Toward Sofware Synthesis for Distributed Applications

Aleta Ricciardi* Paul Grisham†
aleta@ece.utexas.edu grisham@ece.utexas.edu
Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712

1 Introduction

This paper describes Sage, a software environment supporting software development, synthesis, and testing for distributed computing applications. While the principal domain of interest is applications that must be fault-tolerant (*i.e.*, be able to withstand the failure of some of the participants) Sage is not limited to this; it can be extended to distributed applications with no criticality requirements and to those with security requirements. Sage mechanically applies specialized knowledge-theoretic analyses to a distributed application's high-level specification to automatically derive the necessary communication between the participants in the computation. In particular, Sage implements the results of Chandy and Misra [2] and mimics the analyses of others [9, 10, 12, 15] which have previously only been performed theoretically. Sage applies these results to strategies commonly used by programmers of distributed applications, and commonly provided by packaged subsystems for distributed computing (also called "middleware"), to derive and synthesize correct, efficient solutions.

Using the modal logic of knowledge to describe and reason about coordinating distributed entities is well accepted [2, 3, 5, 6, 9, 10, 12, 15], if highly specialized, and the semantic model has been adapted to suit different distributed environments [6, 7, 10, 11]. In particular, epistemic logic has been used to optimize solutions, to prove impossibility, and to prove possibility [2, 5, 7, 9, 10, 12, 13].

Despite the range of applications in the literature, no software development environment exists that applies and implements these techniques to facilitate writing distributed applications.

*Supported by DOD-ARPA under contract number F30602-96-1-0313, and the Texas Higher Education Coordinating Board grant ARP 003658-260.

†Supported by MCD Fellowship and DOD-ARPA contract F30602-96-1-0313.

Equally significant, despite the expertise accumulated by middleware designers, no product yet contains any tool to guide programmers toward proficient use. Sage incorporates both theory and practical experience to assist developers of (fault-tolerant) distributed applications.

In Section 2 we describe the technical aspects of distributed computing that are specifically related to Sage, and give a concise background of how epistemic logic has been used in distributed computing. In Section 2.3 we describe a simple distributed coordination problem and run through a traditional knowledge analysis. Section 3 presents the Sage environment in detail, and discusses the role of Sage's components in analyzing the coordination problem described in Section 2.3.

2 Background

A *distributed system* consists of a finite set of computational elements, called *processes*, that are connected by a network of *channels*. Since the salient factor is communication between any pair of processes, we ignore special network topologies, and assume the channels completely connect the processes. The set of processes is denoted by $\mathsf{Proc} = \{p_1, ..., p_n\}$. Processes execute *events* in linear temporal order, determined by the program they are following. A process fails by *crashing*, which means it executes only a prefix of the events dictated by its program. For the work described here (though this is not a restriction in Sage) crashed processes do not recover. Channels are *reliable* in that messages sent by a process that never crashes, and destined for another process that never crashes, will eventually be received by the destination; otherwise, a message has only non-zero probability of reaching its destination. Channels do not garble or spontaneously generate messages.

In a synchronous distributed system, network delays and differences in processes' speeds are bounded and known; in an *asynchronous* distributed system these delays and variations are not known. While the low-level components of a distributed system can often be considered synchronous, software layers that implement common abstractions, such as processes and reliable channels, 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 networks 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.

2.1 Modeling Execution

A *history* for process p, denoted h_p, is a linear sequence of *events* beginning with the event $start_p$, which occurs exactly once in any h_p. Other events of interest are those describing

communication between processes, for which $send_p(q, m)$ denotes process p sending message m to process q and $recv_q(p, m)$ is the corresponding reception, and the event modeling the failure of process p, $crash_p$. The event $crash_p$ can be followed only by other $crash_p$ events.

A *run over* Proc is an n-tuple of infinite process histories, one for each $p \in$ Proc. Process p is *correct* in a run if the event $crash_p$ does not occur in the run; otherwise it is faulty. A run is well-formed if it reflects the assumptions of our model; *e.g.*, it contains the corresponding send event for every receive event it contains, $crash_p$ events are followed only by other $crash_p$ events, if p and q are correct, then the run contains $recv_q(p, m)$ if it contains $send_p(q, m)$, and so forth. We only consider well-formed runs over Proc.

A *cut over* Proc is an n-tuple of finite process histories, one for each $p \in$ Proc. A cut $c = (h_1, \ldots, h_p, \ldots, h_n)$ is *causally consistent* if and only if, for every $recv_q(p, m)$ event it contains, it also contains the corresponding $send_p(q, m)$ event [8]. A run r *completes* a cut c (written $c \in r$) if and only if the finite history component for p in c is a prefix of the infinite history component for p in r.

Consistent cuts in a run are naturally partially ordered by comparing prefixes of the component histories. Given $c = (h_1, \ldots, h_p, \ldots, h_n)$ and $c' = (h'_1, \ldots, h'_p, \ldots, h'_n)$, both in the same run r, $c \leq c'$ if and only if for each $p \in$ Proc, h_p is a prefix of h'_p. Consistent cuts, not necessarily in the same run, can also be divided into equivalence classes according to process histories; given $c = (h_1, \ldots, h_p, \ldots, h_n)$ and $c' = (h'_1, \ldots, h'_p, \ldots, h'_n)$, $c \sim_p c'$ if and only if $h_p = h'_p$.

Sage focuses on the many variants of the problem of *distributed coordination*, in which, loosely speaking, processes attempt to execute special actions of interest in a consistent (*i.e.*, non-interfering) manner. It can be argued that all of the complex issues in asynchronous distributed computing, and in coordination in particular, are due to the existence of non-determinism in processes' execution histories. Non-determinism arises from the order in which a process receives messages, and whether it suspects a remote process is crashed. If all processes ran the same program and were not subject to non-determinism, then the state of each process at all intermediate points would be identical, and consistency (*i.e.*, the lack of interference) would exist trivially. Otherwise, processes must reach agreement on the relative order, and even the existence of, non-deterministic events.

2.2 Formal Language

We use linear time temporal logic for expressing the variants of the problem of asynchronous distributed coordination; the epistemic aspects arise in analyzing and generating solutions. All formulas are evaluated on consistent cuts, in the context of a run. Such a pair (r, c) is called a *point*. Since the relation $\sim_p$ is insensitive to other processes' histories, it is trivial to extend the relation to points, $(r, c) \sim_p (r', c')$.

2.2.1 Semantics of Logical Formulas

For φ a formula of our language, we write $(r, c) \models \varphi$ to denote that φ is true at the point (r, c). Base propositional formulas include $\text{SEND}_p(q, m)$, which is true at point (r, c) precisely when $send_p(q, m)$ is an event in p's history component of c. Analogous statements hold for the formulas $\text{RECV}_q(p, m)$ and $\text{CRASH}(p)$. The semantics of the *temporal* modal operators are: $(r, c) \models \Box\varphi$ if and only if for each $c' \in r$, such that $c \leq c'$, we have $(r, c') \models \varphi$. The interpretation is that φ holds at (r, c) and throughout the rest of the run. The dual of $\Box\varphi$ is $\Diamond\varphi$ which is interpreted as φ holding at some, but not every, future point (r, c'). The *knowledge* modal operator is $K_p\varphi$ (p knows φ). The semantics capture the idea that, given its current history, p cannot imagine φ being false: $(r, c) \models K_p\varphi$ if and only if for all points $(r', c') \sim_p (r, c)$, we have $(r', c') \models \varphi$. Note that crashed processes can know facts; but, since they cannot execute further send events, they cannot communicate their knowledge.

2.2.2 Knowledge Acquisition

The analyses on which Sage is based are derived from the results of Chandy and Misra describing how one process learns facts about another process [2]. A formula is said to be *local* to process p if at every point, p knows whether it is true. For example, all formulas describing a process's execution history, for example $\text{SEND}_p(q, m)$, $\text{RECV}_q(p, m)$ and $\text{CRASH}(p)$, are local to only that process. The introspection and negative introspection axioms for epistemic logic (the modal logic S5) also mean that knowledge is local to the knower: for each formula φ : $K_p(K_p\varphi) \vee K_p\neg(K_p\varphi)$.

Chandy and Misra show that for formulas local to only one process, others' knowledge of them is attained only by communicating with that process. Because $\text{CRASH}(p)$ is local only to p, this underscores the impossibility of any other process ever knowing that p is crashed. We rephrase their crucial knowledge-acquisition theorem for our purposes.

Proposition 1 *[Chandy-Misra] Suppose φ is local only to p. Let (r, c) and (r, c') be two points with $c < c'$. If $(r, c) \models \neg K_q\varphi$ and $(r, c') \models K_q\varphi$, then c' contains a chain of messages that originated with p and terminates with q, and that implies the continued truth of φ.*

2.3 Uniform Distributed Coordination

In this section, we develop the example of Uniform Distributed Coordination (UDC) [4], showing how traditional knowledge-based analyses proceed, and how the problem can be further generalized. As we have implemented the communication rules described here for UDC in Sage, this example also serves to describe the first stages of the project.

By *distributed coordination* among Proc, we mean that the members of Proc are attempting to each execute certain *actions* of interest. This includes most canonical problems in distributed computing. For the moment, we are not concerned with other requirements such as executing actions in a particular order. Sage has hooks to handle many such specifications, but in this example, we focus only on the eventual execution of these actions. The event whereby p executes action α is denoted $do_p(\alpha)$. As with other events processes execute, the formula $\text{DO}_p(\alpha)$ holds at (r, c) exactly when $do_p(\alpha)$ is an event of h_p, and $\text{DO}_p(\alpha)$ is local only to p.

Because formulas are evaluated on cuts, the clauses specifying UDC can be understood as formulas describing what must be true on that cut when any process, say p, takes an action of interest. This is (usually) a conjunction of formulas describing the state of individual processes. If any of these are not local to p, then messages must have been exchanged. Colloquially, Uniform Distributed Coordination of actions, $\alpha \in \mathcal{A}$, among processes $p \in \mathsf{Proc}$ specifies that if any $p \in \mathsf{Proc}$ executes $\alpha \in \mathcal{A}$ then every correct $q \in \mathsf{Proc}$ eventually also executes α. This is written as,

$$p \in \mathsf{Proc} \wedge \text{DO}_p(\alpha) \Rightarrow \forall q \in \mathsf{Proc} : \Diamond\Big(\text{DO}_q(\alpha) \vee \text{CRASH}(q)\Big) \ . \tag{1}$$

Each action $\alpha \in \mathcal{A}$ is associated with a single process, p_α, that initiates its uniform distributed execution in a run. The event whereby p_α first becomes aware that α should be executed in a run is denoted $try_{p_\alpha}(\alpha)$. The formula $\text{TRY}(\alpha)$ holds at (r, c) exactly when $try_{p_\alpha}(\alpha)$ is an event in h_{p_α}. Since p_α is unique, we have[1]

$$K_p\text{TRY}(\alpha) \Rightarrow (p = p_\alpha) \vee \text{RECV}_p(p_\alpha, \text{TRY}(\alpha)) \ . \tag{2}$$

There are other clauses to the specification of UDC, but Sage, because it reasons backward about what must be true when an action is taken, is primarily interested with Formula 1, the agreement clause. This problem is impossible to solve in asynchronous distributed systems when the number of faulty processes is not known in advance [13]. In Formula 1, a process is not obliged to execute α only if it is crashed. UDC can be generalized by considering abstract *exemptions from coordination*,

$$p \in \mathsf{Proc} \wedge \text{DO}_p(\alpha) \Rightarrow \forall q \in \mathsf{Proc} : \Diamond\Big(\text{DO}_q(\alpha) \vee \text{EXEMPT}(q, \alpha)\Big) \tag{3}$$

The semantics of $\text{EXEMPT}(q, \alpha)$ and the ramifications for UDC depend on how it is instantiated. For example, when $\text{EXEMPT}(q, \alpha)$ is the truth identity $\top$, no real coordination is achieved because the consequent in Formula 3 is trivially satisfied. At the other side of the spectrum, instantiating $\text{EXEMPT}(q, \alpha)$ with the false identity $\bot$ precludes solutions if even one process can crash. Instantiating $\text{EXEMPT}(q, \alpha)$ with $\text{CRASH}(q)$ gives Formula 1.

[1] $\text{TRY}(\alpha)$ is not local to p_α since in each run any process may be the unique initiator of any action. When $\text{TRY}(\alpha)$ does not hold, no p actually knows this, $\neg\text{TRY}(\alpha) \Rightarrow \forall p \in \mathsf{Proc} : \neg K_p \neg \text{TRY}(\alpha)$. With simple restrictions the results in [2] apply.

Knowledge Analysis of UDC Knowledge analyses are based on the fact that 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. For valid formulas, the modal logic *generalization inference rule* introduces the knowledge operator. Starting with UDC and abstract exemptions, knowledge analyses are:

1. Apply modal logic generalization to Formula 3, then tautological reasoning of S5

$$K_p\Big(p \in \mathsf{Proc}\Big) \wedge K_p \mathrm{DO}_p(\alpha) \Rightarrow \forall q \in \mathsf{Proc} : K_p\Big(\Diamond \mathrm{DO}_q(\alpha) \vee \Diamond \mathrm{EXEMPT}(q, \alpha)\Big)$$

2. Local formulas in step 1 antecedent, give $K_p(p \in \mathsf{Proc}) \wedge K_p \mathrm{DO}_p(\alpha) \Leftrightarrow p \in \mathsf{Proc} \wedge \mathrm{DO}_p(\alpha)$.
3. Temporal logic tautology to Validity clause[2], $\Diamond \mathrm{DO}_q(\alpha) \Rightarrow \Diamond K_q \mathrm{TRY}(\alpha)$
4. We have the intermediate statement

$$p \in \mathsf{Proc} \wedge \mathrm{DO}_p(\alpha) \Rightarrow \forall q \in \mathsf{Proc} : K_p\Big(\Diamond K_q \mathrm{TRY}(\alpha) \vee \Diamond \mathrm{EXEMPT}(q, \alpha)\Big) \ .$$

5. Accountability. This is proven in [13]. Obviously, the knowledge operator does not normally distribute over disjunction, but in this particular case, for this problem it does.

$$K_p\Big(\Diamond K_q \mathrm{TRY}(\alpha) \vee \Diamond \mathrm{EXEMPT}(q, \alpha)\Big) \Leftrightarrow K_p \Diamond K_q \mathrm{TRY}(\alpha) \vee K_p \Diamond \mathrm{EXEMPT}(q, \alpha) \ .$$

6. Cannot predict others' future knowledge (also proven in [13]), $K_p \Diamond K_q \mathrm{TRY}(\alpha) \Leftrightarrow K_p K_q \mathrm{TRY}(\alpha)$.
7. From Formula 2 and because $\mathrm{RECV}_q(p_\alpha, \mathrm{TRY}(\alpha))$ is local only to q we have, $K_p K_q \mathrm{TRY}(\alpha) \Leftrightarrow \mathrm{RECV}_p(q, K_q \mathrm{TRY}(\alpha))$.
8. Finally, we have, $p \in \mathsf{Proc} \wedge \mathrm{DO}_p(\alpha) \Rightarrow \forall q \in \mathsf{Proc} : \mathrm{RECV}_p(q, K_q \mathrm{TRY}(\alpha)) \vee K_p \Diamond \mathrm{EXEMPT}(q, \alpha)$.

Rules Needed by Sage To derive a minimal communication solution to UDC, Sage needs only these rules. *Communication rules* describe the interaction of messages (messages received must have been sent), $\mathrm{RECV}_p(q, m) \Rightarrow \mathrm{SEND}_q(p, m)$, and knowledge of message contents[3], $\mathrm{SEND}_p(q, m) \Rightarrow K_p m$. A pleasant artifact of Sage's reasoning is that the analogous rule $\mathrm{RECV}_p(q, m) \Rightarrow K_p m$ is redundant. The *local formula rule* describes Proposition 1, $K_p \varphi \Rightarrow \mathrm{LOCALTO}(p, \varphi) \vee (\mathrm{RECV}_p(q, \varphi) \wedge \mathrm{LOCALTO}(q, \varphi))$. This covers the UDC-specific requirement stated in Formula 2 that $\mathrm{TRY}(\alpha)$ can be considered local to p_α. Knowing formulas that are not local to any process (*e.g.*, certain instantiations of $\mathrm{EXEMPT}(,)$) will obviously need other rules against which to resolve their truth. The *UDC rule* is step 8 in the previous knowledge analyses. Finally, Sage has *exemption rules.* Aside from the simple exemptions discussed, Sage

[2]The Validity clause of UDC requires that only actions arising in a run be taken, not every action enumerable in $\mathcal{A}$, $\mathrm{DO}_p(\alpha) \Rightarrow K_p \mathrm{TRY}(\alpha)$.

[3]For simplicity, we abuse notation to allow a message m to also stand for the formula describing its contents.

implements one that permits non-trivial safe solutions to UDC in asynchronous systems.[4] We call it SIMCRASH(q), because it results in q appearing as if it were crashed. SIMCRASH() is not local to any process, but is a conjunction of local formulas that, roughly speaking, indicate processes' acceptance of q's exemption.

3 Description of Sage

Sage, when completed, will consist of: a *system definition* component, through which users describe the distributed system's physical characteristics; a *specification* component, through which users specify distributed coordination problems at a high level (Sage generates a knowledge specification from this); a *communication graph extractor*, which derives the messages required to solve the problem given the generated knowledge specification; a *laboratory* component, in which users experiment on the derived solution (*e.g.*, changing the order of events, crashing processes, and partitioning the network), after which Sage generates a new solution; an *identification* component to recognize common distributed communication primitives in the extracted graph; and a *software synthesis* component, derived from the identification tool, to produce skeleton program code.

Small parts of the first four of these have been tested and implemented. We describe here only the specification, extractor, laboratory, and identification components.

3.1 Problem Specification Component

The appeal of Sage is that while its underlying formalism is epistemic logic, users are not specifying problems in terms of process knowledge, or even in terms of formal logic. Users simply instantiate 'variables' in the problem specification and system model through a series of simple and restricted interactions. On-going research and experience programming such applications determines which are the relevant aspects and possible values. The following are the most common clauses of distributed coordination.

Uniform Distributed Coordination Formula 3 is the simplest statement of distributed coordination and is the foundation of more complex coordination problems. UDC, with various exemptions, is the minimal problem Sage will solve.

Exemptions These define the conditions under which a process is not required to execute the coordinated action. Knowledge-theoretic analyses show that the SIMCRASH() exemption gives rise to what is called *primary partition behavior* [1] (*i.e.*, only a majority, or otherwise uniquely designated subset of processes, can continue to take actions). Primary partition behavior is

[4]The solution is live if a majority of processes remain in communication with one another.

commonly provided by middleware, though it is not appropriate for wide-area systems, or for applications without strong fault tolerance requirements. Thus, an important corollary of these analyses is that they give an exact way of specifying UDC variants that have solutions in *partitionable systems* (those in which subsets of processes may be unable to communicate with others).

Sage currently can derive necessary and sufficient communication for UDC, with any of the four exemptions $\top, \bot$, CRASH(), SIMCRASH(). Exemptions are made when processes crash, drop messages, and otherwise appear unresponsive.

Sequential Coordination This adds to UDC a generic clause stipulating conditions under which two actions must be taken by all obliged processes in the same relative order. While UDC alone is akin to information dissemination, Sequential UDC is a problem of conflict resolution; these problems are often significantly more difficult. For two actions α and β, if $\Sigma(\alpha, \beta)$ holds, then a process that executes β must already have executed α,[5] $\Sigma(\alpha, \beta) \Rightarrow \forall p \in \mathsf{S} : \text{DO}_p(\beta) \Rightarrow \text{DO}_p(\alpha)$.

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. For example, the difference between instantiating $\Sigma(\alpha, \beta)$ with a clause that results in partially ordering actions, and instantiating it with a clause that results in totally ordering actions is obvious. In the most common instance of partially ordered actions, all the facts a process must learn before it can execute β are knowable when $try_{p_\beta}(\beta)$ occurs; in any required total ordering of actions, further communication (or more restrictive assumptions) is always required to learn facts that are local to processes other than p_β and that contribute to determining the total order.

Other than the explicit instantiation of $\Sigma(\alpha, \beta)$, only two additional rules are needed to express sequential properties. One describes the knowledge required before any process can execute its first action, the other describes the knowledge required to execute any subsequent action: $\text{DO}_p(\alpha) \wedge \neg\text{DO}_p(\beta) \Rightarrow K_p \Box \neg \Sigma(\beta, \alpha)$, and $\text{DO}_p(\beta) \wedge \Sigma(\alpha, \beta) \Rightarrow K_p \Sigma(\alpha, \beta) \wedge \text{DO}_p(\alpha)$.

Termination A termination clause defines conditions under which every obliged process must either have taken an action or have been deemed exempt; for an action α, $\tau(\alpha) \Rightarrow \forall p \in \mathsf{S} : \text{DO}_p(\alpha) \vee \text{EXEMPT}(p, \alpha)$. It turns out that here, again, the knowledge modality does distribute over disjunction meaning that the communication that must be present for $K_p \tau(\alpha)$ to hold is either a message from q indicating $\text{DO}_q(\alpha)$ or (possibly) more complex communication indicating $\text{EXEMPT}(q, \alpha)$.

Knowing termination is one of the harsh realities of finite computer resources, from registers,

[5] It is debatable whether a process that is exempt from doing α may nonetheless do β.

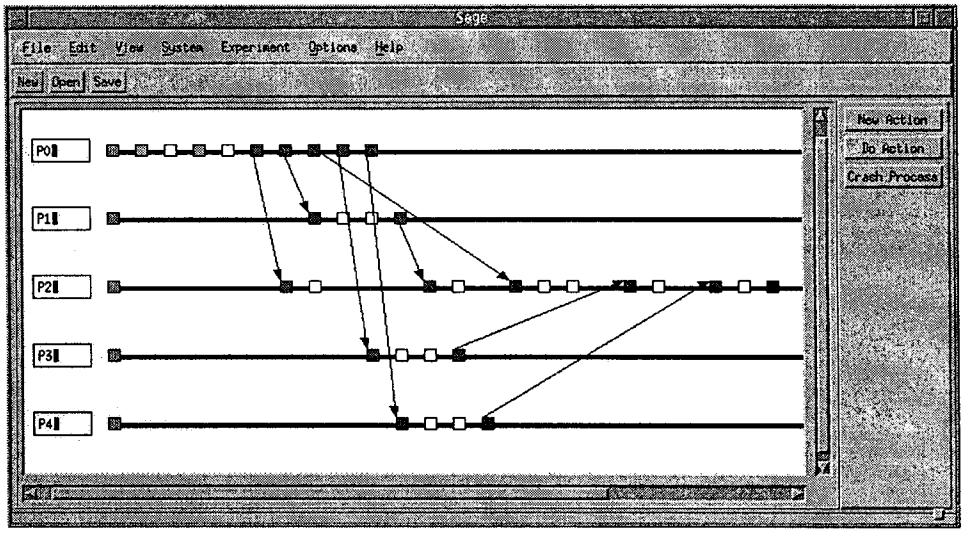

Figure 1: Sage's generated solution for a failure-free run of Uniform Coordination.

to network interface cards, to primary memory. For distributed applications and middleware, the most common occurrence is some flavor of *virtual synchrony* [1]. Loosely speaking, virtual synchrony describes the behavior of group of processes that is trying to appear as if they were a single, fault-tolerant process, historically the most common reason for forming process groups processes and coordinating members' behavior. To express virtual synchrony, Sage uses *barrier actions* and *subordinate actions*. For example, virtual synchrony can seen as ordering message delivery (subordinate) with respect to changes in the membership of the group of processes delivering them (barrier).

3.2 Extractor Component

The extractor component generates an interactive schematic diagram of the messages necessary to solve the problem specified. To run the extractor, a Sage user indicates an action, say α, and selects two processes by clicking on their histories. One process, the start-point, is the initiator of the action (in Figures 1 and 2, $p_\alpha = P_0$), with the event $try_{P_0}(\alpha)$ occurring at the indicated place in its history (the light-blue box, second from left in h_{P_0}). The other process is the end-point (P_2 in these figures), and has executed $do_{P_2}(\alpha)$ at the indicated place in its history (the dark-blue box, rightmost in h_{P_2}). The extractor then builds a solution for the specified problem in a failure-free run. Given this solution, the laboratory component allows users to examine how process and network failures affect the required communication.

From $\text{DO}_{P_2}(\alpha)$, the extractor iteratively resolves formulas until it arrives at `true`, or can resolve no further. Once the end-point has taken an action, Sage determines what must be true

immediately before that. Or, solve for Φ in, $\text{DO}_{P_2}(\alpha) \Rightarrow \Phi$. The UDC rule gives $\Phi = \forall q \in \text{Proc} : \text{RECV}_{P_2}(q, K_q\text{TRY}(\alpha))$. To derive the conditions required for this to hold, solve for Ψ in $\Phi \Rightarrow \Psi$. A basic communication rule gives $\Psi = \forall q \in \text{S} : \text{SEND}_q(P_2, K_q\text{TRY}(\alpha))$. Continuing, we have, $\text{SEND}_q(P_2, K_q\text{TRY}(\alpha)) \Rightarrow K_q\text{TRY}(\alpha) \Rightarrow \text{RECV}_q(P_0, \text{TRY}(\alpha)) \Rightarrow \text{SEND}_{P_0}(q, \text{TRY}(\alpha)) \Rightarrow K_{P_0}\text{TRY}(\alpha) \Rightarrow \texttt{true}$.[6]

While Sage's message extractor uses a reasoning method similar to resolution proofs [14], it is important to note here that Sage is not a resolution theorem prover. Sage is better thought of as a trace analyzer of a pre-determined resolution proof; from the trace, we can determine which events are necessary to have occurred. Sage maintains only the rules necessary to resolve the problem specification and does not implement any of the tautological manipulations or inference rules of epistemic, or even predicate logic. As Sage presents users with a controlled interface, we are not (yet) concerned with logically inconsistent problem specifications.

The primary challenges are to keep the number of rules small, and to avoid situations in which multiple rules could apply. We are optimistic that neither will be an issue. For example, the number of rules for Sequential UDC is remarkably small; as well, if including multiple rules with the same antecedent becomes unavoidable (we have yet to see such a situation) experience writing distributed applications gives heuristics for applying one such rule over another. For pathological cases, Sage can backtrack if necessary.

3.3 Laboratory Component

This component gives users the ability to experiment on a protocol in a simple, graphical manner; as well, it is the foundation for identifying communication primitives and from there, to software synthesis. 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 animates how process and communication failures affect protocols. Because it is a simulation, Sage separates the issues in testing a protocol's required adaptations in the face of failures, from the effects background system conditions have on the testing procedure itself.

Sage users can crash processes at any juncture in the generated protocol, drop or delay any message, and re-order events to discover causal dependencies. To further explore partitionable

[6]Basic communication gives the first implication, Chandy-Misra the second, basic communication the third and fourth, and Chandy-Misra for $\text{TRY}(\alpha)$ gives the last.

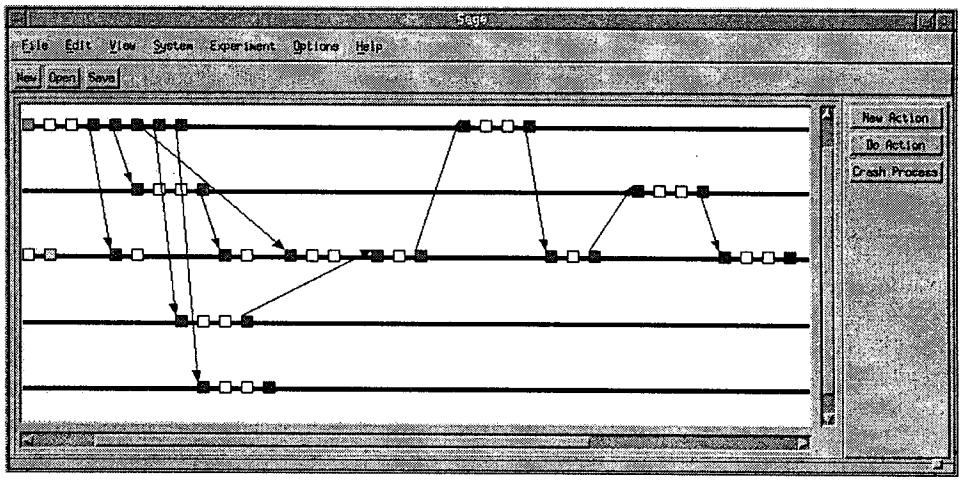

Figure 2: Regenerated solution after a user crashes P_4 before $send_{P_4}(P_2, K_{P_4}\text{TRY}(\alpha).)$

coordination, users will also be able to create network partitions. The reconfigured solution animates the protocol steps needed to adapt to the change, or indicates no progress is possible. Ultimately, the effects of such experiments are determined by the declared exemption. For example, Figure 2 shows a user has crashed P_4 before it executed $send_{P_4}(P_2, K_{P_4}\text{TRY}(\alpha))$; despite this, $do_{P_2}(\alpha)$ has still been taken. To satisfy the UDC rule, $K_{P_2}\Diamond\text{EXEMPT}(P_4, \alpha)$ must hold, and so the reconfigured solution shows the messages necessary for P_0 to attain this knowledge. The exemption in this case is SIMCRASH(), and P_0, P_1 and P_2 comprise the majority subset of Proc agreeing to P_4's exemption.

3.4 Primitives Identification Component

This component is in the early stages of implementation. It will recognize the presence of different middleware primitives, including many implementations of multicast, membership changes and virtual synchrony whose semantics differ only subtly. This helps programmers use them efficiently, and lays the foundation for high level software synthesis. Such primitives are themselves coordination problems and so can be described by the usual logic. Multicast primitives differ from one another in the order in which messages are delivered to all processes. To detect the presence of a particular multicast, Sage will *perform its own experiments* on the solution is has just generated, changing the order in which relevant messages are delivered. If the original solution is insensitive to the experiments (*i.e.*, does not adapt to the change in event order), then the primitive is not necessary.

Conclusions

Sage is unique in embodying much of the complex modeling, formalism, and theory from research in asynchronous distributed computing. By allowing programmers to use these in an intuitive and interactive fashion, the utility of this research can be realized. In the next two months, a group of undergraduates, who have no previous knowledge of distributed computing (not to mention modal logic), will play with Sage. We want their comments to assess Sage's utility as a pedagogical tool, both for academic instruction, and for actual use in industrial software development.

References

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

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

[3] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge.* The MIT Press, 1995.

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

[5] 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.

[6] J. Y. Halpern and R. Fagin. A Formal Model of Knowledge, Action and Communication in Distributed Systems. In *Fourth PODC*, pages 224–236. ACM, 1985.

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

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

[9] M. S. Mazer. A Link Between Knowledge and Communication in Faulty Distributed Systems. In R. Parikh, editor, *Third TARK*, pages 289–304, 1990.

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

[11] P. Panangaden and K. Taylor. Concurrent Common Knowledge: Defining Agreement for Assynchronous Systems. *Distributed Computing*, 6(2):73–94, 1992.

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

[13] A. Ricciardi. A Knowledge-Theoretic Analysis of Partitionable Coordination. Technical Report PDS-1997-013, The Univerisity of Texas, 1997.

[14] E. Rich and K. Knight. *Artificial Intelligence, Second Edition.* McGraw Hill, 1991.

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