activity is happening concurrently in several places in a system, since we can easily fail to consider the subtle interactions between different activities. For example, unexpected race conditions can lead to unexpected (and wrong) behavior. Second, distributed protocols are required to cope with a certain level of nondeterminism in the system, such as variable message delays, variable processor speeds, or even processor failures, and humans find it hard to deal with the exploding number of different possibilities.

For these reasons one is not surprised that there have been several cases where algorithms were published (and implemented) that seemed reasonable, but were later found to be in-

[1]This paper forms part of the first author's Ph.D. thesis "Topics in Distributed Algorithms", Department of Mathematics, Harvard University, August 1987. A preliminary version of the material of this paper has appeared in the Proceedings of the 2nd International Workshop on Distributed Algorithms (July 1987). The work of the second author was supported in part by the Office of Naval Research under Contract N00014-85-K-0168, by the Office of Army Research under contract DAAG29-84-K-0058, by the National Science Foundation under Grants MCS-8306854, DCR-83-02391, and CCR-8611442, and by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125. The work of the third author was supported by an H.T.I. fellowship

correct. A famous example is the ARPAnet routing algorithm. We believe that rigorously proving the correctness of distributed algorithms is an important task, especially for algorithms that are going to be used as building blocks of other protocols. For example, when a distributed leader election protocol is used to choose a primary copy for a replicated relation in a distributed database, any uncertainty about the behavior of the leader election will propagate to undermine confidence in the correctness of the entire database management system.

Despite the reasons presented above, most work in distributed algorithms contains only informal correctness arguments and still omits rigorous proofs of correctness for the algorithms described. The claim is often heard that the formal techniques do not support intuition and the proofs are too complex. Obviously, the complexity of the verification is related to the conceptual complexity of the algorithm but it may also be heavily influenced by the choice of the specific verification procedure.

Good tools for distributed systems analysis have been sought by many researchers for a long time. Temporal logic (e.g. [MP], [HO]) and Floyd-Hoare-style methods (e.g. [OG]) are among the best known and indeed have been used successfully to verify a number of distributed algorithms. While the proofs using these methods do indeed demonstrate correctness of the algorithms, they often do not help the reader to understand why the algorithms are correct. The reader can be lost in the details of the step by step proof and lose the intuition and the global picture.

Partially, the problem stems from the fact that the reader faces the full gap between the low level implementation and the high level specification of the problem. The designer of the algorithm, however, when conceiving the algorithm or explaining it, often first argues in terms of high level activities that comprise the solution, and considers interaction between those. At subsequent design steps those activities are 'implemented' by refining them in turn. Only at the final step are activities of each node in the system fully specified. The method allows each refinement to remain manageably simple. To keep the designer's intuition, ideally, the verification procedure should follow closely the design process. That is, the proof should follow the refinements. The verification procedure then would be structured so that the proof of each refinement could be simple enough and the processes of design and verification would be brought together. To support the stepwise refinement described above, the verification method has to be hierarchical.

Another vital feature of verification procedures is exposed when the designer of the algorithm wishes to change an implementation of some activity, for example for optimization reasons. This obviously results in a new algorithm. Often though, the redesign of one activity does not affect others. In such cases, the verification method should be able to guarantee that only the changed part needs to be proved correct anew. That is, the verification method should be modular or compositional. Compositionality in proofs would also naturally support the fundamental 'off the shelf building block' technique in algorithm design as it allows the use of the correctness proof of the 'building block' in the proof of the algorithm without the need to reexamine it. But we must be particularly careful when considering the intuitive notion of modularity as referred to by algorithm designers. It is too often discussed informally in terms of several pieces needed to solve 'subproblems' although the sense of 'subproblem' is not precise. It is not obvious that the pieces fit together in any precise sense, especially when concurrency is considered. And as the algorithms that one tries to build become more and more complex, the lack of formal notion of modularity becomes more and more of a problem.

The commonly known verification methods do not seem to support both hierarchical and modular reasoning in natural ways. Thus the invariant assertion method allows hierarchical stepwise reasoning, but offers poor support for modularity when distributed systems are concerned. The proofs in temporal logic on the other hand, are composable but leave a large gap between the implementation and the specification.

In this paper we will prove the correctness of a network algorithm using the *I/O automaton* model. The model was introduced by Lynch, Merritt and Tuttle in [LM] and [LT], and it naturally supports both hierarchical and modular reasoning. From our experience with this model, we feel that it enables one to provide rigorous proofs of correctness that follow closely the informal arguments used by the designers of distributed algorithms to explain their work. We describe specifications, intermediate refinements and algorithm as I/O automata, and then show that one 'implements' another. Also, the model includes a natural notion of composition of two automata, that corresponds to the combined use of two algorithms, and its formal semantics are compositional, in that the behavior of the composition can be deduced from the behavior of all the component automata.

An example of hierarchical reasoning in the model can be found in [LT] where it was used to verify correctness of a distributed resource arbiter. The modularity property of the model was exploited in [Wl] to deduce correctness of an n-processor mutual exclusion algorithm, from the correctness of an arbitrary 2-process mutual exclusion algorithm, which is used as a subroutine within the main algorithm. The model has also been successfully applied to describe and verify a number of algorithms for concurrency control, recovery and replication management in nested transaction systems, for example [LM],[FLMW],[GL],[HLMW]. In these, the model's features are used to capture formally some intuitions of system designers, such as 'the correctness of replication management only needs to proved in a serial system, as the correctness of concurrency control for the replicas will then ensure that the replication algorithm is correct in a concurrent system'.

In this paper we demonstrate the ease with which the model allows one to prove the correctness of a network algorithm that uses a superposition of two different algorithms operating concurrently to accomplish almost independent subgoals, using claims that express formally the correctness of the subalgorithms.

1.2 Our proof

The algorithm whose correctness we prove in this paper is a distributed protocol for network synchronization. In designing algorithms to solve problems in a distributed computing environment, it is important to understand the assumptions being made about the processors and the network connecting them. If fewer assumptions are made, it is more likely that they will be satisfied by the hardware available, but it is harder to find algorithms that work correctly whenever the assumptions are satisfied. For example, most networks do not offer reliable bounds on the time a message takes to arrive, so it is important to find algorithms that work correctly in an *asynchronous* system, but it is very much easier to design algorithms if the network is *synchronous*. Awerbuch ([Aw]) proposed the use of a *synchronizer* that would enable one to convert any synchronous graph algorithm into an algorithm that performs correctly in an asynchronous (but failure-free) network. Using a synchronizer in this way has proved a successful methodology for solving asynchronous problems in efficient ways ([Aw2]).

In [Aw], a synchronizer (called γ in that paper) is constructed for a network whose topology is any fixed connected graph provided with a spanning forest subgraph, and a distributed technique is given for finding a spanning forest subgraph for which the resulting algorithm has low time and message complexity. The synchronization algorithm given is, however, asserted to be correct for any spanning forest subgraph. The algorithm is derived as a superposition of a simple synchronizer (called β) executing within each 'cluster' (a connected component of the spanning forest subgraph), and another simple synchronizer (called α) that synchronizes between the clusters. This description helps to explain the detailed algorithm, but no formal proof of correctness is offered in [Aw]. We provide a formal account of an algorithm closely based on Awerbuch's, and rigorously prove results about its correctness. The proof of correctness is modular and hierarchical. It closely follows the outline of the informal arguments of [Aw], by building on claims that express formally the correctness of algorithms α and β. Since these results have also not been formally proved before, we include such proofs for the sake of completeness.

Our account of the synchronizer is given as follows. First we provide a top level specification for any network synchronizer by giving a single I/O automaton S that uses global information about the system. Then we present the γ algorithm itself, as a system DistSysS of I/O automata, including one for each node of the graph with access only to local information and communicating only along the edges of the graph. As this algorithm is a superposition of two algorithms α and β, following Awerbuch's informal reasoning we divide each node-automaton into two automata, one containing the state and operations contributing to intercluster synchronization and the other containing the state and operations contributing to the intracluster synchronization. The two components do not interact at all, except when the node is the root ('leader') of its cluster.

In the language of our model, to verify the correctness of the algorithm we need to prove that the system DistSysS of I/O automata implements the specification automaton S. We proceed in the proof by refining the global specification according to Awerbuch's intuitive construction and defining for each refinement the corresponding correctness claim that needs to be proved, until the level of node algorithms is reached. We start with the global specification S (see Fig. 1) and refine it following the construction in [Aw] by a system SysS that consists of one automaton SL for each cluster, specifying the intracluster synchronization behavior, and also a single coordinator automaton CS that specifies intercluster synchronization (see Fig. 2). The correctness claim for this refinement is that all executions of the composed system SysS are acceptable behaviors of the global specification S.

In the above refinement, automaton SL provides a specification for the intracluster synchronization. According to [Aw] the intracluster synchronization is implemented by algorithm β. Thus, we further refine the intermediate specification SL by the distributed specification SysSL (see Fig. 3), that models the synchronizer β (a simple synchronizer using communication over a tree). The specification includes a separate node automata NDSL for each node in a cluster and a special automaton LESL for the leader, as well as an automaton LISL to represent each link. The correctness claim for this refinement is in fact established by the correctness proof for the algorithm β. If it were already carried out in our model, we could use it here as is.

Next, we consider the specification for the global intercluster synchronization coordinator CS. In [Aw] it is implemented by a distributed algorithm α, in which each cluster is a participant. Thus we refine the global coordinator specification CS with a distributed one SysCS (see Fig. 4), where clusters are modeled by automata CLCS that interact according to algorithm α (a simple synchronizer, using all the edges of the graph). Thus, the correctness claim of this refinement is established by the correctness proof of algorithm α. Here again the proof could be imported if it were available in the model.

Finally we consider the behavior of a cluster participating in α, which is specified by automaton CLCS. Following [Aw] we refine it by a distributed specification SysCLCS that specifies for each node in a cluster its behavior contributing to the cluster's part in algorithm α. This is done by giving a node automaton NDCS for each non-leader node in a cluster and a leader automaton LECS for the leader node, as well as automata LICS for the links (see Fig. 5). The correctness claim for this refinement then requires a proof that the the composed system SysCLCS implements the cluster specification CLCS. This is the last claim for the correctness proof of the network synchronizer. It is due to the support for modularity and

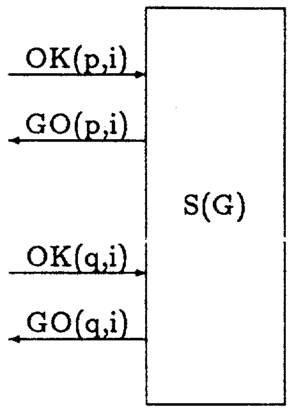

Figure 1: S(G)

hierarchical reasoning provided by the model of [LT], that the results described are sufficient to establish that the detailed node level specification DistSysS correctly implements the high level specification S.

The above discussion has dealt with the safety properties of the algorithm. We also give proofs of the liveness and complexity analysis of the algorithm, by reasoning directly about executions of the detailed system.

This paper shows how the properties of the I/O automaton model enable us to capture formally some of the important intuitions used in designing algorithms. We believe that with this model, it will not be difficult to prove the correctness of other algorithms whose design was guided by these principles of stepwise refinement and modularity. We also hope that the insights into the precise nature of modularity that are gained from this formalization will be useful to the algorithm designers themselves.

2 I/O Automata

The following is a brief introduction to a model that is proving useful for describing and reasoning about distributed systems. The model is developed at length, with extensions to express fairness properties, in [LT], where proofs can be found of many of the claims made

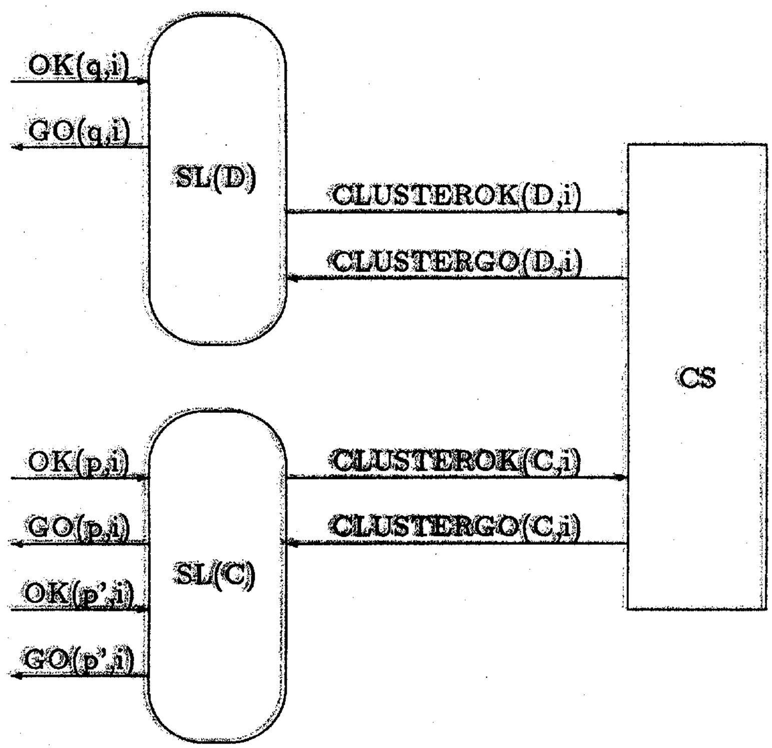

Figure 2: SysS(G)

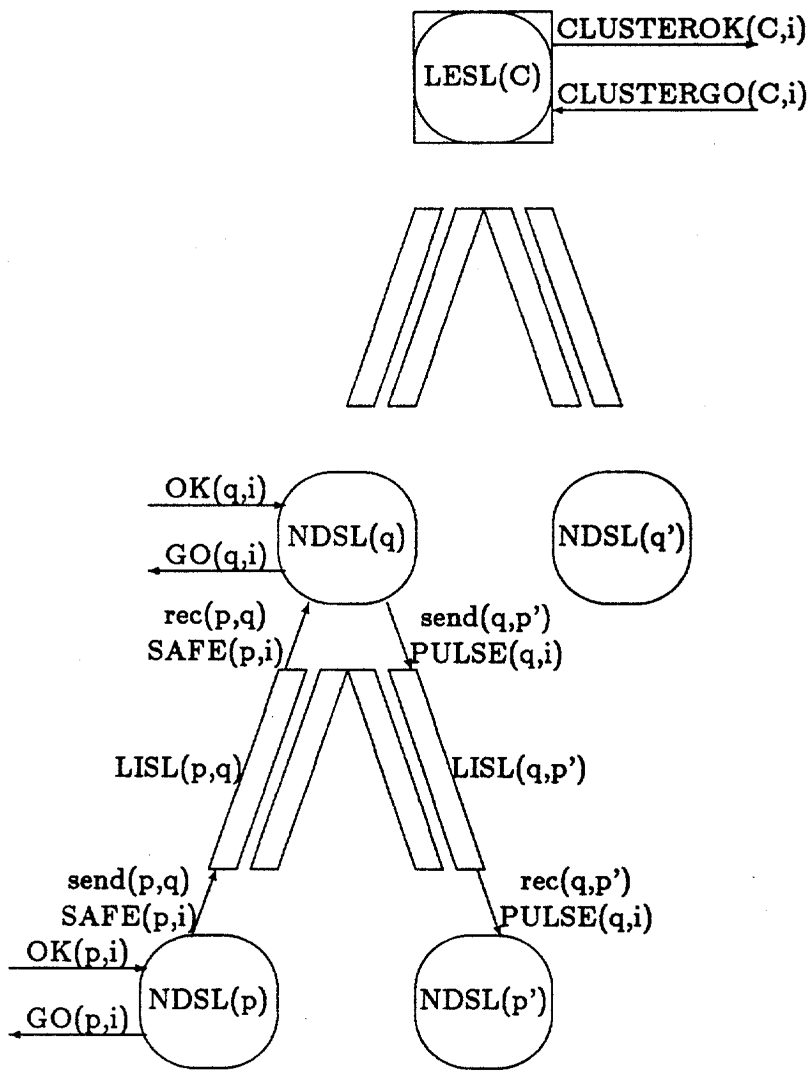

Figure 3: SysSL(C)

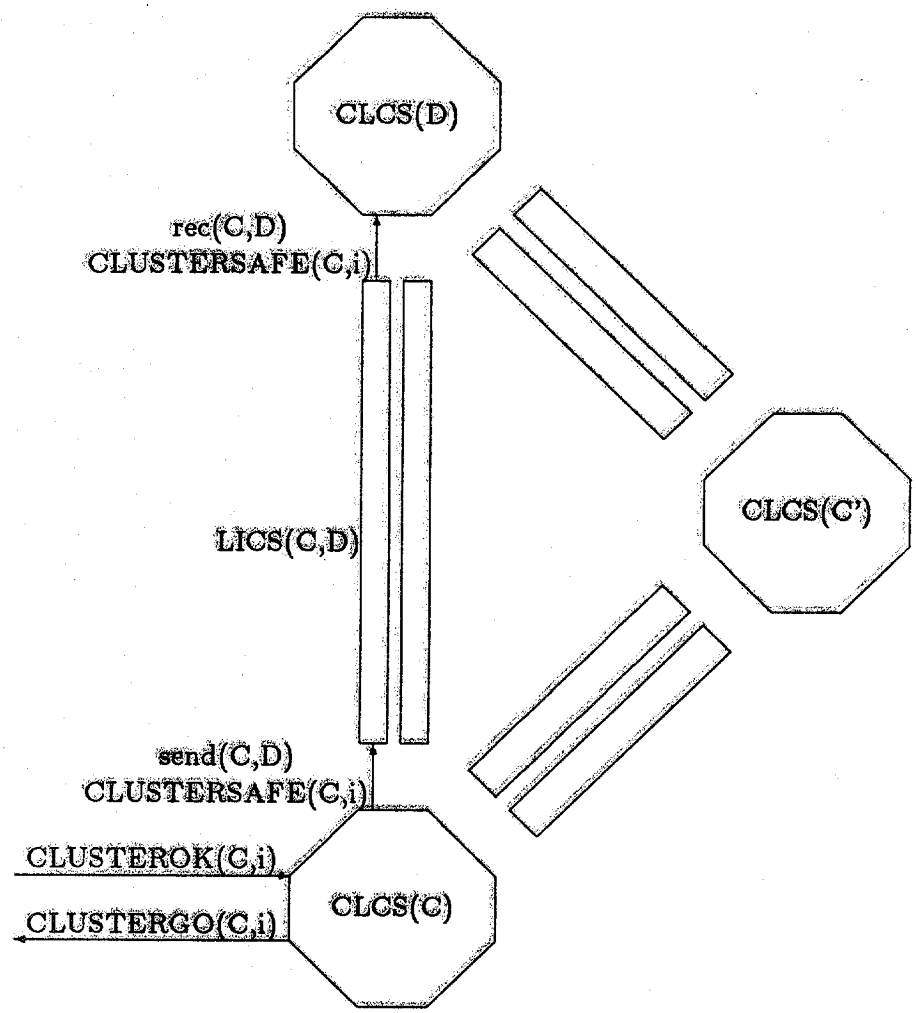

Figure 4: SysCS

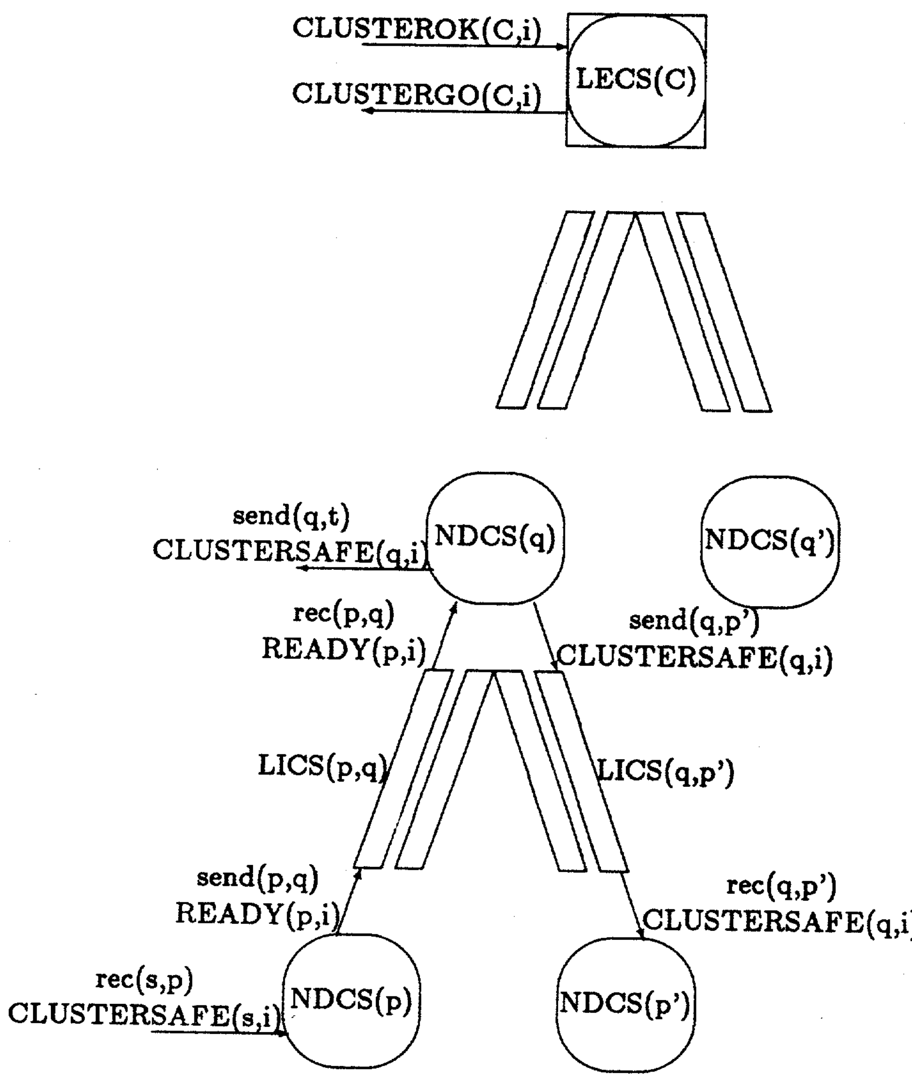

Figure 5: SysCLCS(C)

here.

All components in our system will be modeled by *I/O automata.* An I/O automaton $\mathcal{A}$ has a set of *states*, some of which are designated as *initial states.* It has *operations*, each classified as either an *input operation* or an *output operation*, or an *internal operation.* Finally, it has a transition relation, which is a set of triples of the form (s',π,s), where s' and s are states, and π is an operation. This triple means that in state s', the automaton can atomically do operation π and change to state s. An element of the transition relation is called a *step* of the automaton. The output operations are intended to model the actions that are triggered by the automaton itself, while the input operations model the actions that are triggered by the environment of the automaton. Internal operations are used to model communication within the automaton (when we form an automaton from components, this will include communication between pieces of the automaton). We will always give the transition relation of an automaton by giving pre- and postconditions for each operation π. We give the preconditions as predicates depending on s', and the postconditions as predicates depending possibly on both s' and s. These are to be understood as saying that (s',π,s) is in the transition relationship exactly when the preconditions are true of state s' and the postconditions are true of s' and s.

Given a state s' and an operation π, we say that π is *enabled* in s' if there is a state s for which (s',π,s) is a step. We require the following condition.

Input Condition: Each input operation π is enabled in each state s'.

This condition says that an I/O automaton must be prepared to receive any input operation at any time. This is reflected in the fact that input operations have empty preconditions.

An *execution* of $\mathcal{A}$ is a (finite or infinite) alternating sequence $s_0,\pi_1, s_1,\pi_2,\ldots,\pi_n,s_n,\ldots$ of states and operations of $\mathcal{A}$, beginning with a state, and (if finite) ending with a state. Furthermore, s_0 is a start state of $\mathcal{A}$, and each triple (s',π,s) that occurs as a consecutive subsequence is a step of $\mathcal{A}$. From any execution, we can extract the *schedule*, which is the subsequence of the execution consisting of operations only. Because transitions to different states may have the same operation, different executions may have the same schedule. We say that a schedule α of $\mathcal{A}$ *can leave* $\mathcal{A}$ *in* state s if there is some execution of $\mathcal{A}$ with schedule α and final state s. We say that an operation π is *enabled after* a schedule α of $\mathcal{A}$ if there exists a state s such that α can leave $\mathcal{A}$ in state s and π is enabled in s.

Given a schedule α of automaton $\mathcal{A}$, we define the corresponding *external schedule* $\text{ext}(\alpha)$ to be the subsequence of α consisting of those events that are occurrences of output operations or input operations (that is, we form $\text{ext}(\alpha)$ by removing from α the internal operations). We define the *behavior* of $\mathcal{A}$, $\text{beh}(\mathcal{A})$, to be the set of all sequences that are external schedules of $\mathcal{A}$. Formally, $\text{beh}(\mathcal{A}) = \{\text{ext}(\alpha) : \alpha \text{ is a schedule of } \mathcal{A}\}$. If $\mathcal{A}$ and $\mathcal{B}$ are I/O automata, we say that $\mathcal{B}$ ***implements*** $\mathcal{A}$ if $\mathcal{A}$ and $\mathcal{B}$ have the same output and input operations, and $\text{beh}(\mathcal{B}) \subset \text{beh}(\mathcal{A})$. The intuitive meaning of this is that $\mathcal{B}$ can be safely used for any task for which $\mathcal{A}$ is satisfactory. It is clear that implementation is transitive, that is, if $\mathcal{B}$ implements $\mathcal{A}$ and $\mathcal{C}$ implements $\mathcal{B}$ then $\mathcal{C}$ implements $\mathcal{A}$. When $\mathcal{B}$ implements $\mathcal{A}$ and $\mathcal{A}$ implements $\mathcal{B}$, then we say that $\mathcal{A}$ and $\mathcal{B}$ are ***equivalent***.

We describe systems as consisting of interacting components, each of which is an I/O automaton. It is convenient and natural to view a system itself as an I/O automaton. Thus, we define a composition operation for I/O automata, to yield a new I/O automaton. A set of I/O automata may be composed if, for each component $\mathcal{A}$ the set of internal operations of $\mathcal{A}$ is disjoint from the set of all operations of the other components, and in addition, the sets of output operations of the various automata are pairwise disjoint. A state of the composed automaton is a tuple of states, one for each component, and the start states are tuples consisting of start states of the components. The operations of the composed automaton are those of the component automata. Thus, each operation of the composed automaton is an operation of a subset of the set of component automata. An operation is an output of the composed automaton exactly if it is an output of some component. An operation of the composed automaton is an internal operation exactly if it is an internal operation of some component. An operation of the composed automaton is an input operation exactly if it is not an output or internal operation of any component. (The output operations of a system are intended to be exactly those that are triggered by components of the system, while the input operations of a system are those that are triggered by the system's environment.) During an operation π of a composed automaton, each of the components that has operation π carries out the operation, while the remainder stay in the same state.

An *execution* or *schedule* of a system is defined to be an execution or schedule of the automaton composed of the individual automata of the system. If α is a schedule of a system with component $\mathcal{A}$, then we denote by $\alpha|\mathcal{A}$ the subsequence of α containing all the operations of $\mathcal{A}$. Clearly, $\alpha|\mathcal{A}$ is a schedule of $\mathcal{A}$. The following lemma expresses formally the idea that an operation is under the control of the component of which it is an output.

Lemma 1 *Let α' be a schedule of a system $\mathcal{S}$, and let $\alpha = \alpha'\pi$, where π is an output operation of component $\mathcal{A}$. If $\alpha|\mathcal{A}$ is a schedule of $\mathcal{A}$, then α is a schedule of $\mathcal{S}$.*

We now give the lemma that states that implementation is a compositional property. This is a major reason why modeling algorithms by I/O automata permits modular proofs of correctness.

Lemma 2 *Suppose the automaton $\mathcal{A}$ is the result of composing $\mathcal{A}_i$, and $\mathcal{B}$ is the result of composing $\mathcal{B}_i$. If $\mathcal{B}_i$ implements $\mathcal{A}_i$ for each index i, then $\mathcal{B}$ implements $\mathcal{A}$.*

When we consider a system composed of several components, we are often not interested in the internal working of the system, and so we wish to ignore the operations that model communication between the components. We therefore introduce the *hiding* transformation. If $\mathcal{A}$ is an automaton and π an output operation of $\mathcal{A}$, then the result of hiding π in $\mathcal{A}$ is the automaton with the same states, operations and transition relation as $\mathcal{A}$, but with π classified as an internal operation instead of an output operation. Note that the schedules of the automaton after hiding are exactly the same as the schedules of the original automaton, but the behavior, which is involved in proving implementation, has changed. Clearly if π is an operation of exactly one component of a system, the result of hiding π in that component and then composing the automata, is the same as composing the automata and then hiding π in the composition. We also introduce the transformation that renames an operation of an automaton. So long as the renaming is done consistently throughout a system of automata, and the new name is not already used for any operation of any component, then the result of renaming an operation and then composing is the same as the result of composing and then renaming. Finally we observe that renaming an internal operation of an automaton, as long as the new name is not already used for an operation of the automaton, does not alter the behavior of the automaton.

2.1 Distributed Solutions

We will use I/O automata to model both a global specification of the synchronizer, and the local components of the distributed solution that we will give. Since the fundamental composition mechanism described above is the simultaneous occurrence at several automata of an operation, we have to be careful when modeling asynchronous communication. For example, we would not represent message passing as a single operation shared by sender and receiver. Instead we give explicit automata to represent the communication links, just as we give an explicit automaton to represent each node. Sending a message is an operation that occurs simultaneously at the sender and the link. Similarly, receipt of a message is a shared operation between the link and the recipient. We use nondeterminism within the automaton for the link to capture the asynchrony of the communication network. Thus, we model an asynchronous unidirectional link from p to q, conveying messages from the set $\mathcal{M}$, by the following automaton.

Link Automaton: $\mathrm{LI}_{\mathcal{M}}(\mathrm{p,q})$

Inputs:

send(p,q)M for $M \in \mathcal{M}$

Outputs:

rec(p,q)M for $M \in \mathcal{M}$

state:

multiset contents, initially empty

transitions:

send(p,q)M

Postconditions

s.contents = s'.contents $\cup$ M

rec(p,q)M

Preconditions

$M \in$ s'.contents

Postconditions

s.contents = s'.contents $-$ M

Suppose we are given a distributed problem. This will be specified by an automaton whose schedules are acceptable behaviors for a solution, together with a graph G describing the topology of the network on which a solution has to run, and an assignment locale, that gives for each operation of the specification automaton the node of the network at which it occurs. We now define what it means to say that a system of automata provides a *distributed solution* to this problem. This means that the automaton that results from composing the members of the system and then hiding all operations that are not operations of the specification, is an implementation of the specification in the sense of the previous section, and in addition, the system satisfies the following conditions:

1. The system consists of an automaton NODE(p) for each node p of the graph, together with, for each edge (p,q) of the graph G, two link automata LI(p,q) and LI(q,p) as given above for a suitable choice of message set.

2. For each operation π of the system, either there is a node p such that π is an operation of the node automaton NODE(p) (and no other component), or there are nodes p and q so that π is an input of NODE(p) and an output of LI(q,p) (and an operation of no other component), or there are nodes p and q so that π is an output of NODE(p) and an input of LI(p,q) (and an operation of no other component).

3. Each operation π of the specification automaton is an operation of NODE(p), where p=locale(π) is the node to which the operation is assigned, and of no other component.

3 The Algorithm

The algorithm will run on a network whose topology is given as a connected graph G, described by giving for each node p a set of nodes neighbors(p). The nodes are partitioned into clusters, so that each cluster is connected. Each cluster's subgraph has a distinguished rooted spanning tree. This data is given as follows: for each cluster C there is a node leader(C), and for each node p $\in$ C there is another node parent(p), which is the next node on the path to leader(C). If p = leader(C) then parent(p) = nil. We let children(p) denote the set of nodes q such that parent(q) = p. We say that cluster D is a neighbor of cluster C, written D $\in$ Neighbors(C), if there are nodes p and q with p $\in$ C, q $\in$ D, and q $\in$ neighbors(p). For each pair of neighboring clusters, a single distinguished 'preferred' edge is chosen between them. This is indicated by giving for each node p a set preferred(p) of nodes that are neighbors of p along preferred edges. We say that a node is special if any of its descendants in the tree (that is, itself, or its children, or its children's children, etc.) have neighbors along preferred edges. We let specialchildren(p) denote the subset of children(p) containing special nodes. Thus when there are at least two clusters, the special nodes form the least subtree of a cluster's tree that has the same root and contains all the endpoints of preferred edges.

3.1 The Use of the Synchronizer

We briefly discuss the architecture of the context in which the synchronizer is placed, and show how I/O automata can be used to model all the pieces of such a system. At each node of the asynchronous network is a proccess that executes the code for a graph algorithm in a synchronous system. We model the process at node p by an I/O automaton CLIENT(p), whose operations are synch-receive(p,i)$\mathcal{N}$ and synch-send(p,i)$\mathcal{N}$, where $\mathcal{N}$ is a collection of messages tagged with source or destination information. Round i of the synchronous algorithm at node p is begun when the automaton CLIENT(p) receives an input operation synch-receive(p,i)$\mathcal{N}$, where the messages in the set $\mathcal{N}$ are those that were included with destination p in the sets of messages in preceding synch-send(q,i-1) operations. When the node has finished local processing of these messages, it performs an output operation synch-send(p,i)$\mathcal{N}'$ for a new set of messages and destinations. Different synchronous algorithms will be described by different I/O automata, and we do not constrain the choice except by simple syntactic conditions, such as requiring each p not to perform a synch-send(p,i) operation unless a synch-receive(p,i) operation had occurred earlier, and not to perform a synch-send(p,i) operation if a synch-send(p,i) operation had already occurred.

At each node of the network there is also a process that uses the asynchronous communication system to transmit the messages of the client algorithm, and also to send and receive acknowledgements for such messages. This process has the responsibility of notifying the synchronizer when all the round i messages of the client algorithm have been acknowledged, and it must also delay delivering the collected client algorithm round i messages until the synchronizer has given permission for the start of round i+1 at that node. We model this process at node p by an I/O automaton FRONT-END(p). The operations of CLIENT(p) include synch-send(p,i)$\mathcal{N}$ and synch-receive(p,i)$\mathcal{N}$, which are shared with CLIENT(p). FRONT-END(p) also has operations send(p,q)M(i), rec(q,p)M'(i), send(p,q)ACK-M'(i), and rec(q,p)ACK-M(i), where M and M' are round i messages of the client algorithm. These operations are shared with link automata between p and q. Finally the interaction with the synchronizer is modelled by input operations GO(p,i), which indicate that all round i-1 messages being sent to p have already arrived (and that therefore they can be bundled into a set and delivered to the client algorithm at any time once the client has finished round i-1), and by output operations OK(p,i), which indicate to the synchronizer that acknowledgements have been received at p for all round i messages of the client algorithm that were sent from p.

We give here the explicit construction of the I/O automaton FRONT-END(p). We use the notations described earlier, and also we will assume, for this and for all other I/O automata that we give, that the postconditions of each operation include implicitly the clause s.v = s'.v for each component v of the state s whenever that component s.v is not mentioned in the explicitly given postconditions.

Front-end: FRONT-END(p)

Inputs:

synch-send(p,i)$\mathcal{N}$, for $\mathcal{N}$ a multiset of (message,node) pairs, i positive

rec(q,p)M(i), for q a node, M a message, i positive

rec(q,p)ACK-M(i), for q a node, M a message, i positive

GO(p,i), for i positive

Outputs:

synch-receive(p,i)$\mathcal{N}$, for $\mathcal{N}$ a multiset of (message,node) pairs, i positive

send(p,q)M(i), for q a node, M a message, i positive send(q,p)ACK-M(i), for q a node, M a message, i positive

OK(p,i), for i positive

State:

array GOrec[i], initially all false

array OKsent[i], initially all false

array synchsend[i], initially all false

array synchreceive[i], initially all false

multiset mess, initially empty

multiset ack, initially empty

multiset unacked, initially empty

array of multisets mess-received[i], initially all empty

transitions:

synch-send(p,i)$\mathcal{N}$

Postconditions

  s.synchsend[i] = true

  s.mess = s'.mess $\cup$ {(p,q)M(i) : (M,q) $\in \mathcal{N}$}

rec(q,p)M(i)

Postconditions

  s.ack = s'.ack $\cup$ {(p,q)ACK-M(i)}

  s.mess-received[i] = s'.mess-received[i] $\cup$ {(M,q)}

rec(q,p)ACK-M(i)

Postconditions

  s.unacked = s'.unacked $-$ {(p,q)M(i)}

GO(p,i)

  s.GOrec[i] = true

synch-receive(p,i)$\mathcal{N}$

Preconditions

s'.GOrec[i] = true

i = 1 or s'.synchsend[i-1] = true

s'.synchreceive[i] = false

$\mathcal{N}$ = s'.messreceived[i]

Postconditions

s.synchreceive[i] = true

send(p,q)M(i)

Preconditions

(p,q)M(i) $\in$ s'.mess

Postconditions

s.mess = s'.mess $-$ {(p,q)M(i)}

s.unacked = s'.unacked $\cup$ {(p,q)M(i)}

send(p,q)ACK-M(i)

Preconditions

(p,q)M(i) $\in$ s'.ack

Postconditions

s.ack = s'.ack $-$ {(p,q)M(i)}

OK(p,i)

Preconditions

s'.synchsend[i] = true

s'.unacked $\cup$ s'.mess contains no element (p,q)M(i) for any q or M

s'.OKsent[i] = false

Postconditions

s.OKsent[i] = true

In the next section we will give a specification synchronizer automaton S(G), which uses global information about the OK(q,i) operations at all nodes to determine when to perform GO(p,i+1). In particular, S(G) does not perform GO(p,i+1) until OK(q,i) has occurred for all q $\in$ neighbors(p). In Fig. 6 we illustrate all these automata. When S(G) performs GO(p,i+1), every neighbor of p has received an acknowledgement for every round i message sent. In particular, acknowledgements have been received for every round i message sent to p, and therefore every such message must have arrived at p. Thus FRONT-END(p) will correctly deliver to CLIENT(p) all the round i messages in the synch-receive(p,i+1) operation. It is straightforward to use the techniques of [LM] to turn this argument into a formal proof that the system illustrated behaves (as far as each CLIENT automaton can tell) just like a synchronous system, that is, one in which the clients share their operations with a single communication system automaton, that accepts collections of messages in synch-send input operations from all nodes, sorts out the destinations appropriately, and bundles the messages and delivers them in synch-receive output operations after all client nodes have finished the previous round. In this paper, we concentrate on the problem of showing that a complicated but distributed synchronizer implements the simple but centralized specification synchronizer, where we illustrate the I/O automata model's support for compositional modularity.

3.2 Specification

We give a single specification automaton S(G), called a synchronizer for the graph G. This has an input operation OK(p,i), which is an indication from the front-end at node p that every message it sent in round i has arrived at its destination. When every neighbor q of a node p has issued its OK(q,i-1) operation, the synchronizer can issue an output operation GO(p,i), which indicates to the front-end at node p that it can commence round i of the synchronous algorithm as soon as the client has finished its local processing for round i-1, since there can be no more round i-1 messages in transit to p.

Synchronizer: S(G)

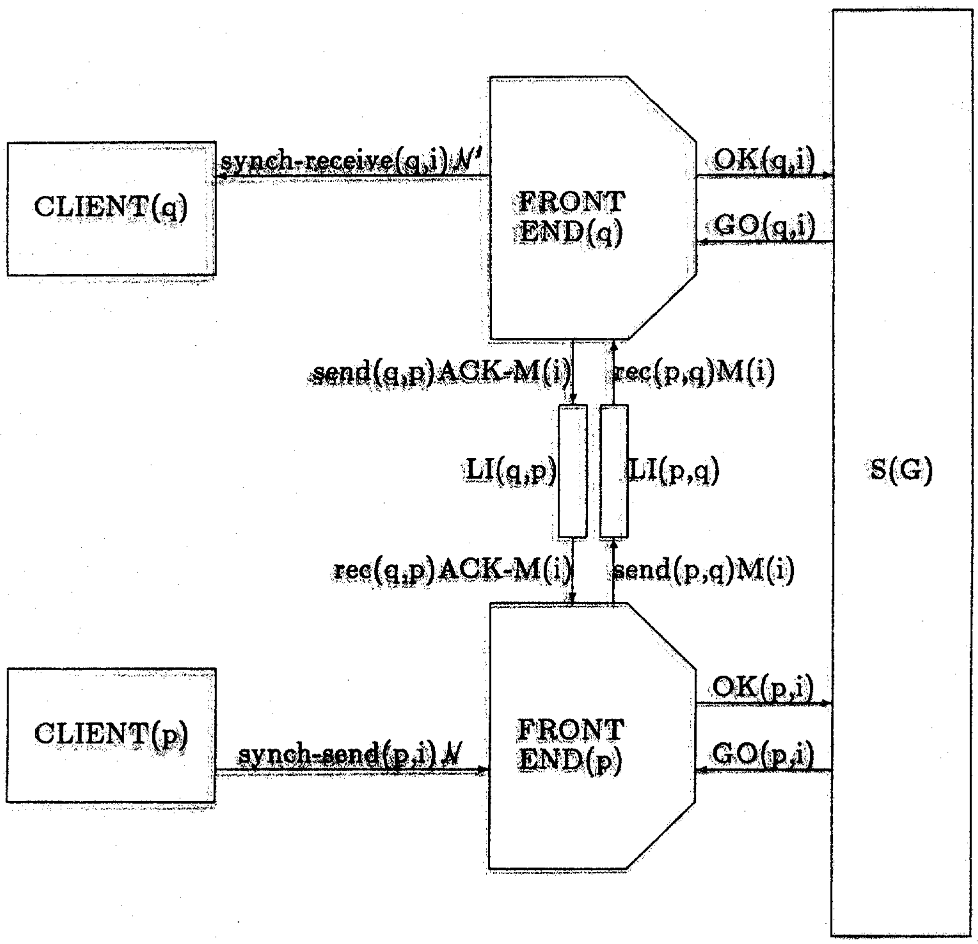

Figure 6: The whole system

Inputs:

OK(p,i) for p $\in$ G, i positive

Outputs:

GO(p,i) for p $\in$ G, i positive

State:

array OKrec[p,i], initially all false

array GOsent[p,i], initially all false

transitions:

OK(p,i)

Postconditions

  s.OKrec[p,i] = true

GO(p,i)

Preconditions

  i = 1 or (s'.OKrec[q,i-1] = true for all q $\in$ neighbors(p))

  i = 1 or s'.GOsent[p,i-1] = true

  s'.GOsent[p,i] = false

Postconditions

  s.GOsent[p,i] = true

3.3 The Detailed Distributed Algorithm

We now give the distributed solution that is closely based on Awerbuch's algorithm γ, translated into the I/O automaton model. We give an automaton ND(p) for each node p of the graph that is not a leader of a cluster, and an automaton LE(C) for the leader of each cluster C. We also give link automata for each edge of the graph G. The detailed code is given in Appendix I, together with an account of the relationship between it and the code in [Aw].

To help the reader understand the algorithm, we give an informal account, paraphrasing [Aw], of the low level working of the system. Once a node p that is a leaf of its cluster's tree has received the OK(p,i) input operation (indicating that the node is safe, that is, every message that node sent in the i-th round has been received) p sends a SAFE(p,i) message to its parent in the tree. Any node p that is not a leaf nor the leader sends a SAFE(p,i) message to its parent only after it has both received the OK(p,i) input and also received SAFE(q,i) messages from all its children. Thus SAFE(p,i) is not sent until every node in the tree that is a descendant of p is safe. This pattern of communication, with a node passing a message to its parent only after receiving it from all its children, is a common paradigm in distributed graph algorithms, and is called *convergecast*. When the leader of cluster C has received SAFE(q,i) messages from all its children q, and also is known to be safe itself (that is, has received OK(p,i)), it issues the CLUSTEROK(C,i) operation.

Once CLUSTEROK(C,i) has occurred, intercluster synchronization begins. The leader sends each of its special children a CLUSTERSAFE(p,i) message. In addition it sends CLUSTERSAFE(p,i) messages over any preferred edges that originate at the leader. Each node p in the tree, after receiving a CLUSTERSAFE(q,i) message from its parent q, sends CLUSTERSAFE(p,i) to its special children, and also along any preferred edges. Thus the CLUSTERSAFE messages are *broadcast* over the subtree of special nodes (this is another standard communication pattern), and are also sent to neighboring trees. The cluster C uses a convergecast of READY(p,i) messages (over the subtree containing only special children) to detect the fact that CLUSTERSAFE(q,i) messages have been received from all neighboring trees along preferred edges. When the leader of the cluster has received READY(q,i) from each of its children, and also has received CLUSTERSAFE(q',i) along any preferred edges that go directly from the leader to neighboring trees, it issues the CLUSTERGO(C,i+1) operation, which indicates the completion of intercluster synchronization for cluster C.

Once the CLUSTERGO(C,i+1) operation has occurred, and also the whole cluster is known to be safe (because the leader has received SAFE(q,i) messages from all its children, and also it has received OK(p,i) itself) the leader p can issue GO(p,i+1) (informing node p that the next round can begin) and it can also send PULSE(p,i+1) messages to each of its children. The PULSE(p,i+1) messages are broadcast over the tree, and when they arrive at each node, that node is able to issue the GO(p,i+1) operation.

We claim that the collection of automata, consisting of all the automata LE(C) for all C, ND(p) for all non-leader nodes p, and LI(p,q) for all p and q such that (p,q) is an edge of G, is a distributed solution to the problem specified by the automaton S(G), the graph G, and the requirement that the operations GO(p,i) and OK(p,i) be assigned to node p. Since it is clear that the system is properly distributed, all that remains is to show that the automaton DistSysS(G), the result of composing the automata and then hiding all operations except GO(p,i) and OK(p,i), implements S(G). This will be done in Theorem 10.

4 The Verification

We now begin the process of verifying that the algorithm given implements the specification. First we divide the code at each node into two pieces, containing the operations and state relevant to inter- and intracluster synchronization, respectively. Then we give the specification SL for an intracluster synchronizer, and remark that the actual code gives an implementation of this using algorithm β. Similarly we note that the collection of automata doing intercluster synchronization in one cluster implements the representative CLCS. In turn, CLCS acts as the whole cluster should, as a piece contributing to intercluster synchronization using algorithm α. Then we give the specification of the coordinator CS, which represents intercluster synchronization, and note that algorithm α is a correct implementation of this. We prove formally that the combination of CS with the automata SL(C) implements the specification S, that is, that synchronization can be achieved by combining intra- and intercluster synchronization. Finally we combine all these results to see that the distributed algorithm γ as described by the detailed code implements the global specification S.

Although the subsidiary claims are given here in a particular bottom-up order, we note that these results are independent, and could be carried out separately and in any order, or even imported from other work (if available).

4.1 The Division between Inter- and Intracluster Algorithms

Following Awerbuch's informal correctness arguments, we will regard the activity of the system as consisting of both inter- and intracluster synchronization. The messages CLUSTERSAFE(p,i) and READY(p,i) are used for intercluster synchronization, while the messages SAFE(p,i) and PULSE(p,i), as well as the operations OK(p,i) and GO(p,i) are part of intracluster synchronization. The operation CLUSTEROK(C,i) serves to communicate from the intracluster synchronizer to the intercluster synchronizer, while CLUSTERGO(C,i) communicates the other way. Thus we give two sets of automata: NDCS(p), LECS(C) and LICS(p,q) to represent the intercluster synchronization, NDSL(p), LESL(C) and LISL(p,q) to represent the intracluster synchronization. The detailed code can be found in Appendix II, as it is extremely similar to the code of the full algorithm. Essentially we divide the operations, state variables and transition relationships of ND(p) between NDCS(p) and NDSL(p) so that each gets the operations, state variables and transitions relevant to its own part of the synchronization. Similarly we divide LE(C) into LECS(C) and LESL(C), and LI(p,q) into LICS(p,q) and LISL(p,q).

It is clear that the composition of the automata NDCS(p) and NDSL(p) is equivalent to the automaton ND(p). The only difference, in fact, is that the composition has two multisets for outgoing messages, while ND(p) has only one multiset buffer. Similarly the composition of LECS(C) and LESL(C) is equivalent to LE(C), and the composition of LICS(p,q) and LISL(p,q) is equivalent to LI(p,q). Therefore DistSysS(G) is equivalent to DistSysS(G)', the result of composing all the automata mentioned in this subsection, and then hiding all the operations except GO(p,i) and OK(p,i). Our task will thus be to prove that DistSysS(G)' implements S(G).

4.2 An Intracluster Synchronizer

The collection of automata that perform intracluster synchronization for a cluster C use algorithm β. The combined activity of these automata is to synchronize the cluster, and in addition to inform the intercluster synchronizer (via CLUSTEROK(C,i)) when the whole cluster is safe, and to delay the GO(p,i) at any node until all neighboring clusters are known to be safe. (The intercluster synchronizer reports this by CLUSTERGO(C,i).) Thus the behavior of the cluster as a whole can be specified by the following automaton:

Modified Synchronizer for cluster C: SL(C)

{This is a slightly modified synchronizer specified, with extra operations that interact with the intercluster synchronizer.}

Inputs:

OK(p,i) for $p \in C$, i positive

CLUSTERGO(C,i) for i positive

Outputs:

GO(p,i) for $p \in C$, i positive

CLUSTEROK(C,i) for i positive

State:

array OKrec[p,i], initially all false

array GOsent[p,i], initially all false

array CLUSTEROKsent[i], initially all false

array CLUSTERGOrec[i], initially all false

transitions:

OK(p,i)

Postconditions

s.OKrec[p,i] = true

CLUSTERGO(C,i)

Postconditions

s.CLUSTERGOrec[i] = true

GO(p,i)

Preconditions

i = 1 or (s'.OKrec[q,i-1] = true for all $q \in$ Neighbors(p) $\cap$ C)

i = 1 or s'.GOsent[p,i-1] = true

s'.CLUSTERGOrec[i] = true

s'.GOsent[p,i] = false

Postconditions

s.GOsent[p,i] = true

CLUSTEROK(C,i)

Preconditions

s'.OKrec[p,i] = true for all p $\in$ C

s'.CLUSTEROKsent[i] = false

Postconditions

s.CLUSTEROKsent[i] = true

In order to express formally the fact that the algorithm β is correct, we let SysSL(C) denote the result of composing the automata LESL(C), NDSL(p) for all p $\in$ C except leader(C), and LISL(p,q) for all p and q so that (p,q) is an edge of G and both p and q are nodes of C, and then hiding all the operations that are not operations of SL(C). Then we have the following lemma, whose proof is found in section 5.1.

Lemma 3 *SysSL(C) implements SL(C).*

4.3 A Cluster Representative for Intercluster Synchronization

In giving his informal account of this algorithm, Awerbuch refers to the intercluster synchronization being performed by using algorithm α between the clusters. Thus, we give, for each cluster C, an automaton that specifies the activity of the whole cluster as a participant in intercluster synchronization, using algorithm α. Thus the cluster sends messages to its neighbors once it has heard (from CLUSTEROK(C,i)) that the cluster is safe, it receives messages from its neighbors indicating that they are safe, and performs CLUSTERGO(C,i) once all the neighboring clusters are known to be safe.

Cluster representative: CLCS(C)

Inputs:

CLUSTEROK(C,i) for i a number

rec(D,C)CLUSTERSAFE(D,i) for D $\in$ Neighbors(C), i positive

Outputs:

CLUSTERGO(C,i) for i positive

send(C,D)CLUSTERSAFE(C,i) for D $\in$ Neighbors(C), i positive

state:

array CLUSTERGOsent[i], initially all false

array CLUSTERSAFErec[D,i], initially all false

multiset mess, initially empty

transitions:

CLUSTEROK(C,i)

Postconditions

s.mess = s'.mess $\cup$ {(C,D)CLUSTERSAFE(C,i) : D $\in$ Neighbors(C)}

rec(D,C)CLUSTERSAFE(D,i)

Postconditions

s.CLUSTERSAFErec[D,i] = true

CLUSTERGO(C,i)

Preconditions

i = 1 or (s'.CLUSTERSAFErec[D,i-1] = true for all D $\in$ Neighbors(C))

i = 1 or s'.CLUSTERGOsent[i] = true

s'.CLUSTERGOsent[i] = false

Postconditions

s.CLUSTERGOsent[i] = true

send(C,D)CLUSTERSAFE(C,i)

Preconditions

(C,D)CLUSTERSAFE(C,i) $\in$ s'.mess

Postconditions

s.mess = s'.mess $-$ {(C,D)CLUSTERSAFE(C,i)}

We denote by SysCLCS(C) the system formed by composing all the automata LECS(C), NDCS(p) for p $\in$ C $-$ leader(C), and LICS(p,q) for p and q in C such that (p,q) is an edge of G, then renaming send(p,q)CLUSTERSAFE(p,i) as send(C,D)CLUSTERSAFE(C,i) and rec(q,p)CLUSTERSAFE(q,i) as rec(D,C)CLUSTERSAFE(D,i) when (p,q) is the preferred edge between C and D, and finally hiding all operations that are not operations of CLCS(C). Then we have the following claim, that the detailed algorithm in each cluster implements the required behavior. Its proof is found in section 5.2.

Lemma 4 *SysCLCS(C) implements CLCS(C).*

4.4 An Intercluster Synchronizer

If we consider all the automata CLCS(C) for each cluster C, together with link automata LICS(C,D) (each of these is just LICS(p,q) for (p,q) the preferred edge between C and D with operations renamed, with p replaced by C and q replaced by D), then these together perform algorithm α to synchronize between the clusters. Thus we introduce an automaton that is just a specification synchronizer for the quotient graph formed by identifying all the nodes in a cluster together, except that each state and operation name is prefixed by 'cluster'.

Intercluster Synchronizer: CS

Inputs:

CLUSTEROK(C,i) for C a cluster, i positive

Outputs:

CLUSTERGO(C,i) for C a cluster, i positive

State:

array CLUSTEROKrec[C,i], initially all false array CLUSTERGOsent[C,i], initially all false

transitions:

CLUSTEROK(C,i)

Postconditions

s.CLUSTEROKrec[C,i] = true

CLUSTERGO(C,i)

Preconditions

i = 1 or (s'.CLUSTEROKrec[D,i-1] = true for all D $\in$ Neighbors(C))

i = 1 or (s'.CLUSTERGOsent[C,i-1] =true)

s'.CLUSTERGOsent[C,i] = false

Postconditions

s.CLUSTERGOsent[C,i] = true

We denote by SysCS the automaton formed by composing the automata CLCS(C) for all clusters C, and LICS(C,D) for all pairs of clusters C and D that are neighbors, and then hiding all operations that are not operations of CS. The fact that algorithm α is correct is expressed simply by the following lemma, whose proof is given in section 5.3.

Lemma 5 *SysCS implements CS.*

4.5 High Level Structure

Consider an automaton SysS(G), which is formed by composing the intracluster synchronizers SL(C) for all clusters C, together with the intercluster synchronizer CS, and then hiding all the operations except GO(p,i) and OK(p,i). The fact that performing inter- and intracluster synchronization is a way to synchronize the whole graph, is expressed in the following simple statement: SysS(G) implements S(G). In order to prove this statement, we first give several results that relate the schedules of the automata involved to the states in which the automata are left. First we discuss the specification automaton S(G).

Lemma 6 *Let α be a schedule of $S(G)$, and let s be the state of $S(G)$ after α. Then*

1. *s.OKrec[p,i]=true if and only if α contains OK(p,i).*
2. *s.GOsent[p,i]=true if and only if α contains GO(p,i).*

Proof: We give the proof of (1), as the proof of (2) is almost the same. We use induction on the length of α. If α is empty, then it does not contain OK(p,i), and s is the initial state, for which s.OKrec[p,i]=false. Thus suppose $\alpha = \alpha'\pi$, and let s' be the state of S(G) after α'. If π is OK(p,i), then α contains OK(p,i), and by the postcondition of the operation OK(p,i), s.OKrec[p,i] = true. Otherwise π is an operation whose postconditions do not mention OKrec[p,i], and so we have s.OKrec[p,i] = true if and only if s'.OKrec[p,i] = true, which by the induction hypothesis occurs if and only if α' contains OK(p,i). But (since π is not OK(p,i)) we also have in this situation that α' contains OK(p,i) if and only if α contains OK(p,i). This completes the proof of (1). Q.E.D.

We next give the lemmas about the state of the components of SysS(G). The proofs are almost identical to that for Lemma 6, and so are left to the reader.

Lemma 7 *Let α be a schedule of CS, and let s be the state of CS after α. Then*

1. *s.CLUSTEROKrec[C,i]=true if and only if α contains CLUSTEROK(C,i).*
2. *s.CLUSTERGOsent[C,i]=true if and only if α contains CLUSTERGO(C,i).*

Lemma 8 *Let α be a schedule of SL(C), and let s be the state of SL(C) after α. Then*

1. *s.OKrec[p,i]=true if and only if α contains OK(p,i).*
2. *s.GOsent[p,i]=true if and only if α contains GO(p,i).*
3. *s.CLUSTEROKsent[i]=true if and only if α contains CLUSTEROK(C,i).*
4. *s.CLUSTERGOrec[i]=true if and only if α contains CLUSTERGO(C,i).*

Now we can prove the claim above, which says that intracluster synchronization and intercluster synchronization combine to provide synchronization for the whole graph G.

Lemma 9 *SysS(G) implements S(G).*

Proof: Since every input and output operation of S(G) is an input or output of some component SL(C) from which the system SysS(G) is formed, we only need to prove that whenever α is a schedule of SysS(G), and β denotes the subsequence of α consisting of the operations of S(G), then β is a schedule of S(G). This is proved by induction on the length of α. If α is empty, then so is β, so that β is a schedule of S(G). So let us assume that $\alpha = \alpha'\pi$. Letting β' denote the subsequence of α' consisting of operations of S(G), we have by the induction hypothesis that β' is a schedule of S(G). If π is not an operation of S(G), then $\beta = \beta'$, and we are done. Otherwise $\beta = \beta'\pi$. If π is OK(p,i), then π is an input to S(G), and so is enabled after any schedule of S(G), by the Input Condition, and therefore β is a schedule of S(G).

Thus we suppose that π is GO(p,i). Let s denote the state of SL(C) after α', where C is the cluster containing p. Let t denote the state of S(G) after β'. We have that π is enabled (as an operation of SL(C)) in t, and we will deduce that it is enabled (as an operation of S(G)) in s. By the preconditions for π, t.GOsent[p,i] = false, and thus by Lemma 8 α' does not contain GO(p,i). Therefore β' does not contain GO(p,i), and so by Lemma 6, s.GOsent[p,i] = false. Also by the preconditions, either i = 1 or t.GOsent[p,i] = true. If i $\neq$ 1, by Lemma 8 α' contains GO(p,i-1), and thus β' contains GO(p,i-1). Therefore, by Lemma 6, either i = 1 or s.GOsent[p,i-1] = true.

Suppose that i $\neq$ 1. Then the preconditions of π as an operation of SL(C) imply that t.CLUSTERGOrec[i] = true and that t.OKrec[q,i-1] = true for all q $\in$ Neighbors(p) $\cap$ C. By Lemma 8, α' contains CLUSTERGO(C,i) and OK(q,i) for all q $\in$ Neighbors(p) $\cap$ C. Now, by examining the preconditions for the operation CLUSTERGO(C,i) of the intercluster synchronizer CS, and Lemma 7, we see that the prefix of α' preceding the CLUSTERGO(C,i) operation must contain CLUSTEROK(D,i-1) for all clusters D that are neighbors of C. Therefore, by the preconditions of the operation CLUSTEROK(D,i-1) of SL(D) and Lemma 8, we deduce that the prefix of α' preceding each CLUSTEROK(D,i-1) contains the operations OK(q,i-1) for all nodes q in cluster D. Thus α' (and hence β') contains OK(q,i-1) for all q $\in$ Neighbors(p), as any such q is either in Neighbors(p) $\cap$ C, or else is a member of a cluster D that is in Neighbors(C). By Lemma 6, s.OKrec[q,i-1] = true for any q $\in$ Neighbors(p).

Thus we have shown that s.GOsent[p,i] = false, that i = 1 or s.GOsent[p,i-1] = true, and that i=1 or (s.OKrec[q,i-1] = true for all q $\in$ Neighbors(p)). That is, we have shown that π is enabled in state s, completing the proof. Q.E.D.

4.6 The Main Theorem

We can now combine the results given above to verify the correctness of the detailed algorithm for network synchronization.

Theorem 10 *DistSysS(G) implements S(G).*

Proof: We first consider DistSysCS, the automaton that results from composing all the automata NDCS(p), LECS(C) and LICS(p,q), and then hiding all operations except CLUSTERGO(C,i) and CLUSTEROK(C,i). By the associativity of composition (and the fact that renaming and hiding behave well in composition), this is equivalent to composing all the automata SysCLCS(C) and LICS(C,D), and then hiding the remaining operations except CLUSTERGO(C,i) and CLUSTEROK(C,i). Since by Lemma 4, SysCLCS(C) implements CLCS(C) for each C, we have that DistSysCS implements SysCS by Lemma 2. Since by Lemma 5, SysCS implements CS, we deduce that DistSysCS implements CS.

Now DistSysS(G) is equivalent to DistSysS(G)', the result of composing all the automata NDCS(p), NDSL(p), LECS(C), LESL(C), LICS(p,q) and LISL(p,q), and then hiding all operations except GO(p,i) and OK(p,i). But DistSysS(G)' is, by the associativity of composition, equivalent to the result of composing DistSysCS with all the automata SysSL(C), and then hiding operations. Since by Lemma 3 SysSL(C) implements SL(C), and, as we saw above, DistSysCS implements CS, we can deduce from Lemma 2 that DistSysS(G)' implements SysS(G), the result of composing CS with all the automata SL(C) and then hiding all operations except GO(p,i) and OK(p,i). By Lemma 9, SysS(G) implements S(G), and therefore DistSysS(G)' implements S(G). Thus DistSysS(G) implements S(G). Q.E.D.

5 Subsidiary Correctness Proofs

We will now give the proofs of the claims made and used in the previous section about the correctness of the simpler algorithms such as synchronizers α and β. First, we prove the fundamental lemmas about the behavior of a link automaton, as these are used repeatedly in the following proofs.

Lemma 11 *Let α be a schedule of $LI_{\mathcal{M}}(p,q)$, and let s be the state of $LI_{\mathcal{M}}(p,q)$ after α. Then for $M \in \mathcal{M}$, the multiplicity of M as an element of s.contents is $x-y$, where x is the number of occurrences in α of send(p,q)M and y is the number of occurrences in α of rec(p,q)M.*

Proof: By induction on the length of α. The base case, when α is empty, is trivial since then s is the initial state, so s.contents is empty and the multiplicity of M is zero. On the other hand x and y are also both zero. Thus we suppose $\alpha = \alpha'\pi$, and let s' be the state of $LI_{\mathcal{M}}$(p,q) after α'. If π is send(p,q)M' or rec(p,q)M' for M' $\neq$ M, then by the postconditions above the multiplicity of M is the same in s.contents as in s'.contents. Also the number of occurrences of send(p,q)M and rec(p,q)M are the same in α as in α'. Thus the lemma follows from the inductive hypothesis that the multiplicity of M in s'.contents equals the difference between the number of occurrences of send(p,q)M and rec(p,q)M in α'.

If π is send(p,q)M, the multiplicity of M in s.contents is one more than its multiplicity in s'.contents. On the other hand α contains one more occurrence of send(p,q)M than α', and α and α' contain the same number of occurrences of rec(p,q)M. Therefore the lemma follows from the induction hypothesis. If π is rec(p,q)M the multiplicity of M in s.contents is one less than its multiplicity in s'.contents but α contains the same number of occurrences of send(p,q)M than α', and α contains one more occurrence of rec(p,q)M than α'. Thus the lemma follows from the induction hypothesis. An obvious consequence of this lemma is the following:

Lemma 12 *Let α be a schedule of $LI_{\mathcal{M}}(p,q)$ and let $M \in \mathcal{M}$. Then α contains at least as many occurrences of send(p,q)M as of rec(p,q)M.*

5.1 Correctness of Intracluster Synchronization

We prove Lemma 3, which says that algorithm β is correct.

We first study the components out of which SysSL(C) is formed.

Lemma 13 *Let α be a schedule of NDSL(p) and let s be the state of NDSL(p) after α. Then*

1. *s.OKrec[p,i]=true if and only if α contains OK(p,i).*

2. *s.SAFErec[q,i]=true if and only if α contains rec(q,p)SAFE(q,i).*

3. *s.GOsent[p,i]=true if and only if α contains GO(p,i).*

4. *s.pulse[i]=true if and only if α contains rec(parent(p),p)PULSE(parent(p),i).*

5. *The multiplicity of (p,q)PULSE(p,i) as an element of s.mess equals $x-y$ where x is the number of occurrences of rec(parent(p),p)PULSE(parent(p),i) in α and y is the number of occurrences of send(p,q)PULSE(p,i) in α.*

6. *The multiplicity of (p,parent(p))SAFE(p,i) as an element of s.mess equals $x-y$ where x is the number of occurrences in $\alpha-\beta$ of any of operations OK(p,i) or rec(q,p)SAFE(q,i) for $q \in$ children(p) (where β is the longest prefix of α not containing at least one occurrence of each of the operations OK(p,i) and rec(q,p)SAFE(q,i) for $q \in$ children(p)), and y is the number of occurrences of send(p,parent(p))SAFE(p,i) in α.*

Immediate consequences of the previous lemma are given next.

Lemma 14 *Let $q \in$ children(p). If α is a schedule of NDSL(p) then α contains at least as many occurrences of rec(parent(p),p)PULSE(parent(p),i) as of send(p,q)PULSE(p,i).*

Lemma 15 *If α is a schedule of NDSL(p) that contains send(p,parent(p))SAFE(p,i) then α contains rec(q,p)SAFE(q,i) for all $q \in$ children(p), and α also contains OK(p,i) .*

Lemma 16 *Let α be a schedule of LESL(C) and let s be the state of LESL(C) after α. Then*

1. *s.OKrec[q,i]=true if and only if α contains OK(q,i).*

2. *s.GOsent[q,i]=true if and only if α contains GO(q,i).*

3. *s.SAFErec[q,i]=true if and only if α contains rec(q,p)SAFE(p,i), where p=leader(C).*

4. *s.CLUSTERGOrec[q,i]=true if and only if α contains CLUSTERGO(C,i).*

5. *s.clustersafe[i]=true if and only if α contains OK(p,i) and rec(q,p)SAFE(q,i) for p= leader(C) and all $q \in$ children(p).*

6. *s.pulse[i]=true if and only if α contains CLUSTERGO(C,i) and either i=1 or s.clustersafe[i-1]=true.*

7. *s.CLUSTEROKsent[i]=true if and only if α contains CLUSTEROK(C,i).*

8. *For p = leader(C), the multiplicity of (p,q)PULSE(p,i) as an element of s.mess equals $x-y$ where x is the number of occurrences in $\alpha-\beta$ of any of the operations CLUSTERGO(C,i), OK(p,i-1) or rec(q,p)SAFE(q,i-1) (where β is the longest prefix of α not containing CLUSTERGO(C,i) and (if $i \neq 1$) at least one occurrence of each of OK(p,i-1) and rec(q,p)SAFE(q,i-1) for $q \in$ children(p)), and y is the number of occurrences of send(p,q)PULSE(p,i) in α.*

We next give an immediate consequence of part (7) of the Lemma above.

Lemma 17 *Let p = leader(C), and $q \in$ children(p). If α is a schedule of LESL(C) that contains send(p,q)PULSE(p,i) then α contains CLUSTERGO(C,i) and (if $i \neq 1$) OK(p,i-1) and rec(q,p)SAFE(q,i-1) for all $q \in$ children(p).*

The next result is an immediate consequences of the preconditions for CLUSTEROK(C,i) as an operation of LESL(C), and (5) of Lemma 16.

Lemma 18 *Let p = leader(C). If α is a schedule of LESL(C) that contains CLUSTEROK(C,i), then α contains OK(p,i) and rec(q,p)SAFE(q,i) for all $q \in$ children(p).*

We next prove the fundamental invariants of the system SysSL(C) that capture the principles of the broadcast and convergecast paradigms of message flow. We recall that SysSL(C) is formed by composing NDSL(p) for p $\in$ C $-$ leader(C), LESL(C), and LISL(p,q) for p and q in C, and then hiding certain operations, so its schedules are just schedules of the composition.

Lemma 19 *Let α be a schedule of the automaton that results form composing NDSL(p) for p $\in C -$ leader(C), LESL(C), and LISL(p,q) for p and q in C. If α contains send(p,parent(p))-SAFE(p,i) for some p such that $p \in C$, $p \neq$ leader(C), then α contains OK(q',i) for all q' such that q' is a descendant of p in the tree of C.*

Proof: We use induction on the height of p in the tree of C. The basis case, when p has height 1, is when p is a leaf of the tree. In this case we need only check that α contains OK(p,i), as p has no descendants except itself. This case is immediate from Lemma 15. So suppose that the Lemma has been proved for all non-leader nodes of height at most k, and that p has height k+1, for $k > 1$. By Lemma 15, α contains rec(q,p)SAFE(q,i) for all q $\in$ children(p), and also OK(p,i). By Lemma 12, α must contain send(p',p)SAFE(p',i) for all p' $\in$ children(p), but such p' have height at most k, and none is leader(C). Thus the induction hypothesis implies that α contains OK(q',i) for all q' such that q' is a descendant of p' where p' is a child of p. However any q' that is a descendant of p is either p itself or a descendant of a child of p. Thus α contains OK(q',i) for all q' that are descendants of p in the tree.

Q.E.D.

Lemma 20 *Let α be a schedule of the automaton that results form composing NDSL(p) for $p \in C - leader(C)$, LESL(C), and LISL(p,q) for p and q in C. Let s be the state of LESL(C) after α. If s.clustersafe[i]=true then α contains OK(q',i) for all q' $\in$ C.*

Proof: By Lemma 16 α contains an OK(p,i) for p=leader(C) and a rec(q,p)SAFE(q,i) for all q $\in$ children(p). By Lemma 12 α contains a send(q,p)SAFE(q,i) for all q $\in$ children(p) that then by Lemma 19 implies that α contains OK(q',i) for all q' descendants of all q $\in$ children(p). Thus we have shown that α contains OK(q',i) for all q' $\in$ C . Q.E.D.

Lemma 21 *Let α be a schedule of the automaton that results form composing NDSL(p) for $p \in C - leader(C)$, LESL(C), and LISL(p,q) for p and q in C. Suppose that s.pulse[i]=true, where s is the state of the NDSL(p) (or LESL(C) if p=leader(C)) after α. Then α contains CLUSTERGO(C,i) and also, either i=1 or α contains OK(q,i-1) for all $q \in C$.*

Proof: We use induction on the depth of p in the tree of C. The basis case, when p has depth 1, is when p=leader(C). From Lemma 16, we see that α contains CLUSTERGO(C,i) and that either i=1 or else s.clustersafe[i-1]=true. By Lemma 20, either i=1 or α contains OK(q,i-1) for all q $\in$ C. Thus we suppose that the lemma has been proved for all nodes of depth at most k, and that p has depth k+1, for $k > 1$. Then p is not the leader of C. By Lemma 13 s.pulse[i]=true implies α contains rec(parent(p),p)PULSE(parent(p),i), which by Lemma 12 implies that α contains a send(parent(p),p)PULSE(parent(p),i). Now the preconditions of send(parent(p),p)PULSE(parent(p),i) imply s'.pulse[i]=true, where s' is the state of NDSL(parent(p)) (or LESL(C), if parent(p)=leader(C)), immediately before the operation send(parent(p),p)PULSE(parent(p),i). But parent(p) has depth k, and so the induction hypothesis implies that a prefix of α, and thus α itself, contains CLUSTERGO(C,i) and also that either i=1 or α contains OK(q,i-1) for all q $\in$ C. Q.E.D.

Now we are ready to prove the claim, given as Lemma 3, that SysSL(C) acts as a modified synchronizer for the whole cluster C, by following algorithm β.

Lemma 22 *SysSL(C) implements SL(C).*

Proof: Since every input and output operation of SL(C) is an input or output of SysSL(C), we only need to prove that whenever α is a schedule of the composition SysSL(C), and β denotes the subsequence of α consisting of operations of SL(C), then β is a schedule of SL(C). This is proved by induction on the length of α. If α is empty, then so is β, so that β is a schedule of SL(C). Therefore let us assume that $\alpha = \alpha'\pi$. Letting β' denote the subsequence of α' consisting of operations of SL, we have by the induction hypothesis that β' is a schedule of SL. If π is not an operation of SL, then $\beta = \beta'$, and we are done. Otherwise $\beta = \beta'\pi$. If π is CLUSTERGO(C,i) or OK(p,i) where then π is an input to SL(C), and so is enabled after any schedule of SL(C), by the Input Condition, and therefore β is a schedule of SL(C).

If π is CLUSTEROK(C,i), then by preconditions for π as operation of LESL(C) and Lemma 16, α' must not contain CLUSTEROK(C,i) and also s.clustersafe(i)=true, where s is the state of LESL(C) after α'. By Lemma 20, α' contains OK(p,i) for all p $\in$ C . Therefore, transferring these facts to β', we see that β' contains OK(p,i) for all p $\in$ C, and that β' does not contain CLUSTEROK(C,i). Let t denote the state of SL(C) after β'. By Lemma 8, t.OKsent[p,i]=true for all p $\in$ C, and t.CLUSTEROKsent[i]=false. Examining the preconditions for π as an operation of SL(C), we see that π is enabled after β', and thus β is a schedule of SL(C).

If π is GO(p,i), then let s denote the state after α' of NDSL(p) (or LESL(C) if p=leader(C)). By the preconditions for π as an operation of NDSL(p) or LESL(C), and Lemma 16 or Lemma 13, α' does not contain GO(p,i) and also, if i$\neq$1, α' contains GO(p,i-1). Also, the precondition s.pulse[i]=true for π as an operation of NDSL(p) or LESL(C), implies by Lemma 21 that α' contains CLUSTERGO(C,i) and also that, if i $\neq$ 1, α' contains OK(q,i-1) for all q $\in$ C. Thus β' does not contain GO(p,i) and contains CLUSTERGO(C,i), and if i $\neq$ 1, also contains GO(p,i-1) and OK(q,i-1) for all q $\in$ C. Now, by the preconditions for π as an operation of SL(C), and by Lemma 8, we have that π is enabled after β', so β is a schedule of SL(C) as required. Q.E.D.

5.2 Correctness of the Cluster Representative

Now we prove Lemma 4, which says that the broadcast and convergecast, used by the automata NDCS(p) and LECS(C) to communicate within a cluster C, work as the cluster representative CLCS(C) is supposed to. Once again, we first relate the schedules of the automata involved to the states in which the automata are left.

Lemma 23 *Let α be a schedule of CLCS(C), and let s be the state of CLCS(C) after α. Then*

1. *s.CLUSTERGOsent[i]=true if and only if α contains CLUSTERGO(C,i).*
2. *s.CLUSTERSAFErec[D,i]=true if and only if α contains rec(D,C)CLUSTERSAFE(D,i).*
3. *the multiplicity of (C,D)CLUSTERSAFE(C,i) as an element of s.mess equals $x-y$, where x is the number of occurrences of CLUSTEROK(C,i) in α and y is the number of occurrences of send(C,D)CLUSTERSAFE(C,i) in α.*

For later use, we observe the following immediate consequence of (3) above.

Lemma 24 *Let α be a schedule of CLCS(C). Then α contains at least as many occurrences of CLUSTEROK(C,i) as of send(C,D)CLUSTERSAFE(C,i).*

We now study the components out of which SysCLCS(C) is formed.

Lemma 25 *Let α be a schedule of NDCS(p) and let s be the state of NDCS(p) after α. Then*

1. *s.CLUSTERSAFErec[q,i]=true if and only if α contains rec(q,p)CLUSTERSAFE(q,i).*
2. *s.READYrec[q,i]=true if and only if α contains READY(q,i).*
3. *If $q \in$ specialchildren(p) $\cup$ Preferred(p), the multiplicity of (p,q)CLUSTERSAFE(p,i) as an element of s.mess equals $x-y$ where x is the number of occurrences of rec(parent(p),p)-CLUSTERSAFE(parent(p),i) in α and y is the number of occurrences of send(p,q)-CLUSTERSAFE(p,i) in α.*

4. *The multiplicity of (p,parent(p))READY(p,i) as an element of s.mess equals* $x-y$ *where* x *is the number of occurrences in* $\alpha-\beta$ *of any of the operations rec(q,p)READY(q,i) for* $q \in$ *specialchildren(p) or rec(q',p)CLUSTERSAFE(q',i) for q'* $\in$ *Preferred(p), (where* β *is the longest prefix of* α *not containing at least one occurrence of all the operations rec(q,p)READY(q,i) for* $q \in$ *specialchildren(p) and rec(q',p)CLUSTERSAFE(q',i) for q'* $\in$ *Preferred(p)), and* y *is the number of occurrences of send(p,parent(p))READY(p,i) in* α.

Immediate consequences of (3) and (4) of the previous lemma are given next.

Lemma 26 *Let* $q \in$ *children(p)* $\cup$ *Preferred(p). If* α *is a schedule of NDCS(p) then* α *contains at least as many occurrences of rec(parent(p),p)CLUSTERSAFE(parent(p),i) as of send(p,q)CLUSTERSAFE(p,i).*

Lemma 27 *If* α *is a schedule of NDCS(p) that contains send(p,parent(p))READY(p,i) then* α *contains rec(q,p)READY(q,i) for all* $q \in$ *specialchildren(p), and* α *also contains rec(q',p)CLUSTERSAFE(q',i) for all q'* $\in$ *Preferred(p).*

We similarly study LECS(C).

Lemma 28 *Let* α *be a schedule of LECS(C) and let s be the state of LECS(C) after* α. *Then*

1. *s.READYrec[q,i]=true if and only if* α *contains rec(q,p)READY(q,i), where p=leader(C).*
2. *s.CLUSTERSAFErec[q,i]=true if and only if* α *contains rec(q,p)CLUSTERSAFE(q,i), where p=leader(C).*
3. *s.CLUSTERGOsent[i]=true if and only if* α *contains CLUSTERGO(C,i).*
4. *For* $p =$ *leader(C) and* $q \in$ *specialchildren(p)* $\cup$ *Preferred(p), the multiplicity of (p,q)-CLUSTERSAFE(p,i) as an element of s.mess equals* $x-y$ *where* x *is the number of occurrences of CLUSTEROK(C,i) in* α *and* y *is the number of occurrences of send(p,q)-CLUSTERSAFE(p,i) in* α.

We next give an immediate consequence of (4) above.

Lemma 29 *Let p = leader(C), and $q \in$ children(p) $\cup$ Preferred(p). If α is a schedule of LECS(C) then α contains at least as many occurrences of CLUSTEROK(C,i) as of send(p,q)CLUSTERSAFE(p,i).*

The next result is an immediate consequence of the preconditions for CLUSTERGO(C,i) as an operation of LECS(C), and (2) of Lemma 28.

Lemma 30 *Let p = leader(C). If α is a schedule of LECS(C) that contains CLUSTERGO(C,i) for a value $i > 1$, then α contains rec(q,p)READY(q,i-1) for all $q \in$ specialchildren(p), and α also contains rec(q',p)CLUSTERSAFE(q',i-1) for all q' $\in$ Preferred(p).*

We next prove the fundamental invariants of the system SysCLCS(C) that capture the principles of the broadcast and convergecast paradigms of message flow. We recall that SysCLCS(C) is formed by composing NDCS(p) for p $\in$ C $-$ leader(C), LECS(C), and LICS(p,q) for p and q in C, and then renaming and hiding certain operations.

Lemma 31 *Let α be a schedule of the automaton that results form composing NDCS(p) for $p \in C -$ leader(C), LECS(C), and LICS(p,q) for p and q in C. Let p and q be such that $p \in C$ and $q \in$ specialchildren(p) $\cup$ Preferred(p). Then α contains at least as many occurrences of CLUSTEROK(C,i) as of send(p,q)CLUSTERSAFE(p,i).*

Proof: We use induction on the depth of p in the tree of C. The basis case, when p has depth 1, is when p=leader(C). This case is immediate from Lemma 29. So suppose that the lemma has been proved for all nodes of depth at most k, and that p has depth k+1, for $k > 1$. Then p is not the leader of C. Let x denote the number of occurrences of send(p,q)CLUSTERSAFE(p,i) in α. By Lemma 26, α contains at least x occurrences of rec(parent(p),p)CLUSTERSAFE(parent(p),i), and therefore by Lemma 12, it contains at least x occurrences of send(parent(p),p)CLUSTERSAFE(parent(p),i). However parent(p) has depth k, and so the induction hypothesis implies that α contains at least x occurrences of CLUSTEROK(C,i), as required. Q.E.D.

Lemma 32 *Let α be a schedule of the automaton that results form composing NDCS(p) for p $\in C -$ leader(C), LECS(C), and LICS(p,q) for p and q in C. If α contains send(p,parent(p))-READY(p,i) for some p such that $p \in C$, $p \neq$ leader(C), then α contains rec(q,q')CLUSTER-*

SAFE(q,i) for all q and q' such that q' is a descendant of p in the tree of C, and q $\in$ Preferred(q').

Proof: We use induction on the height of p in the tree of C. The basis case, when p has height 1, is when p is a leaf of the tree. In this case we need only check that α contains rec(q,p)CLUSTERSAFE(q,i) for q $\in$ Preferred(p), as p has no descendants except itself. This case is immediate from Lemma 27. So suppose that the Lemma has been proved for all non-leader nodes of height at most k, and that p has height k+1, for k > 1. By Lemma 27, α contains rec(q,p)CLUSTERSAFE(q,i) for all q $\in$ Preferred(p), and also rec(p',p)READY(p',i) for all p' $\in$ specialchildren(p). By Lemma 12, α must contain send(p',p)READY(p',i) for all p' $\in$ children(p), but such p' have height at most k, and none is leader(C). Thus the induction hypothesis implies that α contains rec(q,q')CLUSTERSAFE(q,i) for all q and q' such that q' is a descendant of p' where p' is a special child of p, and such that q $\in$ Preferred(q'). However for any q' that is a descendant of p and for which q $\in$ Preferred(q'), q' is either p itself or a descendant of a special child of p. Thus we have completed the proof. Q.E.D.

Now we are ready to prove the claim, Lemma 4 that SysCLCS(C) acts as a representative of the whole cluster C, within algorithm α.

Lemma 33 *SysCLCS(C) implements CLCS(C).*

Proof: Since every input and output operation of CLCS(C) is an input or output of SysCLCS(C), we only need to prove that whenever α is a schedule of the composition SysCL$_{CS}$(C), and β denotes the subsequence of α consisting of operations of CLCS(C), then β is a schedule of CLCS(C). This is proved by induction on the length of α. If α is empty, then so is β, so that β is a schedule of CLCS(C). Therefore let us assume that $\alpha = \alpha'\pi$. Letting β' denote the subsequence of α' consisting of operations of CS, we have by the induction hypothesis that β' is a schedule of CS. If π is not an operation of CS, then $\beta = \beta'$, and we are done. Otherwise $\beta = \beta'\pi$. If π is CLUSTEROK(C,i) or rec(D,C)CLUSTERSAFE(D,i) where then π is an input to CLCS(C), and so is enabled after any schedule of CLCS(C), by the Input Condition, and therefore β is a schedule of CLCS(C).

If π is send(C,D)CLUSTERSAFE(C,i), then before renaming (as an operation of the automaton that results form composing NDCS(p) for p $\in$ C $-$ leader(C), LECS(C), and

LICS(p,q) for p and q in C), π was send(p,q)CLUSTERSAFE(p,i) where p $\in$ C, q $\in$ Preferred(p), and q $\in$ D. Then by Lemma 31, α (and hence α' and β') contains at least x occurrences of CLUSTEROK(C,i), where x is the number of occurrences of send(C,D)CLUSTERSAFE(C,i) in α, since these were exactly the occurrences of send(p,q)CLUSTERSAFE(p,i) before renaming. Thus β' contains x-1 occurrences of send(C,D)CLUSTERSAFE(C,i). By Lemma 23, (C,D)CLUSTERSAFE(C,i) is an element of t.mess, where t is the state of CLCS(C) after β', and thus π is enabled in state t. Thus β is a schedule of CLCS(C).

If π is CLUSTERGO(C,i), then before renaming (as an operation of the automaton that results form composing NDCS(p) for p $\in$ C $-$ leader(C), LECS(C), and LICS(p,q) for p and q in C), π was also CLUSTERGO(C,i). By the preconditions for π as an operation of LECS(C) and Lemma 28, α' must not contain CLUSTERGO(C,i). Also, if i$\neq$1, α' (before renaming) must contain CLUSTERGO(C,i-1) and rec(q,p)CLUSTERSAFE(q,i-1) for p = leader(C) and all q $\in$ Preferred(p), and rec(p',p)READY(p',i-1) for p = leader(C) and all p' $\in$ children(p). Then, by Lemma 12, α' (before renaming) contains send(p',p)READY(p',i-1) for all p' $\in$ children(p), and hence (by Lemma 32) before renaming, α' contains rec(q,q')CLUSTERSAFE(q,i-1) for all q' descended from a child of p, and q $\in$ Preferred(q'). Thus we have shown that, before renaming, α' contains rec(q,q')CLUSTERSAFE(q,i-1) for all q' descended from p (that is, all q' $\in$ C), and all q $\in$ Preferred(q'). Therefore (after renaming) α' contains CLUSTERGO(C,i-1) and rec(D,C)CLUSTERSAFE(D,i-1) for all D $\in$ Neighbors(C). We can transfer all the above conclusions to β', deducing that β' does not contain CLUSTERGO(C,i), and if i $\neq$ 1, β' contains CLUSTERGO(C,i-1) and rec(D,C)CLUSTERSAFE(D,i-1) for all D $\in$ Neighbors(C). By the preconditions for π as an operation of CLCS(C) and Lemma 23, we have that π is enabled after β', so β is a schedule of CLCS(C) as required.

Q.E.D.

5.3 Correctness of Intercluster Synchronization

We next prove the claim of Lemma 5, that algorithm α provides correct synchronization between the clusters.

Lemma 34 *SysCS implements CS.*

Proof: Since every input and output operation of CS is an input or output of SysCS, we only need to prove that whenever α is a schedule of SysCS, and β denotes the subsequence of α consisting of the operations of CS, then β is a schedule of CS. This is proved by induction on the length of α. If α is empty, then so is β, so that β is a schedule of CS. Therefore let us assume that $\alpha = \alpha'\pi$. Letting β' denote the subsequence of α' consisting of operations of CS, we have by the induction hypothesis that β' is a schedule of CS. If π is not an operation of CS, then $\beta = \beta'$, and we are done. Otherwise $\beta = \beta'\pi$. If π is CLUSTEROK(C,i), then π is an input to CS, and so is enabled after any schedule of CS, by the Input Condition, and therefore β is a schedule of CS.

Thus we suppose that π is CLUSTERGO(C,i). Let s denote the state of CLCS(C) after α'. Let t denote the state of CS after β'. We have that π is enabled (as an operation of CLCS(C)) in t, and we will deduce that it is enabled (as an operation of CS) in s. By the preconditions for π, t.CLUSTERGOsent[i] = false, and thus by Lemma 23 α' does not contain CLUSTERGO(C,i). Therefore β' does not contain CLUSTERGO(C,i), and so by Lemma 7, s.CLUSTERGOsent[C,i] = false. Also by the preconditions, either i = 1 or t.CLUSTERGOsent[i] = true. If i $\neq$ 1, by Lemma 23 α' contains CLUSTERGO(C,i-1), and thus β' contains CLUSTERGO(C,i-1). Therefore, by Lemma 7, either i = 1 or s.CLUSTERGOsent[C,i-1] = true.

Suppose that i $\neq$ 1. Then the preconditions of π as an operation of CLCS(C) imply that t.CLUSTERSAFErec[D,i-1] = true for all D $\in$ Neighbors(C). Thus by Lemma 23 α' contains rec(D,C)CLUSTERSAFE(D,i-1) for all D $\in$ Neighbors(C), and hence by Lemma 12 α' contains send(D,C)CLUSTERSAFE(D,i-1). By Lemma 24 applied to CLCS(D), α' contains CLUSTEROK(D,i-1). Therefore β' contains CLUSTEROK(D,i-1), and so by Lemma 7 s.CLUSTEROKrec[D,i-1] = true for all D $\in$ Neighbors(C).

Thus we have shown that s.CLUSTERGOsent[C,i] = false, that i = 1 or s.CLUSTERGOsent[C,i-1] = true, and that i=1 or (s.CLUSTEROKrec[D,i-1] = true for all D $\in$ Neighbors(C)). That is, we have shown that π is enabled in state s, completing the proof. Q.E.D.

6 Message and Time Analysis

We will now show that operational reasoning in the I/O model can be used to prove results about the message and time performance of the algorithm, as well as the safety property of implementing a specification. In order to do this, however we will need to restrict the environment of the system, that is, the ways in which the input operations OK(p,i) arrive. We say that a schedule of the distributed synchronization system DistSysS(G) is *well-formed* if any occurrence of OK(p,i) is preceded by GO(p,i) and is not preceded by OK(p,i). Thus a well-formed schedule reflects the behavior of the system when the environment is issuing only one OK message at each node for each round, and is not issuing that until the synchronizer has allowed the round to start.

We now show that in a well-formed schedule every operation can occur at most once.

Lemma 35 *Let α be a well-formed schedule of DistSysS(G). Then α contains at most one occurrence of each operation.*

Proof: Since the DistSysS(G) is equivalent to DistSysS(G), we can and will regard α as a schedule of DistSysS(G)'. We use induction on the length of α. The basis case, when α is empty, is trivial. Thus we suppose $\alpha=\alpha'\pi$, and that α' contains at most one occurrence of each operation. In order to show the same for α, we need only prove that α' does not contain π.

If π is OK(p,i) this is immediate from the definition of well-formed.

If π is rec(p,q)M for some message M, this follows from Lemma 12, since by the induction hypothesis α' (and thus α) contains at most one occurrence of send(q,p)M.

If π is GO(p,i) or CLUSTERGO(C,i) or CLUSTEROK(C,i), this is a consequence of the preconditions for π as an operation of the appropriate component automaton. Each of these operations has a precondition that checks that the operation has not already occurred, for example s'.GOsent[i]=false is a precondition for GO(p,i), and by Lemma 13 this means that α' does not contain GO(p,i).

If π is send(p,q)PULSE(p,i) and p is not the root of its tree, this follows from part (5) of Lemma 13, since the multiplicity of a message in a multiset cannot be negative, and by the induction hypothesis α' (and hence α) contains at most one occurrence of rec(parent(p),p)PULSE(parent(p),i). If π is send(p,q)PULSE(p,i) where p=leader(C), the lemma follows similarly from part (8) of Lemma 16, since by the induction hypothesis each operation CLUSTERGO(C,i), OK(p,i-1) and rec(q',p)SAFE(q',i-1) can occur at most once in α' and so all except one of these (namely the one that occurs last) occur in a prefix of α not containing all of them.

If π is send(p,q)SAFE(p,i) the lemma follows from part (6) of Lemma 13, since the multiplicity of a message in a multiset is non-negative, and only the last one of the operations OK(p,i) or rec(q',p)SAFE(q',i) for q' $\in$ children(p), will not occur in a prefix of α not containing all of these operations.

If π is send(p,q)READY(p,i) the lemma follows from part (4) of Lemma 25 in the same way.

If π is send(p,q)CLUSTERSAFE(p,i) the lemma follows from part (4) of Lemma 28, or part (3) of Lemma 25, depending on whether or not p is the leader of its tree.

Thus we have proved the lemma for each possibility for π. Q.E.D.

6.1 Message Complexity

We now show how we can bound the number of messages sent in an execution of the algorithm. We will speak of the messages PULSE(p,i), SAFE(p,i-1), CLUSTERSAFE(p,i-1) and READY(p,i-1) as all *belonging to* round i, because they are sent in preparation for issuing a GO(p,i) operation. We note that if α is a schedule of DistSysS(G) containing an operation send(p,q)M for a message M belonging to round i, and i $\neq$ 1, then α contains at least one operation OK(p',i-1). If M is SAFE(p,i-1) this is proved in Lemma 19. If M is CLUSTERSAFE(p,i-1) then Lemma 31 implies that α contains CLUSTEROK(C,i-1), whose precondition s'.clustersafe[i-1]=true implies by Lemma 20 that α contains OK(p,i-1) as desired. If M is READY(p,i-1) then Lemma 32 shows that α contains some rec(q',q")-CLUSTERSAFE(q',i-1) operation, for q' a descendant of p, and thus a send(q',q")CLUSTER-SAFE(q',i-1) operation, and hence some OK(p',i-1) operation, by the above. Finally if M is PULSE(p,i) then α contains OK(q',i-1) for all q' in p's cluster, by Lemma 21. This result implies for a well-formed schedule of DistSysS(G), that if it contains a message belonging to round i, then it contains GO(p,i-1) for some p.

Now we can prove that the number of messages used per round is bounded by four times the number of edges that are preferred edges or tree edges. We say that round i is *commenced* in the execution α if α contains some GO(p,i) operation.

Lemma 36 *Suppose α is a well-formed schedule of DistSysS(G) for which i_0 is the largest round number commenced. Then the number of send(q,q')M operations in α is at most $4(i_0+1)$ times the number of tree or preferred edges.*

Proof: The observations above show that α contains no operation send(q,q')M where M is a message belonging to a round greater than i_0+1. Since no the link automata on edges, other than tree or preferred edges, have empty message sets, and each of the two automata on a preferred or tree edge has at most 2 messages belonging to each round in its message set, the result is immediate from Lemma 35. Q.E.D.

6.2 Time Complexity and Liveness

In order to discuss the time complexity of the algorithm, we introduce the idea of a 'timed execution'. We call the combination of an execution $s_0,\pi_1,s_1,\pi_2,s_2,\ldots$ of automaton $\mathcal{A}$ and a nondecreasing sequence of nonnegative real numbers ('times') $t_1,t_2,\ldots,$, where there are the same number of t_i as there are operations π_i in the execution, a *timed execution* of $\mathcal{A}$. Intuitively, we understand this combination as indicating that π_i occurred at time t−i. As a convention we put $t_0 = 0$. For any nonnegative t, we say that s_i is *a state of the automaton at time* t if $t_i \leq t \leq t_{i+1}$. Note that since the times need not be strictly increasing, there may be several states at a given time. We refer to the subsequence of the execution up to, but not including, the first operation π_i for which $t_i > T$, as the execution *up to time T*, so that the state s_{i-1} that ends this is the last state of the automaton at time T. Thus the operations π_i that occur in the execution up to time T are exactly those whose times t_i are less than or equal to T. In order to prove any bounds on the time the synchronizer algorithm takes, we will need to assume that the component automata take steps promptly. Thus we introduce the notion of a 1-admissible timed execution of an automaton $\mathcal{A}$. We say that a timed execution of $\mathcal{A}$ is *1-admissible*[2] if whenever there is an output or internal operation

[2]This is a special case of a more general definition due to Tuttle.

π, a state s and a time T, such that $s = s_i$ is a state of the automaton at time T and π is enabled in state s, then there is some index $j > i$ such that the operation $\pi = \pi_j$ and $t_j \leq T+1$. In particular, in a 1-admissible timed execution, any operation (other than an input) enabled in a state at time T, occurs in the execution of the system up to time T+1.

Now, an output or internal operation is enabled for an automaton formed by composing components and hiding operations, exactly when it is enabled for the unique component automaton of which the operation is not an input operation. It follows that in applying the definition of 1-admissible timed execution to the system DistSysS(G), we can consider the states of the component automata separately. For example, when we consider the link automaton $LI_{\mathcal{M}}(p,q)$, we see that the definition implies that in a 1-admissible timed execution of a distributed solution, any message sent is delivered within one unit of time. We also remark that all the automata discussed in this paper have the property that once an output or internal operation is enabled, it remains enabled until it occurs.

We first prove that the system DistSysS(G) begins by issuing GO(p,1) operations promptly.

Lemma 37 *Let H be the greatest depth of a tree in the spanning forest for G. Then any 1-admissible timed execution of DistSysS(G) contains GO(p,1) for all p, in the execution up to time 2H.*

Proof: We prove that for any node p, the operations GO(p,1) and send(p,q)PULSE(p,1) occur in the execution up to time 2k, where k is the depth of p in its cluster's tree. This statement clearly implies the truth of the lemma, and we will prove it by induction on k.

The basis case, when k=1, is when p=leader(C) for some cluster C. Notice that for each cluster C, the operation CLUSTERGO(C,1) of LE(C) is enabled in the initial state of the system, and so is enabled in a state at time 0. Therefore the operation occurs by time 1. Examining the postconditions of CLUSTERGO(C,1), and the preconditions of GO(p,1) and send(p,q)PULSE(p,1) for $q \in$ children(p), we see that each operation GO(p,1) and send(p,q)PULSE(p,1) is enabled in the last state of the system at time 1, unless it has occurred already in the execution up to time 1. In either case, we deduce that each operation GO(p,1) and send(p,q)PULSE(p,1) occurs in the execution up to time 2.

Now we suppose the statement proved for all nodes of depth k-1, and prove it for a node p of depth k, for some value $k > 1$. Since $k \neq 1$, p is not leader(C), so let p'=parent(p). Then p' has depth k-1, the induction hypothesis shows that the execution up to time 2k-2 contains send(p',p)PULSE(p',1). Therefore, considering the preconditions for rec(p',p)PULSE(p',1) as an operation of LI(p',p), rec(p',p)PULSE(p',1) is enabled in the last state of the system at time 2k-2 unless rec(p',p)PULSE(p',1) has occurred in the execution to time 2k-2. In any case, rec(p',p)PULSE(p',1) must occur in the execution up to time 2k-1. Examining the postconditions of rec(p',p)PULSE(p',1) as an operation of ND(p), we see that the preconditions of each of the operations GO(p,1) and send(p,q)PULSE(p,1) for q $\in$ children(p) are satisfied in the last state at time 2k-1, unless the operation in question has already occurred in the execution up to time 2k-1. In any case, each operation must occur in the execution up to time 2k. This completes the inductive step of the proof of the statement, and thus completes the proof of the lemma. Q.E.D.

Now we prove that the algorithm has good time performance, as claimed in [Aw].

Lemma 38 *Let H be the greatest depth of a tree in the spanning forest for G. Suppose i is a positive integer. Then any 1-admissible well-formed timed execution of DistSysS(G) that contains OK(p,i) for every node p in the execution up to time T, contains GO(p,i+1) for every node p in the execution up to time T+8H.*

Proof: We first prove the statement that for any node p, whose height in its cluster's tree is k, the execution up to time T+2k-2 contains rec(p',p)SAFE(p',i) for all p' $\in$ children(p). This is proved by induction on the height k. The basis case, when k=1, is when p is a leaf. This case is trivial as there are no elements of children(p). Therefore we assume that k > 1, and that the statement has been proved for all nodes of height less than k. Fix any p' $\in$ children(p), so p' has height at most k-1, and so by the induction hypothesis, the execution up to time T+2k-4 contains rec(p",p')SAFE(p",i) for every p" $\in$ children(p'). Examining the postconditions of the operations OK(p',i) and rec(p",p')SAFE(p",i), we see that the last of these to occur causes (p',p)SAFE(p',i) to be placed in the outgoing message buffer of ND(p'), and so (since all have occurred in the execution to time T+2k-4) the operation send(p',p)SAFE(p',i) is enabled in the last state at time T+2k-4, unless it has already occurred in the execution to time T+2k-4. In any case send(p',p)SAFE(p',i) must occur in the execution to time T+2k-3. Considering the link automaton LI(p',p), we see that rec(p',p)SAFE(p',i) is enabled in the last state at time T+2k-3, unless it has already occurred, and so rec(p',p)SAFE(p',i) must occur in the execution to time T+2k-2. Since p' was an arbitrary child of p, this establishes the truth of the statement.

Next we prove the statement that for any special node p, whose depth in its cluster's tree is k, the execution up to time T+2H+2k-2 contains send(p,q)CLUSTERSAFE(p,i) for every q $\in$ specialchildren(p) $\cup$ Preferred(p). This time we use induction on the depth k. The basis case, when k=1, is when p=leader(C). Examining the preconditions of the CLUSTEROK(C,i) operation of the automaton LE(C), we deduce from the previous statement (since p has height at most H in its tree) that CLUSTEROK(C,i) is enabled in the last state at time T+2H-2, unless it has occurred earlier. In any case, CLUSTEROK(C,i) must occur in the execution to time T+2H-1. Examining the postconditions of CLUSTEROK(C,i), we see that, for every q $\in$ specialchildren(p) $\cup$ Preferred(p), send(p,q)CLUSTERSAFE(p,i) is enabled in the last state at time T+2H-1, unless it has occurred already. In any case, send(p,q)CLUSTERSAFE(p,i) occurs in the execution up to time T+2H, proving the statement for k=1. Assuming the result proved for nodes of depth less than k, we prove the statement for a special node p of depth k > 1. Since parent(p) is special, and has depth k-1, the induction hypothesis implies that the execution to time T+2H+2k-4 contains send(parent(p),p)CLUSTERSAFE(parent(p),i). Thus the execution up to time T+2H+2k-3 contains rec(parent(p),p)CLUSTERSAFE(parent(p),i). Examining the postconditions of this operation of ND(p), we see that each operation send(p,q)CLUSTERSAFE(p,i) for q $\in$ specialchildren(p) $\cup$ Preferred(p) is enabled in the last state at time T+2H+2k-3, unless it has already occurred. In any case each of these operations must occur in the execution to time T+2H+2k-2, completing the proof of this statement.

Next we prove the statement that for every special node p, whose height in its cluster's tree is k, the execution up to time T+4H+2k-3 contains rec(p',p)READY(p',i) for all p' $\in$ specialchildren(p). The basis case, when k=1, is trivial, as then p is a leaf of the tree and has no children at all. Therefore, we assume that k > 1, and that the statement has been proved for all special nodes of height less than k. Fix any p' $\in$ specialchildren(p), so p' has height at most k-1. Examining the postconditions of all the operations rec(q,p')READY(q,i) for q $\in$ specialchildren(p'), and rec(q',p')CLUSTERSAFE(q',i) for q' $\in$ Preferred(p'), we see that the last of these to occur causes (p',p)READY(p',i) to be placed in the outgoing message buffer of ND(p'). However each of rec(q,p')READY(q,i) occurs in the execution up to time T+4H+2k-5, by the induction hypothesis, and each of rec(q',p')CLUSTERSAFE(q',i) occurs in the execution up to time T+4H-1 since send(q',p')CLUSTERSAFE(q',i) occurs in the execution up to time T+4H-2 (by the previous statement). Since p is special, the set of events rec(q,p')READY(q,i) for q ∈ specialchildren(p') and rec(q',p')CLUSTERSAFE(q',i) for q' ∈ Preferred(p'), is not empty, and so send(p',p)READY(p',i) is enabled in the state at time T+4H+2k-5 unless it occurred already. In any case, send(p',p)READY(p',i) occurs in the execution up to time T+4H+2k-4, and so rec(p',p)READY(p',i) occurs in the execution up to time T+4H+2k-3.

Finally we observe that we can prove by induction on the depth, that for any node p, whose depth in its cluster's tree is k, and any q ∈ children(p), the operations GO(p,i+1) and send(p,q)PULSE(p,i+1) occur in the execution up to time T+6H+2k-3. This statement clearly implies the truth of the lemma. The basis case, when k=1, is when p=leader(C) for some cluster C. Since the schedule we are considering is well formed, it contains GO(p',i) for every p' ∈ G, and therefore (considering the preconditions for GO(p,i)), also contains CLUSTERGO(C,i). Thus the operation CLUSTERGO(C,i+1) of LE(C) is enabled in the last state at time T+6H-3, unless it has occurred already, since the execution up to time T+6H-3 contains rec(p',p)READY(p',i) for all p' ∈ specialchildren(p), by the previous statement, and the execution up to time T+4H-1 contains rec(q',p)CLUSTERSAFE(q',i) for all q' ∈ Preferred(p), because send(q',p)CLUSTERSAFE(q',i) occurred by time T+4H-2. We can deduce that CLUSTERGO(C,i+1) occurs in the execution up to time T+6H-2. Examining the postconditions of whichever occurs last of the operations CLUSTERGO(C,i+1), OK(p,i) and rec(p',p)SAFE(p',i) for p' ∈ children(p), we see that each of the operations GO(p,i+1) and send(p,q)PULSE(p,i+1) is enabled in the last state of the system at time T+6H-2, unless it has occurred already. Therefore each occurs in the execution up to time T+6H-1. The case where k > 1 is straightforward, since then parent(p) has depth k-1, and so the induction hypothesis says that send(parent(p),p)PULSE(parent(p),i+1) occurs in the execution up to time T+6H+2k-5, and thus rec(parent(p),p)PULSE(parent(p),i+1) occurs by time T+6H+2k-4. The postconditions of this operation show that each of GO(p,i+1) and send(p,q)PULSE(p,i+1) is enabled in the last state at time T+6H+2k-4, unless it has occurred earlier, and so each occurs by time T+6H+2k-3, as required. Q.E.D.

Even without assuming that the system performs actions within time 1, as we did above, we can show that the system satisfies a liveness condition, as long as each output or internal operation is performed eventually, once it is enabled. Thus we say that an execution $s_0, \pi_1, s_1, \pi_2, \ldots$ is *admissible* if for every i and every operation π that is enabled in state s_i, there is an index j with $j > i$ such that $\pi_j = \pi$. The following lemmas have proofs that are almost identical to those of the two previous lemmas concerning timed executions, except that references to specific times are deleted, and instead operations are deduced to occur 'eventually'.

Lemma 39 *Any admissible execution of DistSysS(G) contains GO(p,1) for all p.*

Lemma 40 *Suppose i is a positive integer. Any admissible well-formed execution of DistSysS(G) that contains OK(p,i) for every node p, contains GO(p,i+1) for every node p.*

7 Summary and Further Directions

In this paper we have offered a formal, rigorous proof of the correctness of Awerbuch's algorithm for network synchronization. We specified both the algorithm and the correctness condition using the I/O automaton model. Our proof of correctness followed closely the intuitive arguments made by the designer of the algorithm by exploiting the model's natural support for such important design techniques as stepwise refinement and modularity. In particular, since the algorithm uses simpler algorithms for synchronization within and between 'clusters' of nodes, our proof could have imported as lemmas the correctness of these simpler algorithms, if these had been proved before. Alternatively, the understanding of the modularity that the proof gives us would allow us to see how to safely change the choices of implementation of the separate parts of the synchronizer, independently of one another. Also, we clearly benefit from having carried out the correctness proof in the I/O automaton model which supports modularity, since the network synchronizer is often used as an 'off-the-shelf building block' component in a larger system, and proofs of the correctness of the system will be able to use our proof without change.

In the future, we hope to study other network protocols in the same way. We still need to understand how to use the model to capture the intuition behind other, less clear-cut, forms of 'modularity'. For example many network algorithms operate over spanning forests that change with time, and so seem to be hard to represent as intermediate specifications implemented by collections of automata. Nonetheless, we expect that the I/O automaton model will provide support for verifying many protocols, once we understand the precise nature of the modularity.

8 Bibliography

[**Aw**] Awerbuch, B., 'Complexity of Network Synchronization,' JACM, *32*, *4*, 804-823 (1985).

[**Aw2**] Awerbuch, B., 'Reducing Complexities of Distributed Maximum Flow and Breadth-First Search Algorithms by means of Network Synchronization,' Networks, *15*, 425-437 (1985).

[**FLMW**] Fekete, A., Lynch, N., Merritt, M., and Weihl, W., 'Nested Transactions and Read/Write Locking,' Proceedings of 6th ACM Symposium on Principles of Database Systems, 1987.

[**GL**] Goldman, K., and Lynch, N., 'Nested Transactions and Quorum Consensus,' Proceedings of 6th ACM Symposium on Principles of Distributed Computation, 1987.

[**HLMW**] Herlihy, M., Lynch, N., Merritt, M., and Weihl, W., 'Correctness of Orphan Elimination Algorithms,' Proceedings of 17th IEEE Symposium on Fault-Tolerant Computing, 1987.

[**HO**] Hailpern, B., and Owicki, S., 'Verifying Network Protocols Using Temporal Logic,' Proceedings of IEEE Conference on Trends and Applications: 1980, Computer Network Protocols.

[**LM**] Lynch, N., and Merritt, M., 'Introduction to the Theory of Nested Transactions,' Technical Report MIT/LCS/TR-367, MIT Laboratory for Computer Science, Cambridge, MA., July 1986.

[LT] Lynch, N., and Tuttle, M., 'Hierarchical Correctness Proofs for Distributed Algorithms,' Proceedings of 6th ACM Symposium on Principles of Distributed Computation, 1987.

[MP] Manna, Z., and Pnueli, A., 'Verification of Concurrent Programs: the Temporal framework,' In *The Correctness Problem in Computer Science*, R. Boyer and J. Moore, eds, Academic Press, 1981.

[OG] Owicki, S., and Gries, D., 'An Axiomatic Proof Technique for Parallel Programs I,' Acta Informatica *6*, *4*, 319-340 (1976).

[Wl] Welch, J., 'Synthesis of Efficient Mutual Exclusion Algorithms,' manuscript

Appendix I: The Detailed Code for the Synchronization Algorithm

We give the code for each automaton ND(p) for a non-leader node p, and also for each automaton LE(C) for the leader node of cluster C. Afterwards, we discuss the code for two operations, to give the interested reader some feeling for the model. We also discuss the way our algorithm is developed from the code in [Aw], which is written for an interrupt-driven model.

Non-leader node: ND(p)

Inputs:

rec(q,p)READY(q,i) for $q \in$ children(p), i positive

rec(q,p)CLUSTERSAFE(q,i) for $q \in$ Preferred(p) or $q =$ parent(p), i positive

OK(p,i) for i positive

rec(q,p)SAFE(q,i) for $q \in$ children(p), i positive

rec(q,p)PULSE(q,i) for $q =$ parent(p), i positive

Outputs:

send(p,q)READY(p,i) for $q =$ parent(p), i positive

send(p,q)CLUSTERSAFE(p,i) for $q \in$ children(p) $\cup$ Preferred(p), i positive

GO(p,i), for i positive

send(p,q)SAFE(p,i) for q = parent(p), i positive

send(p,q)PULSE(p,i) for q ∈ children(p), i positive

state:

array CLUSTERSAFErec[q,i], initially all false

array READYrec[q,i], initially all false

array OKrec[i], initially all false

array GOsent[i], initially all false

array SAFErec[q,i], initially all false

array pulse[i], initially all false

multiset mess, initially empty

transitions:

rec(q,p)READY(q,i)

Postconditions

  s.READYrec[q,i] = true

  if q ∈ specialchildren(p)

    and (s'.READYrec[q',i] = true for all q' ∈ (specialchildren(p)-{q}))

    and (s'.CLUSTERSAFErec[q',i] = true for all q' ∈ Preferred(p))

  then s.mess = s'.mess ∪ {(p,parent(p))READY(p,i)}

rec(q,p)CLUSTERSAFE(q,i)

Postconditions

  s.CLUSTERSAFErec[q,i] = true

  if q = parent(p)

  then s.mess = s'.mess ∪ {(p,p')CLUSTERSAFE(p,i) : p' ∈ specialchildren(p) ∪ Preferred(p)}

  if q ∈ Preferred(p)

    and (s'.READYrec[q',i] = true for all q' ∈ specialchildren(p))

    and (s'.CLUSTERSAFErec[q',i] = true for all q' ∈ (Preferred(p)-{q}))

  then s.mess = s'.mess ∪ {(p,parent(p))READY(p,i)}

OK(p,i)

Postconditions

s.OKrec[i] = true

if (s'.SAFErec[q,i] = true for all q $\in$ children(p))

then s.mess = s'.mess $\cup$ {(p,parent(p))SAFE(p,i)}

rec(q,p)SAFE(q,i)

Postconditions

s.SAFErec[q,i] = true

if (s'.SAFErec[q',i] = true for all q' $\in$ children(p)-{q}

and s'.OKrec[i] = true)

then s.mess = s'.mess $\cup$ {(p,parent(p))SAFE(p,i)}

rec(q,p)PULSE(q,i)

Postconditions

s.pulse[i] = true

s.mess = s'.mess $\cup$ {(p,p')PULSE(p,i) : p' $\in$ children(p)}

send(p,q)READY(p,i)

Preconditions

(p,q)READY(p,i) $\in$ s'.mess

Postconditions

s.mess = s'.mess $-$ {(p,q)READY(p,i)}

send(p,q)CLUSTERSAFE(p,i)

Preconditions

(p,q)CLUSTERSAFE(p,i) $\in$ s'.mess

Postconditions

s.mess = s'.mess $-$ {(p,q)CLUSTERSAFE(p,i)}

GO(p,i)

Preconditions

s'.pulse[i] = true

i = 1 or s'.GOsent[i-1] = true

s'.GOsent[i] = false

Postconditions

s.GOsent[i] = true

send(p,q)SAFE(p,i)

Preconditions

(p,q)SAFE(p,i) ∈ s'.mess

Postconditions

s.mess = s'.mess − {(p,q)SAFE(p,i)}

send(p,q)PULSE(p,i)

Preconditions

(p,q)PULSE(p,i) ∈ s'.mess

Postconditions

s.mess = s'.mess − {(p,q)PULSE(p,i)}

Leader: LE(C)

Inputs:

rec(q,p)READY(q,i) for p = leader(C), q ∈ children(p), i positive

rec(q,p)CLUSTERSAFE(q,i) for p = leader(C), q ∈ preferred(p), i positive

OK(p,i) for p = leader(C), i positive

rec(q,p)SAFE(q,i) for p = leader(C), q ∈ children(p), i positive

Outputs:

CLUSTERGO(C,i) for i positive

send(p,q)CLUSTERSAFE(p,i) for p = leader(C), q $\in$ children(p) $\cup$ preferred(p), i positive

GO(p,i), for p = leader(C), i positive

CLUSTEROK(C,i) for i positive

send(p,q)PULSE(p,i) for p = leader(C), q $\in$ children(p), i positive

state:

array READYrec[q,i], initially all false

array CLUSTERSAFErec[q,i], initially all false

array clustergo[i], initially all false

array OKrec[i], initially all false

array GOsent[i], initially all false

array SAFErec[q,i], initially all false

array clustersafe[i], initially all false

array pulse[i], initially all false

array CLUSTEROKsent[i], initially all false

multiset mess, initially empty

transitions:

rec(q,p)READY(q,i)

Postconditions

s.READYrec[q,i] = true

rec(q,p)CLUSTERSAFE(q,i)

Postconditions

s.CLUSTERSAFErec[q,i] = true

OK(p,i)

Postconditions

s.OKrec[i] = true

if (s'.SAFErec[q,i] = true for all q $\in$ children(p)) then (s.clustersafe[i] = true

if (s'.SAFErec[q,i] = true for all q ∈ children(p)

and s'.clustergo[i+1] = true)

then (s.mess = s'.mess ∪ {(p,q)PULSE(p,i+1) : p ∈ children(p)}

and s.pulse[i+1] = true))

rec(q,p)SAFE(q,i)

Postconditions

s.SAFErec[q,i] = true

if (s'.SAFErec[q',i] = true for all q' ∈ children(p)-{q}

and s'.OKrec[i] = true)

then s.clustersafe[i] = true

if (s'.SAFErec[q',i] = true for all q' ∈ children(p)-{q}

and s'.OKrec[i] = true and s'.clustergo[i+1] = true)

then (s.mess = s'.mess ∪ {(p,q)PULSE(p,i+1) : p ∈ children(p)}

and s.pulse[i+1] = true)

CLUSTERGO(C,i)

Preconditions

i = 1 or ((s'.READYrec[q,i-1] = true for all q ∈ specialchildren(p))

and (s'.CLUSTERSAFErec[q,i-1] = true for all q ∈ Preferred(p)))

i = 1 or s'.clustergo[i-1] = true

s'.clustergo[i] = false

Postconditions

s.clustergo[i] = true

if (i = 1 or s'.clustersafe[i-1] = true)

then (s.mess = s'.mess ∪ {(p,p')PULSE(p,i) : p' ∈ children(p)}

and s.pulse[i] = true)

send(p,q)CLUSTERSAFE(p,i)

Preconditions

(p,q)CLUSTERSAFE$(p,i) \in s'$.mess

Postconditions

s.mess = s'.mess $-$ $\{(p,q)$CLUSTERSAFE$(p,i)\}$

GO(p,i)

Preconditions

s'.pulse[i] = true

i = 1 or s'.GOsent[i-1] = true

s'.GOsent[i] = false

Postconditions

s.GOsent[i] = true

CLUSTEROK(C,i)

Preconditions

s'.clustersafe[i] = true

s'.CLUSTEROKsent[i] = false

Postconditions

s.CLUSTERTOKsent[i] = true

s.mess = s'.mess $\cup$ $\{(p,q)$CLUSTERSAFE(p,i) : $q \in$ (specialchildren(p) $\cup$ Preferred(p))$\}$

send(p,q)PULSE(p,i)

Preconditions

(p,q)PULSE$(p,i) \in s'$.mess

Postconditions

s.mess = s'.mess $-$ $\{(p,q)$PULSE$(p,i)\}$

For each p and q for which (p,q) is an edge of G, we let LI(p,q) be a link automaton from p to q, for the message set $\mathcal{M}$ described next: if (p,q) is a preferred edge, then $\mathcal{M}$ is the set of messages CLUSTERSAFE(p,i) for positive i; if p = parent(q) then $\mathcal{M}$ is the set of CLUSTERSAFE(p,i) and PULSE(p,i) for positive i; if p $\in$ children(q) then $\mathcal{M}$ is the set of READY(p,i) and SAFE(p,i) for positive i; if (p,q) is neither a preferred edge nor a tree edge then $\mathcal{M}$ is the empty set (so in this case the link automaton is the trivial automaton with no operations!).

As an aid in understanding the code above, we consider the pre- and postconditions for the operation rec(q,p)CLUSTERSAFE(q,i) of the non-leader node automaton ND(p). This is an input operation, and so it has no preconditions, since it can occur at any time. When it occurs, the fact that it has happened is recorded in the state by setting the value of CLUSTERSAFErec[q,i] to true. The other effects depend on whether this is a message being broadcast over p's own cluster (this is the case if q is p's parent) or whether this is a message from a neighboring cluster (when q is a neighbor of p over a preferred edge). In the first case, a CLUSTERSAFE(p,i) message to p' is added to the multiset of outgoing messages, for each p' among p's children and also for each p' that is a neighbor along a preferred edge. In the second case, the node checks to see whether all the conditions are now satisfied, in order to play its part in the convergecast of READY messages. The convergecast can occur if a READY(q',i) message has been received from every special child q' (as recorded in the state of the READYrec[q',i] variables) and if a CLUSTERSAFE(q',i) message has been received from every neighbor q' along a preferred edge (except, of course, for q itself). If all of these have been received, the node places a READY(p,i) message for its parent, in its buffer of outgoing messages.

As another example, consider the operation GO(p,i) for a non-leader node p. This can occur provided the PULSE(q,i) message has arrived from p's parent (a fact reflected by the variable pulse[i] being true) and if the previous GO operation (if any) has already occurred, and if the GO(p,i) itself has not occurred (this is necessary as the other conditions once true, remain true forever). The fact that the operation has occurred is reflected in the state by setting GOsent[i] to true.

The Relationship to Awerbuch's Original Algorithm

We have given the detailed algorithm for network synchronization by using I/O automata, where a node changes state after receiving a message, and a message can be sent (and the node's state can change accordingly) whenever the send(p,q)M operation is enabled. In his account, Awerbuch used the interrupt-driven model that is more common among designers of network algorithms, where the effects of a message receipt include (atomically) both changes in the state of the node involved and the sending of messages from that node, but where messages are not generated spontaneously. As the reader can see, we have expressed the interrupt-driven code 'on receipt of M from q: change the value of variable v from v-old to v-new = f(v-old), and send M_1 to q_1, M_2 to q_2, etc.' by an input operation rec(q,p)M with no precondition, and postcondition s.v = f(s'.v), s.mess = s'.mess $\cup$ $\{(p,q_1)M_1,(p,q_2)M_2,\ldots\}$. Also we have, for example, an output operation send$(p,q_1)M_1$ with precondition $(p,q_1)M_1$ $\in$ s'.mess and postcondition s.mess = s'.mess $-$ $(p,q_1)M_1$. Thus our model does not send out messages atomically on receipt of a trigger message, but rather places them in a multiset of outgoing messages, and sends them at some later time. We note that this difference is not important for the correctness of the algorithm. After all, even in the interrupt-driven model, the time of message receipt is delayed arbitrarily, and so additional uncertainty, about the delay before the message is sent, does not cause trouble.

Some other differences between our presentation of the algorithm and the original version in [Aw] should be mentioned. The first is that we have 'hard-wired' the distinction between the leader of a cluster and other nodes, while Awerbuch gives a uniform algorithm for every node that branches, depending on whether or not the node is a leader. Also Awerbuch uses several subroutines that are called from different places, whereas we have included these 'in-line' at every occurrence. Another minor difference is that the events that we call CLUSTERGO(C,i) and CLUSTEROK(C,i), and treat as operations of the leader of cluster C, are regarded by Awerbuch as the leader sending itself a message (PULSE and CLUSTERSAFE, respectively). None of these differences is at all significant for the correctness or performance of the algorithm.

There is one respect, however, in which our algorithm is significantly altered from the one given by Awerbuch. In that version, each node delayed sending the READY message to its parent until it had received the CLUSTERSAFE message for its own cluster, as well as the CLUSTERSAFE message for every neighboring cluster along a preferred edge and the READY message from every child. In contrast, we allow the READY messages to be sent without waiting for the cluster itself to be safe. Instead we check only at the leader, before commencing the broadcast of PULSE messages. We therefore use only the subtree containing special nodes, rather than the whole tree, for the convergecast. Similarly, the CLUSTERSAFE messages are broadcast only over the subtree of special nodes. This alteration does not affect correctness, and may improve running time by allowing the convergecast of READY messages to overlap the broadcast of CLUSTERSAFE messages. It may also reduce the number of messages sent. The change also makes the verification simpler, as it increases the degree of independence between the inter- and intracluster synchronization.

Appendix II: Detailed Code for the Divided Algorithm

Non-leader node: NDCS(p)

Inputs:

rec(q,p)READY(q,i) for q ∈ children(p), i positive

rec(q,p)CLUSTERSAFE(q,i) for q ∈ Preferred(p) or q = parent(p), i positive

Outputs:

send(p,q)READY(p,i) for q = parent(p), i positive

send(p,q)CLUSTERSAFE(p,i) for q ∈ children(p) ∪ Preferred(p), i positive

state:

array CLUSTERSAFErec[q,i], initially all false

array READYrec[q,i], initially all false

multiset mess, initially empty

transitions:

rec(q,p)READY(q,i)

Postconditions

  s.READYrec[q,i] = true

  if q ∈ specialchildren(p)

    and (s'.READYrec[q',i] = true for all q' ∈ (specialchildren(p)-{q})) and (s'.CLUSTERSAFErec[q',i] = true for all q' ∈ Preferred(p))

then s.mess = s'.mess ∪ {(p,parent(p))READY(p,i)}

rec(q,p)CLUSTERSAFE(q,i)

Postconditions

s.CLUSTERSAFErec[q,i] = true

if q = parent(p)

then s.mess = s'.mess ∪ {(p,p')CLUSTERSAFE(p,i) : p' ∈ specialchildren(p) ∪ Preferred(p)}

if q ∈ Preferred(p)

and (s'.READYrec[q',i] = true for all q' ∈ specialchildren(p))

and (s'.CLUSTERSAFErec[q',i] = true for all q' ∈ (Preferred(p)-{q}))

then s.mess = s'.mess ∪ {(p,parent(p))READY(p,i)}

send(p,q)READY(p,i)

Preconditions

(p,q)READY(p,i) ∈ s'.mess

Postconditions

s.mess = s'.mess − {(p,q)READY(p,i)}

send(p,q)CLUSTERSAFE(p,i)

Preconditions

(p,q)CLUSTERSAFE(p,i) ∈ s'.mess

Postconditions

s.mess = s'.mess − {(p,q)CLUSTERSAFE(p,i)}

Leader: LECS(C)

Inputs:

CLUSTEROK(C,i) for i positive

rec(q,p)READY(q,i) for p = leader(C), q ∈ children(p), i positive rec(q,p)CLUSTERSAFE(q,i) for $p = \text{leader}(C)$, $q \in \text{preferred}(p)$, i positive

Outputs:

CLUSTERGO(C,i) for i positive

send(p,q)CLUSTERSAFE(p,i) for $p = \text{leader}(C)$, $q \in \text{children}(p) \cup \text{preferred}(p)$, i positive

state:

array READYrec[q,i], initially all false

array CLUSTERSAFErec[q,i], initially all false

array CLUSTERGOsent[i], initially all false

multiset mess, initially empty

transitions:

rec(q,p)READY(q,i)

Postconditions

  s.READYrec[q,i] = true

rec(q,p)CLUSTERSAFE(q,i)

Postconditions

  s.CLUSTERSAFErec[q,i] = true

CLUSTEROK(C,i)

Postconditions

  s.mess = s'.mess $\cup$ {(p,q)CLUSTERSAFE(p,i) : $q \in (\text{specialchildren}(p) \cup \text{Preferred}(p))$}

CLUSTERGO(C,i)

Preconditions

  i = 1 or ((s'.READYrec[q,i-1] = true for all $q \in \text{specialchildren}(p)$)

    and (s'.CLUSTERSAFErec[q,i-1] = true for all $q \in \text{Preferred}(p)$))

  i = 1 or s'.CLUSTERGOsent[i-1] = true

  s'.CLUSTERGOsent[i] = false

Postconditions

s.CLUSTERGOsent[i] = true

send(p,q)CLUSTERSAFE(p,i)

Preconditions

(p,q)CLUSTERSAFE(p,i) $\in$ s'.mess

Postconditions

s.mess = s'.mess $-$ {(p,q)CLUSTERSAFE(p,i)}

Tree Link: LICS(p,q)

If q $\in$ children(p), this is a link automaton from p to q for the messages CLUSTERSAFE(p,i). If q = parent(p), this is a link automaton from p to q for the messages READY(p,i). If (p,q) is a preferred edge, this is a link automaton from p to q for the messages CLUSTERSAFE(p,i). Otherwise, this is a link automaton for no messages.

Non-leader node: NDSL(p)

Inputs:

OK(p,i) for i positive

rec(q,p)SAFE(q,i) for q $\in$ children(p), i positive

rec(q,p)PULSE(q,i) for q = parent(p), i positive

Outputs:

GO(p,i), for i positive

send(p,q)SAFE(p,i) for q = parent(p), i positive

send(p,q)PULSE(p,i) for q $\in$ children(p), i positive

state:

array OKrec[i], initially all false

array GOsent[i], initially all false

array SAFErec[q,i], initially all false

array pulse[i], initially all false

multiset mess, initially empty

transitions:

OK(p,i)

Postconditions

s.OKrec[i] = true

if (s'.SAFErec[q,i] = true for all q $\in$ children(p))

then s.mess = s'.mess $\cup$ {(p,parent(p))SAFE(p,i)}

rec(q,p)SAFE(q,i)

Postconditions

s.SAFErec[q,i] = true

if (s'.SAFErec[q',i] = true for all q' $\in$ children(p)-{q}

and s'.OKrec[i] = true)

then s.mess = s'.mess $\cup$ {(p,parent(p))SAFE(p,i)}

rec(q,p)PULSE(q,i)

Postconditions

s.pulse[i] = true

s.mess = s'.mess $\cup$ {(p,p')PULSE(p,i) : p' $\in$ children(p)}

GO(p,i)

Preconditions

s'.pulse[i] = true

i = 1 or s'.GOsent[i-1] = true

s'.GOsent[i] = false

Postconditions

s.GOsent[i] = true

send(p,q)SAFE(p,i)

Preconditions

(p,q)SAFE(p,i) $\in$ s'.mess

Postconditions

s.mess = s'.mess $-$ {(p,q)SAFE(p,i)}

send(p,q)PULSE(p,i)

Preconditions

(p,q)PULSE(p,i) $\in$ s'.mess

Postconditions

s.mess = s'.mess $-$ {(p,q)PULSE(p,i)}

Leader: LESL(C)

Inputs:

OK(p,i) for p = leader(C), i positive

CLUSTERGO(C,i) for i a number

rec(q,p)SAFE(q,i) for p = leader(C), q $\in$ children(p), i positive

Outputs:

GO(p,i), for p = leader(C), i positive

CLUSTEROK(C,i) for i positive

send(p,q)PULSE(p,i) for p = leader(C), q $\in$ children(p), i positive

state:

array OKrec[i], initially all false

array GOsent[i], initially all false

array SAFErec[q,i], initially all false

array CLUSTERGOrec[i], initially all false

array clustersafe[i], initially all false

array pulse[i], initially all false

array CLUSTEROKsent[i], initially all false

multiset mess, initially empty

transitions:

OK(p,i)

Postconditions

s.OKrec[i] = true

if (s'.SAFErec[q,i] = true for all q ∈ children(p))

then (s.clustersafe[i] = true

if (s'.SAFErec[q,i] = true for all q ∈ children(p)

and s'.CLUSTERGOrec[i+1] = true)

then (s.mess = s'.mess ∪ {(p,q)PULSE(p,i+1) : p ∈ children(p)}

and s.pulse[i+1] = true))

rec(q,p)SAFE(q,i)

Postconditions

s.SAFErec[q,i] = true

if (s'.SAFErec[q',i] = true for all q' ∈ children(p)-{q}

and s'.OKrec[i] = true)

then s.clustersafe[i] = true

if (s'.SAFErec[q',i] = true for all q' ∈ children(p)-{q}

and s'.OKrec[i] = true and s'.CLUSTERGOrec[i+1] = true)

then (s.mess = s'.mess ∪ {(p,q)PULSE(p,i+1) : p ∈ children(p)}

and s.pulse[i+1] = true)

CLUSTERGO(C,i)

Postconditions

s.CLUSTERGOrec[i] = true

if (i = 1 or s'.clustersafe[i-1] = true)

then (s.mess = s'.mess ∪ {(p,p')PULSE(p,i) : p' ∈ children(p)}

and s.pulse[i] = true)

GO(p,i)

Preconditions

s'.pulse[i] = true

i = 1 or s'.GOsent[i-1] = true

s'.GOsent[i] = false

Postconditions

s.GOsent[i] = true

CLUSTEROK(C,i)

Preconditions

s'.clustersafe[i] = true

s'.CLUSTEROKsent[i] = false

Postconditions

s.CLUSTERTOKsent[i] = true

send(p,q)PULSE(p,i)

Preconditions

(p,q)PULSE(p,i) $\in$ s'.mess

Postconditions

s.mess = s'.mess $-$ {(p,q)PULSE(p,i)}

Tree Link: LISL(p,q)

If q $\in$ children(p), this is a link automaton from p to q for the messages PULSE(p,i). If q = parent(p), this is a link automaton from p to q for the messages SAFE(p,i). Otherwise, this is a link automaton for no messages.