Modelling Knowledge and Action in Distributed Systems*

Joseph Y. Halpern
Ronald Fagin

IBM Almaden Research Center
San Jose, CA 95120
email: halpern@ibm.com, fagin@ibm.com

Abstract: We present a formal model that captures the subtle interaction between knowledge and action in distributed systems. We view a distributed system as a set of *runs*, where a run is a function from time to *global states* and a global state is a tuple consisting of an *environment state* and a *local state* for each process in the system. This model is a generalization of those used in many previous papers. *Actions* in this model are associated with functions from global states to global states. A *protocol* is a function from local states to actions. We extend the standard notion of a protocol by defining *knowledge-based protocols*, ones in which a process' actions may depend explicitly on its knowledge. Knowledge-based protocols provide a natural way of describing how actions should take place in a distributed system. Finally, we show how the notion of one protocol *implementing* another can be captured in our model.

*Some material in this paper appeared in preliminary form in [HF85]. An abridged version of the paper appeared in *Proceedings of Concurrency 88* (Fritz Vogt, ed.), Springer Verlag, 1988, pp. 18–32. This version is essentially as one that appeared in *Distributed Computing* **3**:4, 1989, pp. 159–177.

... you act, and you know why you act, but you don't know why you know that you know what you do.

Umberto Eco, *The Name of The Rose*

1 Introduction

It has been argued [HM90] that the right way to understand and reason about distributed protocols is in terms of how the *knowledge* of the processes in a system changes. In this paper we look carefully at the notion of knowledge in distributed systems. In particular, we consider the interaction between knowledge and action. Intuitively, a process' actions depend on its knowledge, and its knowledge changes as a result of actions. The precise interaction between knowledge and action can be subtle, as is demonstrated by the analyses performed in such papers as [CM86, DM90, HM90, Leh84, MDH86, MT88]. Our aim is to understand and clarify these subtleties.

We start by providing a formal model of distributed systems. There are a number of ways that one can model a system of interacting processes or agents; it is doubtful that there is a "best" model. Whereas one approach might lend itself naturally to a certain type of analysis, it might not be useful for another. Ideally we would like an approach that is abstract and general, and yet can be easily specialized to capture important special cases of systems such as asynchronous message-passing systems, shared-memory models of parallel computation (PRAMs), or systems of communicating human agents or robots. Of course, we also want the model to be natural and intuitive, and lend itself easily to most types of formal analysis.

We describe a model here that we believe fulfills these properties. The model is motivated by previous work on knowledge-based analyses of protocols [CM86, DM90, FHV92, FI86, Had87, HF85, HM90, HZ92, MT88, NT93, PR85] (see [Hal87] for an overview). In all of these papers (and many others that have appeared in the literature), a protocol is identified with a set of *runs* or *executions*. We intuitively think of a run as a complete description of all the relevant events that occur in a system over time, where for convenience we think of time as ranging over the natural numbers. At every step in a run, the system is in some *global state*, where a global state is a description of each process' current local state and the current state of the *environment*. We use the environment component of the global state to capture everything that is relevant to the system that is not described by the states of the processes.

Like the models of [LF81, Lam86] and others, our model is geared to describing the behavior of a distributed system in a natural way. However, unlike Milner's CCS [Mil80] or Pratt's notion of *pomsets* [Pra85], we do not attempt to give a calculus which allows us to view a protocol as being put together from other protocols via various combining forms (such as composition). While we could define ways of combining simpler systems to form more complicated systems, our framework does not lend itself naturally to such an approach. We can view the distinction between our type of formalism and that of Milner and Pratt as somewhat like the distinction between Temporal Logic [Pnu77], which focuses on the analysis of a given system, and Dynamic Logic [Har79, Pra76], which explicitly allows programs to be combined into more complicated programs.

We incorporate knowledge into the model by using the basic framework described in [HM90]. Given a global state s and a process i, there may be many global states consistent with i's information in s, that is, many global states s' where i has the same local state as in s. We say that process i *knows* a fact φ at a certain point in a run if φ is true at all the points where φ has the same local state. This notion of knowledge in distributed systems can be easily shown to satisfy the axioms of the classical modal logic S5 (see [HM92] for more details). Note that it is an *external* notion of knowledge. We do not assume that the processes somehow think about the world and do some introspection in order to obtain their knowledge. Rather, this is knowledge that is ascribed by us (or the system designer) to the processes. A process cannot necessarily answer questions based on this notion of knowledge. Nevertheless, this definition has been shown to capture much of the intuitive reasoning that is done about protocols. One often hears statements such as "Process p_1 should send an acknowledgment because p_2 does not know that p_1 got the message." The phrase "p_2 does not know that p_1 got the message" can easily be given a formal interpretation in this model. Moreover, it is an interpretation that directly captures the intuitions of the system designers doing such reasoning.

We define runs, actions, and systems formally in Section 2, and show how to ascribe knowledge to processes in a distributed system in Section 3. The notion of *protocol* is discussed in Section 4. We view a protocol as a function from a process' local state to *actions*. This definition of protocol is quite a general one, and certainly includes all protocols that can be described in current programming languages. However, using such *standard protocols* we cannot naturally describe situations where a process' actions depend explicitly on its knowledge. For example, consider a protocol such as "If I know that you are planning to attack, then I will attack too." This is an example of what we call a *knowledge-based protocol*. Knowledge-based protocols give us a way to directly describe the relationship between knowledge and action, and thus provide a convenient high-level description of what a process should do in certain situations. We discuss knowledge-based protocols in detail in Section 5.[1] In Section 6, we consider the *cheating husbands problem*, informally discussed in [MDH86], and show how it can be captured in our framework. This example also shows how the same knowledge-based protocol corresponds to distinct standard protocols in different systems. In Section 7 we discuss what it means for one protocol to *implement* another in our context. We conclude in Section 8 by suggesting some directions for further research.

2 Runs, actions, and systems

As we mentioned in the introduction, we identify a system with its set of possible runs, where a run is a description of the system's behavior over time. But how can we best describe a system's behavior?

In most papers on distributed systems, two key notions that appear repeatedly are *states* and *actions*. Consider a very simple distributed system, consisting of only one process running a sequential program. As Lamport points out [Lam85], a run of this program can be viewed as a sequence

$$s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \cdots$$

[1] Some of the material in Sections 2–5 appeared in preliminary form in [HZ92].

where $s_0, s_1, s_2, \ldots$ are states and $\alpha_0, \alpha_1, \alpha_2, \ldots$ are actions. In this view a process is an automaton, which is always in one of a (possibly infinite) number of internal states. We do not make any additional assumptions about the structure of these internal states, although, of course, there will invariably be extra structure once we consider more concrete applications. In this framework, an action is simply a state transformer, a function mapping one state into another.

We want to extend this viewpoint to more complicated systems. If we have, say, n processes in the system, the state of the system will clearly have to include the state of each of the processes. But, in general, more than just the state of the processes may be relevant when doing an analysis of the system. If we are analyzing a message-based system, we may want to know about messages that have been sent but not yet delivered or about the status of a communication link (such as whether it is up or down).

Motivated by these observations, we conceptually divide a system into two components: the processes and the *environment*, where we view the environment as "everything else that is relevant". We define a *global state* of a system with n processes or agents to be an $(n+1)$-tuple $(s_e, s_1, \ldots, s_n)$, where s_e is the state of the environment and s_i is the (local) state of process i.

The way we divide the system into processes and environment will depend on the system being analyzed. In a message-based system, we could view the message buffer as one of the processes or as part of the environment. If it is viewed as a process, then its state could encode which messages have been sent but not yet delivered. Similarly, we could view a communication line as a process, whose local state might describe (among other things) whether or not it is up, or we could have the status of the communication lines be part of the environment.

As in the single-process case, actions change the global state of the system. But, unlike the single-process case, we can no longer look at individual actions in isolation. Actions performed simultaneously by different components of the system may interact. For example, we must explain what happens when two processes simultaneously try to write to the same register. To this end, we define we define a *joint action* to be a tuple $(a_e, a_1, \ldots, a_n)$, where a_e is an action performed by the environment, and a_i is an action performed by process i. We associate with each joint action a global state transformer. If τ is the function that associates a global state transformer with every joint action, then $\tau(a_e, a_1, \ldots, a_n)(g)$ is the global state that results when the actions $a_e, a_1, \ldots, a_n$ are performed simultaneously by the environment and the processes when the system is in global state g. Note that we allow the environment, not just the processes, to perform actions in our framework. Although we do require that processes perform an action at every step, this is not a serious restriction. We can use a null action to capture the possibility that no non-trivial action is performed. It may also seem that we are assuming that all actions are *atomic*, but this is not the case. We return to this point in our examples below.

We can now extend the picture described above for the single-process case by viewing a run as a sequence of global states, joined by arcs labelled by a joint action. It turns out that for many of our applications we can (and do) essentially ignore the actions and "erase" them from the picture.[2] Formally then, we take a *run* to be a function from "real time" to global states. For convenience, we take "real time" here to range over the natural numbers. We could perfectly well have taken time to range over the real numbers or, in fact, any other linear order. We do *not* assume that processes in the system necessarily have access to real time; if they do,

[2]It is interesting that in Milner's CCS the dual approach is taken; the states are erased, leaving only the actions.

this would simply be encoded as part of their local state.

Let L_e be a set of possible (local) states for the environment, and let L_i, $i = 1, \ldots, n$, be local states for each of the processes. Let $\mathcal{G} \subseteq L_e \times L_1 \times \cdots \times L_n$ be a set of global states. A *run over* $\mathcal{G}$ is a function from the natural numbers to $\mathcal{G}$. Thus a run over $\mathcal{G}$ can be thought of as a sequence of global states. Intuitively, $r(m)$ is the global state of the system at time m in run r. If $r(m) = (s_e, s_1, \ldots, s_n)$, we define $r_i(m) = s_i$ for $i = 1, \ldots, n$. We refer to a pair (r, m) consisting of a run r and time m as a *point*; thus, $r_i(m)$ is process i's local state at the point (r, m). A *system over* $\mathcal{G}$ is a set of runs over $\mathcal{G}$. We say that (r, m) is a point in system $\mathcal{R}$ if $r \in \mathcal{R}$. We remark that, in practice, the set of runs making up the system will be chosen by the system designer or the person analyzing the system, who presumably has a model of what the possible executions of the protocol are.

There are two major assumptions we have made here that, while relatively common, are not made in a number of other papers: namely, that we can view time as a linear order, rather than viewing it as just a partial order, and that it makes sense to talk about the global state of the system. While these assumptions can be relaxed, they make our presentation far easier, and they seem to be appropriate for the systems we wish to analyze. (See [Lam85, Pra82] for some discussion about and arguments against these assumptions; see [PT92] for a discussion of how knowledge can be captured in a situation where we have partial orders.) There is a third assumption that we make for simplicity, namely, that there is a fixed set of n processes in the system. We could easily extend the notion of global state to allow for processes leaving and joining the system; we do not do this in order for our main points to come across clearly.

We view runs here as infinite objects, describing events over all time. It is occasionally convenient to consider *finite runs*, which are functions from an initial segment of the natural numbers to global states. Given a run $r \in \mathcal{R}$, let $r|_m$, the *restriction* of r up to time m, to be the finite run with domain $\{0, \ldots, m\}$ that agrees with r on their common domain. We say that ρ is a *prefix* of r if $\rho = r|_m$ for some $m \geq 0$. If a finite run ρ has domain $\{0, \ldots, m\}$, we say that its *length*, denoted $|\rho|$, is m. (We can think of $|\rho|$ as the number of transitions in ρ.) Given a system $\mathcal{R}$, let $\mathrm{Pref}(\mathcal{R})$ consist of the runs in $\mathcal{R}$ together with all the finite prefixes of runs in $\mathcal{R}$. If $\rho, \rho' \in \mathrm{Pref}(\mathcal{R})$, then we say that ρ is a *prefix* of ρ', and write $\rho \preceq \rho'$, if for some $r \in \mathcal{R}$ and $m \leq m'$, we have $\rho = r|_m$ and $\rho' = r|_{m'}$.

Systems can often be characterized by the types of actions that are allowed. Typical actions in a system might include reading and writing a shared variable, sending a message, receiving a message, and local computations. How these actions change the global state of the system will depend to some extent on the details of how we model the processes' local states and the state of the environment. At this point, the choice of how to model a system, including choosing the state space for the processes and the environment and deciding on the set of runs that make up the system, is more of an art than a science. We give some examples below (and in later sections of the paper) to show how this formalism can be used to capture a number of situations that arise in distributed and parallel computing.

Example 2.1: In an *asynchronous message-based system*, we assume that there are three types of actions: *sending* messages, *delivering* messages, and *local computations*. We assume that the environment state is simply a description of the message buffer: those messages that have been sent but not yet delivered. When a process sends a message, the effect of this action is to put the message (marked with its intended recipient) into the message buffer, and perhaps to change the sending process' state so as to record that the message has been sent. The action of delivering a message is performed by the environment; it results in that message being removed from the message buffer and the state of the recipient process perhaps being changed in some way to record the fact that it has received a message. Local computations affect only the state of the process performing the action.

By assuming that the environment delivers only messages that are in its buffer and removes a message once it is delivered, we have made a number of implicit assumptions about message delivery. Although messages can come out of order, we do not allow messages to be corrupted or duplicated. Moreover, the environment cannot deliver a message that was never sent (although it is possible that a message that was sent will never be delivered). Of course, we can easily alter the model to accommodate all of these possibilities. For example, if we want to allow messages to be duplicated (so that the same message can be delivered a number of times), we simply change the semantics of message delivery so that delivering a message does not result in that message being removed from the buffer. We can allow for corruption in a number of ways. Perhaps the most elegant is to view the delivery of a message as a nondeterministic action, which can transform a process' state in a number of ways (intuitively, one corresponding to each of the ways the message could be corrupted).

There are a number of other restrictions on message delivery that one frequently wants to capture. For example, we may want to require that all messages are eventually delivered, or that a message is either delivered within some time T or not delivered at all. We could capture these restrictions in our framework in several ways. One approach is to take the system to consist only of runs where the restriction is met. Another is to have the environment's state include the time, and to attach a delivery time to each message in its buffer. Thus, when a process p sends the message $\mathbf{m}$ to q, the effect of this action is that the tuple $(p, \mathbf{m}, q, T')$ is inserted into the message buffer, where T' is the time the message will be delivered, chosen (nondeterministically) to be consistent with the assumptions about message delivery. Still other approaches are possible.

We have not been specific here about exactly how the state of a process changes as a result of sending or receiving messages. A common choice made in the literature is to assume that the process' state contains a complete record of all messages sent and delivered. Of course, this choice assumes an unbounded number of possible states in general, so is not always realistic. ∎

Example 2.2: In the previous example we implicitly assumed that processes were always enabled, so that whenever a process tried to send a message, the message was actually sent. It is often convenient to assume that processes are not always enabled, but rather are scheduled by a scheduler.

We can model the effect of the scheduler by augmenting the set of actions that the environment can perform to include actions of the form "processes in I are not scheduled", where I is a subset of the set of processes. If process i sends a message at the same time that the environment performs a "processes in I not scheduled" action, and $i \in I$, then the action is disabled (the message is not added to the message buffer, nor is process i's state changed to record the fact that the message is sent). Alternatively, we can assume that the environment's state includes a tuple $(x_1, \ldots, x_n)$ such that $x_i = 1$ if i is currently enabled and 0 otherwise. If i tries to perform a send action in a global state where it is not enabled, then the action has no effect. Clearly these two ways of modelling a scheduler are essentially equivalent.

One often wants to capture various fairness properties of a scheduler, such as the fact that a process is scheduled infinitely often. This is best done by restricting the set of runs of the system to ones where the appropriate fairness property holds. ∎

Example 2.3: Consider a CRCW PRAM (concurrent-read concurrent-write parallel random access machine) [FW78]. In this case a system consists of n processes together with an m-cell shared memory. Computation proceeds in synchronous rounds. Each computation step consists of three phases, each of which takes one round. In the first phase, every process may read one memory cell. In the second phase, every process may perform local computation. In the third phase, every process may attempt to write into a cell of shared memory. Any number of processes may attempt to simultaneously read or write from the same memory cell. There are a number of mechanisms for resolving write conflicts that appear in the literature. For example, in the MINIMUM model of [Gol82], priority is given to the process of lowest index; in the ARBITRARY model of [Vis83], an arbitrary process succeeds.

Once we fix a mechanism for resolving write conflicts, it is straightforward to model this situation in our framework. The shared memory is the environment component of the global state. We assume that each process' state includes a special *read* variable r. During the read phase, a process can perform only the null action Λ (we always use Λ to denote the special null action), or an action of the form $\mathit{read}(i)$, $i = 1, \ldots, m$, where $\mathit{read}(i)$ means that the value of the local read variable r should be set equal to the contents of cell i of shared memory. The environment performs the Λ action at the read phase (and in every other phase). Since read actions do not interfere with each other, the effect of performing a tuple $(\Lambda, \alpha_1, \ldots, \alpha_n)$ of actions, where α_j is either $\mathit{read}(i)$ or Λ, is simply the result of performing each of these actions separately, in any order. Similarly, a local action performed by process i changes just its local state, with no effect on any other local states. Again, there is no interference between the local actions performed by the processes in the computation phase.

In the write phase, a process can perform either a Λ action or one of the form $\mathit{write}(i, v)$, $i = 1, \ldots, m$. If, for a fixed value of i, only one process performs a $\mathit{write}(i, v)$ action, the result is that the value v is written into cell i in the environment; the local state of the process performing the action changes to record the fact that a write was attempted. If more than one process performs a $\mathit{write}(i, v)$ action, then the result depends on how we choose to resolve write conflicts. For example, in the MINIMUM model, the resulting value is that written by the process of lowest index. In the ARBITRARY model, the result of a write conflict is nondeterministic. Note that a process will not know whether it has succeeded after a *write* action in the ARBITRARY model. This is reflected in the fact that its state changes in the same way whether or not the write succeeds. ∎

This example should already indicate the flexibility of this formalism; it also serves to point out that the state transformer associated with a joint action cannot necessarily be computed by just somehow composing the effects of each of the individual actions.

Example 2.4: In the previous example, reading and writing were viewed as atomic, taking place in one round. We could easily modify this example to allow non-atomic reads and writes. The intuition here is that although we may want to think at a high level of the reads and writes as taking one unit of time, they may in fact be implemented by a sequence of lower level actions, and thus take place over a period of time. This means we will have to describe what happens if a read starts during one write and finishes after that write (possibly after several other writes have completed and during yet another write). Again, a number of choices are possible (cf. [Lam85]). We describe one here.

The basic idea is quite simple: the effect of a *read* action is now to indicate that the process has begun trying to read. The environment decides when the read is complete (by performing a *read.ended* action). Similar comments hold in the case of writing. Suppose for simplicity that we are trying to model an n-reader, 1-writer register. This means that exactly one process can write to that register, and n processes can try to read it. For definiteness, let us assume that we have $n+1$ processes; processes $1, \ldots, n$ are the only ones that can read the register and $n+1$ is the only process that can write into it. We take the environment state now to consist not only of the value of the register, but also of a description of which processes are currently trying to read or write into the register, the value that is currently being written (if any), and the value currently in the register. Thus we can view the environment's state as an $(n+3)$-tuple $(x_1, \ldots, x_{n+3})$. We take x_i, $i = 1, \ldots, n$, to be 1 if process i is currently trying to read, and 0 otherwise. Similarly, x_{n+1} is 1 if process $n+1$ is currently trying to write, and 0 otherwise. We take x_{n+2} to be the value that $n+1$ is currently trying to write if process $n+1$ is trying to write. Finally, x_{n+3} is the current value of the register. Similarly, we assume that process i, $i = 1, \ldots, n$, has a special variable *reading* that is 1 if process i is trying to read, and 0 otherwise. Similarly process $n+1$ has a *writing* variable.

We assume that a process can perform a *read* action only if its *reading* variable is set to 0 (i.e., it cannot start reading while it has another read in progress). When process i, $i = 1, \ldots, n$, performs a *read* action, its effect is simply to set the x_i component to 1 and to set the *reading* variable in its local state to 1. Thus the fact that it is reading is recorded in both its state and the environment's state. Similarly, when process $n+1$ performs a $write(v)$ action, which it can only do if its *writing* variable is 0, its effect is simply to set x_{n+1} to 1 and x_{n+2} to v and to set its *writing* variable to 1. Since reading and writing actions are not assumed to be atomic, they can go on for a number of steps. The environment can now perform actions which we call $read.ended(i)$, $i = 1, \ldots, n$, and *write.ended*. As the names suggest, these actions signal that a read (resp. write) action has ended. The action $read.ended(i)$ can be performed only if i is currently trying to read the register, i.e., if $x_i = 1$. By recording in the environment state the fact that i is reading, we allow the environment's actions to depend only on its state. Had we not done this, the environment's actions would also have to depend on the state of the processes. The effect of $read.ended(i)$ is to set x_i to 0, set i's *reading* variable to 0, and set i's read variable r to x_{n+3}, the current value of the register. Similarly, *write.ended*, which can only be performed if $x_{n+1} = 1$, sets both x_{n+1} and process $n+1$'s *writing* variable to 0, and sets x_{n+3} to x_{n+2}. ∎

It should be clear by now that many naturally-occurring systems can be captured in this framework in a straightforward way. We remark that not all aspects of systems behavior can be defined in terms of runs. In particular, the "branching behavior" of programs cannot be defined (although it can be defined, for example, in the framework of CCS [Mil80]). The branching behavior of a process becomes visible only when it is composed with other processes. Since our concern in applying the tools of knowledge is usually in analyzing particular protocols, rather than composing them, this branching behavior will not be of great concern to us.

3 Incorporating knowledge

It is easy to incorporate knowledge into our framework. As we mentioned in the introduction, the intuition we want to capture is that a process knows a given fact at a certain point in a system if that fact is true at all other points in the system where the process has the same local state. To make this precise, suppose we have a set Φ of primitive formulas, which we can think of as describing basic facts about the system. These might be such facts as "the value of the variable x is 0", "process 1's initial input was 17", "process 3 sends the message **m** at round 5 of this run", or "the system is deadlocked". In practice, basic facts depend only on the global state, although we do not make this a requirement (so that we allow a fact such as "the protocol eventually terminates" to be a basic fact, although its truth might depend on a future global state). In fact, in many cases a basic fact p will be *local* to a particular process i, so that the truth of p depends only on the local state of i.

Starting with the basic facts in Φ, we can extend the language to have formulas that express conjunctions, negations, and statements about knowledge. Thus, if φ and ψ are formulas, then so are $\varphi \wedge \psi$, $\neg\psi$, and $K_i\psi$ (read "process i knows ψ"). In order to assign truth values to these formulas, we need to first assign truth values to the basic facts in Φ.

Definition: An *interpreted system* $\mathcal{I}$ consists of a pair $(\mathcal{R}, \pi)$, where $\mathcal{R}$ is a system and π assigns truth values to the basic facts at each point in $\mathcal{R}$, so that for every $p \in \Phi$ and point (r, m) in $\mathcal{R}$, we have $\pi(r, m)(p) \in \{\mathbf{true}, \mathbf{false}\}$. We say that the point (r, m) is in interpreted system $\mathcal{I} = (\mathcal{R}, \pi)$ if $r \in \mathcal{R}$. ∎

Given an interpreted system $\mathcal{I} = (\mathcal{R}, \pi)$ and a point (r, m) in $\mathcal{I}$, we define a satisfiability relation $\models$ between the tuple $(\mathcal{I}, r, m)$ and a formula φ. For a basic fact $p \in \Phi$, we have

$$(\mathcal{I}, r, m) \models p \quad \text{iff} \quad \pi(r, m)(p) = \mathbf{true}.$$

We extend the $\models$ relation to conjunctions and negations in the obvious way:

$$\begin{array}{rcl} (\mathcal{I}, r, m) \models \neg\varphi & \text{iff} & (\mathcal{I}, r, m) \not\models \varphi \\ (\mathcal{I}, r, m) \models \varphi \wedge \psi & \text{iff} & (\mathcal{I}, r, m) \models \varphi \text{ and } (\mathcal{I}, r, m) \models \psi. \end{array}$$

In order to capture the intuition described above for formulas involving knowledge, define two points (r, m) and (r', m') to be *indistinguishable to* i, written $(r, m) \sim_i (r', m')$, if $r_i(m) = r'_i(m')$. Thus (r, m) and (r', m') are indistinguishable to i if i has the same local state at both of these points. Now define

$$(\mathcal{I}, r, m) \models K_i\varphi \quad \text{iff} \quad (\mathcal{I}, r', m') \models \varphi \text{ for all } r' \text{ and } m' \text{ such that } (r, m) \sim_i (r', m').$$

This interpretation of knowledge is well known to satisfy the axioms of the modal logic S5. In particular, it satisfies the axioms:

- $K_i\varphi \Rightarrow \varphi$
- $(K_i\varphi \wedge K_i(\varphi \Rightarrow \psi)) \Rightarrow K_i\psi$
- $K_i\varphi \Rightarrow K_iK_i\varphi$

- $\neg K_i\varphi \Rightarrow K_i\neg K_i\varphi$,

together with the rule of inference:

- From φ infer $K_i\varphi$

The first of these axioms says that a process knows only true facts. The next one says that a process' knowledge is closed under logical implication. In combination with the rule of inference, which says that processes know all *valid* formulas (i.e., formulas that are true at every point), this says that we can view processes as "perfect reasoners". Although this property may be inappropriate for analyzing the notion of knowledge as applied to humans, recall that we are considering here an external notion of knowledge, one ascribed by the system designer to the processes. We do not assume that the processes compute their knowledge in any way. The last two axioms are axioms of introspection. They say that a process knows what it knows and knows what it does not know. It can be shown that these axioms and inference rule, together with the axioms and inference rules of propositional logic, give a complete axiomatization for the logic (see [HM92] for a discussion and proof).

We can easily extend this logic further to capture the important notion of *common knowledge* (see [HM90] for further discussion and applications to distributed systems). Intuitively, a group G has common knowledge of a fact φ if everyone in G knows φ, everyone in G knows that everyone in G knows φ, etc. In order to deal with this, we add two further operators to the logic, E_G and C_G, for each subgroup G of processes, read "everyone in the group G knows φ" and "φ is common knowledge among the group G", respectively.

$(M, s) \models E_G\varphi$ iff $(M, s) \models K_i\varphi$ for all $i \in G$

$(M, s) \models C_G\varphi$ iff $(M, s) \models E_G^k\varphi$ for all $k \geq 1$, where $E_G^1\varphi$ is an abbreviation for $E_G\varphi$, and $E_G^{k+1}\varphi$ is an abbreviation for $E_G E_G^k\varphi$.

It is well known (again, see [HM92]) that we can get a complete axiomatization for this extended language by adding the axioms:

- $E_G\varphi \equiv \bigwedge_{i \in G} K_i\varphi$
- $(C_G\varphi \wedge C_G(\varphi \Rightarrow \psi)) \Rightarrow C_G\psi$
- $C_G\varphi \equiv E_G(\varphi \wedge C_G\varphi)$,

together with the rule of inference:

- From $\varphi \Rightarrow E_G\varphi$ infer $\varphi \Rightarrow C_G\varphi$.

The first axiom just describes the semantics of the E_G operator, while the second corresponds to the analogous property for knowledge. The third axiom (called the *fixed point axiom*) captures the fact that $C_G\varphi$ is a solution to the fixed point equation $X \equiv E_G(\varphi \wedge X)$. (Actually, in a precise sense it is the greatest such solution; cf. [HM90].) The rule of inference is called the induction rule, because using the fact that $\varphi \Rightarrow E_G\varphi$ is valid, we can show by induction on k that $\varphi \Rightarrow E_G^k\varphi$ is valid for all $k \geq 1$.

We can further extend the language so that we can talk about time, by adding standard temporal operators like $\Box$ ("always"), $\Diamond$ ("eventually"), and U ("until"). This allows us to make statements like "process 3 will eventually know the value of variable x." Doing this gives us quite a rich language for reasoning about knowledge and time. We remark that in general, the temporal operators will be used for reasoning about events that happen along a single run (there is no deadlock, eventually the transaction completes, etc.), while the knowledge operators will be used for reasoning about events that might be happening on other runs, which could be the real run, as far as a given process knows.

If we reason about knowledge and time, we might want to make some assumptions about the relationship between knowledge and time. We discuss two typical assumptions here, referring the reader to [HV89] for more details (as well as a discussion of the impact of these assumptions on the complexity of the validity problem).

A (completely) synchronous system $\mathcal{R}$ is one where, intuitively, there is a global clock and the clock time is part of each process' state. Thus, all processes "know" the time. Formally, $\mathcal{R}$ is a *synchronous system* if for all processes i and points $(r, m), (r', m')$ in $\mathcal{R}$, if $(r, m) \sim_i (r', m')$, then $m = m'$. We say that an interpreted system $\mathcal{I} = (\mathcal{R}, \pi)$ is synchronous if $\mathcal{R}$ is synchronous. Note that a system is synchronous exactly if a process can always distinguish points in the present from points in the future.

We say that processes do not forget if, intuitively, their set of possibilities always stays the same or decreases over time (this notion has also been called *unbounded memory* [HV86] or *cumulative knowledge* [FHV92, Moo85]). To make this precise, we define *process i's state sequence at the point* (r, m) to be the sequence of local states it has gone through in run r up to time m, without consecutive repetitions. Thus, if from time 0 through time 4 in run r process i has gone through the sequence $\langle s, s, s', s, s\rangle$ of local states, then its state sequence at $(r, 4)$ is $\langle s, s', s\rangle$. We say that *process i does not forget in system* $\mathcal{R}$ if at all points (r, m) and (r', m') in $\mathcal{R}$, if $(r, m) \sim_i (r', m')$, then process i has the same state sequence at both (r, m) and (r', m'). Thus process i does not forget if it "remembers" its state sequence. It is easy to see that no forgetting requires an unbounded number of local states in general (one for each distinct state sequence). A typical situation where we obtain no forgetting is if a process records its complete message history, as discussed in Example 2.1. However, as we pointed out, this assumption is often unreasonable in practice.

4 Protocols

Processes usually perform actions according to some *protocol* (or *algorithm*, or *strategy*; we tend to use the words interchangeably). Intuitively, a *protocol for process i* is a description of what actions process i takes as a function of its local state. To make this precise, we fix a set A_i of *actions* for process i, and define a protocol over state space L_i to be a function (possibly probabilistic) from L_i to nonempty sets of actions in A_i. The fact that a protocol maps a local state into a set of actions is used to capture the possible nondeterminism of the protocol. As we shall see, at any step only one of the possible actions of the protocol is actually performed. Of course, a deterministic protocol maps states to singleton sets of actions. For now we leave the set A_i unspecified, but in typical applications it consists of a small set of basic actions such as reading a data element, writing a value, sending a message, or making a move in a game.

Just as it is useful to view the environment as performing an action, it is also useful to view the environment as running a protocol. We can use the environment's protocol to capture the possibility that messages are lost or that messages may be delivered out of order; input from the outside world can be modelled by messages from the environment. Thus, we fix a set A_e of actions for the environment, and define a protocol for the environment to be a function from L_e to nonempty sets of actions in A_e.

We remark that our notion of protocol is quite general. For example, we do not constrain the function defining the protocol to be computable, although we could easily do so. But note that in contrast to, for example, [FI86], we require a protocol to be a function from *local* states to sets of actions, rather than a function on global states. It is crucial to most of our knowledge-based analyses that what a process does can depend only on its local state, and not on the whole global state.

We define a *joint protocol* P to be a tuple $(P_e, P_1, \ldots, P_n)$ consisting of a protocol P_e for the environment, and protocols P_i, $i = 1, \ldots, n$ for each of the processes. When analyzing a protocol, it is often convenient to associate with it a system, which intuitively consists of the set of runs of the protocol. In order to associate a set of runs with the joint protocol $P = (P_e, P_1, \ldots, P_n)$, where $P_e\colon L_e \rightarrow 2^{A_e} - \emptyset$ and $P_i\colon L_i \rightarrow 2^{A_i} - \emptyset$, $i = 1, \ldots, n$, fix a set $\mathcal{G} \subseteq L_e \times L_1 \times \cdots \times L_n$ of global states, a set $\mathcal{G}_0 \subseteq \mathcal{G}$ of *initial states*, and a *transition function* τ that associates with every joint action $(a_e, a_1, \ldots, a_n) \in A_e \times A_1 \times \ldots \times A_n$ a global state transformer $\tau(a_e, a_1, \ldots, a_n)$, i.e., a function from $\mathcal{G}$ to $\mathcal{G}$. We say that a run r is *consistent* with the joint protocol P if

1. $r(0) \in \mathcal{G}_0$ (so $r(0)$ is a legal initial state).
2. For all $m \geq 0$, if $r(m) = (s_e, s_1, \ldots, s_n)$, then there is a joint action $(a_e, a_1, \ldots, a_n) \in P_e(s_e) \times P_1(s_1) \times \cdots \times P_n(s_n)$ such that $r(m+1) = \tau(a_e, a_1, \ldots, a_n)(r(m))$ (so $r(m+1)$ is the result of transforming $r(m)$ by a joint action that could have been performed from $r(m)$ according to P).

We use $\mathcal{R}(P)$ to denote the set of all runs consistent with the joint protocol P. It is usually the system $\mathcal{R}(P)$ we refer to when we speak of "the runs of protocol P." However, if we are given some global constraint on the system (such as a fairness constraint), then we would consider the subset of $\mathcal{R}(P)$ satisfying that constraint.

5 Knowledge-based protocols

While we have argued that our notion of protocol is sufficiently general to include all algorithms that can be written in any programming language currently in use, it cannot be used to give a high-level system-independent description of the relationship between knowledge and action. This issue is perhaps best understood by considering the type of problems that one sees in puzzle books (for example, [Smu78]), where a man meets someone on the road who is known to be either a Truth-teller (who always tells the truth) or a Liar (who always lies). The problem is to determine which he is by asking some questions. The rules of the game are that in response to a question of the form "Is φ the case?", the Truth-teller answers "Yes" if φ is true and "No" if φ is false. Similarly, the Liar answers "Yes" if φ is false and "No" if φ is true.

A closer inspection shows that these rules are not well specified. Suppose the Truth-teller is asked about φ and he doesn't know whether φ is true or false. Then he cannot follow this protocol appropriately (unless he manages to guess right). It seems clear that we should reinterpret these rules so that, when asked about φ, the Truth-teller responds "Yes" if he *knows* that φ is true, "No" if he knows that φ is false, and "I don't know" otherwise. The Liar is similarly constrained.[3]

With this reinterpretation, both the Truth-teller and Liar can be viewed as running protocols with explicit tests for knowledge. For example, if we take the Truth-teller to be process 1, then we can view his protocol P_1 as a function that, in a state where the question φ is asked, has the form

if $K_1\varphi$ **then** say "`Yes`"
else if $K_1\neg\varphi$ **then** say "`No`"
else say "`I don't know`".

This cannot be viewed as a standard protocol, since the truth value of the test $K_1\varphi$ cannot be determined by looking at process 1's local state in isolation. Its truth depends on the truth of φ at other points (all the ones with global states that process 1 cannot distinguish from the current global state). Thus, whereas a standard protocol for process i is a function from i's local states to actions, we can view a knowledge-based protocol for process i as a program that contains statements of the form "**if** $K_i\varphi_1$ **then** a **else if** $K_i\varphi_2$ **then** a' ...", where a and a' are actions in A_i. We then need to have an interpreted system to decide whether the tests are true.

Although it is useful to think of a knowledge-based protocol as a function from states to **if-then-else** statements with tests for knowledge, it is technically more convenient to view it as a function from a pair consisting of a local state *and an interpreted system* to actions. (We essentially took the former approach in [HF85], where the notion of knowledge-based protocol was introduced; the approach we take here to the definition of knowledge-based protocols was taken in [NT93].) Formally, fix a set $\mathcal{G} \subseteq L_e \times L_1 \times \cdots \times L_n$ of global states and a set A_i of actions for process i, and let $INT(\mathcal{G})$ be the set of all interpreted systems $\mathcal{I} = (\mathcal{R}, \pi)$ such that for every run $r \in \mathcal{R}$, all the global states in r are in $\mathcal{G}$. Then a knowledge-based protocol for process i is a function P_i from $L_i \times INT(\mathcal{G})$ to nonempty sets of actions in A_i. For example, suppose the Truth-teller is in state ℓ after being asked the question "Is φ the case?". Then we have

$$P_1(\ell, \mathcal{I}) = \begin{cases} \text{say "}\texttt{Yes}\text{"} & \text{if } (\mathcal{I}, r, m) \models \varphi \text{ for all points } (r,m) \text{ where } r_1(m) = \ell \\ \text{say "}\texttt{No}\text{"} & \text{if } (\mathcal{I}, r, m) \models \neg\varphi \text{ for all points } (r,m) \text{ where } r_1(m) = \ell \\ \text{say "}\texttt{I don't know}\text{"} & \text{otherwise.} \end{cases}$$

Note that the only difference between the formal definition of knowledge-based protocols and standard protocols is that a knowledge-based protocol takes an interpreted system as one of its arguments. Once we fix an interpreted system $\mathcal{I}$, then a knowledge-based protocol reduces to a standard protocol. Thus we can view knowledge-based protocols as functions from

[3] It is not clear exactly how the Liar should respond if he doesn't know whether φ is true or false. In the solutions to such Truth-teller/Liar puzzles, the questions are always carefully chosen so that the person answering knows the answer; thus, this issue does not arise.

interpreted systems to standard protocols. A standard protocol can be viewed as a special case of a knowledge-based protocol where the function is independent of the interpreted system.

Like our definitions for standard protocols, we can define a joint knowledge-based protocol to be a tuple $(P_e, P_1, \ldots, P_n)$ of knowledge-based protocols, all defined with respect to the same set $\mathcal{G}$ (i.e., there is a set $\mathcal{G} \subseteq L_e \times L_1 \times \cdots \times L_n$ such that P_e is a function from $L_e \times INT(\mathcal{G})$ to nonempty sets of actions in A_e and P_i is a function from $L_i \times INT(\mathcal{G})$ to nonempty sets of actions in A_i, $i = 1, \ldots, n$). We would also like to define the notion of a run being consistent with a knowledge-based protocol, in analogy to our definition for standard protocols. In order to do this, we must also specify an interpreted system, since a knowledge-based protocol takes an interpreted system as one of its arguments. Given a joint knowledge-based protocol P as above, a set $\mathcal{G}_0 \subseteq \mathcal{G}$ of initial states, a transition function τ, and an interpreted system $\mathcal{I}$, we define a run r to be *consistent with* P *relative to* $\mathcal{I}$ just as we defined the notion of a run being consistent with a standard protocol P, except that now the joint action $(a_e, a_1, \ldots, a_n)$ in clause (2) is in $P_e(s_e, \mathcal{I}) \times P_1(s_1, \mathcal{I}) \times \cdots \times P_n(s_n, \mathcal{I})$ rather than $P_e(s_e) \times P_1(s_1) \times \cdots \times P_n(s_n)$. An interpreted system $\mathcal{I} = (\mathcal{R}, \pi)$ is *consistent with knowledge-based protocol* P if every run $r \in \mathcal{R}$ is consistent with P relative to $\mathcal{I}$.

This completes our description of standard protocols and knowledge-based protocols. A detailed example of how the semantics of both standard and knowledge-based protocols can be specified in this framework is given in [HZ92].

The definition of an interpreted system being consistent with a knowledge-based protocol has some inherent circularity. This can perhaps be better seen if we define $Con(P, (\mathcal{R}, \pi))$ to be the set of runs consistent with knowledge-based protocol P relative to the interpreted system $\mathcal{I} = (\mathcal{R}, \pi)$. Then $\mathcal{I}$ is consistent with P if $\mathcal{R} \subseteq Con(P, (\mathcal{R}, \pi))$. Given this circularity, it may not seem too surprising that, in contrast with the situation for standard protocols, we cannot talk about *the* interpreted system consistent with a knowledge-based protocol. There may not be any interpreted systems consistent with a given knowledge-based protocol; more often, there will be many interpreted systems consistent with a given knowledge-based protocol.

This contrast between standard protocols and knowledge-based protocols is quite relevant when proving that these protocols are correct or satisfy certain specifications. In order to prove that a standard protocol P satisfies certain specifications, we typically prove that these specifications hold for all the runs in $\mathcal{R}(P)$ (or perhaps all the runs satisfying some constraint such as fairness). In order to prove that a knowledge-based protocol P satisfies certain specifications, what we prove is that these specifications hold for *all* the interpreted systems consistent with P (cf. the proofs in [HZ92]).

We would often like to think of a knowledge-based protocol as specifying a unique set of runs. To understand what may prevent us from doing so, fix a knowledge-based protocol P, a set $\mathcal{G}$ of global states, a subset $\mathcal{G}_0 \subseteq \mathcal{G}$, and a transition function τ. The obvious way of constructing a set of runs is to proceed by constructing all prefixes of consistent runs of length m, by induction on m. Suppose we have managed to construct the prefixes of length m. In order to know what action to perform next at a certain point (r, m), we must know the result of a test of the form $K_i\varphi$. However, this result depends on the truth value of φ at other points where i has the same state as the current point. Thus, there are two reasons why we cannot determine the truth value of $K_i\varphi$. The first is that there may be points in the future, that we have not yet constructed, where i has the same local state as it does at (r, m). And even if i

can always distinguish points in the present from points in the future, it may be the case that the truth of φ itself depends on the future (for example, φ may be of the form $\Diamond p$, so it is true only if p eventually holds).

Recall that in a synchronous systems, processes have access to a global clock, so they can always distinguish points in the present from points in the future. Thus, we can avoid the first problem by restricting attention to synchronous systems. If we also restrict attention to knowledge-based protocols whose actions depend only on the past, we can avoid the second problem. Once we do this, we still do not get a *unique* set of runs corresponding to the knowledge-based protocol, but we do get in some sense a *canonical* set of runs, which we can think of as *the* set of runs determined by the knowledge based protocol. We now make this intuition precise.

Suppose we are given a joint knowledge-based protocol $P = (P_e, P_1, \ldots, P_n)$, defined with respect to a set $\mathcal{G}$ of global states and a transition function τ. Since we now must consider *interpreted systems* rather than just systems, we must also have a function π that assigns truth values to the primitive propositions in our language. We assume (as is the case in most real-world protocols) that the truth value of a primitive proposition depends only on the global state. Thus, we assume that we have a function σ such that for each global state $g \in \mathcal{G}$ and each primitive proposition $p \in \Phi$, we have $\sigma(g)(p) \in \{\mathbf{true}, \mathbf{false}\}$. Given an interpreted system $\mathcal{I} = (\mathcal{R}, \pi)$, we say that π is *based on* σ if $\pi(r, m) = \sigma(r(m))$ for all points (r, m) in $\mathcal{R}$. For the remainder of this section, we restrict our attention to synchronous interpreted systems $\mathcal{I} = (\mathcal{R}, \pi)$ in $INT(\mathcal{G})$ where π is based on σ.

Since we restrict attention to synchronous systems in any case, we assume for simplicity that process i's local state is of the form $(m, \ldots)$, where the first component is the time; similarly for the environment's local state. Thus, the global state at a point (r, m) of a synchronous system is of the form $((m, \ldots), (m, \ldots), \ldots, (m, \ldots))$. We assume that the global states in $\mathcal{G}$ are of this form. We further assume that our transition function τ has the property that an action always result in an increase in time by one unit. Thus, for any joint action $(a_1, \ldots, a_n)$, we have $\tau(a_1, \ldots, a_n)((m, \ldots), \ldots, (m, \ldots)) = ((m + 1, \ldots), \ldots, (m + 1, \ldots))$.

Finally, as mentioned above, in order to construct a unique system consistent with P, we must also assume that the actions in P depend only on the past. To make this precise, given a system $\mathcal{R}$, let $\mathcal{R}^m$ consist of all the prefixes of runs in $\mathcal{R}$ of length m. We say that two interpreted systems $\mathcal{I} = (\mathcal{R}, \pi)$ and $\mathcal{I}' = (\mathcal{R}', \pi')$ *agree up to time* m if $\mathcal{R}^m = (\mathcal{R}')^m$. We say that P_i's actions *depend only on the past* if, for all times m, given two synchronous interpreted systems $\mathcal{I}, \mathcal{I}'$ that agree up to time m and points (r, m) in $\mathcal{I}$ and (r', m) in $\mathcal{I}'$ such that $r(m) = r'(m) = s$, then $P_i(s, \mathcal{I}) = P_i(s, \mathcal{I}')$. We can similarly define what it means for the environment protocol P_e's actions to depend only on the past. We say that the actions of $P = (P_e, P_1, \ldots, P_n)$ depend only on the past if the actions of each of its components do. Note that if we had viewed a knowledge-based protocol as a function that maps local states to objects such as **if** $K_i\varphi$ **then** $\ldots$, then a knowledge-based protocol where the tests φ were restricted to involve only *past-time temporal operators* (cf. [LPZ85]), whose truth at a point (r, m) depends only on the prefix of the run up to time m, would in fact be a protocol whose actions depended only on the past.

Theorem 5.1: *Fix $\mathcal{G}$, σ, and τ as above, and suppose that the actions of P depend only on the past. Let $\mathcal{G}_0 \subseteq \mathcal{G}$ be such that all the global states in $\mathcal{G}_0$ are of the form $((0, \ldots), \ldots, (0, \ldots))$.*

Then there is a synchronous interpreted system $\mathcal{I} = (\mathcal{R}, \pi)$ consistent with P, such that the initial global states of the runs in $\mathcal{R}$ are precisely the elements of $\mathcal{G}_0$ and π is based on σ, and such that $\mathcal{I}$ is a maximal interpreted system with this property (i.e., if $\mathcal{I}' = (\mathcal{R}', \pi')$ is a synchronous interpreted system such that the initial global states of the runs in $\mathcal{R}'$ are precisely the elements of $\mathcal{G}_0$ and π' is based on σ, then it is not the case that $\mathcal{R}$ is a proper subset of $\mathcal{R}'$).

Proof: We just sketch the construction of $\mathcal{I}$ here. We first construct all the prefixes of runs in $\mathcal{R}$ of length 0. These are the functions ρ with domain $\{0\}$ such that $\rho(0) \in \mathcal{G}_0$. Suppose we have constructed all the prefixes of length m; call this set of prefixes $\mathcal{R}^m$. Let $(\mathcal{R}^m)^*$ consist of all runs extending $\mathcal{R}^m$ whose global states at all times m' have the form $((m', \ldots), \ldots, (m', \ldots))$. Let $\mathcal{I}^m = ((\mathcal{R}^m)^*, \pi')$, where π' is determined by σ. Let $\mathcal{R}^{m+1}$ consist of all those finite runs ρ' of length $m+1$ such that there is a prefix ρ in $(\mathcal{R}^m)^*$ of length m and a joint action $(a_e, a_1, \ldots, a_n)$ satisfying

1. $\rho \preceq \rho'$,
2. if $\rho(m) = (s_e, s_1, \ldots, s_n)$, then $(a_e, a_1, \ldots, a_n) \in P(s_e, \mathcal{I}^m) \times P(s_1, \mathcal{I}^m) \times \ldots \times P(s_n, \mathcal{I}^m)$,
3. $\rho'(m+1) = \tau((a_e, a_1, \ldots, a_n))(\rho(m))$ (i.e., $\rho'(m+1)$ is the result of performing the joint action $(a_e, a_1, \ldots, a_n)$ in $\rho(m)$).

It is easy to see that every prefix in $\mathcal{R}^m$ has some extension in $\mathcal{R}^{m+1}$. Moreover, our assumptions on τ guarantee that all the global states in $\mathcal{R}^{m+1}$ have the right form, so that we stay within the framework of synchronous systems.

Let $\mathcal{R}$ consist of all runs r such that for all m, the prefix of r of length m is in $\mathcal{R}^m$. Let π be the truth assignment of $\mathcal{R}$ determined by σ. Using the fact that (by construction) $\mathcal{I}$ and $\mathcal{I}^m$ agree up to time m and that P's actions depend only on the past, it is easy to check that $\mathcal{I} = (\mathcal{R}, \pi)$ is consistent with P. Roughly speaking, we have constructed $\mathcal{I}$ so that at every step $m+1$, we have all possible extensions of runs at step m that allow us to remain consistent with P. From this it easily follows that $\mathcal{I}$ is maximal in the sense given by the statement of the theorem. ∎

It may seem that we require quite a few assumptions for this construction to go through, but systems meeting all these assumptions arise regularly in the literature. A typical example is provided in the next section.

6 An example: the "cheating husbands" puzzle

In this section we examine the "cheating husbands" puzzle and some of its variants, dealt with at great length in [MDH86]. We show how it can be captured in our framework, and how viewing it as a knowledge-based protocol, it gets transformed into a number of different standard protocols, depending on the assumptions about the system. The cheating husbands puzzle is essentially isomorphic to the "muddy children" puzzle discussed in [HM90], so our analysis holds for the latter puzzle as well.

We begin by reviewing the essential elements of the puzzle. The following passage is taken from [MDH86]:

> It has always been common knowledge among the women of Mamajorca [that all of them are perfect reasoners], their queens are truthful, and that the women are obedient to the queens. It was also common knowledge that all women hear every shot fired in Mamajorca. Queen Henrietta I awoke one morning with a firm resolution to do away with the male infidelity problem in Mamajorca. She summoned all of the women heads of households to the town square, and read them the following statement:
>
>> There are (one or more) unfaithful husbands in our community. Although none of you knew before this gathering whether your own husband was unfaithful, each of you knows which of the other husbands are unfaithful. I forbid you to discuss the matter of your own husband's fidelity with anyone. However, should you discover that your husband is unfaithful, you must shoot him on the midnight of the day you find out about it.
>
> Thirty-nine silent nights went by, and on the fortieth night, shots were heard.

Of course, the puzzle is to explain how many cheating husbands there were, and why they were shot on the fortieth night.

We can in fact show that there must have been forty unfaithful husbands. We prove by induction that if there were exactly k unfaithful husbands, they would have been shot on the k^{th} night, and no earlier. Clearly if there were only one unfaithful husband, his wife, not knowing of any other unfaithful husbands, would realize as soon as she heard the queen say that there were some unfaithful husbands, that her own husband was unfaithful. Thus, if $k = 1$, the one unfaithful husband is shot on the first night. For the general case, suppose there are $k + 1$ unfaithful husbands. Their wives know of k unfaithful husbands (since they know of all unfaithful husbands besides their own). When no shots are heard on the k^{th} night, they realize that there could not have been exactly k unfaithful husbands, since if there were, by the induction hypothesis, they would have been shot on the k^{th} night. Thus each wife of an unfaithful husband can deduce on the $(k + 1)^{\text{st}}$ day that her husband is unfaithful, and so will shoot him on that night. She could not have deduced this fact any earlier, because she could not have distinguished before then the actual situation from the one where her husband was faithful and there really were only k unfaithful husbands.

As is pointed out in [HM90], there is a somewhat deeper puzzle here. In the story, there were actually forty unfaithful husbands. Thus it seems that the queen's initial statement that there were unfaithful husbands in the community is unnecessary. All the wives were already aware of this fact. Yet, as is shown in [HM90], none of the women would have been able to conclude anything about their own husbands' faithfulness without this apparently useless statement. It is clear that our proof by induction breaks down in the base case without the queen's statement, but this does not seem to be a very satisfactory explanation as to why the queen's statement is necessary. As is shown in both [HM90] and [MDH86], we can get a better understanding of what is going on here by doing an analysis of the state of knowledge of the women, and how it changes over time. We now do this in our formal framework.

It is clear that all of the women are following a very simple knowledge-based protocol, namely "For all days $k = 1, 2, 3, \ldots$, if you know that your husband is unfaithful, then shoot him at midnight; otherwise do nothing." This is a protocol that is being run in a synchronous system whose actions depend only on the past, so we can construct a canonical interpreted system consistent with this protocol, as discussed in the previous section. In order to understand the structure of this system, we first present a fairly complete informal discussion, and then sketch how it can be formalized.

Suppose there are n couples in the village. We number them $1, ..., n$. We can describe the situation in the village by an n-tuple of 0's and 1's of the form $(x_1, ..., x_n)$, where $x_i = 1$ if husband i is unfaithful, and $x_i = 0$ otherwise. Thus, if $n = 5$, then a tuple of the form $(1, 0, 1, 0, 0)$ would say that there are exactly two unfaithful husbands, that of woman number 1 and that of woman number 3. Suppose the actual situation is described by the tuple $(x_1, ..., x_n)$. Because wife number 1 knows of all the unfaithful husbands besides her own, she initially considers two situations possible: $(0, x_2, ..., x_n)$ and $(1, x_2, ..., x_n)$. Her husband may be faithful or may be unfaithful. Similarly, wife number 2 considers two situations possible: $(x_1, 0, x_3, ..., x_n)$ and $(x_1, 1, x_3, ..., x_n)$. Note that, in general, two tuples cannot be distinguished by woman i exactly iff they differ only in the i^{th} component. Thus, on day 0 (before the queen has spoken), we have 2^n possible initial situations, described by these n-tuples. We can also describe the indistinguishability relationship easily. Suppose we join two n-tuples by an edge (labelled i) if they are indistinguishable by i. Then it is easy to see that we have precisely an n-dimensional cube.[4]

What happens on day 1, after the queen has spoken? The queen said that there were some unfaithful husbands in the village. This eliminates the situation $(0, 0, ..., 0)$ from the picture. Thus we end up with a "truncated" cube, which is missing one vertex. Going back to our example with $n = 5$, if the initial situation were $(1, 0, 1, 0, 0)$, although everyone knew before the queen spoke that there were some unfaithful husbands, woman 1 considered the situation $(0, 0, 1, 0, 0)$ possible before the queen spoke. In that situation, woman 3 would consider $(0, 0, 0, 0, 0)$ possible. Thus, before the queen spoke, woman 1 thought it was possible that woman 3 thought it was possible that there were no cheating husbands. After the queen spoke, it was *common knowledge* that there were some cheating husbands. This is represented by the fact that the cube is truncated. Thus, the queen's initial statement does change the group's knowledge. Even though every woman knew there were some unfaithful husbands before the queen spoke, it wasn't common knowledge.

On day 2, when no shots were heard the night before, all the women can further truncate the cube. They can eliminate all vertices with exactly one 1. The reasoning parallels that which we did above. If the actual situation were described by, say, the tuple $(1, 0, ..., 0)$, then initially woman 1 would have considered two situations possible: $(1, 0, ..., 0)$ and $(0, 0, ..., 0)$. Since it is common knowledge that $(0, 0, ..., 0)$ is not possible, she would know that the situation is described by $(1, 0, ..., 0)$, so she would know that her husband was unfaithful. Since there were no shots, she could not know this. Thus, the situation cannot be $(1, 0, ..., 0)$. Similar reasoning allows all the women to eliminate every situation with exactly one 1. Since it is common knowledge that all of the women are perfect reasoners, on day 2 (before midnight), it is common knowledge that there are at least 2 unfaithful husbands.

Every day we can truncate the cube a little more. Similar reasoning to that above shows that we can eliminate all the vertices with exactly k 1's after midnight of day k. Thus, on day $k + 1$ (before midnight), it is common knowledge that there are at least $k + 1$ unfaithful

[4]This graphical interpretation of the situation as an n-cube was pointed out to us by Moshe Vardi.

husbands. (Notice that knowledge is changing here even in the absence of communication!) If the true situation is described by a tuple with exactly $k+1$ 1's, then on the $(k+1)^{\text{st}}$ day, those women with unfaithful husbands will know the exact situation, and consequently shoot their husbands on that night.

We now capture this situation formally in our framework. Our first step is to describe the possible local states. We take the environment's state to consist of a pair (m, x), where m is the day and x is a complete description of which husbands are faithful (a tuple of 0's and 1's, as described above). We take woman i's state to be a triple of the form (m, y, h), where again m is the day, y is a description of what woman i originally knows about which husbands are unfaithful, and h is a sequence of length m describing what woman i has heard on all the previous days. If x is a complete description of which husbands are faithful, then y is x^i, where x^i is just like x except that there is a $*$ in the i^{th} component, indicating that woman i does not know whether her own husband is unfaithful (although she does know about everyone else). The initial global states are thus of the form $((0, x), (0, x^1, \langle\rangle), \ldots, (0, x^n, \langle\rangle))$.

We assume that Queen Henrietta sends her message on day 0. For ease of exposition, we assume that she sends either the message described above, or no message at all. On later days we just append 1 or 0 to h, depending on whether or not there were shots the previous day. There are some obvious constraints on the global state: all the times must be the same (i.e., the first components of each woman's state and the environment's state must be the same); woman i's view of the situation must be the same as the true situation (as described in the environment's state), except with a $*$ in the i component; and all the women must hear the same thing (so that the h components are all the same). Note that this embodies the implicit assumptions that it is common knowledge that all the women can see and hear, and are paying attention. If some woman considered it possible that some other woman considered it possible ...that some woman was deaf or not paying attention, then there would be a global state in the model where the h components were different.

The only non-null action performed by the women is that of shooting; the only non-null action performed by the environment is that of possibly broadcasting the queen's message on day 0. Since we assume the queen is telling the truth, this message can be broadcast only if the environment's state $(0, x)$ is such that there is at least one 1 in the tuple x (so that there is at least one unfaithful husband).

As we remarked before, woman i is following the knowledge-based protocol "for day $k = 1, 2, 3 \ldots$, **if** K_i(husband i unfaithful) **then** shoot." The environment is running the protocol which (nondeterministically) either sends the queen's message or does nothing on day 0 (and does nothing on all later days). This captures the fact that, *a priori*, the women do not know whether the queen will send a message.

We can now construct the runs corresponding to this knowledge-based protocol by induction, as described in the previous section, using the ideas in the informal analysis above. A straightforward induction shows that at any day $k > 1$, we have precisely $2^n - 1$ prefixes of length k of runs where the queen sends a message, one corresponding to each of the initial states where at least one of the husbands is unfaithful, and a further 2^n prefixes of length k of runs where the queen does not send a message. Moreover, if we put an edge between two time k points if some process cannot distinguish them, then we get a "truncated cube" at the points corresponding to runs where at least k husbands are unfaithful and the queen does send a message. On this subset of runs, woman i knows that her husband is unfaithful (and therefore shoots him) at time k on those runs where there are exactly k unfaithful husbands, one of whom is woman i's. These observations allow us to extend from prefixes of length k to prefixes of length $k+1$. In this interpreted system, the knowledge-based protocol is equivalent to a simple standard protocol: "If you heard the queen's initial statement, your initial state has k 1's (i.e., if you initially knew of k unfaithful husbands), and there are no shots on the k^{th} night, then shoot your husband on the $(k+1)^{\text{st}}$ night; otherwise do nothing."

Several other variants of the cheating husband problem are considered in [MDH86]. For example, the next queen of Mamajorca introduces a mail system, and sends out to all her subjects an exact copy of her mother's message, as well as a letter describing the crucial property of the mail system, namely, that all letters are guaranteed to eventually arrive. In this setting, it is shown that if there is more than one unfaithful husband, then no husband will ever be shot. In our formal model, what is going on here is that although the initial situations now are the same as they were before, and each woman still follows the same knowledge-based protocol as before (once she gets the queen's message), the set of possible runs has changed because the environment's protocol has changed. It is still nondeterministic, but now not only is it possible that the queen sent no message, but, if she did send a message, there is nondeterminism in how long it takes to arrive (so that different women can append the message to the history component h on different days). In this system, it can be shown that the women can never deduce whether their husbands are unfaithful whenever there is more than one unfaithful husband.

In another variant considered in [MDH86], the mail system is improved so that all messages are guaranteed to arrive no later than one day after they are sent (i.e., either on the same day or on the next day). This fact is also made known to all the women by a letter. In our framework, this means that in any given run, the women all receive the queen's message within one day of each other. Moreover, the women are told to shoot their husbands on midnight of the day *after* they first know he is unfaithful. In this situation, it is shown that all the unfaithful husbands (and only the unfaithful husbands!) are shot, but the reasoning is much different from that in the first story. We do not go through the details here, but observe that although the wives execute essentially the same knowledge-based protocol here as in the case discussed above, the corresponding standard protocol becomes: "If your initial state has k 1's and there are no shots for the first $3k$ nights after you get the queen's message, then shoot your husband on the $(3k+2)^{\text{nd}}$ night; otherwise do nothing." The difference between this standard protocol and the one that was equivalent to the knowledge-based protocol in the original scenario is due to the difference in the environment's protocol. It is the environment's protocol that is being used to capture the different assumptions about the system. By using the high-level language of knowledge-based protocols, we can capture the intuition that the women are in some sense running the same protocol.

7 What it means for one protocol to implement another

It is often convenient when designing protocols to first design a joint protocol P that uses high-level constructs, then implement these constructs in a protocol P' using low-level commands. It is usually relatively straightforward to prove the correctness of P; one then proves the correctness of P' by showing that in a precise sense it is an implementation of P. This is particularly the case when starting with knowledge-based protocols (see [HZ92] for examples). Although system designers have good intuitions about when one protocol implements another, making this notion precise has not been so easy. Lamport gives a definition of what it means for one system to implement another in [Lam86], using the framework developed there. We now consider this question in our framework.

Since we identify a protocol with a system, it is clear that an implementation should be a function from one system to another. Thus, if $\mathcal{R}$ (resp. $\mathcal{R}'$) is the system corresponding to protocol P (resp. P'), the fact that P' implements P is captured by having a function h from $\mathcal{R}'$ to $\mathcal{R}$. However, our intuition about implementations will surely not be captured by simply having an arbitrary function from one system to another. An implementation is only interesting if it preserves certain relevant features of the runs (such as reads and writes). We make this notion precise below, but first we consider a number of other properties that we might want an implementation to have.

One condition we might impose is that not only do we have a function from runs to runs, but also from prefixes of runs to prefixes of runs. The intuition is that a prefix of a run of P' should map to a prefix of a run of P where the corresponding steps have been performed. Of course, we would expect that longer prefixes of runs of P' map to longer prefixes of runs of P. Thus we get the following definition.

Definition: A mapping h from (finite and infinite) runs to runs is *monotonic* if $\rho \preceq \rho'$ implies $h(\rho) \preceq h(\rho')$. *$\mathcal{R}'$ is a monotonic implementation of* $\mathcal{R}$ if there is a monotonic mapping $h\colon \mathrm{Pref}(\mathcal{R}') \rightarrow \mathrm{Pref}(\mathcal{R})$. ∎

We can impose an additional requirement; that of continuity. The intuition here is that a low-level protocol can take several steps to implement a step of a high-level protocol, but it eventually does so. Thus, if P' implements P, then a prefix of a run of P' should correspond to a prefix of a run of P where the corresponding high-level steps have been performed. By taking longer and longer prefixes of a run in P' we should be able to reconstruct the run of P that it implements. This leads us to the following definition.

Definition: A mapping h taking runs into runs is *continuous* if, given that $\rho_1 \preceq \rho_2 \preceq \cdots$ and $\cup_i \rho_i = r$, and that ρ_i, $i = 1, 2, \ldots$ and r are all in the domain of h, then $h(\rho_1) \preceq h(\rho_2) \preceq \ldots$ and $h(r) = \cup_i h(\rho_i)$. Note that continuity implies monotonicity. *$\mathcal{R}'$ is a continuous implementation of* $\mathcal{R}$ if there is a continuous mapping $h : \mathrm{Pref}(\mathcal{R}') \rightarrow \mathrm{Pref}(\mathcal{R})$. ∎

As Martín Abadi has pointed out to us, both of these requirements (monotonicity and continuity) on implementations may be too strong. For example, consider the simple situation where a message is sent by p to q and, at the high level, the environment decides how many steps it will take before delivering a message by tossing an infinite-sided coin, with faces labelled $1, 2, 3, \ldots, \infty$. The outcome of the coin toss determines when the message will be delivered; if the coin lands ∞ then the message is never delivered. One way to capture this by having the delivery time (the outcome of the coin toss) be part of the environment's initial state. Thus, we can take the system $\mathcal{R}$ to consist of all runs where p sends the message $\mathbf{m}$ to q at time 0, and this message is delivered at the time k specified by the environment's initial state. We can implement this by having the environment toss a two-sided coin at every step, and deliver the message when the coin lands heads the first time. This corresponds to a set $\mathcal{R}'$ of runs where the environment's state at time k is determined by whether it tossed heads or tails at the last coin toss. Unfortunately, there is no monotonic mapping from Pref($\mathcal{R}'$) to Pref($\mathcal{R}$). The problem is that for a prefix of a run in $\mathcal{R}'$ where the environment has tossed tails at every step, we do not know what prefix of a run in $\mathcal{R}$ to map it to. We cannot commit yet to delivering the message at a fixed time and still maintain monotonicity.

If we modify $\mathcal{R}$ so that the environment's state contains the outcome of the infinite-sided coin toss only after time 1 (and at time 0 the environment was in some special initial state), we can get a monotonic map. It is easy to check, however, that we can do this only by mapping a prefix of a run of $\mathcal{R}'$ where the environment tosses tails at every step to a length 0 prefix of a run of $\mathcal{R}$ (all length 0 prefixes are the same, so it does not matter which run we choose). But this map is not continuous. Consider the run of $\mathcal{R}'$ where the environment tosses tails at every step. All of its prefixes are mapped to a length 0 prefix of $\mathcal{R}$. Thus we do have monotonicity, but not continuity.

Despite this counterexample, it still seems to be the case that most examples of implementations that arise in practice are continuous (and hence also monotonic). Indeed, we usually expect even more of an implementation. We want certain properties of runs to be preserved, for example, what data elements are read or written onto a disk. The fact that a certain property holds at a certain point corresponds to a formula being true. Thus, in order to capture this intuition, we need to consider interpreted systems.

We say that the formula φ *depends only on the past* in interpreted system $\mathcal{I}$ if $(\mathcal{I}, r, m) \models \varphi$ and $r|_m = r'|_m$ implies $(\mathcal{I}, r', m) \models \varphi$. Intuitively, a formula depends only on the past if its truth at the point (r, m) depends only on the global states in r up to time m.[5] Formulas depending only on the past arise frequently in practice. Typical examples include "there were three reads and two writes up to this time" and "the message **m** was sent". Note that if a formula depends only on the past in interpreted system $\mathcal{I}$, then it makes sense to write $(\mathcal{I}, \rho, m) \models \varphi$, where ρ is a finite prefix of a run of length at least m. We can view this as an abbreviation of the statement $(\mathcal{I}, r, m) \models \varphi$, where r is any run extending ρ (it does not matter which one we take, since φ depends only on the past).

Suppose $\mathcal{I} = (\mathcal{R}, \pi)$ and $\mathcal{I}' = (\mathcal{R}', \pi')$ are interpreted systems, and Φ is a collection of formulas that depend only on the past in $\mathcal{I}$ and $\mathcal{I}'$. The reader should think of the formulas in Φ as describing the properties of interest in $\mathcal{I}$ and $\mathcal{I}'$.

Definition: *$\mathcal{I}'$ is a monotonic implementation of $\mathcal{I}$ with respect to Φ* if there is a monotonic function $h : \text{Pref}(\mathcal{R}') \to \text{Pref}(\mathcal{R})$ such that for all formulas $\varphi \in \Phi$ and $\rho \in \text{Pref}(\mathcal{R}')$, we have $(\mathcal{I}', \rho, |\rho|) \models \varphi$ iff $(\mathcal{I}, h(\rho), |h(\rho)|) \models \varphi$.[6] Similarly, *$\mathcal{I}'$ is a continuous implementation of $\mathcal{I}$ with respect to Φ* if there is a map h as above which is continuous.

These last definitions do seem to come close to the spirit of the notion of implementation as used in practice. In particular, in [HZ92] the correctness of a knowledge-based protocol for the *sequence transmission problem* (where a sender must transmit a sequence of data elements to a receiver over a potentially faulty channel) is proved; it is shown that every interpreted

[5]The notion of a formula depending on the past is different from, but related to, the previously defined notion of a knowledge-based protocol's actions depending only the past. If we restrict attention to systems based on a function σ, then it is easy to see that any formula φ that involves only past-time temporal operators depends only on the past.

[6]This notion of implementation was inspired by the notion of an isomorphism between two interpreted systems, as defined in the revised version of [Had87].

system consistent with the knowledge-based protocol satisfies appropriate safety and liveness properties. The correctness of certain standard protocols (including ones which correspond to the well-known *Alternating Bit Protocol* [BSW69] and protocols given by Aho, Ullman, and Yannakakis [AUY79, AUWY82]) is proved by showing that the system consisting of the set of runs for the standard protocol is a continuous implementation with respect to a certain set Φ of one of the interpreted systems consistent with the knowledge-based protocol. The set Φ is chosen so that the implementation preserves the reading and writing of data elements. Thus Φ consists of formulas formulas of the form "the value i^{th} data element is j", the "the i^{th} data element has been read", and "the i^{th} data element was written", for $i = 1, 2, 3, \ldots$.[7]

8 Conclusions

We have presented a general model of knowledge and action in distributed systems. This area has seen quite an upsurge of interest recently. The main contribution of this work has been to focus in on the interaction between knowledge and action, and, in particular, to define and give a formal treatment of knowledge-based protocols.

There are a number of obvious directions for further work along these lines. We have not carefully considered probabilistic or randomized protocols in our discussion. Such protocols give rise in a natural way to a probability measure on the set of runs. In order to reason about probability in our framework, we want probabilities on the points, not the runs. This allows us to extend our language with such formulas as $K_i^\alpha \varphi$, which holds if φ holds on a set of measure at least α of the points that process i considers possible. Probability has always been incorporated into the economists' models of knowledge (cf. [Aum76, MZ85]), although the economists do not use a formal language for reasoning about knowledge and probability. We have recently extended the model presented here in order to deal with reasoning about knowledge and probability; see [FH88b] for details.

Another interesting line of research is that of trying to axiomatize certain properties of communication (e.g., the fact that communication is guaranteed, or, for that matter, that communication is *not* guaranteed). The idea would be to capture these notions by describing how they affect a process' knowledge. Some work along these lines is described in [FHV92, FV86].

Perhaps most interesting of all is the continued investigation of knowledge-based protocols. Knowledge-based protocols seem to be a particularly useful high-level tool for analyzing many natural situations that arise in distributed computing. It is certainly much more natural to describe the wives' protocol in essentially all the variants of the cheating husbands puzzle presented in [MDH86] as "For all days $k = 1, 2, 3, \ldots$, if you know that your husband is unfaithful, then shoot him at midnight; otherwise do nothing", rather than trying to explain the appropriate standard protocol for each variant.

A particularly intriguing notion is that of having a programming language that would directly allow us to write knowledge-based protocols, with details of how the knowledge is computed being invisible to the programmer. Such a high-level programming language would require

[7] In [HZ92] the set Φ is not explicitly described, but it is clear from the description there that a continuous implementation with respect to the set Φ described above is actually constructed.

a "compiler" that could translate knowledge-based tests to knowledge-free tests. Presumably this could only be done by imposing restrictions on both the language of communication and the environment (perhaps restricting attention to a situation where communication is guaranteed and proceeds in rounds, and there are no failures).

Before we could hope to have such a language, of course, much further work needs to be undertaken to understand all the subtleties of translating knowledge-based protocols to standard protocols. The work of [DM90] and [MT88] can be viewed as taking some steps in this direction. In [DM90], Dwork and Moses give a simple knowledge-based protocol that guarantees simultaneous Byzantine agreement in an optimal number of rounds for all runs, under the assumption that the only failures are crash failures (where a process can fail only by crashing, and once it does so, it sends no further messages). They show that this knowledge-based protocol can be efficiently transformed into a standard protocol. In [MT88], Moses and Tuttle extend these results by showing how the knowledge-based protocol can be converted to a standard protocol if the only failures are omission failures (where the only faulty behavior a process may exhibit is in not sending a message, but all the messages it sends are those it should send according to the protocol). The conversion to a standard protocol is more difficult here, but it can still be done efficiently (in time polynomial in the number of processes in the network). However, it is also shown that for a slightly more general notion of failure, where a process may either fail to send a message or fail to receive one, although the knowledge-based protocol is still correct and can be converted to a standard protocol, this conversion is NP-hard (in the size of the network).

This leads us to one last issue. As we mentioned before, the notion of knowledge we consider is an external one, ascribed by the system designer to the processes. There is no notion of a process computing its knowledge. Thus it may seem somewhat strange to consider knowledge-based protocols where processes perform actions based on their knowledge, if this is knowledge that they might not be able to compute. To the extent that we view a knowledge-based protocol as a convenient specification used by the system designer, there is no problem here. For many applications, it may also be the case that the necessary knowledge to carry out a knowledge-based protocol can be computed easily (although the results in [MT88] mentioned above show that this is not always the case). These observations point out the need for a notion of knowledge in distributed systems that takes into account the computation required to obtain that knowledge. Such a notion of knowledge would *not* satisfy all the axioms and rules of inference discussed in Section 4. In particular, we would not expect a process' knowledge to be closed under logical implication. Abstract models for notions of knowledge where agents are not perfect reasoners are discussed in many papers in the philosophy and AI literature (cf. [FH88a, Lev84]); a semantics that seems to be appropriate for distributed systems is given in [Mos88].

We feel that a deeper analysis of the interaction of knowledge, action, and communication will be useful in order to improve our understanding of distributed systems. We have clearly only scratched the surface here; we hope that much more work will be done.

Acknowledgments: Yoram Moses made numerous invaluable suggestions that helped improve the style and presentation of the ideas; chief among these was his suggestion (and insistence!) that a system should be viewed as a set of runs. Lenore Zuck also made a number of useful suggestions, including ones that helped simplify the presentation of the material on knowledge-based protocols which determine a canonical set of runs. We would also like to thank Martín Abadi, Peter van Emde Boas, Shel Finkelstein, Vassos Hadzilacos, Leslie Lamport, Nimrod Megiddo, Michael Merritt, Ray Strong, Moshe Vardi, and Pierre Wolper for their helpful comments and criticisms.

References

[Aum76] R. J. Aumann. Agreeing to disagree. *Annals of Statistics*, 4(6):1236–1239, 1976.

[AUWY82] A. V. Aho, J. D. Ullman, A. D. Wyner, and M. Yannakakis. Bounds on the size and transmission rate of communication protocols. *Computers and Mathematics with Applications*, 8(3):205–214, 1982. This is a later version of [AUY79].

[AUY79] A. V. Aho, J. D. Ullman, and M. Yannakakis. Modeling communication protocols by automata. In *Proc. 20th IEEE Symp. on Foundations of Computer Science*, pages 267–273, 1979.

[BSW69] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12:260–261, 1969.

[CM86] K. M. Chandy and J. Misra. How processes learn. *Distributed Computing*, 1(1):40–52, 1986.

[DM90] C. Dwork and Y. Moses. Knowledge and common knowledge in a Byzantine environment: crash failures. *Information and Computation*, 88(2):156–186, 1990.

[FH88a] R. Fagin and J. Y. Halpern. Belief, awareness, and limited reasoning. *Artificial Intelligence*, 34:39–76, 1988.

[FH88b] R. Fagin and J. Y. Halpern. Reasoning about knowledge and probability: preliminary report. In M. Y. Vardi, editor, *Proc. Second Conference on Theoretical Aspects of Reasoning about Knowledge*, pages 277–293. Morgan Kaufmann, San Francisco, CA, 1988. An expanded version of this paper appears as IBM Research Report RJ 6020, 1990; to appear in *Journal of the ACM*.

[FHV92] R. Fagin, J. Y. Halpern, and M. Y. Vardi. What can machines know? On the properties of knowledge in distributed systems. *Journal of the ACM*, 39(2):328–376, 1992.

[FI86] M. J. Fischer and N. Immerman. Foundations of knowledge for distributed systems. In J. Y. Halpern, editor, *Theoretical Aspects of Reasoning about Knowledge: Proc. 1986 Conference*, pages 171–186. Morgan Kaufmann, San Francisco, CA, 1986.

[FV86] R. Fagin and M. Y. Vardi. Knowledge and implicit knowledge in a distributed environment: preliminary report. In J. Y. Halpern, editor, *Theoretical Aspects of Reasoning about Knowledge: Proc. 1986 Conference*, pages 187–206. Morgan Kaufmann, San Francisco, CA, 1986.

[FW78] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. 10th ACM Symp. on Theory of Computing*, pages 114–118, 1978.

[Gol82] L. Goldschlager. A unified approach to models of synchronous parallel machines. *Journal of the ACM*, 29(4):1073–1086, 1982.

[Had87] V. Hadzilacos. A knowledge-theoretic analysis of atomic commitment protocols. In *Proc. 6th ACM Symp. on Principles of Database Systems*, pages 129–134, 1987. A revised version has been submitted for publication.

[Hal87] J. Y. Halpern. Using reasoning about knowledge to analyze distributed systems. In J. F. Traub, B. J. Grosz, B. W. Lampson, and N. J. Nilsson, editors, *Annual Review of Computer Science, Vol. 2*, pages 37–68. Annual Reviews Inc., Palo Alto, CA, 1987.

[Har79] D. Harel. *First-Order Dynamic Logic.* Lecture Notes in Computer Science, Vol. 68. Springer-Verlag, Berlin/New York, 1979.

[HF85] J. Y. Halpern and R. Fagin. A formal model of knowledge, action, and communication in distributed systems: preliminary report. In *Proc. 4th ACM Symp. on Principles of Distributed Computing*, pages 224–236, 1985.

[HM90] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, 1990. A preliminary version appeared in *Proc. 3rd ACM Symposium on Principles of Distributed Computing*, 1984.

[HM92] J. Y. Halpern and Y. Moses. A guide to completeness and complexity for modal logics of knowledge and belief. *Artificial Intelligence*, 54:319–379, 1992.

[HV86] J. Y. Halpern and M. Y. Vardi. The complexity of reasoning about knowledge and time. In *Proc. 18th ACM Symp. on Theory of Computing*, pages 304–315, 1986.

[HV89] J. Y. Halpern and M. Y. Vardi. The complexity of reasoning about knowledge and time, I: lower bounds. *Journal of Computer and System Sciences*, 38(1):195–237, 1989.

[HZ92] J. Y. Halpern and L. D. Zuck. A little knowledge goes a long way: knowledge-based derivations and correctness proofs for a family of protocols. *Journal of the ACM*, 39(3):449–478, 1992.

[Lam85] L. Lamport. Paradigms for distributed computing. In M. Paul and H. J. Siegert, editors, *Methods and tools for specification, an advanced course*, Lecture Notes in Computer Science, Vol. 190, pages 19–30, 454–468. Springer-Verlag, Berlin/New York, 1985.

[Lam86] L. Lamport. On interprocess communication, Part I: basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

[Leh84] D. J. Lehmann. Knowledge, common knowledge, and related puzzles. In *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, pages 62–67, 1984.

[Lev84] H. J. Levesque. A logic of implicit and explicit belief. In *Proc. National Conference on Artificial Intelligence (AAAI '84)*, pages 198–202, 1984.

[LF81] N. A. Lynch and M. J. Fischer. On describing the behavior and implementation of distributed systems. *Theoretical Computer Science*, 13:17–43, 1981.

[LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In Rohit Parikh, editor, *Proc. Workshop on Logics of Programs*, Lecture Notes in Computer Science, Vol. 193, pages 196–218. Springer-Verlag, Berlin/New York, 1985.

[MDH86] Y. Moses, D. Dolev, and J. Y. Halpern. Cheating husbands and other stories: a case study of knowledge, action, and communication. *Distributed Computing*, 1(3):167–176, 1986.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science, Vol. 92. Springer-Verlag, Berlin/New York, 1980.

[Moo85] R. C. Moore. A formal theory of knowledge and action. In J. Hobbs and R. C. Moore, editors, *Formal Theories of the Commonsense World*, pages 319–358. Ablex Publishing Corp., Norwood, NJ, 1985.

[Mos88] Y. Moses. Resource-bounded knowledge. In M. Y. Vardi, editor, *Proc. Second Conference on Theoretical Aspects of Reasoning about Knowledge*, pages 261–276. Morgan Kaufmann, San Francisco, CA, 1988.

[MT88] Y. Moses and M. R. Tuttle. Programming simultaneous actions using common knowledge. *Algorithmica*, 3:121–169, 1988.

[MZ85] J. F. Mertens and S. Zamir. Formulation of Bayesian analysis for games of incomplete information. *International Journal of Game Theory*, 14(1):1–29, 1985.

[NT93] G. Neiger and S. Toueg. Simulating real-time clocks and common knowledge in distributed systems. *Journal of the ACM*, 40(2):334–367, 1993.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. on Foundations of Computer Science*, pages 46–57, 1977.

[PR85] R. Parikh and R. Ramanujam. Distributed processing and the logic of knowledge. In R. Parikh, editor, *Proc. Workshop on Logics of Programs*, pages 256–268, 1985.

[Pra76] V. R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proc. 17th IEEE Symp. on Foundations of Computer Science*, pages 109–121, 1976.

[Pra82] V. R. Pratt. On the composition of processes. In *Proc. 9th ACM Symp. on Principles of Programming Languages*, pages 213–223, 1982.

[Pra85] V. R. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1985.

[PT92] P. Panangaden and S. Taylor. Concurrent common knowledge: Defining agreement for asynchronous systems. *Distributed Computing*, 6(2):73–94, 1992.

[Smu78] R. Smullyan. *What is the Name of this Book?* Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Vis83] U. Vishkin. Implementation of simultaneous memory access in models that forbid it. *Journal of Algorithms*, 4:45–50, 1983.