The Sage Project: Software Engineering for Distributed Applications

Aleta Ricciardi and Chun Chan

TR TR-PDS-1996-007 April 1996

Parallel & Distributed Systems group

Department of Electrical & Computer Engineering

University of Texas at Austin

Austin, Texas 78712

The Sage Project: Software Engineering for Distributed Applications

Aleta Ricciardi* and Chun Chan†

April 16, 1996

Abstract

This paper describes *Sage*, a research project in software engineering for (fault-tolerant) distributed applications that assists programmers write correct, efficient code. Specifically, Sage uses the modal logic of knowledge and applies theoretical results detailing how processes learn facts about each other's state to derive the minimal communication graph for a given distributed coordination problem. An environment that applies intricate theoretical results, but hides the theory from users is particularly important as few distributed applications programmers hold advanced degrees in computer science.

The specification interface is controlled, yet expressive enough to capture canonical distributed coordination problems and weaker variants appropriate for wide-area and object-oriented systems. The resulting graphical display shows programmers which messages must be received. Sage allows users to experiment on the derived protocol by crashing processes, reordering events, losing messages, and partitioning the network. Under these conditions, Sage regenerates the communication graph, or indicates a solution is now impossible. This has two benefits: it animates the effects of unpredictable system events on distributed applications, and it greatly simplifies testing distributed algorithms. Sage completely separates the issues in testing a protocol's behavior in the face of failures, from the effects background system conditions (*e.g.*, processor or network loads) can have on the testing procedure itself.

As development continues, other components will highlight required group and communication primitives, generate skeleton code, and verify user-drawn protocols. Thus, Sage will be an integrated development, testing and verification environment for distributed applications.

Keywords: software fault tolerance; highly-available software systems; distributed systems; specification and design of reliable software-based systems.

*Supported by the École Polytechnique Fédérale de Lausanne and a University of Texas Research Institute Summer Research Fellowship.

†Partially supported by NSF grant ASC-9318151.

1 Introduction

The last decade has seen an enormous change in industry, engineering and scientific computing from monolithic systems to desktop workstations. In theory, the ability of the sum of these distributed desktop systems surpasses that of its centralized predecessor. In practice this is often not the case, as the act of distributing processors calls for new programming strategies and techniques; simply distributing computers does not give distributed computing. Correct distributed applications must, in addition to sequential correctness criteria, be able to account for the states of processes participating in the computation. In asynchronous systems, accurately reasoning about the current state of remote processes is impossible, meaning that programs attempting seemingly rudimentary coordination can either be safe or live, but not both [8]. Certainly, some very useful coordination problems do not require accurate information of *all* remote processes' states [9]. For those that do, it is possible to implement a *virtually synchronous* system [3], allowing one to weaken a coordination problem so that it has solutions in an asynchronous system, but ensuring the executions of the weaker problem are indistinguishable to external observers from executions of the original problem.

The Isis Toolkit [4] (and its intellectual descendants Transis [1], Phoenix [17], and Horus [28], and the Consul system [18]) provides basic programming primitives for virtually synchronous distributed computing; these include operations on groups of processes, different multicasts within and between groups of processes, as well as more sophisticated subsystems. However experience is showing that users often do not understand the subtle differences between the various primitives. To ensure applications are correct, programmers will use the computationally costly 'atomic' mechanisms that are easier to understand when lighter-weight mechanisms would have sufficed. If, on the other hand, programmers use primitives that are not strong enough, the program will not be correct. These difficulties are exacerbated in the trend to deconstruct strong primitives to weaker microprotocols [13, 28]. While it appears that programmers need a solid foundation in the underlying theory to use these toolkits effectively, they rarely have this background. With networked desktop machines proliferating, more programmers than ever are, and will be, writing distributed applications.

This paper describes *Sage*, a software development environment that assists distributed applications programmers write correct, efficient code. Sage applies the results of a wide body of theoretical research in (asynchronous) distributed computing to the very real problems faced by programmers writing distributed applications. The underlying theory itself uses the modal logics of knowledge and time, and results that detail how remote processes 'learn' facts about each other's state to derive the minimal (*i.e.*, essential) communication graph for the problem described. Sage implements the underlying theory, provides a user-interface that shields programmers from the theory, and uses the theory to derive protocols. Users describe a distributed coordination problem in simple language through a series of pull-down menus. Thus, applications programmers can thus use the theory simply and naturally, without having to learn and be proficient in it.

To further assist users understand asynchrony and fault tolerance issues and solutions, Sage allows users to experiment with the simulated solution by changing the order of events in the derived graph, crashing and recovering processes in the 'middle' of a computation, dropping messages, and partitioning and repairing the network. While the initial protocol is shown assuming a perfect (*i.e.*, failure-free) run, users can nonetheless examine it under the most common failure scenarios. Significantly, the experimental facility decouples the difficulty in debugging distributed applications from the effects non-deterministic, ambient system conditions have on the debugging process itself. As development continues, Sage will highlight the (necessary) presence of different communication and group-based primitives, helping programmers use available toolkits effectively. It will also allow users to describe graphically and verify their own protocols.

It is likely CORBA will remove much of the difficulty in writing distributed applications. However, there are few ORBs available, fewer available inexpensively, and only two simple, experimental ORBs for fault-tolerant distributed object computing. There, our experiences with Electra [16] (and investigations into Orbix+Isis) show the efficiency of the reliable ORB and using it efficiently, depend on thoroughly understanding issues in fault-tolerant computing. For example, at present, Electra's replicated objects are implemented as Isis peer groups (the simplest, but most expensive option) and communication between replicas is only by atomic broadcast (also the simplest, but most costly communication primitive). Irrespective of the programming model, processes or distributed objects, Sage will be useful to applications programmers, as well as ORB developers.

Section 2 describes the theoretical background on which Sage is based. In Section 3 we describe each component of the full system, while Section 4 discusses the current status of the project.

2 Research and Theoretical Background

Research in distributed computing has increased our understanding of fundamental issues in distributed coordination. Distributed coordination can be broadly stated as follows:

> If a member of some set of processes performs some action of interest, then all the other processes in that set are obliged to eventually perform the same action, or are eventually determined to be exempt.

Variants of the problem are obtained by allowing the set of obliged processes to change composition, the urgency with which an action must be taken by all, whether any actions conflict and must be ordered, and the conditions under which exemptions are made. Distributed coordination characterizes a wide range of canonical problems, including agreement, mutual exclusion, reliable communication, transaction commit, and clock synchronization. There are quantifiable message-complexity results [15], as well as qualitative expressions of impossibility [8, 12], possibility [5, 6], and problems' comparative difficulty [5, 10, 24]. Using modal logics of knowledge and time to describe and reason about distributed coordination is well accepted, though unfortunately complex [11, 12, 19, 22, 23, 26]. Rarely do these sorts of results find their way to applications programmers; rarer still is the programmer who can spare the time to learn the theories.

2.1 The Computing Model (Briefly)

For our work, a distributed computing system consists of processes executing on processors, and an abstract network of communication channels connecting processes. Processes may crash and recover, but otherwise follow their assigned protocol. The communication channels ensure only that each message has some probability of arriving at its destination. In a *synchronous* distributed system network delays and differences in processors' speeds are known and bounded; in an *asynchronous* distributed system these delays and variations are not known. Even if the low-level components can be considered synchronous, software layers that implement common abstractions (*e.g.*, reliable channels, processes, objects) and the inability to predict network and processor loads make the high-level system, where distributed applications are developed and execute, asynchronous. Asynchronous behavior is the norm in wide-area distributed systems where large geographic distances and frequent network partitions exacerbate communication delays. Therefore, Sage assumes an asynchronous computing model and is based on the research done in asynchronous distributed computing.

The execution of a process can be represented as a linear sequence of events. Events can be local (such as incrementing a private variable) or can have remote effects (such as sending a message). The execution of the distributed system of processes is the tuple of all processes' executions. While all the events in a process's history can be linearly ordered, this is not so for all events in a system execution. A *global state* of a distributed system is a tuple of prefixes of process's executions; it is *consistent* if, for each receive event it contains, it also contains the corresponding send event [14].

2.2 Modal Logics of Time and Knowledge (Briefly)

We present only an informal discussion of the semantics of the formal language; full treatments can be found in Chellas [7], van Benthem [27], and the works already cited. The formulas of the logic are evaluated at consistent global states, in the context of an entire execution of the system. Given a propositional formula ϕ, the semantics of the modal formulas are:

Time $\Box\phi$ (always) is true if and only if ϕ holds at that global state, and every global state in the future of that execution.

$\Diamond\phi$ (eventually) is true if and only if ϕ holds at some global state in the future of that execution.

$\diamondminus\phi$ (in the past) is true if and only if ϕ held at some global state in the past of that execution.

$\boxminus\phi$ (always in the past) is true if and only if ϕ holds at that state and held at all global states in the past of that execution.

Knowledge $K_p\phi$ (p knows) is true if and only if ϕ holds at that global state, and at all global states in which p has the same local state.

An algorithm is a solution to a problem if and only if the formulas specifying the problem are *valid* over the set of all runs generated by the algorithm.

2.3 Using Knowledge Logics

Sage implements the theory of Chandy and Misra [6], in which it is explicitly stated how to understand necessary communication between processes. Informally, a formula is said to be *local* to a process exactly when that process, and only that process, always knows (in the modal logic sense just described) whether the formula is true. One of the Chandy and Misra results states that if ϕ is local to p and $K_q\phi$ holds, then q has received a message that originated at p, and that asserts (the continued truth of) ϕ.

In distributed coordination, it is safe for a process to take an action of interest only when certain conditions about the global state hold. Because a global state is a tuple of process execution histories, if any of these conditions involve formulas that are local to processes other than the process taking the action, messages must have been exchanged.

2.3.1 An Example

To make this concrete, we consider the example of Uniform Coordination [9] (UC). Let S be the set of processes participating in the distributed coordination of actions $\alpha \in A$. The Agreement and Termination conditions state that if p is a participant and takes action α, then eventually every other participant either takes α as well, or is declared exempt from doing so. The formula expressing this is,[1]

$$p \in \mathsf{S} \ \wedge \ \textsc{did}_p(\alpha) \Rightarrow \forall q \in \mathsf{S} : \Diamond\textsc{did}_q(\alpha) \ \vee \ \Diamond\textsc{exempt}_q(\alpha) \ . \tag{1}$$

Because Formula 1 must be valid in any solution, the modal logic Generalization Inference Rule applies; specifically, we can introduce the knowledge operator to describe what must be the state of knowledge *in any solution* to UC when process p executes action α. After introducing the K_p knowledge operator, logically manipulating Formula 1, and applying research results described in [20] (the K_p operator does not tautologically distribute over disjunction) we obtain

$$p \in \mathsf{S} \ \wedge \ \textsc{did}_p(\alpha) \Rightarrow \forall q \in \mathsf{S} : K_p\Diamond\textsc{did}_q(\alpha) \ \vee \ K_p\Diamond\textsc{exempt}_q(\alpha) \ . \tag{2}$$

Formula 1 means that p cannot know only that one of the two disjuncts will hold; it must be able to account definitively for each process regarding which one will hold. That is, in asynchronous systems actions are not taken without explicit knowledge of every other participating process.

It turns out that $\Diamond\textsc{did}_q(\alpha)$ is local to q, and that the interpretation of $\Diamond\textsc{exempt}_q(\alpha)$ needed to implement virtually synchronous behavior involves sub-formulas that are themselves local other processes, which must form a majority subset of S.[2] In short, before it is safe for p to execute α, it must, for each other q, either have received a message informing it of q's intention to eventually execute α, or have received messages from a majority of S informing p of each of their acceptance of q's exemption.

---

[1] We trust that the basic propositions are self-explanatory.

[2] See [20] for the full technical derivation.

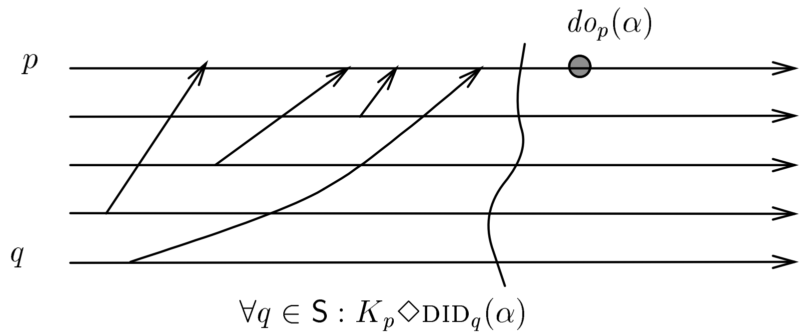

Figure 1: Messages required before $do_p(\alpha)$ is safe. The messages to p are promises from each q to eventually execute $do_q(\alpha)$.

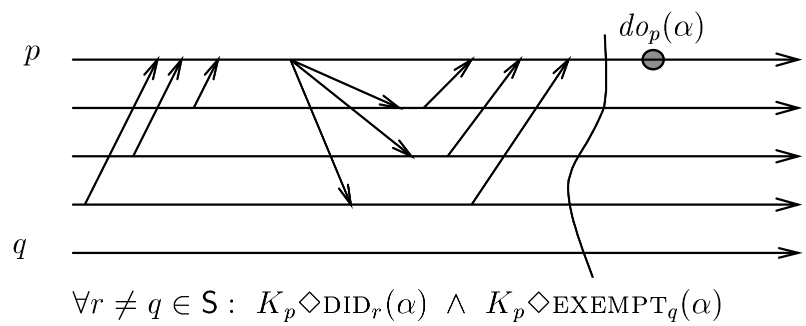

Figure 2: Messages required when q is unresponsive. The second set of messages to p indicate acceptance of p's proposal to exempt q.

Figure 1 illustrates these messages for the simple case when all processes have responded to p; they are indicating their willingness to eventually execute α.[3] In Figure 2 process q is unresponsive; before it can execute α, p must first correspond with at least a majority of S ascertaining, in addition to their willingness to execute α, their acceptance of q's exemption.

[3] In our figures, process histories proceed from left to right and messages between processes are indicated by arrows from one process to another. The vertical slice across all process histories is a global state.

3 Sage Description and Goals

Sage, when completed, will consist of:

- A *system definition tool* through which users describe the physical characteristics of a distributed system (Section 3.1).
- A *specification tool* through which users specify distributed coordination problems. Sage generates a knowledge specification from this (Section 3.2).
- A *communication graph extractor* which extracts from the generated knowledge specification the messages required to solve the problem (Section 3.3).
- A *laboratory* in which users experiment on the solution (*e.g.*, changing the order of events, crashing and recovering processes, dropping messages, and partition the network). Sage generates a new solution (Section 3.4).
- A *verifier*, in which users graphically describe a protocol. The verifier determines whether and how efficiently the input protocol solves the specified problem (Section 3.5).
- An *identification tool* to recognize various communication primitives within the extracted communication graph.
- A *code generator*, which would work with the identification and extractor components to produce skeleton code.

The first four of these are implemented in our prototype, and the research for the identification tool and verifier is underway. The code generator requires closer study and hands-on use of the actual distributed programming toolkits.

3.1 System Definition

The characteristics of the physical system include the number of processes participating in the computation, the types of failures they can experience, the topological connectivity of the network, and any communication guarantees the user would like to assume are provided. In the example shown in Figure 1, five processes are participating and the network is assumed completely connected (the default); had the network been specified as a ring topology only one of the other processes would have been able to send directly to p.

The type of failures processes can experience determines the complexity of the resulting communication graph; more information is required to detect (and mask) malicious failures than benign. In addition, replication requirements are different depending on failure types.

Three formulas express the fundamental rules of communication: receiving a message implies it was sent, $\textsc{recv}_p(q, m) \Rightarrow \textsc{send}_q(p, m)$, and receiving or sending a message implies knowledge of its contents, $essqpm \rightarrow K_q(*m)$, and $\textsc{recv}_p(q, m) \Rightarrow K_p(*m)$.[4]

3.2 Problem Specification

The success of Sage depends on the fact that while its underlying formalism is the esoteric modal logic of knowledge, users are not specifying problems in terms of process knowledge, or even in terms of formal logic. Sage uses the logic of knowledge to distill the communication essentials from the problem specification and failure model, but users simply instantiate variables in the problem specification and system model. Our research has determined (and will continue to determine) the set of rules the extractor should apply. Section 4 describes how we have encoded these rules in the prototype.

Sage's default assumption is that some form of coordination is being attempted, but it does not impose assumptions about ordering the coordinated actions, terminating them, or even exempting processes from coordinating on specific actions. In the prototype, users can choose from exempting processes that have crashed, or that are unresponsive and so appear to be crashed.

Sage's expressiveness depends on a full understanding of the many variants of distributed coordination. We now discuss the expressive capabilities of the specification tool by describing each of the aspects of distributed coordination.

3.2.1 Uniform Coordination

Uniform Coordination is the simplest statement of distributed coordination, and is the foundation of more complex coordination problems.[5] In dissecting UC [20] we derived necessary and sufficient

[4] Here, $*m$ expresses m's contents as a logical formula.

[5] We choose UC, rather than non-Uniform, as it is better suited to systems that experience only benign failures; in these systems, UC more accurately describes behaviors desired of fault-tolerant services.

communication for any solution to this problem in asynchronous systems that can experience process crashes; Sage embodies these results in the form of rules for constructing a message-minimal solution. To implement virtually synchronous behavior, we showed it is necessary for processes to conspire against an exempt process, and we showed this conspiracy requires processes to stop communicating with the exempt process. In Figure 2, the second set of messages sent to p (indicating processes' acceptance of q being exempt from taking α) are pledges to eventually cease communicating with q, while the messages p sent to the others were requests for them to do so.

From our research, only two formulas are needed to express the communication structure for UC, a further derivation from Formula 2, and a formula describing how to effect this exemption (see also Sections 2.3.1 and 3.2.2).

3.2.2 Exemptions

Sage's specification tool embodies the generic notion of *exempting* a processes from coordinating; as such, Sage can express problems that have solutions in asynchronous systems that can experience crash failures. Certainly crashed processes must be exempt from coordinating, but permitting only this strong exemption makes problems impossible [8]. Our research has shown that virtually synchronous behavior is directly attributable to a strong exemption that makes an exempt process behaviorally indistinguishable from a crashed process; we call this exemption DIS'D() (the exempt process is disabled from taking further actions). We then showed that any problem with the DIS'D() exemption is equally hard irrespective of properties of the communication channels, narrowing the solution space for Sage.

Defining weaker exemptions for distributed coordination problems gives rise to problems that have solutions in asynchronous systems, as well as in systems prone to partitioning (*e.g.*, wide-area and cellular networks). These coincide with weak virtual synchrony models [25] and we are following these ideas to expand Sage's expressiveness.

3.2.3 Sequential Coordination

We broaden UC by defining a generic *sequence clause* [21]. A sequence clause defines conditions under which two actions must be taken by all obliged processes in the same relative order. This decomposition isolates communication costs due solely to ordering, giving a clear way of comparing coordination problems, and a simple accounting of the extra costs incurred by sequencing. While our prototype only implements unsequenced UC, we have completed much of the research to be able to concisely express sequence clauses, and to distill the necessary communication. Canonical problems in sequential coordination include linear and causal multicast, leader election, mutual exclusion, serializability, and group membership changes.

Formally, a *Sequence Clause* requires that whenever two actions are related by some condition, all jointly-obliged processes take both actions in the same order,

$$\Sigma(\alpha,\beta) \Rightarrow \forall p \in \left(\mathsf{S}_\alpha \cap \mathsf{S}_\beta\right) : \textsc{did}_p(\beta) \Rightarrow \textsc{did}_p(\alpha) \ . \tag{3}$$

Partial Order To define a partial order that respects causality, we instantiate $\Sigma(\alpha,\beta)$ to reflect the causal relationship between the originating events for α and β. Let $Annc_{p_\alpha}(\mathsf{S}_\alpha,\alpha)$ denote the event by which p_α, the process initiating α, announces α to S_α, the processes obliged to participate in the coordinated action. $\textsc{before}(e,e')$ is true if e causally precedes e'. expected meaning for events e and e'.

$$\Sigma(\alpha,\beta) \stackrel{\text{def}}{=} \textsc{before}(Annc_{p_\alpha}(\mathsf{S}_\alpha,\alpha), Annc_{p_\beta}(\mathsf{S}_\beta,\beta)) \ .$$

Interpreting $\textsc{did}_p(\alpha)$ as p's delivery of message α and $Annc_{p_\alpha}(\mathsf{S}_\alpha,\alpha)$ as the multicast of message α to process group S_α, specifies causal delivery order of multicast messages.

Total Order A total order may have $\Sigma(\alpha,\beta)$ reflect some pre-ordained rule if α and β originate concurrently and adhere to causality otherwise. $\textsc{concurrent}(e,e')$ has the expected meaning for events e and e', and $p_\alpha < p_\beta$ reflects any commonly known way of using processes' identifiers to break ties for concurrent events.

$$\begin{aligned}\Sigma(\alpha,\beta) \stackrel{\text{def}}{=} \ & \textsc{before}(Annc_{p_\alpha}(\mathsf{S}_\alpha,\alpha), Annc_{p_\beta}(\mathsf{S}_\beta,\beta)) \vee \\ & \Big((p_\alpha < p_\beta) \ \wedge \ \textsc{concurrent}(Annc_{p_\alpha}(\mathsf{S}_\alpha,\alpha), Annc_{p_\beta}(\mathsf{S}_\beta,\beta))\Big) \ .\end{aligned}$$

This definition is frequently used in linearizing the partial order on events, implementing causal-atomic multicast, checkpointing, and in serializability.

Virtual Synchrony A sequence clause requiring any change to an action's obligation set to occur after the action itself can define weak interpretations of virtual synchrony. Stronger variants are obtained provided these changes are themselves coordinated (Section 3.2.5).

$$\Sigma(\alpha,\beta) \stackrel{\text{def}}{=} \beta \in \Big\{\mathsf{S}_\alpha \leftarrow \mathsf{S}_\alpha \cup \{r\} \ , \ \mathsf{S}_\alpha \leftarrow \mathsf{S}_\alpha - \{q\}\Big\} \ . \tag{4}$$

3.2.4 Termination Guarantees

We have also defined a generic *termination clause* which, when instantiated with specific formulas, defines conditions under which every obliged process must either have taken the action or have been deemed exempt.[6]

$$\tau(\alpha) \;\Rightarrow\; \forall p \in \mathsf{S}_\alpha : \textsc{did}_p(\alpha) \vee \textsc{exempt}_p(\alpha) \;. \tag{5}$$

Common examples occur in buffer flushing for implementing reliable messaging (*e.g.*, TCP) and in garbage collection: a process should remove actions from local storage when it knows it no longer needs to store them for future reference. Termination requirements also exist whenever a group of processes coordinate to implement a service whose behavior must be that of a single, fault-tolerant process; that is, the previous coordinated action must be terminated before before any replica can take a next coordinated action. This is most common in the family of resource allocation problems (*e.g.*, leader election, mutual exclusion, and token holder). As can be seen in Formula 4, virtual synchrony also defines a class a completion clauses. As with sequence clauses, our prototype does not implement termination clauses, though we have completed much of the background research.

Simultaneity Simultaneous termination requires an action to be taken at the same global instant by all processes, $\tau(\alpha) \stackrel{\text{def}}{=} \exists p \in \mathsf{S}_\alpha : \textsc{did}_p(\alpha)$.

Lock-Step Lock-step termination precludes runs in which some errant processes may, for a finite period, ignore the behavior of the group; it requires any action any process has taken to be terminated before any other process is able to take a subsequent action. For simplicity, assume $\mathsf{S}_\alpha = \mathsf{S}_\beta = \mathsf{S}$,[7]

$$\tau(\alpha) \stackrel{\text{def}}{=} \exists p \in \mathsf{S} : \textsc{did}_p(\alpha) \;\wedge\; \exists \beta : \Big(\big(\forall q \in \mathsf{S} : \neg\textsc{did}_q(\beta) \big) \wedge \big(\exists q \in \mathsf{S} : K_q \textsc{safe}_q(\beta) \big) \Big) \;.$$

3.2.5 Termination and Virtual Synchrony

In practice, virtual synchrony is obtained when a change to an action's obligation set is the terminating condition for the action. 'Flush' protocols effect this, and ensure that no partially executed

[6]Termination is a kind of timeliness assertion for asynchronous systems: if some condition holds, then α is terminated. In synchronous systems, this condition involves the global clock time. It is important that $\tau(\alpha)$ be locally determinable.

[7]A process must know it is safe to take an action before it can take that action.

distributed actions exist across changes to an obligation set. Virtual synchrony can be generalized as termination with respect to a class of other distributed events. Let $\mathcal{E}$ be a class of events, and as before let $do_p(e)$ denote process p executing event $e \in \mathcal{E}$. Then termination of α with respect to $\mathcal{E}$ is,

$$\tau(\alpha) \stackrel{\text{def}}{=} \exists e \in \mathcal{E} : \exists p \in \mathsf{S}_\alpha : \text{BEFORE}(do_p(\alpha), do_p(e)) \ .$$

Further requiring events in $\mathcal{E}$ to adhere to other properties gives stronger virtual synchrony variants. For example, if $\mathcal{E}$ is membership changes which are themselves delivered in the same sequence at all affected sites, the type of virtual synchrony obtained results in all obliged processes taking an action during the same instance of the obligation set's evolving composition. This is the virtual synchrony provided by Isis. Extended virtual synchrony [2] is obtained by sequencing membership changes, and by terminating α before any obliged process changes S_α,

$$\mathcal{E} \ = \ \left\{ \mathsf{S}_\alpha \leftarrow \mathsf{S}_\alpha \cup \{r\} \ , \ \mathsf{S}_\alpha \leftarrow \mathsf{S}_\alpha - \{q\} \right\} \ .$$

Ensuring the relevant properties for changing the obligation set is another instance of distributed coordination (implemented by a *Membership Service*). This separation simplifies specifying dynamic coordination and building composable systems (see [13] for another approach to specifying membership services).

3.2.6 Remarks

We have shown that Sage's formal model is capable expressing many canonical coordination problems (*e.g.*, leader election, atomic multicast, causal multicast, resource allocation, and virtual synchrony variants). Of course, these complex logical formulas are not presented to the user; instead, our research clarifies what the variables are in instantiating sequence and termination clauses to obtain useful problems. We present these choices to the user.

While powerful in LANs, existing toolkits for distributed applications programming make assumptions about connectivity and network reliability that do not apply in WANs. Sage's contributions in this area are twofold: Sage will assist WAN applications programmers who are once again on their own implementing distributed coordination, and Sage will assist designers of toolkits for WAN-based distributed programming specify, understand, and implement communication and virtual synchrony primitives that are weaker and more dynamic than those that can be used in LANs.

3.3 Extracting the Communication Graph

To run the extractor, a user indicates an action and selects two processes (possibly the same one) by clicking on their histories. One process, the start-point, is the initiator of the action (the first among the participants to be aware of it), the other, the end-point, has actually taken the action. The Extractor then builds a solution for the specified problem (or indicates none is possible) assuming a failure-free run. Indeed, Sage must assume a failure-free run; we cannot put the onus on the user to predict which will be the critical junctures in a solution as this would imply the user knew, or could anticipate, the structure of a solution. However, given a solution in a perfect run, users can then examine how process and communication failures affect the communication. Section 4 describes how the extractor applies its rules to build the communication graph.

3.4 The Laboratory

This is the most interesting part of Sage; it gives users the ability to experiment methodically on a protocol in a simple, graphical manner. At present, creating the appropriate testing scenarios for distributed applications is extremely difficult. Failures must be programmed or caused to occur at precisely the right moment; completely understanding their effects requires that this be done in a controlled, repeatable fashion. Unfortunately, non-determinism, and ambient system and network conditions inevitably mean the experimental state cannot be exactly duplicated. For debugging purposes, one must rely on log files or strategies to predict entire runs; these can neither duplicate conditions nor express all conditions. Sage's laboratory is essential for understanding how process and communication failures affect protocols.

Users can 'crash' processes at any juncture in the generated protocol, and drop or delay any message. The reconfigured solution animates the protocol steps needed to adapt to the change or failure (or indicates no progress is possible). Ultimately, the effects of such experiments are linked to which exemption the user selects. For example, if a user 'crashed' process q before it responded to the end-point the altered protocol indicates that the coordinated action has been taken but without a response from q. To satisfy Formula 2, the end-point must know $\Diamond\text{EXEMPT}_q(\alpha)$, and so the reconfigured solution will show the messages necessary for the end-point to attain this knowledge. Figures 3 and 4 show this sequence, as run in our Sage prototype.

In the future, users will be able to re-order displayed events.[8] To complete the laboratory, we

[8]Equivalently, the extractor could graphically indicate waves of events that need not be ordered. Figure 3 already

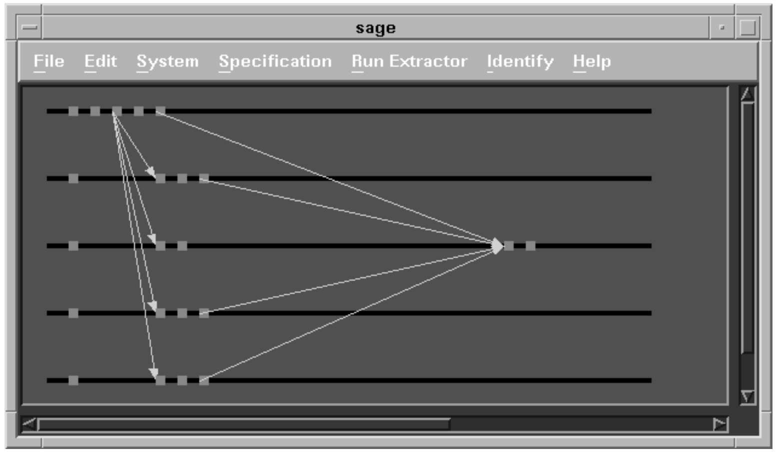

Figure 3: Sage's generated solution for a failure-free run of Uniform Coordination.

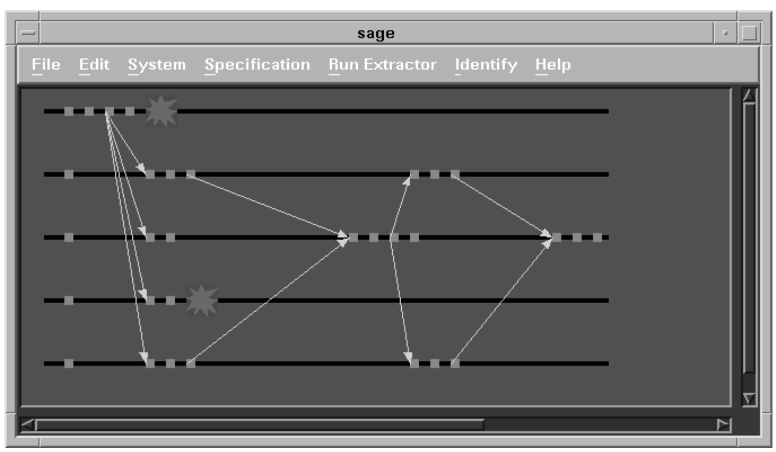

Figure 4: Sage's regenerated solution after a user crashes two processes.

plan to add full dynamic process group capability, allowing processes to join (or rejoin) an ongoing coordinated activity. Our current research is contributing to this component.

3.5 Verifying Protocols

At present, Sage is a one-way tool: users specify a problem and Sage derives a solution. The verifier will allow users to graphically describe a protocol, by drawing the relevant processes, actions, messages, and their contents. Sage's rules will be used to determine the knowledge accumulated by each process with each message received (rather than reason backward about what must have been the global state). If the protocol has not accumulated sufficient knowledge when the action is taken, it is incorrect; we are not concerned (yet) with liveness. As well, we can easily discover and highlight inefficiencies (unnecessary messages) in the user's protocol.

4 Prototype Implementation

Sage is written in C++ and uses the Xwindow/Motif toolkit. We chose C++, rather than a pure rule-based language, for modularity, maintenance, and pragmatic concerns (*e.g.*, programmer support, experience, portability). As well, Sage's few rules are sufficiently small in number compared to typical rule-based applications. We chose the Xwindow toolkit solely for pragmatic reasons; we are considering other environments for future versions.

Sage has three main components. The *graphical user interface* (GUI) provides simple, controlled access to the Sage environment. Users describe the physical system (see Figure 5), specify a distributed coordination problem, run the extractor, and conduct experiments on protocols. The *configuration* completely describes the user-specified system conditions and problem to be solved (from the GUI). These instance-specific rules and conditions determine the solution domain. One library contains the communication theorems for solving Uniform Coordination in the presence of process crash failures, lost messages, and the virtual synchrony exemption $\textsc{dis'd}(q)$; another contains the fundamental communication theorems (for example, a process knows the information content of messages it sends and receives). Finally, the *extractor* is Sage's computational core.

---

shows that correctness is insensitive to the order in which end-point receives messages.

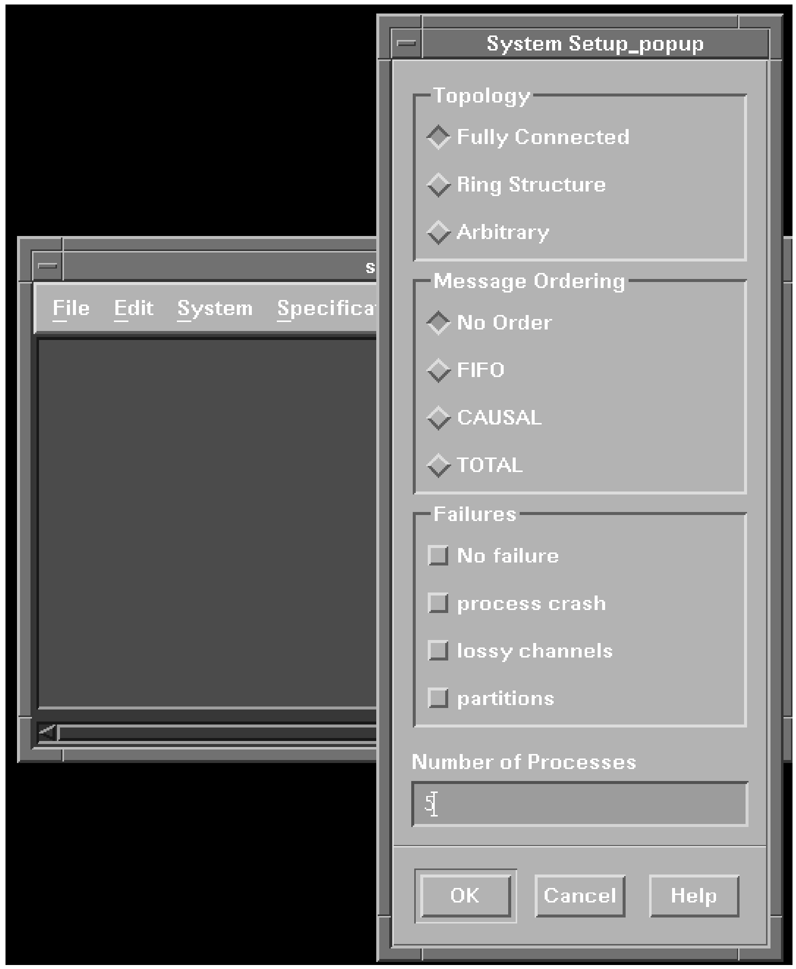

Figure 5: User interface for physical system configuration.

4.1 Representing Knowledge in the Configuration

Although there are many ways to express the same formula, Sage uses implication. This is ultimately due to the style of reasoning used to derive protocols: given that the end-point has taken an action, what must be true immediately before the action is taken, $\textsc{did}_p(\alpha) \Rightarrow \Phi$? Answer: $\Phi = \forall q \in \mathsf{S} : \textsc{recv}_p(q, m(\alpha))$. What, in turn, were the conditions necessary to cause this, $\forall q \in \mathsf{S} : \textsc{recv}_p(q, m(\alpha)) \Rightarrow \Psi$? Answer: $\Psi = \forall q \in \mathsf{S} : \textsc{send}_q(p, m(\alpha))$. This questioning continues until we reach the originator of α or the beginning of the system run.[9] Three classes represent the different symbolic categories of the formal logic.

System System objects describe non-logical entities; they are the system- and problem-specific building blocks of formulas. For instance, to encode the formula $\textsc{send}_p(q, m)$, we need processes and messages; to encode $\textsc{did}_p(\alpha)$, we also need actions. In Sage, these concepts are represented by the elementary classes `Process`, `Message`, and `Action`.

Primitive Primitive objects are the base propositions. *True* and *False* are constants, *Initiator*() formalizes p_α, the start-point for α, and *About*() asserts the existence of an action.

$$\begin{aligned} \texttt{Primitive} ::= \;& True \mid False \mid \mathit{Initiator}(\texttt{Process}, \texttt{Action}) \mid \mathit{About}(\texttt{Action}) \mid \\ & Send(\texttt{Process}, \texttt{Process}, \texttt{Message}) \mid \mathit{Recv}(\texttt{Process}, \texttt{Process}, \texttt{Message}) \mid \\ & Did(\texttt{Process}, \texttt{Action}) \mid Crash(\texttt{Process}) \mid Dis'd(\texttt{Process}) \;. \end{aligned}$$

Formula Formula objects are primitives or the propositional and modal operators. As the rules Sage applies deal only with the present tense, temporal operators are not (yet) necessary.

$$\begin{aligned} \texttt{Formula} ::= \;& \texttt{Primitive} \mid \mathit{And}(\texttt{Formula}, \texttt{Formula}) \mid Or(\texttt{Formula}, \texttt{Formula}) \mid \\ & Not(\texttt{Formula}) \mid \mathit{Implies}(\texttt{Formula}, \texttt{Formula}) \mid Know(\texttt{Process}, \texttt{Formula}) \end{aligned}$$

4.2 Using Knowledge in the Extractor

Sage's extractor is rule-based. Given a query, the extractor considers each rule in the configuration until a rule's antecedent matches the query; it returns the consequent, which is then fed back as a new query (assuming the antecedent holds, the consequent must hold to validate the rule). Rules

[9] Continuing, $\textsc{send}_q(p, m(\alpha)) \Rightarrow K_q`\alpha' \Rightarrow \textsc{recv}_q(p_\alpha, m(\alpha)) \Rightarrow \textsc{send}_{p_\alpha}(q, m(\alpha)) \Rightarrow K_{p_\alpha}`\alpha'$, where '$\alpha$' is a formula representing information about α and $m(\alpha)$ is any message mentioning α.

with the same antecedent are ordered in the configuration list based on our understanding of their utility, though at present, this does not occur in Sage.

In Figure 3, selecting an action, start-point and end-point instantiated the action, its initiator, and generated the queries (as in footnote 9): *Did*(end-point, action), then *Recv*(end-point, AllProc, action),[10] then *Send*(AllProc, end-point, action), then *Know*(AllProc, action), then *Recv*(AllProc, start-point, action), then *Send*(start-point, AllProc, action), then *Know*(start-point, action), and finally *True*.

The extractor uses the method functions of primitive and formula objects objects (we describe these only briefly). These method functions define the semantics of the syntactic objects of our grammar, and provide the standard interface to the extractor to operate on all `Formula` objects.

`bind()` binds the identifiers of a generic formula to the identifiers of a predicate. For example, if the generic formula is $Implies(Recv(q,p,m),\ Send(p,q,m))$, and predicate is $Recv(p_1,p_2,m_0)$, `bind()` links q to p_1, p to p_2 and m to m_0. Thereafter, any occurrence of q in resolved formulas refers to p_1, and so on. The bindings are stored in a list.

`assign()` complements `bind()`. It creates and returns an instance of an object (the type of the instance depends on which object invokes the function). It checks the binding list to assign values to the identifiers. If the $Send(p,q,m)$ object invokes `assign()` on $Recv(p_1,p_2,m_0)$, then $Send(p_2,p_1,m_0)$ is created and returned.

`match()` checks whether a `Formula` matches any of the configuration `Formulas`. $\phi(p_1,...p_n)$ matches configuration `Formula` $\psi(f_1,...f_n)$, if ϕ and ψ are identical and either there are no bindings for any of $f_1,...f_n$ (in which case they are bound) or $(f_1,p_1),...(f_n,p_n)$ are all in the binding list. Otherwise `match()` detects a mismatch.

`resolve()` embodies the semantics of the formula; given an input query, `resolve()` returns the `Formula` that must be true for the query to be true. Thus, query ϕ resolved with `Formula` $And(\phi,\psi)$, returns ψ; in our examples, query $\phi =$*Did*(end-point, action) matches and is then resolved with *Implies*(ϕ,*Recv*(end-point, AllProc, action), returning *Recv*(end-point, AllProc, action), and so on.

[10]Here, AllProc is shorthand for a conjunction of *Recv*(end-point, *, action) objects over all processes.

5 Conclusions

Sage will evolve to incorporate rules for sequential and terminating UC variants, weaker exemptions, and dynamic coordination. Because of the small number of rules needed to describe complex problems and our ability to prioritize rules with the same antecedents, we are not hindered by the back-tracking and complexity issues usually seen in resolution systems. New system, primitive, and formula (*e.g.*, temporal operators) objects can be added as needed, and their semantics defined using the functions already available. In addition to expanding the realm of problems Sage can solve, our other main thrust is in verifying user-input protocols, making Sage an integrated development, testing, and verification tool for distributed programming.

References

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *22nd FTCS*, pages 76–84. IEEE, 1992.

[2] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership Algorithms for Multicast Communication Groups. In *13th IEEE ICDCS*, pages 551–560, 1993.

[3] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *11th Symposium on Operating System Principles*, pages 123–138, November 1987.

[4] K. P. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.

[5] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Asynchronous Systems. In *Tenth PODC*, pages 325–340. ACM, 1991. (also Cornell University TR93-1374).

[6] K. M. Chandy and J. Misra. How Processes Learn. *Distributed Computing*, 1(1):40–52, 1986.

[7] B. F. Chellas. *Modal Logic: an Introduction.* Cambridge University Press, 1980.

[8] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, April 1985.

[9] A. Gopal and S. Toueg. Reliable Broadcast in Synchronous and Asynchronous Environments. In *3rd WDAG. Springer Verlag (LNCS 392)*, pages 110–123, 1989.

[10] R. Guerraoui. Revisiting the Relationship Between Non-Blocking Atomic Commitment and Consensus Problems. In *9th WDAG. Springer Verlag (LNCS 972)*, pages 87–100, 1995.

[11] J. Y. Halpern. Using Reasoning About Knowledge to Analyze Distributed Systems. *Annual Review of Computer Science, II*, pages 37–68, 1987. Ed. J.F.Traub.

[12] J. Y. Halpern and Y. Moses. Knowledge and Common Knowledge in a Distributed Environment. *JACM*, 37(3):549–587, 1990.

[13] M. Hiltunen and R. Schlichting. Understanding Membership Services. Technical Report 95-07, University of Arizona, 1995.

[14] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[15] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[16] S. Maffeis. *Run-Time Support for Object-Oriented Distributed Programming.* PhD thesis, Universität Zürich, 1995.

[17] C. Malloth and A. Schiper. View Synchronous Communication in Large Scale Networks. Technical Report TR94/84, Ecole Polytechniqe Fédérale de Lausanne, October 1994.

[18] S. Mishra, L. L. Peterson, and R. D. Schlichting. Consul: A Communication Substrate for Fault-tolerant Distributed Programs. *Distributed Systems Engineering Journal*, 1(2):87–103, 1993.

[19] G. Neiger and R. Bazzi. Using Knowledge to Optimally Achieve Coordination in Distributed Systems. In *Fourth TARK*, pages 43–59, 1992.

[20] A. Ricciardi. Dissecting Distributed Coordination. In *9th WDAG. Springer Verlag (LNCS 972)*, pages 101–115, 1995.

[21] A. Ricciardi. Sequential Distributed Coordination. Technical Report ECE-PDS-9503, The Univerisity of Texas, 1995.

[22] A. Ricciardi. Impossibility of (Repeated) Reliable Broadcast. Technical Report ECE-PDS-9603, The Univerisity of Texas, 1996.

[23] A. M. Ricciardi. Practical Utility of Knowledge-Based Analyses. In *Fourth TARK*, pages 15–28, March 1992.

[24] L. Sabel and K. Marzullo. Election Vs. Consensus in Asynchronous Systems. Technical Report CS95-411, University of California, San Diego, 1995.

[25] A. Schiper and A. Ricciardi. Virtually-Synchronous Communication Based on a Weak Failure Suspector. In *23rd FTCS*, pages 534–543, June 1993.

[26] M. R. Tuttle. *Knowledge and Distributed Computation*. PhD thesis, M.I.T., Cambridge, MA, 1989. Electrical Engineering and Computer Science.

[27] J. van Benthem. *Modal Logic and Classical Logic*. Bibliopolis, Milan, 1983. Monographs in Philosophical Logic an Formal Linguistics III.

[28] R. van Renesse, T. Hickey, and K. Birman. Design and Performance of Horus: A Lightweight Group Communications System. Technical Report 94-1441, Cornell University, 1994.