Report Number: WUCS-92-10

1992

Hierarchical Correctness Proofs for Recursive Distributed Algorithms using Dynamic Process Creation

Bala Swaminathan and Kenneth J. Goldman

We present a new proof methodology that uses dynamic process creation to capture the structure of recutsive distributed algorithms> Each recursive invocation of a distributed algorithm is modeled as a separate process, encouraging local reasoning about the individual recursive invocations and making explicit the communicatino that takes place among the concurrently executing invocations. Our methodology involves the construction of hierarchical correctness proofs in which the state of each individual call in a refined algorithm is mapped to the state of a corresponding call in a simpler or more abstract algorithm. Algorithm optimizations that result in the creation of fewer... Read complete abstract on page 2.

Hierarchical Correctness Proofs for Recursive Distributed Algorithms using Dynamic Process Creation

Bala Swaminathan and Kenneth J. Goldman

Complete Abstract:

We present a new proof methodology that uses dynamic process creation to capture the structure of recutsive distributed algorithms> Each recursive invocation of a distributed algorithm is modeled as a separate process, encouraging local reasoning about the individual recursive invocations and making explicit the communicatino that takes place among the concurrently executing invocations. Our methodology involves the construction of hierarchical correctness proofs in which the state of each individual call in a refined algorithm is mapped to the state of a corresponding call in a simpler or more abstract algorithm. Algorithm optimizations that result in the creation of fewer recursive calls are treated cleanly in the hierarchical proofs with the use of a hiding operator that makes explicit exactly which recursive calls of the abstract algorithms are optimized away in the refined algorithm. The proof methodology is presented and illustrated in the context of an extended example, the cloture voting Byzantine agreement algorithm of Berman, Garay and Perry. Dynamic process creation is used to capture the recursive structure of the cloture voting agorithm, and a complete hierarchical correctness proof for teh algorithm is given. The structure provided by dynamic process creation and the hierarchical correctness proof provides considerable insight into this rather complicated distributed algorithm.

Hierarchical Correctness Proofs for Recursive Distributed Algorithms using Dynamic Process Creation

Bala Swaminathan
Kenneth J. Goldman

WUCS-92-10

Revised April 1993

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130–4899

Hierarchical Correctness Proofs for Recursive Distributed Algorithms using Dynamic Process Creation*

Bala Swaminathan
bs@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO 63130

Kenneth J. Goldman†
Department of Computer Science
Washington University
St. Louis, MO 63130

April 27, 1993

Abstract

We present a new proof methodology that uses dynamic process creation to capture the structure of recursive distributed algorithms. Each recursive invocation of a distributed algorithm is modeled as a separate process, encouraging local reasoning about the individual recursive invocations and making explicit the communication that takes place among the concurrently executing invocations. Our methodology involves the construction of hierarchical correctness proofs in which the state of each individual call in a refined algorithm is mapped to the state of a corresponding call in a simpler or more abstract algorithm. Algorithm optimizations that result in the creation of fewer recursive calls are treated cleanly in the hierarchical proofs with the use of a hiding operator that makes explicit exactly which recursive calls of the abstract algorithm are optimized away in the refined algorithm.

The proof methodology is presented and illustrated in the context of an extended example, the cloture voting Byzantine agreement algorithm of Berman, Garay and Perry. Dynamic process creation is used to capture the recursive structure of the cloture voting algorithm, and a complete hierarchical correctness proof for the algorithm is given. The structure provided by dynamic process creation and the hierarchical correctness proof provides considerable insight into this rather complicated distributed algorithm.

Keywords: distributed computing, recursion, dynamic process creation, Byzantine agreement, input/output automata, hierarchical proofs, possibilities mappings

---

*This research was supported in part by the National Science Foundation under Grant CCR-91-10029.

†Contact author: Kenneth J. Goldman, Department of Computer Science, Campus Box 1045, Washington University, One Brookings Drive, St. Louis, MO 63130-4899, (314) 935-7542, kjg@cs.WUSTL.EDU

1 Introduction

Recursion is a powerful technique for designing sequential algorithms that are both efficient and easy to express. Therefore, it is not surprising that recursion has found its way into distributed algorithms. For example, Peterson's tournament tree algorithm for n-process mutual exclusion can be viewed as the recursive instantiation of a two-process mutual exclusion algorithm [16]. Other examples of recursive distributed algorithms include the original Byzantine agreement algorithm of Lamport, Pease and Shostak [23] and, more recently, the cloture voting algorithm for Byzantine agreement of Berman, Garay and Perry [7].

What *is* surprising is that relatively few distributed algorithms have actually been designed using the recursive paradigm. In part, this may be reflection on the problem domain, but the difficulty involved in reasoning about recursive distributed algorithms may be largely to blame. In this paper, we argue that if one views recursive distributed algorithms in terms of dynamic process creation, then it is easier to make sense of recursive distributed algorithms. We introduce a proof methodology, based on dynamic process creation, that offers a new way to describe, understand and reason about recursive distributed algorithms.

Dynamic process creation, the ability to create new processes in a system in the course of performing a computation, is virtually ubiquitous in both programming languages and models for distributed computing. Dynamic process creation, in its simplest form, is handled in programming languages with a system call, such as the UNIX **fork**, that spawns a new process. Some programming languages provide language constructs, such as Ada's **task** and task access type, that provide a higher level mechanism for creating new processes at run-time. Taken to its logical extreme, dynamic process creation can be used to partition a given task into multiple subtasks, each handled by a separate process for greater concurrency and improved modularity. When nested in this way, dynamic process creation not only yields efficient programs that are easier to understand, but also provides a useful mechanism for masking failures in a distributed system. This kind of nesting can be seen in systems such as ISIS [8] and Argus [25], and has been studied formally [26].

In addition to the work in programming languages supporting dynamic process creation, there has also been a great deal of interest in developing models and proof systems to support reasoning about systems with dynamic process creation. For example, CCS is a mathematical model of

concurrent composition where the number of new processes and their communication can change dynamically [29]. CCS allows a single composition to create an unbounded number of processes. The Actor model [1] provides dynamic process creation by allowing an actor to giving a continuation leading to more than one actor. Formal proof systems that accommodate dynamic process creation have been developed. For example, [3] provides a formal proof system for a simplified version of the parallel programming language called POOL [2] where parallelism is modeled by dynamic process creation.

The reason dynamic process creation is so prevalent in programming languages and models is that it is so convenient for structuring large concurrent systems. Within distributed computing, dynamic process creation is useful because it provides a natural mechanism for handling both local and remote subroutine calls. Subroutines have appeared in distributed algorithms in various forms. The simplest and most standard form is a subroutine invoked as part of the local computation of one processor in the system. More ingenious uses involve one distributed algorithm calling another (different) distributed algorithm as a subroutine. For example, the drinking philosophers algorithm of Chandy and Misra [9] can be understood in terms of a generalized dining philosophers algorithm that is called as a subroutine by the drinking philosophers algorithm in order to establish priority for the resources and prevent starvation [35]. In this paper, we are concerned with a third type of subroutine call in which a particular distributed algorithm calls *itself* to solve a smaller problem. This is what we mean by a *recursive distributed algorithm.*

Recursive distributed algorithms differ substantially from recursive sequential algorithms, and these differences make reasoning about recursive distributed algorithms much more difficult than reasoning about their sequential counterparts. The most important difference is that in recursive distributed algorithms, recursive calls at different levels in the recursion may proceed in parallel and even communicate information to each other as they execute. So, although one would like to reason locally about each recursive call, one cannot consider a single invocation of a recursive distributed algorithm executing in isolation, but must consider the interactions that may occur among the different invocations. In fact, it is sometimes the case that these interactions are seemingly so important that algorithm designers find it necessary to describe their algorithms in terms of a single global state and a single thread of control, rather than in terms of separate threads and local states for each recursive calls. Unfortunately, this makes local reasoning even more difficult. Therefore,

it would be desirable to have a methodology for describing and proving the correctness of recursive distributed algorithms that would enable us to reason locally about each recursive invocation of the algorithm and to make explicit the information that is shared among concurrently executing invocations.

In order to accomplish this, we advocate a methodology that exploits dynamic process creation for describing recursive distributed algorithms in a structured way and for constructing careful proofs of their correctness. The methodology pulls together several different techniques, principally dynamic process creation [24] and hierarchical proofs [27], and brings them to bear on the problem of studying recursive distributed algorithms. Using dynamic process creation to model recursive calls in a distributed system allows each recursive call to be studied in isolation. This permits local reasoning about the individual invocations. At the same time, it is possible to treat information sharing among different recursive calls on the same processor using explicit accesses to shared variables. This makes it clear exactly what information is being shared among the calls. Hierarchical correctness proofs can be constructed that relate each invocation in a refined efficient algorithm to a corresponding invocation in a more abstract inefficient algorithm. Furthermore, optimizations that result in fewer recursive calls in the refined algorithm can be treated in the hierarchical proof with the use of a hiding operator. Our methodology may be summarized as the following sequence of steps.

1. State a careful specification of the problem in terms of the interactions between a module and its environment.

2. Describe a simple (but probably inefficient) recursive algorithm in terms of dynamic process creation. Ideally, this simple algorithm should not require shared information among different invocations on the same processor, except for information transmitted as parameters on invocation (passed down the recursion tree) or results returned (passed up the recursion tree).

3. Prove that the algorithm given in (2) meets the specification given in (1). This proof may be carried out using a variety of techniques, but when the information flow is restricted as described in (2), the dynamic process creation structure is well-suited to a simple inductive argument and local reasoning about each of the recursive calls.

4. Describe an efficient (but more complicated) recursive algorithm in terms of dynamic process

creation. Describe any information sharing among different invocations on a the same processor (other than parameter and return values) in terms of atomic accesses to shared variables. To simplify later reasoning, variable sharing should be kept to a minimum.

5. Construct a *possibilities mapping*[1] from the states of each of the processes in the complicated efficient algorithm to the states of the *corresponding* processes in the simple inefficient algorithm. That is, structure the mapping on a process-by-process basis that allows one to see the correspondences in the two systems.

6. Prove that the mapping constructed in (5) implies that the efficient algorithm "simulates" the simple algorithm at the level of the interactions that occur between the algorithm and its environment. Since the refined algorithm is more efficient, fewer recursive calls may occur in the refined algorithm than in the simple one. These optimizations may be handled in the proof by a hiding operator that makes explicit exactly where these optimizations take place.

Note that, as in any hierarchical proof, steps (4), (5), and (6) may be repeated for successively more refined algorithms, resulting in a proof that is completed in several stages.

Because dynamic process creation is available in many programming languages and models for distributed computing, it was not necessary for us to construct a new model in order to develop and illustrate our methodology. Instead, we have adopted the I/O automaton model of Lynch and Tuttle [27, 28] with the shared memory extensions of Goldman and Lynch [18]. Reasons for choosing the I/O automaton model include the fact that it is well-suited for modeling distributed systems and provides mechanisms for constructing precise problem specifications, writing unambiguous algorithm descriptions, and constructing careful correctness proofs. However, our primary reason for choosing the I/O automaton model is that it allows us to build upon previous work in the area of hierarchical proof techniques for distributed algorithms [27, 34, 13], as well as work on modeling dynamic process creation [24, 26], that has been carried out within that model. We emphasize, though, that the methodology we describe is not necessarily limited to the context of this particular model. In fact, any model that supports dynamic process creation, supports both message passing and shared memory, and is amenable to hierarchical proof techniques (or proofs by refinement) should be a suitable context for the methodology we describe.

[1] Possibilities mapping is a kind of state mapping (see Section 2.7).

We illustrate our methodology by presenting a structured description and complete hierarchical correctness proof for the cloture voting Byzantine agreement algorithm of Berman, Garay and Perry [7]. In Byzantine agreement protocols, a set of non-faulty processors must agree on a value in the presence of arbitrary messages sent by faulty processors. In the simplest of the protocols, namely the EIG (*exponential information gathering*) protocol, the algorithm runs in a series of "rounds". Initially, each processor exchanges its value with every other processor and creates one node in a "computation tree" for each value received. In the following rounds values received in the earlier rounds are exchanged and the trees grow. At the end, each processor determines its final value using a "bottom-up" majority taking scheme. The ESFM (*early stopping and fault masking*) protocol [7] is more efficient and more complicated than EIG, but not as complicated nor as efficient as the cloture voting algorithm. ESFM is an optimized version of EIG where the computation tree does not necessarily expand completely. The final values of some nodes are predicted and once a processor discovers that another processor is faulty, it "pretends" that all values reported by this faulty processor are 0. In the cloture voting protocol [7], each non-faulty processor runs several "debates" of decreasing size concurrently with a main process that secures the votes from these debates. Each debate is similar to an ESFM protocol. Although there are multiple debates in each processor, each debate exchanges messages only with the corresponding debates in other processors in order to grow its computation tree. A new debate is initiated at each round. The first debate is started at each processor with the initial value assigned to that processor by the environment. If it can be determined that there has been sufficient communication to safely assume that both the values 0 and 1 are represented as initial values of non-faulty processors, then a subsequent debate may be started with initial value 0. Otherwise, each subsequent debate is started with initial value 1. So, if any debate returns 0, then a default decision value may be taken immediately. When all the debates on a given processor return, then that processor can determine the agreed-upon value.

Berman, Garay and Perry do not present the recursive structure of their algorithm in terms of dynamic process creation. Rather, in their formulation, each debate is a single process that maintains all information and handles all interaction at all levels in the computation tree. Here, we reformulate their algorithm by treating each node in the computation tree as a separate process that is created by its parent in the tree and returns a result to that parent. That is, we use dynamic process creation to model each recursive call within a debate rather than representing the state of

the computation as one large data structure kept by a single process. Note, however, that we have not changed the algorithm itself. Rather, we have exposed the recursive structure of the algorithm and described it in terms of dynamic process creation.

Because our proof is carried out in several stages, it is *longer* than the proof given by Berman, Garay, and Perry [7]. However, we will demonstrate that considering the cloture voting algorithm in terms of our methodology yields several benefits. Dynamic process creation makes the structure of the algorithms more explicit; it is easier to see what information is used by each recursive call since that information is kept by a separate process. In addition, representing each recursive call by a separate automaton permits us to construct possibilities mappings that relate the corresponding automata in two systems while hiding the actions of those automata representing processes that are "optimized away" in the more efficient algorithms. Also, modeling the recursive calls as separate automata permits us greater flexibility in constructing our simulation arguments, where we rely on local commutativity arguments to permit the reordering of actions within a given automaton's execution. Such reordering is used to construct canonical equivalent executions that simplify our arguments. Modeling each recursive call separately also permits us to make explicit the pattern of communication between recursive calls in different processors. We describe the algorithms and carry out our proofs in an asynchronous setting, and then make explicit a set of conditions under which the executions of the cloture voting protocol are correct. We then show that these conditions are guaranteed in all synchronous executions. In this way, we make explicit exactly how synchronization contributes to the algorithm's correctness.

The remainder of the paper is organized as follows. In Section 2, we provide a review of the I/O automaton model, including a discussion of possibilities mappings and the mechanisms for dynamic process creation and shared memory. Then, we begin our extended example in Section 3 with a brief statement of the Byzantine agreement problem. This is followed by our hierarchical correctness proof, which parallels the development of the cloture algorithm as presented by Berman, Garay and Perry.

The proof is carried out in several stages. We begin with the modeling of the EIG algorithm. In Section 4, we recast the EIG algorithm in terms of dynamic process creation and provide a formal description of the EIG algorithm in the I/O automaton model. We then prove that the EIG algorithm solves the Byzantine agreement problem. The EIG algorithm is used as the basis of our

hierarchical proof. That is, if a system simulates (in a formal sense to be explained later) an EIG system, then that system solves the Byzantine agreement problem.

In Section 5, we describe the ESFM algorithm. The algorithm is again recast using dynamic process creation to model the recursive structure. ESFM is then proved to be correct using a *possibilities mapping*, a particular kind of state mapping (see Section 2). By exhibiting a possibilities mapping from the states of ESFM to the states of EIG, we show that ESFM systems "simulate" EIG systems, and therefore solve the Byzantine agreement problem. To simplify the proof, we exhibit the possibilities mapping only for a restricted set of canonical executions, and prove based on local commutativity arguments that every ESFM execution has an equivalent canonical execution. In the proof that our mapping is indeed a possibilities mapping, we use a hiding technique to capture the optimizations used in the ESFM algorithm. That is, we hide the "extra" steps taken by the EIG protocol to "keep up" with ESFM. This hiding operator makes explicit exactly where the optimizations of ESFM result in savings over the EIG protocol, and takes advantage of the fact that we model each recursive call as a separate automaton.

Finally, in Section 6, we present the cloture algorithm, reformulated in terms of dynamic process creation, and its proof of correctness. The hierarchical proof is completed with a possibilities mapping from the cloture protocol to ESFM. The cloture voting algorithm gains efficiency by initiating a series of "debates" of decreasing size. Each debate is essentially another instantiation of the ESFM protocol. In order to permit local reasoning about each of the separate "debates" in the cloture voting algorithm, we introduce an *oracle* that abstracts away those properties of the rest of the system upon which each of the debates depends. That is, as we consider each debate, we view the remaining debates as being encapsulated within a single oracle that guarantees certain properties. The oracle models the faults reported from other debates and captures the flow of information among many simultaneous instantiations of the algorithm. It is helpful in isolating and reasoning about individual instantiations of the algorithm. We show that every ESFM system with an oracle simulates an "ordinary" ESFM system. Since in an asynchronous system the debates can be created at any time and since the size of successive debates decreases, we stipulate certain conditions and prove the cloture algorithm as a conditional property based on these hypotheses. Given these conditions, we show that each "debate" in the cloture algorithm simulates an ESFM system with an oracle. We then define a synchronous executions of the cloture voting algorithm

and show that the hypotheses are satisfied in all such executions. The correctness proof for the synchronous cloture algorithm follows directly.

2 The I/O Automaton Model

The I/O Automaton model [27, 28] is used to present the algorithms and proofs of correctness. The model provides a composition operator that permits the modular construction of large systems and a specification mechanism for writing precise statements of the problems to be solved by modules in concurrent systems. The model encourages very careful algorithm descriptions, and can be used to construct rigorous correctness proofs, including hierarchical correctness proofs. In addition, the model can be used for carrying out complexity analysis and for proving impossibility results.

The I/O automaton model is significantly different from CCS [30] and CSP [19] in that input and output actions in the I/O automaton model are distinguished, and an I/O automaton cannot block an input action from occurring. In that sense, I/O automata are similar to I/O-systems [20], a model that also provides support for composition and verification using simulation arguments [21, 22].

In this section, we provide a brief introduction to the model, including a discussion of techniques for modeling shared memory and dynamic process creation. The details of the model and these techniques may be found elsewhere [18, 24, 28].

2.1 I/O Automata

An I/O automaton is essentially a nondeterministic (possibly infinite-state) automaton with an action labeling each transition. An automaton's actions are classified as either 'input', 'output', or 'internal'. An automaton can establish restrictions on when it will perform an output or internal action, but it is unable to block the performance of an input action.

Formally, an *action signature* S is a partition of a set $acts(S)$ of *actions* into three disjoint sets $in(S)$, $out(S)$, and $int(S)$ of *input actions*, *output actions*, and *internal actions*, respectively. We denote by $ext(S) = in(S) \cup out(S)$ the set of *external actions*. We denote by $local(S) = out(S) \cup int(S)$ the set of *locally-controlled actions*. An I/O automaton A consists of five components: an action signature $sig(A)$; a set $states(A)$ of *states*; a nonempty set $start(A) \subseteq states(A)$ of *start states*; a transition relation $steps(A) \subseteq states(A) \times acts(A) \times states(A)$ with the property that for

every state s' and input action π there is a transition (s', π, s) in $steps(A)$; and an equivalence relation $part(A)$ partitioning the set $local(A)$ into a countable number of *equivalence classes*. Each class of a relation may be thought of as a separate process. We refer to an element (s', π, s) of $steps(A)$ as a *step* of A. If (s', π, s) is a step of A, then π is said to be *enabled* in s'. Since every input action is enabled in every state, automata are said to be *input-enabled*. This means that an automaton is unable to block its input.

An *execution* of A is a finite sequence $s_0, \pi_1, s_1, \ldots, \pi_n, s_n$ or an infinite sequence $s_0, \pi_1, s_1, \pi_2, \ldots$ of alternating states and actions of A such that $(s_i, \pi_{i+1}, s_{i+1})$ is a step of A for every i, and $s_0 \in start(A)$. The *schedule* of an execution α, denoted $sched(\alpha)$, is the subsequence of α consisting of the actions appearing in α. The *behavior* of an execution or schedule α, denoted $beh(\alpha)$, of A is the subsequence of α consisting of *external* actions. The sets of executions, finite executions, schedules, finite schedules, behaviors, and finite behaviors are denoted $execs(A)$, $finexecs(A)$, $scheds(A)$, $finscheds(A)$, $behs(A)$, and $finbehs(A)$, respectively. A particular action may occur several times in an execution or a schedule; we refer to a particular occurrence of an action as an *event*. An *extended step* of A is a triple $(s_i, \pi_1 \ldots \pi_n, s_{i+n})$ such that there exists states s_j for $i \leq j < i+n$ such that $(s_j, \pi_{j+1-i}, s_{j+1})$ is a step of A for every j.

2.2 Composition

We can construct an automaton that models a complex system by composing automata that model the simpler system components. When we compose a collection of automata, we identify an output action π of one automaton with the input action π of each automaton having π as an input action. Consequently, when one automaton having π as an output action performs π, all automata having π as an action perform π simultaneously (automata not having π as an action do nothing).

Since we require that at most one system component controls the performance of any given action, any collection of automata to be composed must have disjoint sets of locally-controlled actions and no automaton in the collection may have as an input action an internal action of another automaton.

In the *composition* $A = \prod_{i \in I} A_i$ of a compatible collection of automata $\{A_i\}_{i \in I}$, the output actions of A are all the output actions of the components $\{A_i\}_{i \in I}$, the internal actions are all the internal actions of the components, and the input actions are the remaining actions of the

components. Thus, an action that is an output of one component and an input of another becomes an output action of the composition. In our proofs, it is sometimes convenient to hide such actions. If A is an automaton and Σ is a set of output actions of A, then $Hide_\Sigma(A)$ is the automaton differing from A only in that $out(Hide_\Sigma(A)) = out(A) - \Sigma$ and $int(Hide_\Sigma(A)) = int(A) \cup \Sigma$. We also define the set of equivalence classes of the composition to be the union of all the sets of equivalence classes of the component automata.

Each state of the automaton A resulting from the composition is a vector of states of the component automata. Given an execution $\alpha = \vec{s_0}\pi_1\vec{s_1}\ldots$ of A, let $\alpha|A_i$ (read "α projected on A_i") be the sequence obtained by deleting $\pi_j\vec{s_j}$ when $\pi_j \notin acts(A_i)$ and replacing the remaining $\vec{s_j}$ by $\vec{s_j}[i]$, the i^{th} component of the state vector $\vec{s_j}$.

2.3 Fairness

Of all the executions of an I/O automaton, we are primarily interested in the 'fair' executions — those that permit each of the automaton's primitive components (i.e., its classes or processes) to have infinitely many chances to perform output or internal actions. The definition of automaton composition says that an equivalence class of a component automaton becomes an equivalence class of a composition, and hence that composition retains the essential structure of the system's primitive components. In the model, therefore, being fair to each component means being fair to each equivalence class of locally-controlled actions. A *fair execution* of an automaton A is defined to be an execution α of A such that the following conditions hold for each class C of $part$(A):

1. If α is finite, then no action of C is enabled in the final state of α.
2. If α is infinite, then either α contains infinitely many events from C, or α contains infinitely many occurrences of states in which no action of C is enabled.

We denote the set of fair executions of A by $fairexecs(A)$. We say that β is a *fair behavior* of A if β is the behavior of a fair execution of A, and we denote the set of fair behaviors of A by $fairbehs(A)$. Similarly, β is a *fair schedule* of A if β is the schedule of a fair execution of A, and we denote the set of fair schedules of A by $fairscheds(A)$.

2.4 Shared Memory Extensions

The I/O automaton model has been extended to allow modeling of shared memory systems, as well as systems that have both shared memory and shared action communication [18]. Here, we use the shared memory extensions to model data structures shared by processes running within the same processor, whereas message passing is used to model communication between processes running on different processors. A special kind of output action, called a *shared memory action* is used to model an atomic access to shared memory. Each shared memory action contains information that corresponds to the contents of the shared memory before and after the action occurs. Shared memory actions are used by *shared memory automata* to access the shared variables. Each shared memory automaton must be prepared to handle any value it may observe in the shared variables, so the preconditions of an output action may not depend on the values of the shared variables.

Since shared memory automata are special cases of I/O automata, all the I/O automaton model definitions (notably composition and fairness) apply to shared memory automata as well. Shared memory automata operate in a system in which the environment is free to change the contents of the shared memory at any time. However, a *closeout* operator C is provided for taking a shared memory automaton A and a set of variables X and producing a new shared memory automaton $C(A, X)$ in which the given set of variables is made private (absorbed into the local state of the composed automaton). This discussion should be sufficient to understand the use of shared variables in the algorithms presented here. For more details on the shared memory extensions see [18].

2.5 Dynamic Process Creation

To model dynamic process creation, we adopt the technique described in [24]. A system in which processes are created and destroyed can be modeled as the set of all processes (where each process is an I/O automaton) that could possibly be created during any execution of the system. Creation, then, simply "wakes up" the desired automaton, which becomes "alive". For each automaton, a predicate is defined which indicates whether or not it is alive in a given state; the automaton does not actually do anything when it is not alive. Only live automata may take steps; however, when an automaton is not alive it may take a step involving an input action which is a creation action. An additional requirement is that a creation action has no effect if the automaton is already alive.

2.6 Problem Specification

A 'problem' to be solved by an I/O automaton is formalized as a set of (finite and infinite) sequences of external actions. We define a *schedule module* H to consist of two components, an action signature $sig(H)$, and a set $scheds(H)$ of *schedules*. Each schedule in $scheds(H)$ is a finite or infinite sequence of actions of H.

Although the model does not allow an automaton to block its environment or eliminate undesirable inputs, we can formulate our problems (i.e., correctness conditions) to require that an automaton exhibits some behavior only when the environment observes certain restrictions on the production of inputs. A useful notion for discussing such restrictions is that of a module 'preserving' a property of behaviors. Informally, a module *preserves* a property $\mathcal{P}$ iff the module is not the first to violate $\mathcal{P}$: as long as the environment only provides inputs such that the cumulative behavior satisfies $\mathcal{P}$, the module will only perform outputs such that the cumulative behavior satisfies $\mathcal{P}$. A formal definition and a proof that a composition preserves a property if each of the component automata preserves the property are given elsewhere [28].

An automaton is said to *solve* a problem P provided that its set of fair behaviors is a subset of P and the automaton is said to *implement* the problem P if its set of finite behaviors is a subset of P. Sometimes it is sufficient to establish a correspondence between two concurrent systems only from the point of view of each "user" rather than in a global perspective. That is, we want the behavior at each user to be the same in the two systems, but we are not interested in the relative order of actions at different users. Let U be a set of disjoint subsets of the external actions of an automaton B. That is, $U = \{U_i\}_{i \in \mathcal{I}}$ of users for some index set $\mathcal{I}$, where $ext(B) \supseteq \bigcup_{i \in \mathcal{I}} U_i$ and $\forall i, j \in \mathcal{I}, i \neq j \Rightarrow U_i \cap U_j = \emptyset$. We say that an automaton A *simulates* an automaton B *with respect to* users $U = \{U_i\}_{i \in \mathcal{I}}$ if and only if

1. $ext(B) = ext(A)$, and
2. $\forall \beta \in scheds(A), \exists \alpha \in scheds(B)$ such that $\forall i \in \mathcal{I}, \beta | U_i = \alpha | U_i$.

Thus, if one system simulates another, then all users in the first system should see the same sequence of actions they could have seen in some execution of the second system.

Let B be a schedule module or an automaton and let $U = \{U_i\}_{i \in \mathcal{I}}$ be a set of disjoint subsets of the external actions of B. We say that B is *closed under reordering* with respect to U iff

$\forall \beta \in sched(B)$, for every permutation β_u of $\beta|U$, if $\beta_u|U_i = \beta|U_i \ \forall i \in \mathcal{I}$, then there exists a schedule $\beta' \in sched(B)$ such that $\beta_u = \beta'|U$. The following lemma follows from the definitions:

Lemma 1: Let A and B be automata or schedule modules such that $ext(A) = ext(B)$. If B is closed under reordering with respect to U and A simulates B with respect to U, then A solves B.

2.7 Hierarchical Proofs in the I/O Automaton Model

In a hierarchical proof, one shows that the system of interest solves some intermediate system, which in turn solves the problem. We now describe one method for proving that an automaton A solves an automaton B [27]. This method makes use of the notion of a possibilities mapping, a correspondence between the states of the two automata that can be used to prove that A solves B.

Suppose A and B are automata with the same external action signature, and suppose h is a mapping from $states(A)$ to the power set of $states(B)$. The mapping h is said to be a *possibilities mapping* from A to B if the following conditions hold:

1. For every start state a_0 of A, there is a start state b_0 of B such that $b_0 \in h(a_0)$.
2. If a' is a reachable state of A, $b' \in h(a')$ is a reachable state of B, and (a', π, a) is a step of A, then there is an extended step (b', γ, b) of B such that $\gamma|ext(B) = \pi|ext(A)$, and $b \in h(a)$.

If a is a state of A, then a state $b \in h(a)$ in B is referred to as a *possibility* for a. The first condition of possibilities mapping says that every start state of A has as one of its possibilities a start state of B. The second condition says that steps of A and B preserve possibilities. By exhibiting a possibilities mapping we prove that A implements B. When every fair execution is finite (as in the systems in this paper) and A implements B, it follows that A solves B. The interested reader is referred to [28].

Sometimes, we are interested in showing a correspondence not for all executions of a system, but only for a restricted subset of its executions. Therefore, for a system A with execution α, we say that h is a *possibilities mapping from A to a system B for execution α* iff

1. if a_0 is the initial state of α, then there is a start state b_0 of B such that $b_0 \in h(a_0)$, and
2. if a' is a state of α, $b' \in h(a')$ is a reachable state of B, and (a', π, a) is a step in α, then there is an extended step (b', γ, b) of B such that $\gamma|ext(B) = \pi|ext(A)$, and $b \in h(a)$.

If Φ is the subset of executions of A of interest and h is a possibilities mapping from A to B for all $\alpha \in \Phi$, then we know that for all behaviors of A of interest, there exists an execution of B with the same behavior. One may think of this restricted set of executions as an external constraint on the set of reachable states in the original definition.

The following corollary follows immediately from the definitions.

Corollary 2: Let h be a possibilities mapping from a system A to another system B for execution α for all $\alpha \in \Phi$, where $\Phi \subseteq execs(A)$. Let U be a set of disjoint subsets of $ext(A)$ (i.e. U is the set of "users" of A). If $\forall \alpha' \in execs(A), \exists \alpha \in \Phi$ such that $sched(\alpha')$ is a permutation of $sched(\alpha)$ and for all $u \in U$ $sched(\alpha')|u = sched(\alpha)|u$, then A simulates B.

3 The Byzantine Agreement Problem

In the Byzantine agreement problem [23], a set of non-faulty processors must agree on a value in the presence of arbitrary messages sent by faulty processors. More precisely, consider n processors, t of which may be faulty. If we let P denote the set of processors and T denote the set of faulty processors, then the set of processors $NF = P - T$ are called *non-faulty*. We assume that the non-faulty processors initially do not know the set T. Each processor begins with an initial value, either 0 or 1, and processors must decide on final values according to the following conditions:

- *agreement*: If two non-faulty processors decide on final values v and v', then $v = v'$.
- *validity*: If all non-faulty processors start with the same initial value v, then v is the only allowable decision by a non-faulty processor.
- *termination*: All non-faulty processors eventually decide on a final value.

Faulty processors may behave maliciously in order to confuse the rest of the system. In other words, they may send conflicting messages to different non-faulty processors in order to "fool" them into deciding on different final values.

Typical measures for the complexity of a Byzantine agreement algorithm include resiliency (the number t of faulty processors tolerated in a system of n processors), time (the number of "rounds" r of message exchange required), and the maximum message size m. The Byzantine agreement

problem has been studied extensively. Lower bounds on resiliency ($n > 3t$) [23, 15][2], time ($r = t+1$ rounds) [12, 14], and message size ($m = 1$) are known. So far, no algorithm has achieved all three of these bounds, but several algorithms have achieved two of the three, or have come close to all three bounds [4, 5, 6, 7, 10, 11, 12, 17, 23, 31, 33]. For a comparison of these algorithms in terms of these complexity measures, see [11] and [17]. The cloture voting protocol, the central example of this paper, works when $n > 4t$, uses $t+1$ rounds of communication, and message size polynomial in n. These results hold for synchronous protocols only, and it has been shown that every completely asynchronous protocol for this problem has the possibility of nontermination, even with only one faulty processor [16].

3.1 Schedule Module B

We now define Schedule Module B, a specification of the Byzantine Agreement problem. The schedule module captures the problem to be solved by the set of non-faulty processors in the face of an environment that controls the initiation of the protocol at each non-faulty processor, the scheduling of all message delivery events, and the contents of all messages delivered to non-faulty processors on behalf of faulty processors. Well-formedness and liveness requirements of the environment are stated explicitly as part of the specification. The problem is formulated in terms of asynchronous computation. Later, we identify conditions (guaranteed in synchronous executions) under which the problem is solvable. An illustration of the module B and its relationship to the environment is shown in Figure 1.

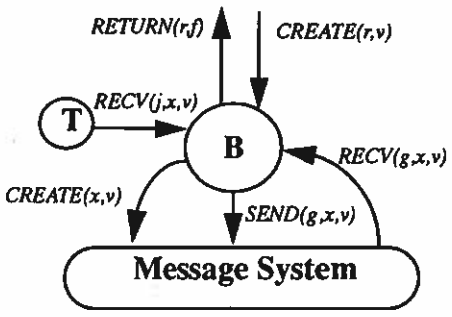

Figure 1: Schedule module B and its environment.

[2]Although randomized algorithms can be used to solve Byzantine agreement with 'high' probability, no deterministic algorithm can solve Byzantine agreement when more than one third of the processes are faulty [23, 15].

Because we are working in an asynchronous setting, we need to place liveness conditions on the behaviors of the faulty processors. Faulty processors must continue to send messages *as if* they are participating in the algorithm (but may send conflicting messages to different processors). However, the faulty processors must still adhere to the communication patterns of a given algorithm; otherwise they could be trivially detected as faulty. Since different algorithms have different patterns of communication, we would like a general mechanism for defining the liveness conditions. For this, we introduce the following definitions of node-names, root-names, and peers.

Recall that P is the set of processor names, with $T \subset P$ the set of names of faulty processors and $NF = P - T$ the set of names of non-faulty processors. Let *node-names* be an arbitrary set of names that we will use to identify automata in our system. Let *root-names* be a set of names containing exactly one element for each $g \in NF$, where we write $root\text{-}name_g$ to denote the particular element of *root-names* corresponding to g. We write *node* x or *automaton* x to refer to the automaton with name x. If $x \in$ *root-names*, then node x is called a *root node*. We will require the environment to create each root node to initiate the algorithm at each processor. The other nodes are used to model the processes created dynamically as the execution unfolds. Let *peers* be a mapping from *node-names* to the power set of P. For a node x, $peers(x)$ denotes the set of all processors from which x is willing to accept a message. Let $\mathcal{M}$ be the universal set of messages. This set captures the allowable set of messages that may be exchanged between the nodes in an algorithm.

Since we want the behavior of a faulty processor to be constrained by the choice of the algorithm, we do not fix the values for node-names, root-names, peers, and $\mathcal{M}$ here; instead we treat them generally in our problem statement. Later, when we present specific algorithms, these sets will be given concrete values. Now the signature of B can be given.

Input actions:

$CREATE(r, v, P, t)$ where $r \in$ *root-names*, $v \in \{0, 1\}$

$RECV(h, x, \mu)$ where $x \in$ *node-names*, $h \in peers(x)$, $\mu \in \mathcal{M}$

Output actions:

$RETURN(r, v)$ where $r \in$ *root-names*, $v \in \{0, 1\}$

$CREATE(x, v, Q, m)$ where $x \in$ *node-names* $-$ *root-names*, $v \in \{0, 1\}, Q \subseteq P, m \leq t$ or $m = \perp$

$SEND(g, x, \mu)$ where $g \in NF$, $x \in$ *node-names*, $\mu \in \mathcal{M}$

We have *CREATE* and *RETURN* actions for every non-faulty processor. The $CREATE(r, v, P, t)$ action with $r = root\text{-}name_g$ is under the control of the environment and initiates the algorithm at processor g with initial value v; the set of processors P involved in the consensus and the number of faulty processors t are also communicated by this action. The $RETURN(r, v)$ action with $r = root\text{-}name_g$ is used by non-faulty processor g to report its final value v.

The $CREATE(x, v, Q, m)$ actions not only are used internally to invoke recursive calls in the algorithm by awakening nodes in the computation tree within a processor but also serve as notifications to the message system that a new automaton x has been awakened, and may now receive messages. The parameters Q and m denote the set of all processors and the number of faulty processors respectively. Each $SEND(g, x, \mu)$ action informs the message system to deliver a value μ to the node x from a node in the non-faulty processor g. The corresponding $RECV(g, x, \mu)$ action is output by the message system to deliver that message. The other $RECV(j, x, \mu)$ actions, with $j \in T$, are the arbitrary messages delivered on behalf of the faulty processors and are under the control of the environment. Note that we explicitly model neither the message system nor the faulty processors, but simply define the weakest possible restrictions on their behaviors. We define an arbitrary sequence β of actions of the schedule module B to be *well-formed* iff

(wf1) $\forall x \in$ *node-names* there exists at most one $CREATE(x, v, Q, m)$ and at most one $RETURN(x, v')$ in β, and the *RETURN* action must occur later than the corresponding *CREATE*,

(wf2) $\forall g \in NF, \forall x \in$ *node-names* there exists at most one action in β of the form $SEND(g, x, v)$ for $v \in \{0, 1\}$, and if $x \in$ *root-names*, then for each $\mu \in \mathcal{M}$, there exists at most one $SEND(g, x, \mu)$ action in β,

(wf3) $\forall h \in P, \forall x \in$ *node-names* there exists at most one action in β of the form $RECV(h, x, v)$ for $v \in \{0, 1\}$, and if $x \in$ *root-names*, then for each $\mu \in \mathcal{M}$, there exists at most one $RECV(h, x, \mu)$ action in β,

(wf4) $\forall j \in NF, \forall x \in$ *node-names* if $RECV(j, x, \mu)$ occurs in β, then the corresponding $SEND(j, x, \mu)$ occurs earlier in β.

The well-formedness conditions say that every node is created at most once, returns only after it is created, there is only one message, with $v \in \{0, 1\}$, for a given source-destination pair, and that

no duplicate messages are directed toward a root node. We say that an arbitrary sequence β of actions of B is *input-live* iff

(il1) $\forall r \in$ *root-names* there exists a $CREATE(r, v, P, t)$ action in β such that $v \in \{0, 1\}$,

(il2) $\forall x \in$ *node-names*, $\forall j \in peers(x) \cap T$ if a $CREATE(x, v, Q, m)$ occurs in β and if x is not a leaf node, then there exists a $RECV(j, x, v)$ action in β such that $v \in \{0, 1\}$, and

(il3) if a $SEND(g, x, v)$ action occurs in β such that $g \in NF$, $x \in$ *node-names*, and a $CREATE(x, v', Q, m)$ occurs in β, then a corresponding $RECV(g, x, v)$ occurs later than both the *SEND* and *CREATE* actions in β.

The first liveness condition (il1) states that the algorithm is initiated at each processor. The second liveness condition (il2) says that the faulty processors must continue to participate in the algorithm. This is needed since we do not assume that the computation proceeds in synchronized rounds. Finally, (il3) says that messages sent by a non-faulty processor are eventually delivered by the message system. Note that (il3) guarantees delivery even if the destination node is not created until after the message is sent.

Now we can define the set of schedules for B. Let β be a sequence of actions in $sig(B)$. Then $\beta \in scheds(B)$ iff

1. B preserves well-formedness in β, and
2. if β is well-formed and input-live, and $n > 3t$ then the following "Byzantine properties" hold:

(b1) **agreement**: if $RETURN(r, v)$ occurs in β for some $r \in$ *root-names* and $v \in \{0, 1\}$, then no $RETURN(r', v')$ occurs in β for any $r' \in$ *root-names* such that $v \neq v'$.

(b2) **validity**: if $\exists v \in \{0, 1\}$ such that for every $r \in$ *root-names*, $CREATE(r, v, P, t)$ occurs in β, then every $RETURN(r, v')$ has $v' = v$.

(b3) **termination**: $\forall r \in$ *root-names*, $RETURN(r, v_r)$ occurs in β for some $v_r \in \{0, 1\}$.

For schedule module B, we define the users $U = \{U_r\}_{r \in roots}$, where $U_r = \{CREATE(r, v, P, t), RETURN(r, v)\}$, where $v \in \{0, 1\}$.

Lemma 3: Schedule module B is closed under reordering with respect to U.

Proof: Let $\beta \in sched(B)$ and let β_u be a permutation of $\beta|U$ such that $\forall U_r \in U, \beta_u|U_r = \beta|U_r$. We construct β' from β_u as follows: Let $\beta' = \beta_u$. Since the order (w.r.t. each user) in which the *CREATE* and *RETURN* actions occur in β_u is same as that in β, β' is well formed and satisfies the three properties (b1), (b2), and (b3). To meet the liveness condition (il2), we simply append the actions of the form $RECV(j, x, v)$ one for each $j \in T$ at the end of β'. ■

4 Exponential Information Gathering

The basis of our hierarchical proof is a simple but inefficient algorithm for Byzantine agreement known as *exponential information gathering* [7] (EIG), a variant of the original Byzantine agreement algorithm [23]. The EIG algorithm proceeds in a series of rounds, although here we do not assume that these rounds are synchronized. In the first round, each processor exchanges its value with every other processor. In the subsequent rounds values received in the previous round are exchanged. After $t + 1$ rounds, each processor determines its *final* value using a "bottom up" majority-taking scheme. In the more efficient versions of this algorithm that we describe later, processors do not exchange all the values. Instead, the algorithm detects faults and masks values sent by the faulty processors to some predetermined default value.

4.1 The Algorithm

We model the EIG algorithm using dynamic process creation to capture its recursive structure. We model each processor as a collection of automata organized into a tree. The environment creates (awakens) the automaton at the root of the tree to initiate the algorithm, and each non-leaf node in the tree of automata awakens its children on the basis of values received from its corresponding automata in the trees of other processors. Each automaton returns a value to its parent based on the majority of the values returned by its children, and finally the root node returns a decision value to the environment. One might argue that our presentation of the simplest of the Byzantine agreement algorithms (exponential information gathering) is made more complicated by the introduction of separate processes to capture the recursive structure of the algorithm. However, the extra effort in modeling this additional structure in the simple algorithm pays us back by permitting straightforward and insightful hierarchical proofs of correctness for the more sophisticated

algorithms (ESFM and cloture).

To describe the algorithm formally, we use a notation similar to that in [7]. For an alphabet

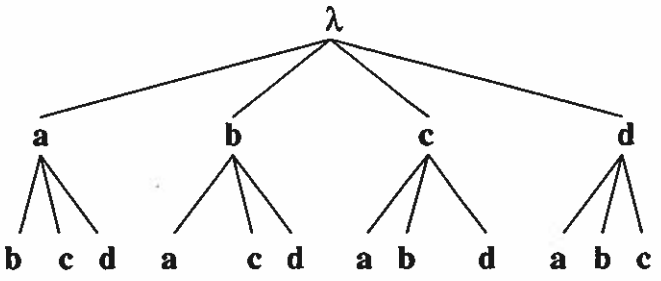

Figure 2: EIG-tree for a with set $\Sigma = \{a, b, c, d\}$ and index $i = 1$.

Σ, let $\Sigma^\bullet$ denote the set of sequences from the set Σ^* without repetitions, and let $\Sigma_k^\bullet$ denote the set of sequences from $\Sigma^\bullet$ of length at most k. Let $\lambda \in \Sigma^\bullet$ be the empty sequence. For $g \in \Sigma$, we define an *EIG-tree* for g with the set Σ and index i to be a tree with root vertex $[\![\lambda, g]\!]$ and with one vertex for each pair of the form $[\![\sigma, g]\!]$ (read "node σ in g") for all $\sigma \in \Sigma_{i+1}^\bullet$, where each node $[\![\sigma, g]\!]$ has as its children all nodes of the form $[\![\sigma j, g]\!]$ such that $\sigma j \in \Sigma_{i+1}^\bullet$. Figure 2 shows an EIG-tree for processor a, where the component σ of a node $[\![\sigma, a]\!]$ is derived as the concatenation of the labels along the path from the root to itself. Note that since every node $[\![\sigma, a]\!]$ in an EIG-tree must have a $\sigma \in \Sigma_{i+1}^\bullet$, no path in the EIG-tree shown in Figure 2 has a repeated label. Although σ represents a sequence, we sometimes abuse the notation and let σ stand for the set of all elements in the sequence. When the processor name g is clear from the context, we sometimes write node σ instead of $[\![\sigma, g]\!]$. Two nodes $[\![\sigma, g]\!]$ and $[\![\sigma', g']\!]$ (in possibly different EIG-trees) are said to be *cousins* iff $\sigma = \sigma'$. From this definition, a node $[\![\sigma, g]\!]$ is a cousin of itself.

In modeling the EIG algorithm, we capture the recursive structure by establishing for each processor a tree of automata as follows. The set of automata are named using sequences from $P^\bullet$ of length at most $t + 1$, where $t = |T|$ is the number of faulty processors in the system; the root automaton is named by λ (the empty sequence); and the children of σ have names of the form $\sigma j \in P^\bullet$. The algorithm starts at each processor with the creation of the root automaton with a particular initial value. Each automaton awakened during the execution sends its value to the cousin nodes in all processors. After the receipt of the value v of an automaton σ on processor j, the automaton σ on processor g wakes up the automaton σj with an initial value v. This continues

until the trees grow to a depth of $t+1$. Then each node establishes a final value. The final value of a leaf is its initial value. Internal nodes compute their final values by taking the majority value of all of their children's final values. The value returned by the root automaton on a given processor is the final value of that processor.

Input Actions:

- $CREATE([\![\sigma, g]\!], v, Q, m)$
  E: $s.v_{\text{initial}}, s.v_{\text{final}} \leftarrow v, v$
  $s.P, s.n, s.t \leftarrow Q, |Q|, m$
  if $|\sigma| \leq t$
  then $s.\text{pending\_sends} \leftarrow Q$
  $s.\text{status} \leftarrow$ 'running'
  else $s.\text{pending\_sends} \leftarrow \emptyset$
  $s.\text{status} \leftarrow$ 'fixed'
- $RECV(j, [\![\sigma, g]\!], v)$, where $j \in s.P$
  E: $s.\text{child} \leftarrow s'.\text{child} \cup \{(\sigma j, v)\}$
- $RETURN([\![\sigma j, g]\!], v)$
  E: **if** $v \neq \perp$ **then** $s.\text{ret}[v] \leftarrow s'.\text{ret}[v]+1$

Output Actions:

- $CREATE([\![\sigma j, g]\!], v, Q, m)$
  P: $s'.\text{status} =$ 'running'
  $(\sigma j, v) \in s'.\text{child}$
  $j \notin s'.\text{created}$
  $m = s'.t$
  $Q = s'.P - \{j\}$
  E: $s.\text{created} \leftarrow s'.\text{created} \cup \{j\}$
- $SEND(g, [\![\sigma, h]\!], v)$
  P: $h \in s'.\text{pending\_sends}$
  $v = s'.v_{\text{initial}}$
  E: $s.\text{pending\_sends} \leftarrow s'.\text{pending\_sends} - \{h\}$
- $RETURN([\![\sigma, g]\!], f)$
  P: $s'.\text{status} =$'fixed'
  $f = s'.v_{\text{final}}$
  E: $s.\text{status} \leftarrow$ 'returned'

Internal Actions:

- $FINAL([\![\sigma, g]\!], v)$ where $v \in \{0, 1\}$
  P: $s'.\text{status} =$ 'running'
  $s'.\text{pending\_sends} = \emptyset$
  $s'.\text{ret}[0] + s'.\text{ret}[1] = s'.n$
  $v = 0 \Rightarrow s'.ret[0] \geq s.n/2$
  $v = 1 \Rightarrow s'.ret[1] > s.n/2$
  E: $s.\text{status}, s.v_{\text{final}} \leftarrow$ 'fixed', v

Figure 3: Transition relation for automaton $[\![\sigma, g]\!]$ in EIG.

We define an EIG system type to describe the structure of an EIG system. An EIG system type is a six-tuple $\langle P, T, nodes, roots, peers, \mathcal{Y} \rangle$, where P and T are sets of processor names with $T \subset P$ denoting the set of faulty processors, *nodes* is a set of automaton names from $P^{\bullet}_{t+1} \times NF$, *roots* $\subseteq$ *nodes* contains the names of automata initiated by the environment, *peers* is a mapping from *nodes* to the power set of P, and $\mathcal{Y}$ is a forest of automaton names. We require that $n > 3t$. For every non-faulty processor $g \in NF$, we define $nodes_g = \{[\![\sigma, g]\!] \mid \sigma \in P^{\bullet}_{t+1}\}$, and $roots_g = [\![\lambda, g]\!]$. For every node σ, $peers(\sigma) = \{j : j \in P \wedge j \notin \sigma\}$, the set of processors from which node σ is willing to receive messages. The allowable set of messages $\mathcal{M} = \{0, 1, \sigma \mid \sigma \in P^{\bullet}_{t+1}\}$. Finally, $\mathcal{Y} = \{\mathcal{Y}_g | g \in NF\}$, where $\mathcal{Y}_g$ is the EIG-tree for g with set P and index t.

To construct an EIG system for a given EIG system type, we associate an automaton $[\![\sigma, g]\!]$ with each element $[\![\sigma, g]\!]$ of *nodes*. Each non-faulty processor g is modeled as the composition of all the automata with names from the set $nodes_g$, and the EIG system itself is the composition of all the non-faulty processors. More formally, for each non-faulty processor $g \in NF$, $X_g = \prod_{A \in nodes_g} A$. Then the EIG system is the composition $\prod_{g \in NF} X_g$.

Each component automaton $[\![\sigma, g]\!]$ for a non-faulty processor g in an EIG system exchanges messages with its cousin automata. If $[\![\sigma, g]\!]$ receives a value of v from j then it wakes up the automaton $[\![\sigma j, g]\!]$ with the value of v. The state of the automaton $[\![\sigma, g]\!]$ consists of the components v_{final}, v_{initial}, *status*, *ret*, *child*, *created* and *pending_sends*. The values v_{initial} and v_{final} are the initial and final values of the automaton respectively. The variable *status* defines the activity of the automaton. We say that an automaton is 'alive' if its status is either 'running' or 'fixed', and we say it is 'confirmed' if its status is either 'fixed' or 'returned'. The variable *ret* maintains the number of children that have returned with a specific final value. The set *child* contains pairs $(\sigma j, v)$ such that $[\![\sigma j, g]\!]$ is a child of $[\![\sigma, g]\!]$ in $\mathcal{Y}_g$ that was already created or is to be created with a value v; the set *created* contains all the children of $[\![\sigma, g]\!]$ that were actually created; and the set *pending_sends* $\subseteq P$ contains the processors to which the initial value of $[\![\sigma, g]\!]$ needs to be sent. Initially, *status* is 'dormant' and v_{initial} and v_{final} are undefined ($\perp$); the values of $ret[0]$ and $ret[1]$ are 0. The sets *pending_sends*, *child*, and *created* are initially empty. In addition we have three variables P, n, and t local to an automaton which hold the set of all possible children, the number of such children, and the number of potentially faulty processors in the system. P, n, and t are undefined initially.

The transition relation for the automaton $[\![\sigma, g]\!]$ in the EIG system is shown in Figure 3. We specify the transition relation of an automaton by giving for each action a list of preconditions (P) and effects (E). An action is enabled from any state s' satisfying the action's preconditions, and the action takes the automaton to the state s such that s can be obtained by modifying s' as indicated by the action's effects. Since input actions are enabled in every state, the obvious precondition (*true*) is omitted.

The actions of the automaton $[\![\sigma, g]\!]$ are as follows. The $CREATE([\![\sigma, g]\!], v, Q, m)$ input action awakens the automaton. It initializes the initial value v_{initial} and the local variables P and t. If $[\![\sigma, g]\!]$ is not a leaf node ($|\sigma| \leq t$) and $m \neq \perp$ it sets the status to 'running'; otherwise the status is set to 'fixed'. The set *pending_sends* is set to Q if the automaton is not a leaf node, and the final

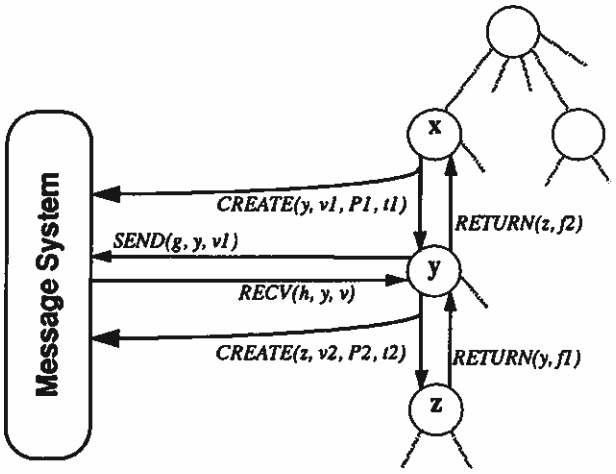

Figure 4: Actions for a node y in an EIG system shared with its parent x, child z, and the message system.

value v_{final} is set to v if $m \neq \perp$. The automaton $[\![\sigma, g]\!]$ sends its initial value to all processors $h \notin \sigma$ using the $SEND(g, [\![\sigma, h]\!], v_{\text{initial}})$ output action. It receives the values sent to it by other processors through the $RECV(h, [\![\sigma, g]\!], v)$ input action. The $RECV(h, [\![\sigma, g]\!], v)$ input action (the receipt of the input value v of the node $[\![\sigma, h]\!]$) places $(\sigma j, v)$ in the set *child* for the later creation of the corresponding child. The $CREATE([\![\sigma j, g]\!], v, Q, m)$ output action creates a child of $[\![\sigma, g]\!]$; the sets *child* and *created* are used to ensure each child is created only once. The $RETURN([\![\sigma j, g]\!], v)$ input action increments $ret[v]$, the number of children with final value v. $FINAL([\![\sigma, g]\!])$, the only internal action, fixes the final value of $[\![\sigma, g]\!]$ based on the majority of the return values of its children. The $RETURN([\![\sigma, g]\!], f)$ output action returns the final value of $[\![\sigma, g]\!]$ to its parent.

The actions of an automaton y shared with its parent x, a child z, and the message system are shown in Figure 4. Each *CREATE* action also informs the message system that an automaton has been awakened and may now receive messages. The messages destined for cousin automata are sent through the message system with the *SEND* action. Messages from cousin automata are delivered by the *RECV* action from the message system. Note that the output *RETURN* action of a child of y is an input action to y. This action also carries the final value of the child to its parent.

4.2 Proof of Correctness

We give an assertional proof to show that every EIG system solves the Byzantine agreement problem. For an automaton name $x \in nodes$, we let s_x denote the state of the automaton named x. We begin by showing that each automaton in an EIG system preserves well-formedness. We then state a number of invariants on the states of well-formed executions. These invariants are used to show that every EIG system solves Byzantine agreement.

Lemma 4: Every automaton $[\![\sigma, g]\!]$ in an EIG system preserves well-formedness.

Proof: Since the effect of the $CREATE([\![\sigma j, g]\!], v, Q, m)$ output action is to add j to the set created, the automaton $[\![\sigma, g]\!]$ creates a child $[\![\sigma j, g]\!]$ at most once. After sending a message to another processor h, h is removed from the set pending_sends, thus allowing for at most one *SEND* to a given processor. The precondition of the *RETURN* requires that the status of the node $[\![\sigma, g]\!]$ be 'fixed'; but after the *RETURN* action its status is set to 'returned'. Thus there will be at most one action of the form $RETURN([\![\sigma, g]\!], v)$ in any execution of $[\![\sigma, g]\!]$. ■

Lemma 5: Let s be a state in a well-formed execution of an EIG system. The following properties hold:

(s1) $\forall x \in nodes$, s_x.ret[v] is equal to the number of children y of x with $s_y.v_{\text{final}} = v$ and $s_y.status =$ 'returned'.

(s2) For any $\sigma \in P_t^\bullet$ such that

$\forall h \in NF, [\![\sigma, h]\!]$ is 'confirmed' and

$\forall \sigma j \in P^\bullet, \forall g \in NF, s_{[\![\sigma j, g]\!]}.v_{\text{final}} = s_{[\![\sigma j, h]\!]}.v_{\text{final}}$,

it follows that $\forall h \in NF, s_{[\![\sigma, g]\!]}.v_{\text{final}} = s_{[\![\sigma, h]\!]}.v_{\text{final}}$.

That is, if a non-leaf node x and its cousins are confirmed and if each child of x and its cousins have a common final value, then x and its cousins will also have some common final value.

(s3) $\forall g, j \in NF, s_{[\![\sigma j, g]\!]}.status \neq$ 'dormant' $\Longrightarrow s_{[\![\sigma j, g]\!]}.v_{\text{initial}} = s_{[\![\sigma, j]\!]}.v_{\text{initial}}$.

That is, the initial values of the children of a node x are determined by the initial values of its cousins in $peers(x)$.

(s4) For any $\sigma \in P_t^\bullet$ and $v \in \{0,1\}$ such that

$\forall g \in NF$, $[\![\sigma, g]\!]$ is 'confirmed' and

there exists a majority of children $[\![\sigma j, g]\!]$ of $[\![\sigma, g]\!]$ with $s_{[\![\sigma j, g]\!]}.v_{\text{final}} = v$,

it follows that $\forall g \in NF, s_{[\![\sigma, g]\!]}.v_{\text{final}} = v$.

Intuitively, if for a node x and its cousins, the majority of each of their children have some common final value, then the node x and its cousins will also have the *same* final value.

(s5) $\forall \sigma \in P_t^\bullet, \forall g, j \in NF$, $[\![\sigma j, g]\!]$ is 'confirmed' $\Longrightarrow s_{[\![\sigma j, g]\!]}.v_{\text{final}} = s_{[\![\sigma, j]\!]}.v_{\text{initial}}$.

That is, the final values of the children of a node x are determined by the initial values of its cousins in $peers(x)$.

(s6) $\forall \sigma \in P_t^\bullet, \forall j \in NF, \exists v \in \{0,1\}$ such that $\forall g \in NF$, $[\![\sigma j, g]\!]$ is 'confirmed' $\Longrightarrow s_{[\![\sigma j, g]\!]}.v_{\text{final}} = v$.

If j and g are non faulty processors, then the final values of the node $[\![\sigma j, g]\!]$ and its cousins will be the same.

(s7) $\forall \sigma \in T^\bullet, \exists v \in \{0,1\}$ such that $\forall g \in NF, [\![\sigma, g]\!]$ is 'confirmed' $\Longrightarrow s_{[\![\sigma, g]\!]}.v_{\text{final}} = v$.

If σ is a non repeating sequence of faulty processors, then a node $[\![\sigma, g]\!]$ and its cousins will have the same final value. Note that $\lambda \in T^\bullet$.

Proof: The invariant (s1) holds initially, since all automata start in the 'dormant' state and $ret[v] = 0$. The only action which makes $s_{[\![\sigma j, g]\!]}.status$ = 'returned' is the shared $RETURN([\![\sigma j, g]\!], v)$ action which has $s_{[\![\sigma j, g]\!]}.v_{\text{final}} = v$ in its precondition; this action also increments $s_{[\![\sigma, g]\!]}.ret[v]$.

Since $s_{[\![\sigma, g]\!]}.status$ = 'dormant' initially, (s2) holds initially. Let $[\![\sigma, g]\!]$ be 'confirmed' for all g. The only action which transforms the automaton from being not confirmed to being 'confirmed' is the $FINAL([\![\sigma, g]\!], v)$ action which assigns the majority of all the values returned from the children of $[\![\sigma, g]\!]$. Since, every child of $[\![\sigma, g]\!]$ has the same value as its cousins, we have the same majority value being assigned to the v_{final} of the automata $[\![\sigma, g]\!]$ for every non-faulty processor g.

Initially, when the status is dormant (s3) holds vacuously. $CREATE([\![\sigma j, g]\!], v, Q, m)$ is the only action which changes the status and also initializes v_{initial} to v. The precondition of the *CREATE* action requires $(\sigma j, v)$ to be in the set *child*. Only the *RECV* action inserts the message $(\sigma j, v)$ into the set *child*. Due to well-formedness conditions, a $SEND(j, [\![\sigma, g]\!], v)$ action must occur earlier. The preconditions on *SEND* require that $v = s'_{[\![\sigma, j]\!]}.v_{\text{initial}}$. Since v_{initial} is "write once," the invariant holds.

Invariant (s4) follows directly from the majority operation at every node $[\![\sigma, g]\!]$ during the $FINAL([\![\sigma, g]\!], v)$ action, which also makes $[\![\sigma, g]\!]$ 'confirmed'.

We prove (s5) by decreasing induction on the length of σ. Note that $|\sigma| \leq t$, so $|\sigma| = t$ is our basis. For $|\sigma| = t$, due to (s3) we have $s_{[\![\sigma j, g]\!]}.v_{\text{initial}} = s_{[\![\sigma, j]\!]}.v_{\text{initial}}$. Since the value of m is always chosen as $s.t$ we have $s_{[\![\sigma j, g]\!]}.v_{\text{final}} = s_{[\![\sigma j, g]\!]}.v_{\text{initial}}$, according to the effect of the *CREATE* action.

Let $|\sigma| < t$ and $s_{[\![\sigma, j]\!]}.v_{\text{initial}} = v$. Assume that $[\![\sigma j, h]\!]$ is 'confirmed' for all $h \in NF$. Since $s_{[\![\sigma, j]\!]}.v_{\text{initial}} = v$, from (s3) we have $s_{[\![\sigma j, g]\!]}.v_{\text{initial}} = v$ for all $g \in NF$. Inductively, for every $k \in NF$, if $\sigma jk \in P^{\bullet}$ then $s_{[\![\sigma jk, g]\!]}.v_{\text{final}} = v$ for all $g \in NF$. Since $|T| = t$, $|\sigma j| \leq t$, and the number of such k's is at least $n - t - t > t$, we can apply property (s4) and get $s_{[\![\sigma j, h]\!]}.v_{\text{final}} = v$.

Property (s6) is a direct consequence of (s5).

We prove (s7) by contradiction. Let $\tau \in T^{\bullet}$ be the deepest node in the EIG tree such that (s7) does not hold. Consider $\tau j \in P^{\bullet}$. If $j \in NF$, then $\exists v$ such that $s_{[\![\tau j, g]\!]}.v_{\text{final}} = v$ for all $g \in NF$ according to (s6). If $j \in T$, then we have $\tau j \in T^{\bullet}$. Again $s_{[\![\tau j, g]\!]}.v_{\text{final}}$ has the same value for all $g \in NF$ due to the assumption. Thus, from (s2) we have a common (the majority) value across all processors for the v_{final} of τ. ■

To show progress and termination of the EIG algorithm, we introduce a variant function $\mathcal{V}$ which satisfies the following: every action decreases $\mathcal{V}$, there is a lower bound for $\mathcal{V}$, and whenever $\mathcal{V}$ is greater than this lowest element, some action is enabled. The existence of such a function proves termination.

The variant function $\mathcal{V}$ has five components, the first three relating to the status of the automata in the system, and the last two relating to the communication between the automata in different EIG trees. Note that the communication actions (*SEND* and *RECV*) do not modify the status variable in the state of any automaton. Let α be a prefix of an execution of an EIG system, and let α end in state s. We define $\mathcal{V}(\alpha) = (\mathcal{D}, \mathcal{R}, \mathcal{F}, \mathcal{P}, \mathcal{W})$, where

$\mathcal{D}$ $= |\{x \in nodes : s_x.\text{status} = \text{'dormant'}\}|$,

$\mathcal{R}$ $= |\{x \in nodes : s_x.\text{status} = \text{'running'}\}|$,

$\mathcal{F}$ $= |\{x \in nodes : s_x.\text{status} = \text{'fixed'}\}|$,

$\mathcal{P}$ $= \sum_{x \in nodes} |s_x.\text{pending\_sends}|$, and

$\mathcal{W}$ $=$ (number of *SEND* events in α) $-$ (number of *RECV* events in α).

Informally, $\mathcal{D}$ is the total number of automata in the system which are 'dormant', $\mathcal{R}$ is the total number of automata that are 'running', $\mathcal{F}$ is the total number of automata that have their final values 'fixed', $\mathcal{P}$ is the number of messages 'pending' in the message system, and $\mathcal{W}$ is the total number of messages waiting to be delivered. We let the domain of values for $\mathcal{V}$ be ordered lexicographically.

Let $treesize(n,t)$ be the size of an EIG tree for a set of n processors and index t. That is, $treesize(n,t) = \sum_{k=0}^{t+1} \prod_{i=0}^{k-1}(n-i)$. Since each non-faulty processor initially has a full tree of dormant automata, the initial value of the variant function is

$$\mathcal{V} = ((n-t) \times treesize(n,t), 0, 0, 0, 0).$$

Note that the domain of all the components in $\mathcal{V}$ is the set of non-negative integers, so $\mathcal{V}$ is bounded from below by $(0,0,0,0,0)$.

Lemma 6: Let $\alpha = \beta\pi s$ be a prefix of an input well-formed execution of an EIG system. Then $\mathcal{V}(\alpha) < \mathcal{V}(\beta)$.

Proof: The proof is a case analysis. Let $\mathcal{V}(\beta) = (\mathcal{D}', \mathcal{R}', \mathcal{F}', \mathcal{P}', \mathcal{W}')$. Based on each possible action π, we compute the value $\mathcal{V}(\alpha)$, and observe that $\mathcal{V}(\alpha) < \mathcal{V}(\beta)$.

- $\pi = CREATE([\![\sigma, g]\!], v, Q, m)$: $\mathcal{V}(\alpha) = \begin{cases} (\mathcal{D}'-1, \mathcal{R}'+1, \mathcal{F}', \mathcal{P}'+n-|\sigma|, \mathcal{W}) & \text{if } |\sigma| < t+1 \\ (\mathcal{D}'-1, \mathcal{R}', \mathcal{F}'+1, \mathcal{P}', \mathcal{W}') & \text{otherwise} \end{cases}$
- $\pi = FINAL$: $\mathcal{V}(\alpha) = (\mathcal{D}', \mathcal{R}'-1, \mathcal{F}'+1, \mathcal{P}', \mathcal{W}')$.
- $\pi = RETURN$: $\mathcal{V}(\alpha) = (\mathcal{D}', \mathcal{R}', \mathcal{F}'-1, \mathcal{P}', \mathcal{W}')$.
- $\pi = SEND$: $\mathcal{V}(\alpha) = (\mathcal{D}', \mathcal{R}', \mathcal{F}', \mathcal{P}'-1, \mathcal{W}'+1)$.
- $\pi = RECV$: $\mathcal{V}(\alpha) = (\mathcal{D}', \mathcal{R}', \mathcal{F}', \mathcal{P}', \mathcal{W}'-1)$.

■

Lemma 7: If α is an input well-formed and input live execution of an EIG system and $\mathcal{V}(\alpha) > (0,0,0,0,0)$ in state s of α, then some action in the system is enabled in state s.

Proof: We show that if at least one of the components of the variant function is positive then there is an action enabled in the system. We analyze case by case.

- Let $\mathcal{W} > 0$. This implies that at least one message is waiting to be delivered. Due to the liveness assumptions on the environment this message eventually will be delivered. Thus a *RECV* action is enabled.

- Let $\mathcal{P} > 0$. There is an automaton $[\![\sigma, g]\!]$ such that $s_{[\![\sigma,g]\!]}$.pending_sends is not empty. Hence the action $SEND(g, [\![\sigma, h]\!], v_i)$ is enabled for some h and v_i.

- Let $\mathcal{F} > 0$. Let $[\![\sigma, g]\!]$ be an automaton such that its status is 'fixed'. The action $RETURN([\![\sigma, g]\!], v_f)$ is enabled for some v_f.

- Let $\mathcal{D} > 0$. Let $[\![\sigma j, g]\!]$ be a node closest to the root that has not yet been created (i.e. the status is 'dormant'). Consider the status of the automaton $[\![\sigma, j]\!]$. There are three cases. Cases (i) and (ii): If $s_{[\![\sigma,j]\!]}$.status = 'returned' or $s_{[\![\sigma,j]\!]}$.status = 'fixed', then we have $\left|s_{[\![\sigma,j]\!]}.\text{pending\_sends}\right| = 0$ implying that the action $SEND(j, [\![\sigma, g]\!], s_{[\![\sigma,j]\!]}.v_{\text{initial}})$ occurs in α. Due to the third liveness condition (il3), the corresponding *RECV* action occurs in α. So the action $CREATE([\![\sigma j, g]\!], v_i, Q, m)$ is enabled. If the *RECV* action does not occur in α, then $\mathcal{W} > 0$, so an action is enabled as argued above. Case (iii): If $s_{[\![\sigma,j]\!]}$.status = 'running', then there are two possibilities. Either $\left|s_{[\![\sigma,j]\!]}.\text{pending\_sends}\right| > 0$ implying $\mathcal{P} > 0$ or $\left|s_{[\![\sigma,j]\!]}.\text{pending\_sends}\right| = 0$ which has already been discussed in the previous cases.

- Let $\mathcal{R} > 0$. Assume that $\mathcal{F} = 0$, or else we already have an action enabled. Let $[\![\sigma, g]\!]$ be the deepest automaton which has its status as 'running'. The children of $[\![\sigma, g]\!]$ must have as their status either 'returned' or 'dormant'. If all the children have returned then the action $FINAL([\![\sigma, g]\!], v)$ will be enabled. If at least one of the children is dormant, then $\mathcal{D} > 0$, which was discussed earlier.

■

Lemma 8: In every input well-formed and input live fair execution of an EIG system, the EIG algorithm terminates.

Proof: Termination also means that all the top-level automata return. The lowest value of the variant function $(0, 0, 0, 0, 0)$ says that there is no automaton with its status 'dormant', 'running', or 'fixed' implying that all the automata have 'returned'. Thus the Lemma is immediate from Lemma 6, Lemma 7, and the definition of fairness. ■

Theorem 9: Every EIG system solves the Byzantine agreement problem (schedule module B).

Proof:

- *Agreement* follows from Lemma 5(s7).
- *Validity*: If all $g \in NF$ start with $s_{[\lambda,g]}.v_{\text{initial}} = v$ then by (s5) of Lemma 5 for every $j \in NF$ we have $s_{[j,g]}.v_{\text{final}} = v$ for every non-faulty processor g whenever $[\![j, g]\!]$ is 'confirmed'. Hence from (s4) of Lemma 5 we have $s_{[\lambda,g]}.v_{\text{final}} = v$.
- *Termination* was proven in Lemma 8.

■

5 Early Stopping and Fault Masking

The next step in our hierarchical proof is to show that an intermediate algorithm called *early stopping and fault masking* (ESFM) simulates EIG using a possibilities mapping. The ESFM strategy allows non-faulty processors to reach agreement with less communication. The final values of some nodes may be predicted before computing all initial values (early stopping). Once a processor discovers that another processor j is faulty (fault discovery), it may pretend that all values reported by j are 0 (fault masking); this increases the number of cases where early stopping applies.

5.1 The Algorithm

We define an ESFM system type to be a six-tuple $\langle P, T, nodes, roots, peers, \mathcal{Y} \rangle$ similar to the EIG system type. An ESFM system is an instance of the ESFM system type with the same properties as that of an EIG system except that we now require that $n > 4t$. Each node is modeled as a shared memory I/O automaton, where each node $[\![\sigma, g]\!]$ has access to the data structure $\mathcal{T}_g$ and the sets $Faulty_g$ and $Stopped_g$. These shared data structures make explicit exactly what information is being shared among the recursive calls on a given processor. We use the set $Faulty_g$ to share the information about the set of faults detected thus far among the automata within processor g. The operation **setfault**$(Faulty_g, j)$ includes the processor j in the set $Faulty_g$ and the operation

getfaults($Faulty_g$) returns the set of processors known to be faulty. This set is initially empty. The structure of $\mathcal{T}_g$ is identical to that of an EIG-tree for g with set P and index t, and each node in the tree holds the state of the node which could be either 'alive' or 'returned'. Initially all nodes are marked as 'returned' and the set *Stopped* is empty. The operations on this tree structure are as follows: **set**($\sigma, \mathcal{T}_g$) records node $[\![\sigma, g]\!]$ as 'alive', **prune**($\sigma, \mathcal{T}_g$) records the status of all the descendants of the node $[\![\sigma, g]\!]$ in $\mathcal{T}_g$ as 'returned' (since these nodes can be treated as if they have returned), and **getstopped**($Stopped_g, \sigma$) returns the set of nodes $[\![\sigma', h]\!]$ that have stopped early such that σ' is a proper ancestor of σ in the tree $\mathcal{T}_g$.

Each processor g has its own variables $Faulty_g$, $Stopped_g$, and $\mathcal{T}_g$ that are shared among all nodes in g. Nodes at each processor access the variables for that processor. Therefore, we define the automaton X_g for every non-faulty processor g as $X_g = \mathcal{C}(\prod_{A \in nodes_g} A, \{Faulty_g, Stopped_g, \mathcal{T}_g\})$, where $\mathcal{C}$ is the *closeout* operator (see Section 2) that makes the set $Faulty_g$ and the tree $\mathcal{T}_g$ private to the composition. The ESFM system is the composition $\prod_{g \in NF} X_g$. The tree structure $\mathcal{T}_g$ is not necessary for the correctness of the ESFM protocol; but we will need the tree data structure in the cloture voting algorithm, so we use it here to set up a correspondence that will simplify our later arguments.

The definition of each automaton $[\![\sigma, g]\!]$ in the ESFM algorithm is given in Figure 5. In addition to the state components of the automata in the EIG algorithm, each automaton maintains a few additional local state variables. The local variable *faulty* is the set of faulty processors known to this automaton, *stop_signals* is the set of processors which must be informed if this automaton stops early, *halted* is the set of automata of interest which have stopped early in other processors, and *marked* is a subset of *halted*. These sets are initially empty. The notation child[v] denotes the number of elements σ such that (σ, v) is in the set *child*. For all $\sigma \in P^\bullet$ such that $|\sigma| \geq 1$ we define $tail(\sigma)$ to be the element j of P such that $\sigma = \sigma' j$ for some $\sigma' \in P^\bullet$ (That is, $tail(\sigma)$ is the last element in the sequence σ). In addition to the actions carried over from the EIG algorithm, we have the actions *DISCOVER_FAULT*, *MASK_FAULT*, *STOP_EARLY*, *UPDATE* and *STOPPED_EARLY*. They are used for fault discovery, fault masking, early stopping, updating both the set of stopped automata and the set of faulty processors known to the automaton, and for processing the set of stopped automata in other processors. It is interesting to see how each optimization is handled by a separate action whose precondition states exactly in what situations

the optimization may be applied.

The $UPDATE([\![\sigma, g]\!])$ action is used to update an automaton's local information by accessing the shared data structures. The action updates the local variable *faulty* to the set of all processors reported as faulty by the automata in the processor g, and updates the local variable *halted* to the set of proper ancestors known to have stopped early in some processor.

The $DISCOVER\_FAULT([\![\sigma, g]\!])$ action inserts the processor name $tail(\sigma)$ into the local state variable *faulty* if there exist at least $t + 1 - F$ items in each of child[0] and child[1], where $F = |\sigma \cap s'.faulty|$.

Using the $MASK\_FAULT([\![\sigma, g]\!])$ action, the node $[\![\sigma, g]\!]$ pretends that values reported by faulty processors are 0; this action removes the entries of the form $(\sigma j, 1)$ from the set *child* and includes those entries of the form $(\sigma j, 0)$ into that set for all $j \in s.faulty$. We will see that this does not affect the correctness of the algorithm, because EIG works regardless of the behavior of the faulty processors. Note that due to asynchronous executions a fault detected by one node may not be immediately marked by other nodes, but only after they execute the *UPDATE* and *MASK_FAULT* actions.

If there is an overwhelming number of the children with the same initial value, the action $STOP\_EARLY([\![\sigma, g]\!], v)$ fixes the final value of $[\![\sigma, g]\!]$ and prunes $\mathcal{T}_g$. Pruning is not necessary in ESFM, but it makes the proof simpler when we move to the cloture voting protocol, where we show a correspondence between parts of cloture voting protocol and ESFM. Pruning the tree makes explicit which recursive calls are "optimized away" in ESFM.

The $STOPPED\_EARLY([\![\sigma, g]\!])$ action is used to identify the nodes that have stopped early in the system and make appropriate changes to the state of $[\![\sigma, g]\!]$. The precondition of the $STOPPED\_EARLY([\![\sigma, g]\!])$ action is that there exists an element $[\![\sigma', h]\!]$ in the set *halted* such that $\sigma' i$ is a proper ancestor of σ. If $h = g$, then $[\![\sigma, g]\!]$ does not need the result of this node or its children. Further more, even if $h \neq g$, $[\![\sigma, g]\!]$ cannot expect its cousin $[\![\sigma, h]\!]$ (or its descendants) to send messages. In such cases, if a node $[\![\sigma'', g]\!]$ has already been created, it should be terminated (for optimization), where σ'' is a descendant of σ. Otherwise, we simply adjust the local variable $s_{[\![\sigma,g]\!]}.P$ so that $[\![\sigma, g]\!]$ never creates its child $[\![\sigma h, g]\!]$.

Input Actions:

- $CREATE(\llbracket \sigma, g \rrbracket, v, Q, m)$
  E: $s.v_{\text{initial}} \leftarrow v$
  $\mathbf{set}(\sigma, \mathcal{T}_g)$
  $s.P, s.n, s.t \leftarrow Q, |Q|, m$
  if $(|\sigma| \leq t \wedge m \neq \perp)$
  then $s.\text{status} \leftarrow$ 'running'
  else $s.\text{status} \leftarrow$ 'fixed'
  if $|\sigma| \leq t$
  then $s.\text{pending\_sends} \leftarrow Q$
  if $m \neq \perp$ **then** $s.v_{\text{final}} \leftarrow v$
- $RECV(j, \llbracket \sigma, g \rrbracket, v)$, where $j \in s.P \wedge v \in \{0, 1\}$
  E: **if** $(\sigma j, 0) \notin s'.\text{child}$
  then $s.\text{child} \leftarrow s'.\text{child} \cup \{(\sigma j, v)\}$
- $RETURN(\llbracket \sigma j, g \rrbracket, v)$
  E: **if** $v \neq \perp$ **then** $s.\text{ret}[v] \leftarrow s'.\text{ret}[v]+1$

Internal Actions:

- $UPDATE(\llbracket \sigma, g \rrbracket)$
  P: $s'.\text{status} =$ 'running'
  E: $s.\text{halted} = \mathbf{getstopped}(Stopped_g, \sigma)$
  s.faulty $\leftarrow$ $\mathbf{getfaults}(Faulty_g)$
- $STOPPED\_EARLY(\llbracket \sigma, g \rrbracket)$
  P: $\llbracket \sigma', h \rrbracket \in s'.\text{halted}$
  $\llbracket \sigma', h \rrbracket \notin s'.\text{marked}$
  where $\sigma = \sigma' i j \gamma$
  $s'.\text{status} \neq$ 'returned'
  E: $s.\text{marked} \leftarrow s'.\text{marked} \cup \{\llbracket \sigma', h \rrbracket\}$
  if $(g = h \vee h \in \gamma)$ **then**
  $s.\text{status}, s.v_{\text{final}} \leftarrow$ 'fixed', $\perp$
  else if $(i \neq h \wedge j \neq h)$ **then**
  $s.P, s.n \leftarrow s'.P - \{h\}, s'.n - 1$
- $DISCOVER\_FAULT(\llbracket \sigma, g \rrbracket)$
  P: $s'.\text{status} =$ 'running'
  $\text{child}[0] \geq s'.t + 1 - |\sigma \cap s'.\text{faulty}|$
  $\text{child}[1] \geq s'.t + 1 - |\sigma \cap s'.\text{faulty}|$
  E: s.faulty $\leftarrow$ $s'.\text{faulty} \cup \{tail(\sigma)\}$
  $\mathbf{setfault}(Faulty_g, tail(\sigma))$

Output Actions:

- $CREATE(\llbracket \sigma j, g \rrbracket, v, Q, m)$
  P: $s'.\text{status} =$ 'running'
  $(\sigma j, v) \in s'.\text{child}$
  $j \notin s'.\text{created}$
  $(\exists v' :: \text{child}[v'] \geq (s'.n - s'.t)) \Rightarrow m = \perp$
  $\neg(\exists v' :: \text{child}[v'] \geq (s'.n - s'.t)) \Rightarrow m = s'.t$
  $Q = s'.P - \{j\}$
  E: $s.\text{created} \leftarrow s'.\text{created} \cup \{j\}$
- $MASK\_FAULT(\llbracket \sigma, g \rrbracket)$
  P: $s'.\text{status} =$ 'running'
  E: $s.\text{child} \leftarrow s'.\text{child} - \{(\sigma j, v) \mid j \in s.\text{faulty}\}$
  $+\{(\sigma j, 0) \mid j \in s.\text{faulty} \wedge j \in s.P\}$
- $STOP\_EARLY(\llbracket \sigma, g \rrbracket, v)$
  P: $s'.\text{status} =$ 'running'
  $s'.\text{pending\_sends} = \emptyset$
  $s'.P \subseteq s'.\text{created}$
  $\text{child}[v] \geq (s'.n - s'.t)$
  E: $\mathbf{prune}(\sigma, \mathcal{T}_g)$
  $s'.\text{status}, s.v_{\text{final}} \leftarrow$ 'fixed', v
- $FINAL(\llbracket \sigma, g \rrbracket, v)$ where $v \in \{0, 1\}$
  P: $s'.\text{status} =$ 'running'
  $s'.\text{pending\_sends} = \emptyset$
  $s'.P \subseteq s'.\text{created}$
  $v = 0 \Rightarrow s'.\text{ret}[0] \geq s'.n/2$
  $v = 1 \Rightarrow s'.\text{ret}[1] > s'.n/2$
  E: $\mathbf{prune}(\sigma, \mathcal{T}_g)$
  $s'.\text{status}, s.v_{\text{final}} \leftarrow$ 'fixed', v
- $SEND(g, \llbracket \sigma, h \rrbracket, v)$
  P: $h \in s'.\text{pending\_sends}$
  $v = s'.v_{\text{initial}}$
  E: $s.\text{pending\_sends} \leftarrow s'.\text{pending\_sends} - \{h\}$
- $SEND(g, \llbracket \lambda, h \rrbracket, \sigma)$ where $h \in P$
  P: $(\exists v' :: \text{child}[v'] \geq (s'.n - s'.t))$
  $h \notin s'.\text{stop\_signals}$
  $s'.\text{stop\_signals} \subset P$
  E: $s.\text{stop\_signals} \leftarrow s'.\text{stop\_signals} \cup \{h\}$
- $RETURN(\llbracket \sigma, g \rrbracket, f)$
  P: $s'.\text{status} =$ 'fixed'
  $f = s'.v_{\text{final}}$
  E: $s.\text{status} \leftarrow$ 'returned'

Figure 5: Transition relation for automaton $\llbracket \sigma, g \rrbracket$ in $ESFM$.

Input Action:

- $RECV(j, \llbracket \lambda, g \rrbracket, \tau)$, where $j \in P$ and $\tau \in P^*$
  E: $\text{Stopped}_g \leftarrow \text{Stopped}_g \cup \{\llbracket \tau, j \rrbracket\}$

Figure 6: Additional action for automaton $\llbracket \lambda, g \rrbracket$ in $ESFM$.

The $RECV(j, [\sigma, g], v)$ action is modified from EIG so that it ignores messages from processors that have been detected as faulty. The *CREATE* and *SEND* actions are modified to facilitate the automata in ESFM to stop early. The *CREATE* output action creates the children with $\perp$ as the value of m whenever early stopping is possible based on the initial values, and the *CREATE* input action sets its status to 'fixed' if the value of m is $\perp$. In this way, an automaton that is created after its parent satisfies the stop early condition simply sends its value to its cousins but does not create any of its own children. The $SEND(g, [\lambda, h], \sigma)$ action informs the processor h that the automaton $[\sigma, g]$ has the potential of stopping early. The root node in each processor uses a special action in addition to those described above. The $RECV(j, [\lambda, g], \tau)$ action receives the information that the automaton $[\tau, j]$ has stopped early and adds it to the variable $Stopped_g$. The special action of the root node of processor g are given in Figure 6.

Although our description of the algorithm is formulated differently, the ESFM algorithm presented here is essentially the same algorithm as that given by Berman, Garay, and Perry [7]. However, there are three differences between our ESFM algorithm and theirs. First, we make explicit that the default value is 0 when the return values from the children are equally split between 0 and 1. Second, we take advantage of the fact that the algorithm works regardless of the behavior of the faulty processors and let fault masking be non-deterministic in our description. Finally, our algorithm description is in an asynchronous model.

5.2 Correctness Proof

We prove the ESFM algorithm correct using the possibilities mapping technique described in Section 2.7. By establishing a possibilities mapping from the states of an ESFM system to the states of an EIG system, we show that ESFM systems simulate EIG systems and hence solve the Byzantine agreement problem. We wish to show that for any execution of an ESFM system A, there exists an execution of an EIG system B with the same system type such that the schedules of the two executions, when projected on the *CREATE*/*RETURN* actions of the root-nodes, are the same. In other words, we are interested only in the external behaviors of the non-faulty processors; we do not require that the behaviors of the faulty processors (as controlled by the environment) are the same in the two systems. For every possible execution of A, we are free to exhibit any possible faulty behavior in B that gives the proper correspondence at the non-faulty nodes. Therefore, in

our proof that there is an execution of B for every possible execution of A, we hide the actions from the the faulty processors. These are the *RECV* actions which have a faulty processor as the sender.

We also hide another set of actions. Since the ESFM system A may create fewer nodes than the EIG system B, we hide the actions of nodes in B which are not created in A so as to show a correspondence. Nevertheless, these nodes in B are created so that the trees of every processor in B can "keep up" with that of A. Thus, when a node in A executes a *RETURN* action, after fixing its final value due to a simple majority of its children's return values, the corresponding node in B can return because it has its complete set of children which would return their final values. In the case where a node in A returns after stopping early, we may not have created all the children of that node in B. So we reorder the execution of A such that we always have all the children of a particular node in B whenever that node returns in A. By showing a correspondence between such a "reordered" canonical execution of A and an execution of B, and by showing that every execution of A is equivalent (from the point of view of its external behavior) to some canonical execution execution we prove that A simulates B. By moving all the *FINAL* and *RETURN* actions to the end, we make sure that before any node returns we create all the nodes in the protocol. The fact that each recursive call is modeled as an automaton with its own state makes it easy to argue that such a reordering is possible, because it is easy to argue locally that two actions of an automaton are "commutative".

The hiding of all the actions of automata not created in A makes explicit the optimizations carried out in the ESFM protocol. We view this as a general technique for handling optimization of algorithms with dynamic process creation. Before presenting the proof that ESFM systems simulate EIG systems we need a few definitions. Let α be an execution of an ESFM system A.

We define $\Sigma_{\mathrm{OPT}}(\alpha) = \bigcup_{x \in A_{\mathrm{OPT}}(\alpha)} acts(x)$ where $A_{\mathrm{OPT}}(\alpha)$ is the set of automata $[\![\sigma, g]\!] \in A$ such that $STOP\_EARLY([\![\tau, g]\!], v)$ is enabled in α, where $[\![\tau, g]\!]$ is a proper ancestor of $[\![\sigma, g]\!]$. Thus $A_{\mathrm{OPT}}(\alpha)$ includes the descendants of all automata that have the potential of stopping early during the execution of α. Note that if the execution α proceeds in a series of "rounds," these automata will not be created. Thus, the set $\Sigma_{\mathrm{OPT}}(\alpha)$ includes all the actions that are "optimized away."

We also hide the set of *RECV* actions which have a faulty processor as the sender and those actions of the automata that are created with $m = \perp$. This allows us to argue only about the actions

which are of interest in EIG. More formally, let $H(\alpha) = \Sigma_{\mathrm{OPT}}(\alpha) \cup \{RECV(j, [\![\sigma, g]\!], v) \mid j \in T\} \cup acts(\bot(\alpha))$, where $\bot(\alpha)$ is the set of automata that are created with $m = \bot$.

Let $\overline{\mathrm{EIG}}$ be the set of actions that are not part of any EIG system. The set $\overline{\mathrm{EIG}}$ would include the actions *DISCOVER_FAULT*, *MASK_FAULT*, *STOP_EARLY*, *STOPPED_EARLY*, and *UPDATE*. For an arbitrary automaton X we define $\mathrm{OPT}_\alpha(X) \equiv Hide_{H(\alpha) \cup \overline{\mathrm{EIG}}}(X)$.

Thus, for an ESFM system A with type $\langle P, T, nodes, roots, peers, \mathcal{Y} \rangle$, $\mathrm{OPT}_\alpha(A)$ defines an optimized form of the corresponding EIG system with type $\langle P, T, nodes, roots, peers, \mathcal{Y} \rangle$. Since the actions in $H(\alpha)$ do not form part of the external interface between the user and the system and since the actions in $\overline{\mathrm{EIG}}$ are "irrelevant" in an EIG system, we exhibit a mapping from executions of optimized ESFM systems of the form $\mathrm{OPT}_\alpha(X)$ to executions of EIG systems while hiding the actions from the set $H(\alpha)$. Now we define what it is for two schedules to be equivalent and what we mean by "reordering".

Commutativity: Two actions π_1 and π_2 of an automaton S are *commutative*, denoted $\pi_1 \| \pi_2$, iff for every state s' and s, $s'\pi_1\pi_2 s$ is an extended step of S exactly when $s'\pi_2\pi_1 s$ is an extended step of S.

Equivalence: Let β_1, β_2 be two schedules of a system S. We say that β_1 is *equivalent* to β_2, denoted $\beta_1 \approx \beta_2$, iff β_2 is a permutation of β_1 and for every state s' and s, $s'\beta_1 s$ is an extended step of S exactly when $s'\beta_2 s$ is an extended step of S.

Thus if $\beta_1 \approx \beta_2$, then for every $s_0 \in start(S)$, if s is reachable from s_0 via schedule β_1 then s is also reachable reachable from s_0 via schedule β_2. It should be noted that not all permutations are equivalent schedules, but if we permute only commutative actions then we will get equivalent executions. This is exactly what we will do in our proof, where we move all the *FINAL* and *RETURN* actions to the end. Let $\Sigma_{\mathrm{FR}}(X) = \{FINAL([\![\sigma, g]\!], v), RETURN([\![\sigma, g]\!], v) \in acts(X)\}$, and let $\overline{\Sigma}_{\mathrm{FR}}(X)$ be the remaining actions of X for an EIG or an ESFM system X.

Reordering: Let α be an execution of A, an ESFM system. We define $\beta = reorder(\alpha)$, the *reordering* of the schedule of α as follows: If $\beta_1 = \alpha | \overline{\Sigma}_{\mathrm{FR}}(A)$, and $\beta_2 = \alpha | \Sigma_{\mathrm{FR}}(A)$, then $\beta = \beta_1 \beta_2'$ where β_2' is obtained from the sequence β_2 after performing a "stable sort" over its actions in decreasing length of σ, where $[\![\sigma, g]\!]$ is the first component of the actions for some $g \in P$.

Let $\mathrm{INT}(g) = acts([\![\lambda, g]\!]) - \{CREATE([\![\lambda, g]\!], v_i, Q, m), RETURN([\![\lambda, g]\!], v_f) \in acts([\![\lambda, g]\!])\}$ and let $U_g = acts(Hide_{\mathrm{INT}(g)}([\![\lambda, g]\!]))$, for $g \in NF$. Intuitively, U_g defines the set of external actions of

a "user process." The following lemma says that the reordering does not change the sequence of actions seen at each user process U_g.

Lemma 10: Let $U_A = \{U_g : g \in NF\}$ for an ESFM system A of type $\langle P, T, nodes, roots, peers, \mathcal{Y}\rangle$. If α is an execution of A, then $scheds(\alpha)|U_g = reorder(\alpha)|U_g$ for all $U_g \in U_A$.

Proof: Let α be an execution of A, let $\beta = sched(\alpha)$, and let $\beta_r = reorder(\alpha)$. Note that the *reorder* mapping moves only the *RETURN* actions. Once the 'status' of an automaton becomes 'fixed', it can be changed only by the *RETURN* action. Thus when a *RETURN* action becomes enabled, it remains enabled forever until it occurs in β. In the construction of β_r we only move the *RETURN* actions past other *CREATE* actions. So, β_r is still a possible schedule of A, and that the order of occurrence of *CREATE* and *RETURN* actions is the same from the point of view of each U_g. In other words for each U_g we have for all $U_g \in U_A$, $\beta|U_g = \beta_r|U_g$. ■

We now show that every ESFM execution has an equivalent canonical execution.

Lemma 11: For every ESFM system A and $\alpha \in execs(A)$ with schedule β, there exists $\alpha' \in execs(A)$ with schedule β' such that $\beta' = reorder(\alpha)$ and $\beta \approx \beta'$.

Proof: Let $\beta' = reorder(\alpha)$. We have to show that $\beta \approx \beta'$. Since in reordering we move only the *FINAL* and *RETURN* output actions to the end, we just have to show that for any action $\pi \in acts(A)$ such that π occurs after $RETURN([\![\sigma, g]\!], v), \pi \| FINAL([\![\sigma, g]\!], v)$ and $\pi \| RETURN([\![\sigma, g]\!], v)$ hold for all possible values of $[\![\sigma, g]\!]$ and v. The preconditions of the *FINAL* action says that $s_{[\![\sigma,g]\!]}$.status = 'running'. Observe that $FINAL([\![\sigma, g]\!], v)$ is the only action in the whole system which can take the automaton $[\![\sigma, g]\!]$ from 'running' to another state when the stop early condition does not hold. Since the output action *SEND* and the input actions *RETURN* and *RECV* of the automaton $[\![\sigma, g]\!]$ have no effect on the preconditions of *FINAL*, this action remains enabled once it is enabled. The other actions of the automaton $[\![\sigma, g]\!]$ are enabled only when $s_{[\![\sigma,g]\!]}$.status = 'running'. Thus $\pi \| FINAL([\![\sigma, g]\!], v)$ holds for all π occurring after $RETURN([\![\sigma, g]\!], v)$. The argument for $RETURN([\![\sigma, g]\!], v)$ is similar.

Since $\beta \approx \beta'$, from the definition of equivalence β' is also a schedule of ESFM. We choose α' to be the execution corresponding to β'. ■

We now define the mapping h from states of ESFM to states of EIG. Let S_{ESFM} be the state of an ESFM system of type $\langle P, T, nodes, roots, peers, \mathcal{Y} \rangle$ and let S_{EIG} be the state of an EIG system of the same type. Let s and t be the state components of the automaton $[\![\sigma, g]\!]$ in S_{ESFM} and S_{EIG} respectively. We say that S_{EIG} is a possibility of S_{ESFM} (i.e. $S_{\text{EIG}} \in h(S_{\text{ESFM}})$) iff for all $[\![\sigma, g]\!] \in nodes$, whenever $[\![\sigma, g]\!]$ is "alive" in both s and t, we have:

$$
\begin{array}{rcl}
s.P & \subseteq & t.P \\
s.n & \leq & t.n \\
s.t & \in & \{t.t, \perp\} \\
s.\text{child} & \sqsubseteq & t.\text{child} \\
s.\text{created} & \subseteq & t.\text{created} \\
s.\text{pending\_sends} & \supseteq & t.\text{pending\_sends} \\
\text{for } v \in \{0,1\}\ s.\text{ret}[v] & \leq & t.\text{ret}[v] \\
\text{for } v \in \{0,1\}\ s.v_{\text{initial}} = v & \Leftrightarrow & t.v_{\text{initial}} = v \\
s.v_{\text{final}} = v \wedge s.\text{status} = \text{`fixed'} & \Leftarrow & t.v_{\text{final}} = v \wedge t.\text{status} = \text{`fixed'}
\end{array}
$$

where we define

$$
\text{s.child} \sqsubseteq \text{t.child} \equiv (\sigma j, v) \in \text{s.child} \Longrightarrow \begin{cases} (\sigma j, v) \in t.\text{child} & \text{if } j \in NF \\ \exists v' \in \{0,1\} : (\sigma j, v') \in t.\text{child} & \text{otherwise} \end{cases}
$$

We now show that h is a possibilities mapping from the states of the automata in ESFM to the states of the automata in EIG. This requires demonstrating that each of the two conditions for possibilities mapping, as defined in Section 2.7 is satisfied by h. We begin by showing, in Lemma 12, that h is a possibilities mapping provided that ESFM never erroneously detects a non-faulty processor as faulty by expanding the children of automata in EIG which are never created in ESFM and hiding the corresponding executions. Then we show, in Lemma 13, that ESFM does indeed detect only "true" faults. The desired result follows as Theorem 14. The following definitions are used in the proofs below. Let B be an EIG system of type $\langle P, T, nodes, roots, peers, \mathcal{Y} \rangle$ and let α be an execution of B.

For $\alpha \in execs(B)$, we define $expansion(\alpha, \sigma)$ to be a sequence of actions $\beta_1\beta_2$ such that β_1 is the schedule of α and $\beta_2 = (\alpha' | \overline{\Sigma}_{\text{FR}}(B)) | \bigcup acts([\![\tau, h]\!])$ where $h \in NF$, σ is a prefix of $\tau \in P^{\bullet}_{t+1}$, and $\alpha\alpha'$ is an execution of B after which no actions are enabled.

For $\alpha \in execs(B)$, we define $compression(\alpha, [\![\sigma, g]\!])$ to be a sequence of actions β such that $\beta = (\alpha|\Sigma_{\mathrm{FR}}(B))|\bigcup acts([\![\tau, g]\!])$ where σ is a proper prefix of $\tau \in P^{\bullet}_{i+1}$.

Informally, $expansion(\alpha, g)$ is the sequence of actions from α followed by the sequence of actions consisting only of *CREATE*, *SEND*, and *RECV* actions from the descendants of the automaton $[\![\sigma, g]\!]$ and its cousins; and $compression(\alpha, [\![\sigma, g]\!])$ is the subsequence of α consisting of the *FINAL* and *RETURN* actions from the proper descendants of the automaton $[\![\sigma, g]\!]$. We use expansion while growing the EIG computation tree and use compression while taking the bottom up majority to fix the final values.

Lemma 12: Suppose that every fault detected by an ESFM system of type $\langle P, T, nodes, roots, peers, \mathcal{Y}\rangle$ is an element of T. Then for all ESFM systems A of type $\langle P, T, nodes, roots, peers, \mathcal{Y}\rangle$ with executions α, there exists an execution α' with $sched(\alpha') \approx sched(\alpha)$ and an EIG system B with system type $\langle P, T, nodes, roots, peers, \mathcal{Y}\rangle$ such that h is a possibilities mapping from $\mathrm{OPT}_{\alpha}(A)$ to $\mathrm{OPT}_{\alpha}(B)$ for execution α'.

Proof: Let α' be an execution of A with schedule $\beta = reorder(\alpha)$. (We know by Lemma 11 that α' exists.) Let $A^o = \mathrm{OPT}_{\alpha}(A)$, $B^o = \mathrm{OPT}_{\alpha}(B)$ and let s and t denote the states of the automata A^o and B^o respectively.

For the start state s_0 of α' we have $t_0 \in h(s_0)$ such that t_0 is the start state of B^o. Since there is no automata in either A^o or B^o that is 'alive', the first condition of the possibilities mapping is met trivially.

To show that the second condition of the mapping also holds we consider each action π individually. Let (s', π, s) be a step of α', let $t' \in h(s')$, and let $\alpha'_B t'$ be an execution of B^o. We have to find a γ such that (t', γ, t) is an extended step in B^o and $t \in h(s)$ such that $\pi|ext(A^o) = \gamma|ext(B^o)$. Let α_B be the execution of B^o after one step in the simulation (i.e. $\alpha_B = \alpha'_B t'\gamma$).

- $\pi = CREATE([\![\sigma, g]\!], v, Q, m)$. If $\pi \in acts(\bot(\alpha))$ then choose $\gamma = CREATE([\![\sigma, g]\!], v, Q, m')$ such that $m' = s_{[\![\lambda, g]\!]}.t$, otherwise choose $\gamma = \pi$. Thus in both cases we have $\pi|ext(A^o) = \gamma|ext(B^o)$. Since m' is chosen such that $m \in \{m', \bot\}$ and since the *CREATE* action has the same effect over other state variables in both systems, the mapping is preserved.

- $\pi = SEND(g, [\![\sigma, h]\!], v)$. Since this action has exactly the same effect in both A^o and B^o, we simply let $\gamma = \pi$ which trivially satisfies the mapping.

- $\pi = RECV(j, [\![\sigma, g]\!], v)$. There are two cases. Case (i) $j \in T$: because of hiding, $\pi|ext(A^o)$ is empty. Let $CREATE((([\![\sigma j, g]\!], v', Q, m)$ be the child created as the result of the action π. We let $\gamma = RECV(j, [\![\sigma, g]\!], v')$. Since this *RECV* action is hidden in both the systems the choice of γ satisfies the condition $\pi|ext(A^o) = \gamma|ext(B^o)$. Case (ii) $j \in NF$: let $\gamma = \pi$. In either case we have $s_{[\![\sigma,g]\!]}$.child $\sqsubseteq t_{[\![\sigma,g]\!]}$.child. Since 'child' is the only state variable that the *RECV* action has any effect on, the mapping is preserved by our choice of γ.

- $\pi|ext(A^o)$ = empty and π is $MASK\_FAULT([\![\sigma, g]\!])$. The state component s.child may be changed to replace the values received from faulty processors to a default. Since ESFM detects only true faults, the relation s.child $\sqsubseteq$ t.child still holds after the *MASK_FAULT* action. So, we simply let γ = 'null', which satisfies the other condition for possibilities mapping ($\pi|ext(A^o)$ = 'null' = $\gamma|ext(B^o)$).

- $\pi|ext(A^o)$ = empty and π is one of *UPDATE*, or *DISCOVER_FAULT*, or *MARK*, or $SEND(g, [\![\lambda, h]\!], \tau)$, or $RECV(j, [\![\lambda, h]\!], \tau)$. Since these actions do not affect any variable that appears in the state mapping, they do not change the state mapping. So letting γ = 'null' trivially satisfies the other condition $\pi|ext(A^o)$ = 'null' = $\gamma|ext(B^o)$.

- $\pi|ext(A^o)$ = empty and π is either $STOPPED\_EARLY([\![\sigma, g]\!], v)$ or $STOP\_EARLY([\![\sigma, g]\!])$. The effect of *STOP_EARLY* is to fix the final value in the state s. Since in state t of B^o the status of $[\![\sigma, g]\!]$ is not yet fixed the mapping holds. The action $\pi = STOPPED\_EARLY$ has the same effect as *STOP_EARLY* or decreases the value of the local variables $s.n$ and $|s.P|$. In the latter case, since corresponding variables are not decreased by B^o in the state t, we still have $s.P \subseteq t.P$ and $s.n \leq t.n$.

- $\pi|ext(A^o)$ =empty such that the remaining actions of α consist only of either the *FINAL* or the *RETURN* actions. It is at this point that we expand the EIG trees in B^o to "keep up" with ESFM. Note that creating those automata in B^o which are never created in A^o does not disturb the mapping. This is because the creation action only adds to the set t.created in B^o and the new automata are not alive in A^o. Further any actions concerning these automata are hidden in both systems leading to $\pi|ext(A^o)$ = 'null' = $\gamma|ext(B^o)$. More formally, the sequence α_B has as its schedule $expansion(\alpha'_B, \lambda)$.

- $\pi = RETURN(\llbracket \sigma, g \rrbracket, v)$. There are two cases. Case (i): If $v \in \{0,1\}$ then we let $\gamma = \pi$ and the mapping is trivially satisfied. Case (ii): If $v = \perp$ then we let $\gamma = FINAL(\llbracket \sigma, g \rrbracket, v')RETURN(\llbracket \sigma, g \rrbracket, v')$ such that $FINAL(\llbracket \sigma, g \rrbracket, v')$ is enabled in B^o. Since we hide these actions in both the systems we have $\pi|ext(A^o) =$ 'null' $= \gamma|ext(B^o)$. In either case the possibilities mapping holds because the variable 'ret' of the parent of the node $\llbracket \sigma, g \rrbracket$ is the only variable affected, and in either case we increment this variable in the system B^o. Note that after the *RETURN* action the node $\llbracket \sigma, g \rrbracket$ is no longer 'alive'. Since $\llbracket \sigma, g \rrbracket$ is not alive in both the systems, the mapping of this node between the two systems A^o and B^o is met trivially. Note that case (ii) occurs when the parent $\llbracket \tau, g \rrbracket$ of the node $\llbracket \sigma, g \rrbracket$ in A^o stops early and returns with an action $RETURN(\llbracket \tau, g \rrbracket, u)$. We must show that the node $\llbracket \tau, g \rrbracket$ still returns u as its final value in the system B. The proof is simple. Since $\llbracket \tau, g \rrbracket$ stopped early, $s_{\llbracket \tau, g \rrbracket}.\text{child}[u] \geq n - t - |\tau|$ and at least $n - 2t$ of its children $\llbracket \tau j, g \rrbracket$ such that $j \in NF$ have initial value u. From the correspondence exhibited earlier for the *SEND*, *RECV*, and *CREATE* actions, it follows that the same holds in the system B. From Lemma 5(s3), we know that at least $n - 2t$ nodes of the form $\llbracket \tau, j \rrbracket$ such that $j \in NF$ had u as their initial values. Since $n - 2t > (n - t - |\tau|)/2$, from Lemma 5(s5) and (s4) we will have u as the eventual final value of $\llbracket \tau, g \rrbracket$ in the system B also.

- $\pi = FINAL(\llbracket \sigma, g \rrbracket, v)$. Whenever such an action occurs in B^o we recursively fix the final values of all the children of $\llbracket \sigma, g \rrbracket$ and allow them to return. Once every child has returned the *FINAL* action of the automaton $\llbracket \sigma, g \rrbracket$ becomes enabled. More formally, the sequence α_B has as its schedule the concatenation of $compression(\alpha'_B, \llbracket \sigma, g \rrbracket)$ and $FINAL(\llbracket \sigma, g \rrbracket, v)$. Note that since the mapping held in the states s' and t', the majority value will not change after the compression.

■

It is interesting to note that the invariants of the EIG system (Lemma 5) are used in the above proof of the ESFM system. This is because those invariants depend only on the actions of EIG and ESFM contains all those actions. The following lemma states that every fault detected in an ESFM system is an element of T. We say that a processor j is marked in a tree $\mathcal{T}_g$ if and only if

$j \in \text{Faulty}_g$, and that the shared tree data structure $\mathcal{T}_g$ is *properly marked* if and only if every fault marked in $\mathcal{T}_g$ is an element of T.

Lemma 13: If α is an execution of an ESFM system, then $\mathcal{T}_g$ is *properly marked* for all $g \in NF$. Therefore, every fault detected by an ESFM system is an element of T.

Proof: By definition $\mathcal{T}_g$ is properly marked initially. Let $s'\pi s$ be a subsequence of α and let $\mathcal{T}_g$ be *properly marked* in state s'. The *DISCOVER_FAULT* action of the ESFM system is the only action which detects a processor to be faulty. Let $[\![\sigma, g]\!] = [\![\sigma' j, g]\!]$ be an automaton in the ESFM system and let $F = \left|\sigma \cap s'_{[\![\sigma,g]\!]}.\text{faulty}\right|$. By the preconditions of the *DISCOVER_FAULT* action, j is marked faulty in the tree $\mathcal{T}_g$ if for each $v \in \{0,1\}$ there exist at least $t + 1 - F$ children of $[\![\sigma, g]\!]$ with initial value v.

Let $n = |P|$, j be non-faulty and $s_{[\![\sigma',j]\!]}.v_{\text{initial}} = v_j$. From Lemma 5(s3) all automata of the form $[\![\sigma' j, h]\!]$ such that $h \in NF$ have v_j as their initial values and the automaton $[\![\sigma, g]\!] = [\![\sigma' j, g]\!]$ has at least $n - |\sigma| - (t - F)$ children with initial value v_j. So there can be at most $t - F$ children with values different from v_j. Thus an ESFM system will not detect j as faulty. ■

Since for all actions in an ESFM system we have the possibilities mapping maintained and since ESFM detects only true faults, we conclude that for reordered executions of an ESFM system there exist an EIG systems exhibiting the corresponding executions. Note that formulating the algorithms in terms of dynamic process creation has permitted us to reason locally about each recursive call. That is, the possibilities mapping is structured as a relationship between the corresponding recursive calls in the two systems. This is one benefit of presenting the EIG algorithm using dynamic process creation.

Theorem 14: ESFM systems simulate EIG systems with respect to the users U (Section 3.1).

Proof: Lemma 12 together with Lemma 13 says that for every execution of an ESFM system there exists an EIG execution corresponding to the reordered execution of the ESFM execution. In Lemma 11, we have shown that for every execution of an ESFM system there is an equivalent reordered execution, and from Lemma 10 we know that the external behavior of the users is not affected by *reordering*. So, from Corollary 2, we see that ESFM systems simulate EIG systems. ■

6 Cloture Voting

In the cloture voting protocol each non-faulty processor runs several agreement protocols, called *debates*, concurrently. A new debate is initiated at each of the $t+1$ rounds. The first debate is started at each processor with the initial value assigned to that processor by the environment. If it can be determined that there has been sufficient communication to safely assume that both the values 0 and 1 are represented as initial values of non-faulty processors, then a subsequent debate may be started with initial value 0. Otherwise, each subsequent debate is started with initial value 1. So, if any debate returns 0 the default decision value may be taken immediately. The debate that is started at round k, *debate*(k), is exactly same as the ESFM protocol except that it runs for $t+1-k$ rounds and that it has some a priori information about faulty processors, communicated to it by other debates. The protocol guarantees that if any of the debates return zero, then at least one correct processor has started with an initial value of zero.

6.1 The Algorithm

We define a cloture system type to be a six-tuple $\langle P, T, nodes, roots, peers, \mathcal{Y} \rangle$ similar to the ESFM system type. The nodes in a cloture system type have three components: debate number, node name, and processor name. Thus, we define $nodes_g = \{[\![d, \sigma, g]\!] \mid 0 \leq d < t+1, \sigma \in P^{\bullet}_{t+1}\}$. For $g \in \Sigma$ we define a *cloture-forest* for g with set Σ and index x to be a collection of EIG-trees for g with set Σ and index i, for $x \geq i \geq 0$, where the nodes in each EIG-tree are of the form $[\![i, \sigma, g]\!]$ such that $\sigma \in \Sigma^{\bullet}_{i+1}$. The trees of names of automata $\mathcal{Y} = \{\mathcal{Y}_g \mid g \in NF\}$, where $\mathcal{Y}_g$ is a cloture-forest for g with processors P and index t. A cloture system is an instance of the cloture system type with $n > 4t$. As in ESFM, we model explicitly the information shared among the nodes on a given processor g by a shared data structure $\mathcal{T}_g$. $\mathcal{T}_g$ is similar to the corresponding ESFM data structure, except that $\mathcal{T}_g$ is an array of trees of decreasing size. Similarly, Stopped_g is now an array of sets, one for each debate. Every debate(k) reads its own tree $\mathcal{T}_g[k]$ and reports a detected fault to the other debates by marking a particular vertex in all trees in $\mathcal{T}_g$. The state information of each node in the shared tree is used by the monitor in determining the initial values of the debates. The transition relation for the automaton $[\![d, \sigma, g]\!]$ is given in Figure 7.

The cloture debate given here is strikingly similar to the ESFM algorithm except for the an

Input Actions:

- $CREATE(\llbracket d, \sigma, g \rrbracket, v, Q, m)$
  E: $s.v_{\text{initial}}, s.v_{\text{final}} \leftarrow v, v$
  $\mathbf{set}(\sigma, \mathcal{T}_g[d])$
  $s.P, s.n, s.t \leftarrow Q, |Q|, m$
  if $(|\sigma| < t + 1 - d \wedge m \neq \perp)$
  then $s.\text{status} \leftarrow$ 'running'
  else $s.\text{status} \leftarrow$ 'fixed'
  if $|\sigma| \leq t + 1 - d$
  then $s.\text{pending\_sends} \leftarrow Q$
  if $m \neq \perp$ **then** $s.v_{\text{final}} \leftarrow v$
- $RECV(j, \llbracket d, \sigma, g \rrbracket, v)$, where $j \in s.P \wedge v \in \{0, 1\}$
  E: **if** $(\sigma j, 0) \notin s'.\text{child}$
  then $s.\text{child} \leftarrow s'.\text{child} \cup \{(\sigma j, v)\}$
- $RETURN(\llbracket d, \sigma j, g \rrbracket, v)$
  E: **if** $v \neq \perp$ **then** $s.\text{ret}[v] \leftarrow s'.\text{ret}[v]+1$

Internal Actions:

- $UPDATE(\llbracket d, \sigma, g \rrbracket)$
  P: $s'.\text{status} =$ 'running'
  E: $s.\text{halted} = \mathbf{getstopped}(Stopped_g[d], \sigma)$
  $\text{s.faulty} \leftarrow \mathbf{getfaults}(Faulty_g)$
- $STOPPED\_EARLY(\llbracket d, \sigma, g \rrbracket)$
  P: $\llbracket d, \sigma', h \rrbracket \in s'.\text{halted}$
  $\llbracket d, \sigma', h \rrbracket \notin s'.\text{marked}$
  where $\sigma = \sigma' i j \gamma$
  $s'.\text{status} \neq$ 'returned'
  E: $s.\text{marked} \leftarrow s'.\text{marked} \cup \{\llbracket d, \sigma', h \rrbracket\}$
  if $(g = h \vee h \in \gamma)$ **then**
  $s.\text{status}, s.v_{\text{final}} \leftarrow$ 'fixed', $\perp$
  else if $(i \neq h \wedge j \neq h)$ **then**
  $s.P, s.n \leftarrow s'.P - \{h\}, s'.n - 1$
- $DISCOVER\_FAULT(\llbracket d, \sigma, g \rrbracket)$
  P: $s'.\text{status} =$ 'running'
  $\text{child}[0] \geq s'.t + 1 - |\sigma \cap s'.\text{faulty}|$
  $\text{child}[1] \geq s'.t + 1 - |\sigma \cap s'.\text{faulty}|$
  E: $\text{s.faulty} \leftarrow s'.\text{faulty} \cup \{tail(\sigma)\}$
  $\mathbf{setfault}(\text{Faulty}_g, tail(\sigma))$

Output Actions:

- $CREATE(\llbracket d, \sigma j, g \rrbracket, v, Q, m)$
  P: $s'.\text{status} =$ 'running'
  $(\sigma j, v) \in s'.\text{child}$
  $j \notin s'.\text{created}$
  $(\exists v' :: \text{child}[v'] \geq (s'.n - s'.t)) \Rightarrow m = \perp$
  $\neg(\exists v' :: \text{child}[v'] \geq (s'.n - s'.t)) \Rightarrow m = s'.t$
  $Q = s'.P - \{j\}$
  E: $s.\text{created} \leftarrow s'.\text{created} \cup \{j\}$
- $MASK\_FAULT(\llbracket d, \sigma, g \rrbracket)$
  P: $s'.\text{status} =$ 'running'
  E: $s.\text{child} \leftarrow s'.\text{child} - \{(\sigma j, v) \mid j \in s.\text{faulty}\}$
  $+\{(\sigma j, 0) \mid j \in s.\text{faulty} \wedge j \in s.P\}$
- $STOP\_EARLY(\llbracket d, \sigma, g \rrbracket, v)$
  P: $s'.\text{status} =$ 'running'
  $s'.\text{pending\_sends} = \emptyset$
  $s'.P \subseteq s'.\text{created}$
  $\text{child}[v] \geq (s'.n - s'.t)$
  E: $\mathbf{prune}(\sigma, \mathcal{T}_g[d])$
  $s'.\text{status}, s.v_{\text{final}} \leftarrow$ 'fixed', v
- $FINAL(\llbracket d, \sigma, g \rrbracket, v)$ where $v \in \{0, 1\}$
  P: $s'.\text{status} =$ 'running'
  $s'.\text{pending\_sends} = \emptyset$
  $s'.P \subseteq s'.\text{created}$
  $v = 0 \Rightarrow s'.\text{ret}[0] \geq s'.n/2$
  $v = 1 \Rightarrow s'.\text{ret}[1] > s'.n/2$
  E: $\mathbf{prune}(\sigma, \mathcal{T}_g[d])$
  $s'.\text{status}, s.v_{\text{final}} \leftarrow$ 'fixed', v
- $SEND(g, \llbracket d, \sigma, h \rrbracket, v)$
  P: $h \in s'.\text{pending\_sends}$
  $v = s'.v_{\text{initial}}$
  E: $s.\text{pending\_sends} \leftarrow s'.\text{pending\_sends} - \{h\}$
- $SEND(g, \llbracket d, \lambda, h \rrbracket, \sigma)$ where $h \in P$
  P: $(\exists v' :: \text{child}[v'] \geq (s'.n - s'.t))$
  $h \notin s'.\text{stop\_signals}$
  $s'.\text{stop\_signals} \subset P$
  E: $s.\text{stop\_signals} \leftarrow s'.\text{stop\_signals} \cup \{h\}$
- $RETURN(\llbracket d, \sigma, g \rrbracket, f)$
  P: $s'.\text{status} =$ 'fixed'
  $f = s'.v_{\text{final}}$
  E: $s.\text{status} \leftarrow$ 'returned'

Figure 7: Transition relation for automaton $\llbracket d, \sigma, g \rrbracket$ in the cloture voting protocol.

Input Action:

- $RECV(j, \llbracket d, \lambda, g \rrbracket, \tau)$, where $j \in P$ and $\tau \in P^*$
  E: $\text{Stopped}_g[d] \leftarrow \text{Stopped}_g[d] \cup \{(\llbracket d, \tau, j \rrbracket)\}$

Figure 8: Additional action for automaton $\llbracket d, \lambda, g \rrbracket$ in the cloture voting protocol.

extra component in the node names (indicating the debate number). Since debate(k) runs only for $t+1-k$ rounds the *CREATE* input action fixes the final value of the automaton that is created in round $t+1-k$ as this will be one of the leaf nodes in that debate. There is a set of special actions for the nodes of the form $[d, \lambda, g]$ similar to that in ESFM (see Figure 8). These actions mark in the tree $\mathcal{T}[d]$ the nodes that have stopped early in debate d.

Input Actions:

- $CREATE([\lambda, g], v, Q, t)$
  E: $s.\text{alive} \leftarrow$ 'true'
  $s.P, s.t \leftarrow Q, t$
  $s.\text{nextdebate} \leftarrow 0$
  $s.\text{initial} \leftarrow v$
- $RETURN([d, , \lambda, g], v_f)$
  E: $s.\text{outcome}[d] \leftarrow v_f$

Internal Actions:

- $UPDATE$
  P: $s'.\text{alive}$
  E: $s.\text{faulty} \leftarrow \textbf{getfaulty}(\mathcal{T}_g)$

Output Actions:

- $CREATE([d, \lambda, g], v_i, Q, m)$
  P: $s'.\text{alive}$
  $d < s'.t + 1$
  $d = s.\text{nextdebate}$
  $f \subseteq s'.\text{faulty}$
  $d = 0 \Rightarrow v_i = s'.\text{initial}$
  $(v_i = 0 \wedge d > 0) \Rightarrow (|f| = d \wedge NTS \geq S)$
  $m = s'.t - |f|$
  $Q = s'.P - f$
  E: $s.\text{nextdebate} \leftarrow s'.\text{nextdebate} + 1$
- $RETURN([\lambda, g], v_f)$
  P: $s.\text{alive}$
  $v_f = \min\{s'.\text{outcome}\} \neq \perp$
  E: $s.\text{alive} \leftarrow$ 'false'

Figure 9: Transition relation for the **monitor** of processor g

We model the root-node for each processor as an automaton we call the *monitor*. This monitor creates a total of $t+1$ debates (see Figure 9). The initial value of the first debate is the initial value assigned to that processor by the environment. The initial values of other debates are determined by whether the total number of nodes whose values are yet to be sent exceeds a threshold $S = 2n^3$ [7]. This total is computed by examining the shared data structure $\mathcal{T}_g$. Let i^{th} *level* of a tree $\mathcal{T}_g[k]$ denote the set of all nodes $[k, \sigma, g]$ in $\mathcal{T}_g[k]$ such that $|\sigma| = i$. We define $NTS(\mathcal{T}_g[d])$ to be the number of nodes in the deepest level of the tree $\mathcal{T}_g[d]$ such that every node in that level is either marked by the "set" operation or by the "prune" operation and $NTS(\mathcal{T}_g)$ is the maximum value of $NTS(\mathcal{T}_g[d])$ such that $0 \leq d \leq t$. The maximum number of the nodes, whose initial values are to be communicated, computed this way across all the debates within the processor g is compared with the threshold S. The state of the monitor consists of the variables *alive* which when set indicates that the monitor is active, *nextdebate* which is the number of the next debate, and *initial* which is the initial value of the first debate, and an array *outcome* which accumulates the return values from all the debates. Note that the set of processors and the number of faulty processors can be

smaller in later debates if enough faults were detected earlier.

6.2 Correctness Proof

We show, using another possibilities mapping, that each debate of the cloture voting algorithm simulates an ESFM system. As an intermediate system for our proof, we define an ESFM′ system to be exactly the same as the ESFM system, except that it has an *oracle* that "miraculously" marks nodes as faulty in the shared data structure of ESFM at arbitrary points in the execution. More formally, an *oracle* is an automaton whose only action is to mark some arbitrary nodes as faulty in the shared data structure of an ESFM system using the **setfault**$(\mathcal{T}_g, j)$ action. We say that an oracle is *correct* if it never marks a non-faulty processor as faulty. We show that, provided the oracle is correct, every ESFM′ system simulates an ESFM system.

To prove the correctness of the cloture algorithm, we show a correspondence between the executions of each debate and the executions of an ESFM′ system. For every step of the debate, ESFM′ also takes a step except that when the debate masks a fault based on a fault detected in some other debate, ESFM′ waits until the oracle marks such a processor to be faulty. Thus, the oracle models the fault detection information propagating back and forth between different debates. By neatly capturing the information (faults detected) contributed by the other debates, the oracle permits reasoning about each debate in isolation.

Unlike Berman, Garay, and Perry [7], we describe the algorithm in an asynchronous setting that allows debates to be created at any time, nondeterministically. A problem with this is that processors might prematurely start debate(k) when only less than k faults are detected. Therefore, we stipulate the following "cloture conditions (CCs)" and prove the correctness of the cloture algorithm as a conditional property based on these hypotheses. In other words, we prove that if an execution of a cloture system satisfies these conditions, then it corresponds to a correct execution of an ESFM. So, any possible implementation of our specification of the cloture algorithm that satisfies these conditions is a correct implementation. At the end of this section, we show that a synchronous implementation is one such possible implementation.

The Cloture Conditions: Let α be an execution of a cloture system.

(CC1) For every automaton $[\![d, \sigma, g]\!]$, a *FINAL* action action occurs in α only if the

STOP_EARLY action is not enabled and no later *RECV* action can result in $s.\text{child}[v] = s.n - s.t$ for some $v \in \{0, 1\}$.

(CC2) If there exists a processor g and debate(k) such that $CREATE(\llbracket k, \lambda, g \rrbracket, 0, Q, t - |f|)$ occurs in α, then for every processor $h \in NF$ there exists a v_i such that a $CREATE(\llbracket k, \lambda, g \rrbracket, v_i, Q, t - |f|)$ occurs in α and that $|f| = k$.

(CC3) Every $CREATE(\llbracket d, \lambda, g \rrbracket, v, Q, m)$ action occurs after all create actions of the form $CREATE(\llbracket d', \sigma, g' \rrbracket, v', Q', m')$ such that $d > d' + |\sigma|$ for all $\sigma \in P^\bullet$ and occurs before any create action of the form $CREATE(\llbracket d', \sigma, g' \rrbracket, v', Q', m')$ such that $d < d' + |\sigma|$ for all $\sigma \in P^\bullet$.

The first cloture condition (CC1) states that every automaton reports its final value only after receiving enough values. In other words, we want it to be the case that if an automaton will be able to stop early on the basis of its initial values, then it will do so on that basis, and not on the basis of values returned by its children. The second condition (CC2) states that if a monitor of a processor starts debate(k) with initial value 0 then it must be the case that this monitor knows about a set of k faulty processors and that this set is also known to other monitors. This information is necessary to ensure agreement, since the debate has fewer than $t + 1$ rounds. The last condition (CC3) states that the monitor creates exactly one debate in each round.

Lemma 15: If the oracle is correct, every fault detected by an ESFM′ system is an element of T.

Proof: The proof follows directly from Lemma 13. ■

Theorem 16: Every ESFM system with type $\langle P, T, nodes, roots, peers, \mathcal{Y} \rangle$ and a correct oracle (the ESFM′ system) simulates an ESFM system.

Proof: The possibilities mapping h defined in Section 5.2 holds from ESFM′ systems to ESFM systems as well. Since we hide all the actions involving faulty processors, the proof is similar. ■

Theorem 17: If the cloture conditions are satisfied, every debate in the cloture system with type $\langle P, T, nodes, roots, peers, \mathcal{Y} \rangle$ simulates an ESFM′ system (an ESFM system with type $\langle P, T, nodes, roots, peers, \mathcal{Y} \rangle$ and a correct oracle $\mathcal{O}$).

Proof: There are two cases, depending on whether the debate is started with initial value 0 at some processor. Let debate(k) be started with an initial value 0 by some processor. From (CC2) we have a set $f \subseteq T$ globally known as faulty; each debate starts with $n-k$ processors with exactly $t-k$ of them faulty such that $|f| = k$. Since $n-k > 4(t-k)$ debate(k) is an ESFM system with type $\langle P-f, T-f, nodes, roots, peers, \mathcal{Y} \rangle$.

The second case is where all debate(k) start with the initial value 1. Let s be the state of the root automaton of this debate. From the safety property (s3) of Lemma 5 we have at least $s.n - s.t$ children of the root with an initial value of 1. After the root sends its initial value to other processors, the *STOP_EARLY* action will be enabled. Due to (CC1) all these debates stop early with the final value 1. Thus all these debates return 1 as their final value.

Since every debate is an ESFM system, the first fault detected by the cloture voting system is in T due to Lemma 13. This fault is known to all the debates in the system. (Note that we use the oracle $\mathcal{O}$ to model the faults reported from other debates.) Every debate continues to detect only correct faults due to Lemma 15. Thus a debate is essentially an ESFM′ system with a correct oracle. ■

Since each debate is a correct ESFM′ protocol, the remainder of the proof is concerned only with proving certain properties of the monitor, which secures the final outcomes from different debates.

Lemma 18: A non-faulty processor starts a debate with value 0 only if there is at least one non-faulty processor with $v_{\text{initial}} = 0$.

Proof: Let all non-faulty processors start with an initial value of 1. From Lemma 5(s3) we know that at least $n-t$ children of the root node in each processor is created with the same initial value 1. The *CREATE* output action of an automaton creates at least one child with $m = \perp$. Since this child never creates any of its own children, this will be at the deepest level where all the automata are either set to 'alive' or marked by prune. Hence for any processor g, $NTS(\mathcal{T}_g)$ does not exceed n, which is well below the threshold S. So the monitor never starts a debate with an initial value of 0. ■

Lemma 19: If all non-faulty processors start with a value 0 then there is at least one debate which returns 0.

Proof: The first debate in all processors are started with the initial value provided by the environment. Since an ESFM′ system simulates an EIG system, the first debate in all processors return the value 0. ■

Theorem 20: If the cloture conditions are satisfied, the cloture algorithm simulates ESFM.

Proof: From Theorem 16 and Theorem 17 we know that every debate in the cloture system simulates an ESFM system. Since all monitor automata use the same function to determine the final decision value, from Lemma 18 and 19 it follows that the cloture algorithm simulates ESFM if the cloture conditions are satisfied. ■

We now give a definition of a synchronous execution of the cloture voting algorithm and show that the cloture conditions are satisfied in all such executions. This is followed by a straightforward theorem to complete the proof.

Define $LEVEL(r,g) = \prod_{0 \le d < t+1} [\![d, \sigma, g]\!]$ such that $|\sigma| = r - d$. That is, the automaton $LEVEL(r,g)$ includes all automata in the processor g that are created during round r in any of its debates. Let α be an execution of the cloture voting system and let

$$\begin{aligned} \Pi(r,g) \quad = \quad & acts(LEVEL(r,g)) \cup \\ & \{SEND(g, [\![d, \lambda, h]\!], \tau), RECV(h, [\![d, \lambda, g]\!], \tau), MARK([\![d, \lambda, g]\!], [\![d, \tau, h]\!])\}, \\ & \text{where } h \in P, 0 \le d \le t, \tau \in P^\bullet \text{ and } |\tau| = r - d. \end{aligned}$$

Let $\beta = \alpha | \Pi(r,g)$. We say that α is *synchronous* iff

(Syn1) The actions of β occur in the following order: first all *CREATE* input actions, then all *SEND* actions that send the initial values to other processors, then *RECV* actions that receive initial values from other processors, then any *DISCOVER_FAULT*s, then all *UPDATE*s, then any *MASK_FAULT*s, then any *STOP_EARLY*s, then any *SEND* actions that inform other processors about stopping early, then *RECV* actions that receive information from other processors about stop early, then all

*STOPPED_EARLY*s, then any *CREATE* output actions, then any *RETURN* input actions, then any *FINAL* actions, and finally all *RETURN* output actions.

(Syn2) For every automaton in the composition $LEVEL(r, g)$, if there is a *CREATE* input action for an automaton in β then there is exactly one *UPDATE*, *DISCOVER_FAULT*, *STOP_EARLY*, and output *RETURN* action for that automaton in β, if they are ever enabled.

(Syn3) If $\pi_1 \in \Pi(r_1, g)$, $\pi_2 \in \Pi(r_2, g)$ and if both π_1 and π_2 are enabled, then π_1 occurs before π_2 whenever $r_1 < r_2$.

The requirement (Syn1) makes sure that execution proceeds in rounds; it also specifies an intuitive ordering of the actions of the system. The requirement (Syn2) along with (Syn1) states that every fault detected by different automata within a processor becomes known to all automata that are created later in that processor, and (Syn3) requires that in a given round, all actions of higher level automata are first executed before any actions of lower level automata. In a synchronous execution α, we say that an action π occurs in round r if and only if $\pi \in \Pi(r, g)$ for some $g \in NF$. Note that in a synchronous execution, when an automaton y stops early due to sufficient agreement among initial values, it would still wait for its children to return before returning itself. Since every child z of y would have been created by a $CREATE(z, v_z, Q_z, \perp)$ action, z would return after sending its initial value to its cousin automata. This synchronous execution of the cloture algorithm is similar to that given in [7].

Now we will show that the synchronous execution of the cloture system satisfies the cloture conditions. The condition (CC2) states that if the monitor creates a debate with an initial value 0 in round r, then there exists a set of at least r processors known to be universally faulty at the beginning of round r. We will show that if this is not the case, then the value of $NTS(\mathcal{T}_g)$ is always below the threshold S. Let $F(r) = \left\{ j \mid j \in s_{[\lambda,g]}.\text{faulty in round } r \right\}$, where $g \in NF$ and $[\lambda, g]$ is the monitor of the processor g. That is, $F(r)$ is the set of processors universally known to be faulty at round r.

Lemma 21: Let α be a synchronous execution of the cloture system and let $\sigma = \tau i$ such that $\tau \in T^*$ and $i \in NF$. Assume that a $CREATE([d, \sigma, g], v, Q, m)$ action occurs in α in round $r \leq t$ of α. Then a $RETURN([d, \sigma, g'], v)$ occurs in round r or in round $r + 1$ in α for all $g' \in NF$.

Proof: Let $s_{[d,\tau,i]}.v_{\text{initial}} = v$. Note that $i \in NF$. Let $g' \in NF$. The node $[\![d, \sigma, g']\!]$ is created as a result of a message received due to the $SEND(i, [\![d, \tau, g']\!], v)$ action. If $r = t$, then a $RETURN([\![d, \sigma, g']\!], v)$ action occurs in α. Otherwise, in the next round of $r+1$ $SEND(g', [\![d, \sigma, h]\!], v)$ occurs in α for all $h \in P - \sigma$. Since the corresponding *RECV* actions follow the *SEND*s in a synchronous execution, we will have $\text{child}[v] \geq n - t - 1 \geq n - t - |\sigma|$ for all nodes $[\![d, \sigma, g]\!]$ such that $g \in NF$. So the $STOP\_EARLY([\![d, \sigma, g]\!], v)$ actions occur in α for all $g \in NF$. The corresponding *RETURN* actions occur immediately after these actions. ∎

The above lemma says that if a node $x = [\![d, \sigma, g]\!]$ is created in round $r \leq t$ of a synchronous execution of the cloture system, then the automaton x and its cousins will be 'pruned' in the next round. Now we show that if a node $x = [\![d, \sigma, g]\!]$ stopped early in round r such that $\sigma = \tau i$ and $\tau \in T^{\bullet}$, then that automaton and its cousins will be 'pruned' in round $r + 1$.

Lemma 22: Let α be a synchronous execution of the cloture system, let $g \in NF$ and let $\sigma = \tau i$ such that $\tau \in T^{\bullet}$ and $i \in P$. Then a $RETURN([\![d, \sigma, g]\!], v)$ action occurring in α for some $g \in NF$ implies that a *RETURN* action occurs in round r or in round $r + 1$ in α for each node $[\![d, \sigma, g']\!]$ such that $g' \in NF$.

Proof: The case where $i \in NF$ is proved in Lemma 21. Consider now $\sigma \in T^{\bullet}$ and a node $[\![d, \sigma, g]\!]$ created in round $r - 1$. If $r = t$, then a $RETURN([\![d, \sigma, g']\!], v)$ action occurs for each $g' \in NF$. If $r < t$, then the $RETURN([\![d, \sigma, g]\!].v)$ action is preceded by either a *STOP_EARLY* action or a *FINAL* action. From the preconditions of these two actions, it follows that the automaton $[\![d, \sigma, g]\!]$ must have at least $n - 2t$ children $[\![d, \sigma j, g]\!]$ such that $j \in NF$ and has v as their initial value. In case of *STOP_EARLY* this requirement is trivial and in case of *FINAL* it follows from Lemma 5(s5). Note that these children are created due to the $SEND(j, [\![d, \sigma, g]\!], v)$ from $j \in NF$. Thus, $CREATE([\![d, \sigma j, g']\!], v, Q', m')$ occurs in α for at least $n - 2t$ children of $[\![d, \sigma, g']\!]$ for each $g' \in NF$. Now the result follows from Lemma 21. ∎

Now we show that if a debate is created with initial value 0 in round r, then there exists a set of at least r processors universally known to be faulty before that debate is created.

Lemma 23: If an action $CREATE([r, \lambda, g], 0, Q, m)$ occurs in an execution α of the cloture voting system, then $|F(r-3)| \geq r$.

Proof: Since g starts the debate with initial value 0, we know that for some debate d in g, $NTS(\mathcal{T}_g[d]) \geq 2n^3$ at the beginning of round r. Let $d = 0$. Consider a node $[\![0, \sigma, g]\!]$ which is 'alive'(i.e. included in computing $NTS(\mathcal{T}_g)$). Let $\sigma = j_1, j_2, \ldots, j_r$. We will first show, by contradiction, that for $1 \leq k \leq r-2$, we have $j_k \in F(k) - F(k-1)$.

Assume that $j_k \in F(k-1)$. Then, at round $k-1$ every processor would set the initial value of the node σ to 0 (the *MASK_FAULT* fault action). In round k, from Lemma 5(s3), σ would have at least $n - t - |\sigma|$ children with initial value 0 and would be pruned (the *STOP_EARLY* action) and will not be included in computing $NTS(\mathcal{T}_g)$.

Assume that $j_k \notin F(k)$. Then for at least one processor g, j_k is not known to be faulty. Therefore, for the node $[\![d, \sigma, g]\!]$ we have child$[v] < t + 1 - |\sigma|$ for some $v \in \{0, 1\}$. This implies that for $v' = 1 - v$ child$[v'] \geq n - t + |\sigma|$. Thus this automaton stops early in round k. From Lemma 22, this implies that in round $k+1$ all of its cousins will be pruned and will not be included in computing $NTS(\mathcal{T}_g)$.

Let $a_k = |F(k) - F(k-1)|$. We proved that for $k \leq r-3, a_k > 1$. The maximum number of possible $j_1 \ldots j_{r-3}$ is at most $\prod_{k=1}^{r-3} a_k$. If $|F(k)| = \sum_{k=1}^{r-3} a_k \leq r$, then the number of possible[3] $j_1 \ldots j_{r-3}$ would be at most 8, thus the number of possible $j_1 \ldots j_{r-2}$ would be at most $8t < 2n$ and the number of possible $j_1 \ldots j_r$ would be less than $2n^3$.

Since a processor g creates a debate with an initial value of 0 in round r only when $NTS(\mathcal{T}_g) > 2n^3$, we have $|F(r-3)| \geq r$, hence $|F(r-1)| \geq r$.

Let $d = k$. This debate must have been started by some processor g with the value 0, otherwise the stop early condition would have been satisfied and all these debates would return in one round. This means that $NTS(\mathcal{T}_g[d']) > 2n^3$ for some $d' < k$ and that $|F(k-1)| \geq k$. By a similar reasoning given for $d = 0$, we can show that $|F(r-3) - F(k)| \geq r - k$. Hence the claim. ∎

Lemma 24: Every synchronous execution of the cloture algorithm satisfies the cloture conditions.

[3] It follows from the claim that for natural numbers $a_1 \ldots a_m, s$ such that $\sum_{i=1}^{m} a_i = m + s$, $\prod_{i=1}^{m} a_i \leq 2^s$ holds. *Proof:* Without loss of generality assume that $a_i > 0$. Then $b_i = a_i - 1$ is a natural number and that $1 + b < 2^b$ for every natural number and the claim reformulates to $\prod_{i=1}^{m}(1 + b_i) \leq 2^{\sum_{i=1}^{m} b_i} = 2^s$.

Proof: The first condition (CC1) is satisfied trivially because all *STOP_EARLY* actions in $LEVEL(r, g)$ occur after all of the *RECV* actions. Since we create a debate with an initial value of 0 only if the total number of nodes whose value is to be sent exceeds the threshold $2n^3$, the second condition (CC2) follows from Lemma 23. The final condition (CC3) is obvious from the definition of $LEVEL(r, g)$. ■

Let us define a *Synchronous Cloture system* to be a cloture system in which all the executions are synchronous executions.

Theorem 25: A Synchronous Cloture system C of type $\langle P, T, nodes, roots, peers, \mathcal{Y} \rangle$ solves the Byzantine agreement problem.

Proof: That is, we have to show that the Synchronous Cloture system C solves the schedule module B. From Theorem 20 and Lemma 24, it follows that the C simulates ESFM with respect to U. Since ESFM systems simulate EIG systems (Theorem 14) and since EIG systems solve Byzantine agreement problem (Theorem 9), by transitivity C simulates B.

Since the Byzantine properties (Section 3.1) were stated using only the *CREATE* and *RETURN* actions of the *roots*, we restrict our attention only to these actions. This allows us to to partition these external actions into sets $U_r = \{CREATE(r, v_i, P, t), RETURN(r, v_f)\}$ for each $r \in roots$. From Lemma 3, we know that schedule module B is *closed under reordering* with respect to $U = \cup_{r \in roots} U_r$. Therefore, we can apply Lemma 1 to conclude that the Synchronous Cloture system solves the Byzantine agreement problem. ■

7 Conclusion and Further Work

We have shown that expressing the recursive structure of a distributed algorithm in terms of dynamic process creation can facilitate understanding of the algorithm by making the recursive structure more explicit, permitting local reasoning about system components, and making possible the use of hiding techniques for handling optimization, particularly in hierarchical proofs. In addition, we have shown that the use of an "oracle" for capturing the flow of information among many simultaneous instantiations of an algorithm can be helpful in isolating and reasoning about individual instantiations of the algorithm. Our treatment of the cloture algorithm was handled entirely in an asynchronous setting, given certain "cloture conditions" that we showed are satisfied

by synchronous executions. This helped to identify the role of synchronization in the original cloture algorithm. We feel that the use of recursion in distributed algorithms can be powerful. It will be interesting to see how the techniques presented here can be used in understanding or designing other recursive distributed algorithms, including the new Byzantine agreement protocol of Garay and Moses [17].

Acknowledgments

We would like to thank Brian Coan, Kenneth Cox, Alan Fekete, Rose Gamble, Terry Idol, Paul McCartney, and Catalin Roman for their helpful comments on earlier drafts.

References

[1] Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press series in AI, Cambridge, MA, 1986.

[2] Pierre America. Issues in the design of a parallel object oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.

[3] Pierre America and Frank de Boer. A proof system for process creation. In *Programming Concepts and Methods: Proceedings of the IFIP Working Group 2.2/2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 303–332, April 1990.

[4] A. Bar-Noy and D. Dolev. Families of consensus algorithms. In *Proceedings of the 3rd Aegean Workshop on computing*, pages 380–390, July 1988.

[5] A. Bar-Noy, D. Dolev, C. Dwork, and H. R. Strong. Shifting gears: Changing algorithms on the fly to expedite byzantine agreement. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, pages 42–57, August 1987.

[6] Piotr Berman and Juan A. Garay. Asymptotically optimal distributed consensus. In *Proceedings of the 16th Colloquium on Automata, Languages and Programming*, pages 80–94, July 1989. Lecture Notes in Computer Science **372**, Springer-Verlag.

[7] Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Towards optimal distributed consensus. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 410–415, October 1989.

[8] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[9] K. Mani Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, October 1984.

[10] Brian A. Coan. A communication-efficient canonical form for fault-tolerant distributed protocol. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing*, pages 63–72, August 1986.

[11] Brian A. Coan and Jennifer L. Welch. Modular construction of a byzantine agreement protocol with optimal message bit complexity. *Information and Computation*, 97(1):61–85, March 1992.

[12] Danny Dolev and H R Strong. Polynomial algorithms for multiple processor agreement. In *Proceedings 14th Annual ACM Symposium on Theory of Computing*, pages 401–407, May 1982.

[13] Alan Fekete, Nancy Lynch, and Liuba Shrira. A modular proof of correctness for a network synchronizer. In *The 2nd International Workshop on Distributed Algorithms*, July 1987. Amsterdam, The Netherlands.

[14] Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, 1982.

[15] Michael J. Fischer, Nancy A. Lynch, and M. Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, January 1986.

[16] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[17] Juan A. Garay and Yoram Moses. Fully polynomial byzantine agreement in $t + 1$ rounds. In *In Proceedings of the* 25^{th} *Annual Symposium in Theory of Computing, San Diego, CA, May 16-18, 1993*, May 1993.

[18] Kenneth Goldman and Nancy Lynch. Modelling shared state in a shared action model. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, June 1990.

[19] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.

[20] Bengt Jonsson. A model and proof system for asynchronous networks. In *Proceedings of the 4th ACM SIGACT–SIGOPS Symposium on Principles of Distributed Computing*, August 1985.

[21] Bengt Jonsson. Compositional specification and verification of distributed systems. Technical Report SICS/R-90/90010, Swedish Institute of Computer Science, October 1990.

[22] Bengt Jonsson. Simulations between specification of distributed systems. In *Proceedings of the 2nd International Conference on Concurrency Theory, LNCS 527*, pages 346–360. Springer–Verlag, August 1991.

[23] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[24] John Leo. Dynamic process creation in a static model. M.S. Thesis, MIT Laboratory for Computer Science, May 1990.

[25] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[26] Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions.* In progress.

[27] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th ACM SIGACT–SIGOPS Symposium on Principles of Distributed Computing*, pages 137–151, August 1987. A full version is available as MIT Technical Report MIT/LCS/TR–387.

[28] Nancy A. Lynch and Mark R. Tuttle. An introduction to Input/Output Automata. *CWI-Quarterly*, 2(3), 1989.

[29] George Milne and Robin Milner. Concurrent processes and their syntax. *Journal of ACM*, 26(2):302–321, April 1979.

[30] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[31] Y. Moses and O. Waarts. Coordinated traversal (t+1)-round byzantine agreement in polynomial time. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 246–255, October 1988.

[32] Sergio Rajsbaum and Yossi Malka. Analysis of distributed algorithms based on recurrence relations. preliminary version, October 1991.

[33] S. Toueg, K. J. Perry, and T. K. Srikanth. Fast distributed agreement. *SIAM Journal on Computing*, 16:445–458, June 1987.

[34] Jennifer Welch, Leslie Lamport, and Nancy Lynch. A lattice-structured proof of a minimum spanning tree algorithm. In *Proceedings of the 7th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 28–43, August 1988.

[35] Jennifer L. Welch and Nancy A. Lynch. Synthesis of efficient drinking philosophers algorithms. Technical Report MIT/LCS/TM-417, MIT Laboratory for Computer Science, November 1989. Revised version submitted for publication as "A Modular Drinking Philosophers Algorithm".