# Implementing Knowledge-Based Programs

Moshe Y. Vardi*
Rice University

**Abstract**

Reasoning about multi-agents systems at the *knowledge level* allows us to abstract away from many concrete details of the systems we are considering. Fagin et al. introduced two notions to facilitate designing and reasoning about systems in terms of knowledge. The first notion is that of *knowledge-based programs*. Knowledge-based programs are defined as syntactic objects: programs with tests for knowledge. The second notion is that of *contexts*, which capture the setting in which a program is to be executed. In a given context, a standard program (one without tests for knowledge) is always *implementable*. A knowledge-based program, on the other hand, might not be implementable. We provide a condition which is necessary and sufficient to guarantee that a given knowledge-based program is implemented by a given protocol, and we completely characterize the complexity of determining whether a given knowledge-based program is implemented by a given protocol in a given *finite-state* context. In particular, we identify an important special case where this problem is tractable.

## 1 Introduction

Reasoning about activities in multi-agent systems at the knowledge level allows us to abstract away from many concrete details of the system we are considering. It is often more intuitive to think in terms of the high-level concepts when we design a protocol, and then translate these intuitions into a concrete program, based on the particular properties of the setting we are considering. This style of program development will generally allow us to modify the program more easily when considering a setting with different properties, such as different communication topologies, different guarantees about the reliability of various components of the system, and the like. See [APP88, BLS93, BLMS94, CM86, DM90, Had87, HMW90, HZ92, MT88, NB92, NT93, Rub89] for examples of knowledge-level analysis of multi-agent systems.

Motivated by these considerations, and following an earlier proposal in [HF85], Fagin et al. [FHMV95b] proposed a notion of *knowledge-based programs*, in which an agent's actions

*Address: Department of Computer Science. Rice University, Mail Stop 132, 6100 S. Main Street, Houston, TX 77005-1892, E-mail: `vardi@cs.rice.edu`, URL: `http://www.cs.rice.edu/~vardi`

depend explicitly on the agent's knowledge. Their goal was to provide a formal semantics for programs with statements such as

**if** $K(x = 0)$ **do** $y := y + 1$,

where $K(x = 0)$ should be read as "you know that $x = 0$". Knowledge-based programs are simply programs with tests for knowledge. Their semantics is defined with respect to *contexts*, which capture the setting in which a program is to be executed. Roughly speaking, a context describes the set of initial states, the environment's protocol (which tells us, for example, whether messages can be lost or reordered), and the effect of each action on the global state of the system. By distinguishing between programs and contexts, and by ascribing meaning to programs in different contexts in a uniform manner, a high-level, model-independent, knowledge-level reasoning is facilitated.

In a given context, a *standard* program (one without tests for knowledge) is always *implementable*, since in any given state of the system the program unambigously prescribes the actions to be performed. That is, the program can be *implemented* by a *protocol*, which is a function from local states to actions. A knowledge-based program, on the other hand, might not be implementable. The reason for this is the circularity inherent in the definition of knowledge-based programs: the actions to be performed depend on the agents' knowledge, but the agents' knowledge depend on the actions that have been performed. A knowledge-based program should be viewed as a high-level specification; various implementations or none may satisfy this specification. If the program is not implementable, then the specification is unsatisfiable.

Of course, if we are to program at the knowledge level, we need to be able to tell whether a given knowledge-based program is implementable. Similarly, we would like to be able to tell whether a given knowledge-based program is implemented by a *given* protocol. Thus, the questions of implementability and implementation are fundamental to knowledge-level reasoning. Implementability was studied in [FHMV95a]. In this paper, we address the related issue of implementation. We provide a condition which is necessary and sufficient to guarantee that a given knowledge-based program is implemented by a given protocol, and we completely characterize the complexity of determining whether a given knowledge-based program is implemented by a given protocol in a given *finite-state* context. In particular, we identify an important special case where this problem is tractable.

In Section 2, we review the framework of [FHMV95b] for systems, protocols, and programs. In Section 3, we provide a characterization of the implementation relation and study its complexity with respect to finite-state contexts.

## 2 Systems, Protocols, and Programs

We assume that at each point in time, each agent is in some *local state*. Informally, this local state encodes the information the agent has observed thus far. In addition, there is also an *environment* state, that keeps track of everything relevant to the system not recorded in the agents' states. The way we split up the system into agents and environment depends on the system being analyzed.

A *global state* is a tuple $g = (\ell_e, \ell_1, \ldots, \ell_n)$ consisting of the environment state $\ell_e$ and the local state $\ell_i$ of each agent $i$. We sometimes use $g_i$ to denote the local state of agent $i$ in a global state $g$ A *run* of the system is a function from time (which, for ease of exposition, we assume ranges over the natural numbers) to a set $\mathcal{G}$ of global states. Thus, if $r$ is a run, then $r(0), r(1), \ldots$ is a sequence of global states that, roughly speaking, is a complete description of what happens over time in one possible execution of the system. We take a *system* to consist of a nonempty set of runs. Intuitively, these runs describe all the possible sequences of events that could occur. We assume that we have a set $\Phi$ of primitive propositions, 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", or "the system is deadlocked". We define an *interpreted system* to be a pair $(\mathcal{R}, \pi)$ consisting of a system $\mathcal{R}$ together with a mapping $\pi$ that assigns a truth value to each primitive proposition in $\Phi$ at each global state.

Given a system $\mathcal{R}$, we refer to a pair $(r, m)$ consisting of a run $r \in \mathcal{R}$ and a time $m$ as a *point*. If $r(m) = (\ell_e, \ell_1, \ldots, \ell_n)$, we define $r_e(m) = \ell_e$ and $r_i(m) = \ell_i$ for $i = 1, \ldots, n$; thus, $r_e(m)$ is the environment state and $r_i(m)$ is process $i$'s local state at the point $(r, m)$. We say that two points $(r, m)$ and $(r', m')$ are *indistinguishable* to agent $i$, and write $(r, m) \sim_i (r', m')$, if $r_i(m) = r'_i(m')$, i.e., if agent $i$ has the same local state at both points.

We can give semantics to formulas with respect to points in an interpreted system in a straightforward way. In particular, given a point $(r, m)$ in an interpreted system $\mathcal{I} = (\mathcal{R}, \pi)$, we have $(\mathcal{I}, r, m) \models K_i\varphi$ (that is, the formula $K_i\varphi$ is satisfied at the point $(r, m)$ of $\mathcal{I}$) if $(\mathcal{I}, r', m') \models \varphi$ for all points $(r', m')$ such that $(r', m') \sim_i (r, m)$. Notice that under this interpretation, an agent knows $\varphi$ precisely if $\varphi$ is true at all the situations the system could be in, given the agent's current information (as encoded by its local state). Interpreted systems also provide semantics for temporal formulas. For example, $(I, r, m) \models \bigcirc\varphi$ if $(I, r, m+1) \models \varphi$. Thus, interpreted systems can provide semantics for the language of knowledge and time.

Intuitively, a *protocol* for agent $i$ is a description of what actions agent $i$ may take, as a function of his local state. We formally define a *protocol* $P_i$ for agent $i$ to be a function from a set $L_i$ of agent $i$'s local states to nonempty sets of actions in a set $ACT_i$ of agent $i$'s possible actions. It is also useful to view the environment as running a protocol. We define a protocol for the environment to be a function from a set $L_e$ of the environment's local states to nonempty subsets of a set $ACT_e$ of the environment's possible actions. For both the agents and the environment, we allow for the possibility of a special *null* action $\Lambda$. While our notion of protocol is quite general, there is a crucial restriction: a protocol is a function on *local* states, rather than a function on *global* states. This captures our intuition that all the information that the agent has is encoded in his local state. Thus, what an agent does can depend only on his local state, and not on the whole global state.

Processes do not run their protocols in isolation; it is the combination of the protocols run by all agents that cause the system to behave in a particular way. We define a *joint protocol* $P$ to be a tuple $(P_1, \ldots, P_n)$ consisting of protocols $P_i$, for each of the agents $i = 1, \ldots, n$.

Note that we do not include the environment's protocol in a joint protocol. This is because of the environment's special role; we usually design and analyze the agents' protocols, while taking the environment's protocol as a given. In fact, when designing multi-agent systems, the environment is often seen as an *adversary* who may be trying to cause the system to behave in some undesirable way.

Since a joint protocol $P$ describes the behavior of the agents in the system, we would like to be able to generate the runs of such a protocol. To do this, we need describe the setting, or *context*, in which $P$ is being run. Formally, a context $\gamma$ is a tuple $(P_e, \mathcal{G}_0, \tau, \Psi)$, where $P_e$ is the environment's protocol, $\mathcal{G}_0$ is a set of initial global states, $\tau$ is a *transition function*, and $\Psi$ is a set of runs. The role of $P_e$ and $\mathcal{G}_0$ should be clear; we now explain the role of $\tau$ and $\Psi$.

With the joint protocol $P$ and $P_e$ in hand, we know how the "participants" in the system will act at any point. But how will their actions affect the system? The transition function $\tau$ characterizes that. Define a *joint action* to be a tuple of the form $(\mathsf{a}_e, \mathsf{a}_1, \ldots, \mathsf{a}_n)$, where $\mathsf{a}_e$ is an action performed by the environment and $\mathsf{a}_i$ is an action performed by agent $i$. We associate with each joint action $(\mathsf{a}_e, \mathsf{a}_1, \ldots, \mathsf{a}_n)$ a *global state transformer* $\tau(\mathsf{a}_e, \mathsf{a}_1, \ldots, \mathsf{a}_n)$, which maps global states to global states. Thus, if $\tau(\mathsf{a}_e, \mathsf{a}_1, \ldots, \mathsf{a}_n)(g) = g'$, then $g'$ is the result of performing the joint action $(\mathsf{a}_e, \mathsf{a}_1, \ldots, \mathsf{a}_n)$ in state $g$.

Given $P_e$, $P$, $\tau$, and $\mathcal{G}_0$, we can generate the possible executions of $(P_e, P)$, starting at global states in $\mathcal{G}_0$. There are times, however, when we do not want to consider *all* the runs generated. This is the case, for example, with a *fairness* assumption such as "all message sent are eventually delivered". We do not want to consider "unfair" runs, where some messages are not delivered. There are a number of ways that we could capture such a restriction. Perhaps the simplest is to specify a set of *admissible* runs, where the appropriate things happen (for example, all messages get delivered). This is the role of the *admissibility* condition $\Psi$ in the context. Often the admissibility condition $\Psi$ can be characterized by temporal formulas and the runs in $\Psi$ are those that satisfy these formulas. For example, to specify reliability of communication, we can use the admissibility condition $Rel$ defined by $Rel = \{r \mid \text{all messages sent in } r \text{ are eventually received}\}$. Let $send(\mu, j, i)$ be a proposition that is interpreted to mean "message $\mu$ is sent to $j$ by $i$", and let $receive(\mu, i, j)$ be a proposition that is interpreted to mean "message $\mu$ is received from $i$ by $j$". Then a run $r$ is in $Rel$ precisely if $\Box(send(\mu, j, i) \Rightarrow \Diamond receive(\mu, i, j))$ holds at $(r, 0)$ (and thus at every point in $r$) for each message $\mu$ and processes $i, j$.

It is only in a context that a joint protocol describes the behavior of a system. The combination of a context $\gamma$ and a joint protocol $P$ for the agents uniquely determines a set of runs. Intuitively, $r$ is a run of a joint protocol $P = (P_1, \ldots, P_n)$ in the context $\gamma = (P_e, \mathcal{G}_0, \tau, \Psi)$ if $r(0) \in \mathcal{G}_0$, $r(m+1)$ follows according to $\tau$ from $r(m)$ by a joint action prescribed by $P_e$ and $P$ in $r(m)$ (i.e., if $r(m) = (\ell_e, \ell_1, \ldots, \ell_n)$, then there is a joint action $(\mathsf{a}_e, \mathsf{a}_1, \ldots, \mathsf{a}_n) \in P_e(\ell_e) \times P_1(\ell_1) \times \cdots \times P_n(\ell_n)$ such that $r(m+1) = \tau(\mathsf{a}_e, \mathsf{a}_1, \ldots, \mathsf{a}_n)(r(m))$), for $0 \leq m$, and $r \in \Psi$. We define $\mathbf{R}^{rep}(P, \gamma)$ to be the system consisting of all runs of $P$ in context $\gamma$, and call it the *system representing protocol $P$ in context $\gamma$*.

In many cases we have a particular interpretation $\pi$ for the primitive propositions in $\Phi$ in mind when we define a context. Just as we went from systems to interpreted systems, we can go from contexts to *interpreted contexts*; an interpreted context is a pair $(\gamma, \pi)$ consisting of a context $\gamma$ and an interpretation $\pi$. Given a joint protocol $P$, we say that $\mathbf{I}^{rep}(P, \gamma, \pi) = (\mathbf{R}^{rep}(P, \gamma), \pi)$ is *the interpreted system representing* $P$ *in interpreted context* $(\gamma, \pi)$.

As discussed above, a protocol is a function from local states to sets of actions. We typically describe protocols by means of *programs* written in some programming language. For example, a receiver $R$, which starts sending an *ack* message after it has received a bit from the sender $S$, can be described by a program such as "**if** $recbit$ **do** $\mathsf{sendack}$". The essential feature of this statement is that the program selects an action based on the result of a test regarding the proposition $recbit$ whose truth depends solely on the local state.

We now describe a simple programming language, which is rich enough to describe protocols, and whose syntax emphasizes the fact that an agent performs actions based on the result of a test that is applied to her local state. A (*standard*) *program* for agent $i$ is a statement of the form:

**case of**
  **if** $t_1$ **do** $\mathsf{a}_1$
  **if** $t_2$ **do** $\mathsf{a}_2$
  ...
**end case**

where the $t_j$'s are *standard tests* for agent $i$ and the $\mathsf{a}_j$'s are actions of agent $i$ (i.e., $\mathsf{a}_j \in ACT_i$). (We call such programs "standard" to distinguish them from the *knowledge-based* programs that we describe later.) A standard test for agent $i$ is simply a propositional formula over a set $\Phi_i$ of primitive propositions (whose truth should depend on $i$'s local state). Intuitively, once we know how to evaluate the tests in the program at the local states in $L_i$, we can convert this program to a protocol over $L_i$: at a local state $\ell$, agent $i$ nondeterministically chooses one of the clauses in the **case** statement whose test is true at $\ell$, and executes the corresponding action.

We want to use an interpretation $\pi$ to tell us how to evaluate the tests. Not just any interpretation, however, will do. We intend the tests in a program for agent $i$ to be *local*, that is, to depend only on agent $i$'s local state. It would be inappropriate for agent $i$'s action to depend on the truth value of a test that $i$ could not determine from her local state. We say that an interpretation $\pi$ on the global states in $\mathcal{G}$ is *compatible* with a program $\mathsf{Pg}_i$ for agent $i$ (with respect to a set $\mathcal{G}$ of global states) if every proposition that appears in $\mathsf{Pg}_i$ is *local* to $i$, that is, if $q$ appears in $\mathsf{Pg}_i$, the states $g$ and $g'$ are in $\mathcal{G}$, and $g \sim_i g'$, then $\pi(g)(q) = \pi(g')(q)$. (This is equivalent to requiring $\pi$ to interpret local propositions in local rather than global states.) If $\varphi$ is a propositional formula all of whose primitive propositions are local to agent $i$ and $\ell$ is a local state of agent $i$, then we write $(\pi, \ell) \models \varphi$ if $\varphi$ is satisfied by the truth assignment $\pi(g)$, where $g$ is a global state such that $g_i = \ell$. Since all the primitive propositions in $\varphi$ are local to $i$, it does not matter which global state $g$ we choose, as long as $i$'s local state in $g$ is $\ell$.

Given a program $\mathsf{Pg}_i$ for agent $i$ and an interpretation $\pi$ compatible with $\mathsf{Pg}_i$, we define a protocol that we denote $\mathsf{Pg}_i^\pi$ by setting:

$$\mathsf{Pg}_i^\pi(\ell) = \begin{cases} \{\mathsf{a}_j : (\pi, \ell) \models t_j\} & \text{if } \{j : (\pi, \ell) \models t_j\} \neq \emptyset \\ \{\Lambda\} & \text{if } \{j : (\pi, \ell) \models t_j\} = \emptyset. \end{cases}$$

Intuitively, $\mathsf{Pg}_i^\pi$ selects all actions from the clauses that satisfy the test, and selects the null action $\Lambda$ if no test is satisfied. In general, we get a nondeterministic protocol, since more than one test may be satisfied at a given state.

Notice that the syntactic form of our standard programs is in many ways more restricted than that of programs in common programming languages such as C or FORTRAN. In such languages, one typically sees constructs such as **for**, **while**, or **if**... **then**... **else**..., which do not have syntactic analogues in our formalism. The semantics of programs containing such constructs depends on the local state containing an implicit *instruction counter*, specifying the command that is about to be executed at the current local state. Since we model the local state of a process explicitly, it is possible to simulate these constructs in our framework by having an explicit variable in the local state accounting for the instruction counter. The local tests $t_j$ used in a program can then reference this variable explicitly, and the actions $\mathsf{a}_j$ can include explicit assignments to this variable. Given that such simulation can be carried out in our framework, there is no loss of generality in our definition of standard programs.

Many of the definitions that we gave for protocols have natural analogues for programs. We define a *joint* program to be a tuple $\mathsf{Pg} = (\mathsf{Pg}_1, \dots, \mathsf{Pg}_n)$, where $\mathsf{Pg}_i$ is a program for agent $i$. We say that an interpretation $\pi$ is *compatible* with $\mathsf{Pg}$ if $\pi$ is compatible with each of the $\mathsf{Pg}_i$'s. From $\mathsf{Pg}$ and $\pi$ we get a joint protocol $\mathsf{Pg}^\pi = (\mathsf{Pg}_1^\pi, \dots, \mathsf{Pg}_n^\pi)$. We say that an interpreted system $\mathcal{I} = (\mathcal{R}, \pi)$ *represents* a joint program $\mathsf{Pg}$ in the interpreted context $(\gamma, \pi)$ exactly if $\pi$ is compatible with $\mathsf{Pg}$, and $\mathcal{I}$ represents the corresponding protocol $\mathsf{Pg}^\pi$. We denote the interpreted system representing $\mathsf{Pg}$ in $(\gamma, \pi)$ by $\mathbf{I}^{rep}(\mathsf{Pg}, \gamma, \pi)$. Of course, this definition only makes sense if $\pi$ is compatible with $\mathsf{Pg}$. From now on we always assume that this is the case.

Our notion of standard programs, in which agents perform actions based on the results of tests that are applied to their local state, is very simple. It is rich enough, however, to describe protocols. Nevertheless, standard programs cannot be used to describe the relationships between knowledge and action that we would often like to capture. The issue is perhaps best understood by considering the well-known muddy children puzzle (cf. [Bar81, HM90]).

Recall that in the muddy children puzzle, the children are asked by the father if they know whether they have mud on their foreheads. If so, they are supposed to answer "Yes"; otherwise they should answer "No". If we take the proposition $p_i$ to represent "child $i$ has mud on his forehead", then it seems quite reasonable to think of child $i$ as following the program $\mathsf{MC}_i$:

**case of**
    **if** *childheard*$_i \wedge (K_i p_i \vee K_i \neg p_i)$ **do** say "Yes"
    **if** *childheard*$_i \wedge \neg K_i p_i \wedge \neg K_i \neg p_i$ **do** say "No"
**end case**.

Here *childheard*$_i$ is a primitive proposition that is true at a given state if child $i$ heard the father's question "Does any of you know whether you have mud on your own forehead?" in the previous round. Unfortunately, $\mathsf{MC}_i$ is not a program as we have defined it. Besides propositional tests, it has tests for knowledge such as $K_i p_i \vee K_i \neg p_i$. Moreover, we cannot use our earlier techniques to associate a protocol with a program, since the truth value of such a knowledge test cannot be determined by looking at a local state in isolation.

We call a program of the form above a *knowledge-based* program, to distinguish it from the standard programs defined earlier. Formally, a knowledge-based program for agent $i$ has the form:

**case of**
  **if** $t_1 \wedge k_1$ **do** $\mathsf{a}_1$
  **if** $t_2 \wedge k_2$ **do** $\mathsf{a}_2$
  ...
**end case**

where the $t_j$'s are standard tests, the $k_j$'s are *knowledge tests* for agent $i$, and the $\mathsf{a}_j$'s are actions of agent $i$. A knowledge test for agent $i$ is a Boolean combination of formulas of the form $K_i\varphi$, where $\varphi$ can be an arbitrary formula that may include other modal operators, including common knowledge and temporal operators. Intuitively, the agent selects an action based on the result of applying the standard test to her local state and applying the knowledge test to her "knowledge state". In the program $\mathsf{MC}_i$, the test *childheard*$_i$ is a standard test, while $K_i p_i \vee K_i \neg p_i$ and $\neg K_i p_i \wedge \neg K_i \neg p_i$ are knowledge tests. We define a *joint* knowledge-based program to be a tuple $\mathsf{Pg} = (\mathsf{Pg}_1, \ldots, \mathsf{Pg}_n)$, with one knowledge-based program for each agent.

The muddy children example shows that knowledge-based programs are better than standard programs in capturing our intuitions about the relationship between knowledge and action. Even when we can capture our intuitions using a standard program, it may be useful to think at the knowledge level, since this allows us to look at things in a more high-level way, abstracting away irrelevant details.

**Example 2.1** Imagine we have two processes, say a *sender* $S$ and a *receiver* $R$, that communicate over a communication line. The sender starts with one bit (either 0 or 1) that it wants to communicate to the receiver. Unfortunately, the communication line is faulty and it may lose messages in either direction in any given round. That is, there is no guarantee that a message sent by either $S$ or $R$ will be received. We call this *the bit-transmission problem*.

One way to solve this problem is by the following protocol: the sender $S$ keeps sending the bit until an acknowledgement is received from the receiver $R$, and the receiver $R$ keeps sending acknowledgements once it has received the bit from the sender $S$. The purpose of the acknowledgement is to inform $S$ that the bit was received by $R$. Thus, another way to describe the sender's behavior is to say that $S$ keeps sending the bit until it *knows* that the bit was received by $R$. This can be described by the knowledge-based program $\mathsf{BT}'_S$:

**if** $\neg K_S(\textit{recbit})$ **do** $\mathsf{sendbit}$.

The advantage of this program over the standard program $\mathsf{BT}_S$

**if** $\neg$*recack* **do** sendbit

is that it abstracts away the mechanism by which $S$ learns that the bit was received by $R$. For example, if messages from $S$ to $R$ are guaranteed to be delivered in the same round in which they are sent, then $S$ knows that $R$ received the bit even if $S$ does not receive an acknowledgement. The knowledge-based framework enables us to abstract even further. The reason that $S$ keeps sending the bit to $R$ is that $S$ wants $R$ to know the value of the bit. Thus, intuitively, $S$ should keep sending the bit until it knows that $R$ knows its value. Let $K_R(\mathit{bit})$ be an abbreviation for $K_R(\mathit{bit} = 0) \vee K_R(\mathit{bit} = 1)$. Thus, $K_R(\mathit{bit})$ is true precisely if $R$ knows the value of the bit. The sender's behavior can be described by the knowledge-based program $\mathsf{BT}''_S$:

**if** $\neg K_S K_R(\mathit{bit})$ **do** sendbit.

This program abstracts away the manner in which $S$ learns that $R$ knows the value of the bit. If messages are guaranteed to be delivered in the same round that they are sent, then $S$ has to send the bit only once. Furthermore, if the value of the bit is common knowledge, then $\mathsf{BT}''_S$ does not require $S$ to send any messages. Thus, programming at the knowledge level enables us to design more efficient programs. ∎

We have described the syntax of knowledge-based programs, and have provided (by example) some intuition for how knowledge-based programs can be used to give high-level descriptions of the agents' behavior. It remains to give formal semantics to knowledge-based programs. Just as we think of a standard program as inducing a protocol that determines an agent's actions in a given context, we also want to think of a knowledge-based program as inducing a protocol. It is not obvious, however, how to associate a protocol with a knowledge-based program. A protocol is a function from local states to actions. To go from a standard program to a protocol, all we needed to do was to evaluate the standard tests at a given local state, which we did using interpretations. In a knowledge-based program, we also need to evaluate the knowledge tests. But in our framework, a knowledge test depends on the whole interpreted system, not just the local state. It may well be the case that agent $i$ is in the same local state $\ell$ in two different interpreted systems $\mathcal{I}_1$ and $\mathcal{I}_2$, and the test $K_i p$ may turn out to be true at the local state $\ell$ in $\mathcal{I}_1$ and false at the local state $\ell$ in $\mathcal{I}_2$.

To deal with this problem, we proceed as follows. Given an interpreted system $\mathcal{I} = (\mathcal{R}, \pi)$, we associate with a joint knowledge-based program $\mathsf{Pg} = (\mathsf{Pg}_1, \ldots, \mathsf{Pg}_n)$ a joint protocol that we denote $\mathsf{Pg}^{\mathcal{I}} = (\mathsf{Pg}_1^{\mathcal{I}}, \ldots, \mathsf{Pg}_n^{\mathcal{I}})$. Intuitively, we evaluate the standard tests in $\mathsf{Pg}$ according to $\pi$, and evaluate the knowledge tests in $\mathsf{Pg}$ according to $\mathcal{I}$. As in the case of standard programs, we require that $\pi$ be compatible with $\mathsf{Pg}$, that is, that every proposition appearing in a standard test in $\mathsf{Pg}_i$ should be local to $i$. Note that we place the locality requirement only on the propositions appearing in the standard tests, not the propositions appearing in the knowledge tests. We wish to define $\mathsf{Pg}_i^{\mathcal{I}}(\ell)$ for all local states $\ell$ of agent $i$. To define this, we first define when a test $\varphi$ holds in a local state $\ell$ with respect to an interpreted system $\mathcal{I}$, denoted $(\mathcal{I}, \ell) \models \varphi$.

If $\varphi$ is a standard test and $\mathcal{I} = (\mathcal{R}, \pi)$ then we define $(\mathcal{I}, \ell) \models \varphi$ iff $(\pi, \ell) \models \varphi$. Since $\varphi$ is a standard test in $\mathsf{Pg}_i$, it must be local to agent $i$, so this definition makes sense. If $\varphi$ is a knowledge test of the form $K_i \psi$, we define

$(\mathcal{I}, \ell) \models K_i\psi$ iff $(\mathcal{I}, r, m) \models \psi$ for all points $(r, m)$ of $\mathcal{I}$ such that $r_i(m) = \ell$.

Finally, for conjunctions and negations[1], we follow the standard treatment (recall that a knowledge test for agent $i$ is a Boolean combination of formulas of the form $K_i\varphi$).

We can now define:

$$\mathsf{Pg}_i^{\mathcal{I}}(\ell) = \begin{cases} \{\mathsf{a}_j : (\mathcal{I}, \ell) \models t_j \wedge k_j\} & \text{if } \{j : (\mathcal{I}, \ell) \models t_j \wedge k_j\} \neq \emptyset \\ \{\Lambda\} & \text{if } \{j : (\mathcal{I}, \ell) \models t_j \wedge k_j\} = \emptyset. \end{cases}$$

Intuitively, the actions prescribed by $i$'s protocol $\mathsf{Pg}_i^{\mathcal{I}}$ are exactly those prescribed by $\mathsf{Pg}_i$ in the interpreted system $\mathcal{I}$.

The mapping from knowledge-based programs to protocols allows us to define what it means for an interpreted system to represent a knowledge-based program in a given interpreted context. We say that an interpreted system $\mathcal{I}$ *represents* $\mathsf{Pg}$ in $(\gamma, \pi)$ if $\pi$ is compatible with $\mathsf{Pg}$ and $\mathcal{I}$ represents $\mathsf{Pg}^{\mathcal{I}}$ in $(\gamma, \pi)$. This means that to check if $\mathcal{I}$ represents $\mathsf{Pg}$, we check if $\mathcal{I}$ represents the protocol obtained by evaluating the knowledge tests in $\mathsf{Pg}$ with respect to $\mathcal{I}$ itself.

We say that a protocol $P$ *implements* the knowledge-based program $\mathsf{Pg}$ in the interpreted context $(\gamma, \pi)$ if $P$ coincides with the protocol $\mathsf{Pg}^{\mathcal{I}}$, where $\mathcal{I} = \mathbf{I}^{rep}(P, \gamma, \pi)$ is the interpreted system that represents $P$ in $(\gamma, \pi)$ (in particular, this requires that $\pi$ is compatible with $\mathsf{Pg}$). Intuitively, $P$ implements $\mathsf{Pg}$ if $P$ and $\mathsf{Pg}$ prescribe the same actions in the interpreted system that represents $P$. In particular, if $P$ implements $\mathsf{Pg}$ in $(\gamma, \pi)$, then the interpreted system $\mathbf{I}^{rep}(P, \gamma, \pi)$ representing $P$ in $(\gamma, \pi)$ also represents $\mathsf{Pg}$ in this interpreted context. Thus, if $\mathsf{Pg}$ satisfies some specification $\sigma$ in a given interpreted context, then $P$ also satisfies $\sigma$ there. (For earlier, related discussions of the concept of implementation see [HF89, Maz91].) Note that $\mathsf{Pg}$ is *implementable* in the interpreted context $(\gamma, \pi)$, i.e., implemented by some protocol $P$ in $(\gamma, \pi)$, precisely if there is some interpreted system that represents $\mathsf{Pg}$ in $(\gamma, \pi)$.

Because of the circularity of the definition, it is not necessarily the case that every knowledge-based program is implementable. Suppose, for example, that we have a system consisting of only one agent, agent 1, who has a bit that is initially set to 0. Suppose agent 1 runs the following simple knowledge-based program $\mathsf{NI}$ (for "not implementable"):

**if** $K_1(\Box(bit \neq 1))$ **do** $bit := 1$.

According to $\mathsf{NI}$, agent 1 sets the bit to 1 if she knows that the bit is never 1, and otherwise does nothing. It should be clear that this knowledge-based program is not implementable, since no protocol implements it. By way of contrast, a standard program is always implementable.

This behavior should not be viewed as a problem. Rather, it suggests that knowledge-based programs are best thought of as knowledge-level specifications. Of course, it is then important to understand when the specification is satisfiable, i.e., when a given knowledge-based program $\mathsf{Pg}$ is implementable in a given interpreted context $(\gamma, \pi)$. This problem was

[1]For convenience we assume that $\wedge$ and $\neg$ are the only propositional connectives.

studied in [FHMV95a]. In this paper we study a related but simpler question. Suppose that we are *given* a protocol $P$ (which perhaps is provided by the program designer, based on his intuitions about the program). How can we tell whether $P$ implements $\mathsf{Pg}$ in $(\gamma, \pi)$?

In the following section we provide a necessary and sufficient conditions for the implementatin relation. This condition enables us to completely characterize the complexity of determining whether a given knowledge-based program is implemented by a given protocol in a given *finite-state* context. A finite-state interpreted context is one in which the set of global states is finite, the set of possible actions is finite, the set of primitive propositions is finite, and the admissibility condition $\Psi$ on runs is given by a temporal logic formula. Such contexts are of importance since a significant number of the communication and synchronization protocols studied in the literature are in essence finite state [Liu89, Rud87].

## 3 Implementing Knowledge-Based Programs

Fagin et al. [FHMV95a], considered the problem of checking whether a given finite knowledge-based program is *implementable* in a given finite-state interpreted context. They proved that the problem is PSPACE-complete. The PSPACE-hardness results from the occurence of temporal formulas in the program or in the context, since satisfiability for linear time temporal logic is PSPACE-complete [SC85]. If we remove all vestiges of temporal logic from the problem then it becomes easier. A *nonrestrictive* interpreted context is one where the admissibility condition on runs is $True$, i.e., all runs are admissible; an *atemporal* knowledge-based program is one where the tests are knowledge formulas, i.e., they do not involve temporal operators. Fagin et al. [FHMV95a] showed that testing implementability of a given atemporal knowledge-based program in a given nonrestrictive finite-state interpreted context is *NP*-complete.

In this paper we study a seemingly easier problem. Instead of asking whether a given knowledge-based program $\mathsf{Pg}$ is *implementable*, we ask whether it is *implemented* by a *given* protocol $P$. Is testing for *implementation* easier then testing for *implementability* (just as testing for *satisfaction* of propositional formulas is believed to be easier than testing for *satisfiability* of such formulas [GJ79][2])?. We show that in general the answer is negative. Testing for implementation is still PSPACE-complete. The problem is easier, however, for atemporal programs and nonrestrictive contexts. While testing for implementability in this case is NP-complete, testing for implementation can be done in polynomial time.

We start by discussing the easier case. Let $\mathcal{F}$ be a set of global states and $\pi$ be an interpretation for the propositions in $\Phi$ over $\mathcal{F}$. We define a *Kripke structure* $M_{\mathcal{F}} = (\mathcal{F}, \mathcal{K}_1, \ldots, \mathcal{K}_n, \pi)$, where $(g, g') \in \mathcal{K}_i$ iff $g_i = g'_i$, that is, if $g$ and $g'$ agree on their $i$th component. Truth of knowledge formulas in $M_{\mathcal{F}}$ can now be defined in the standard way (cf. [HM92]). In particular, we have

$$(M_{\mathcal{F}}, g) \models K_i\varphi \text{ iff } (M_{\mathcal{F}}, g') \models \varphi \text{ for all } g' \text{ such that } (g, g') \in \mathcal{K}_i$$

[2]For example, the distinction between satisfaction and satisfiability has a significant impact in the area of program verification, where checking satisfiability of branching temporal formulas is EXPTIME-complete, while checking satisfaction of such formulas can be done in polynomial time; see [CES86].

Given a set $\mathcal{F}$ of global states, we can associate with an atemporal knowledge-based program $\mathsf{Pg}_i$ for agent $i$ a protocol $\mathsf{Pg}_i^{\mathcal{F}}$ in much the same way we used an interpreted system $\mathcal{I}$ to obtain the protocol $\mathsf{Pg}_i^{\mathcal{I}}$. We start as in Section 2 by defining truth of tests in local states. If $\varphi$ is a standard test and $\ell$ is a local state of agent $i$, then, in analogy to Section 2, we define

$(M_{\mathcal{F}}, \ell) \models \varphi$ iff $(\pi, \ell) \models \varphi$.

Since $\varphi$ is a standard test in $\mathsf{Pg}_i$, it must be local to agent $i$, so this definition makes sense. If $\varphi$ is a knowledge test $K_i\psi$, we define

$(M_{\mathcal{F}}, \ell) \models K_i\psi$ iff $(M_{\mathcal{F}}, g) \models \psi$ for all global states $g$ in $\mathcal{F}$ such that $g_i = \ell$.

Finally, for conjunctions and negations, we follow the standard treatment.

We can now define:

$$\mathsf{Pg}_i^{\mathcal{F}}(\ell) = \begin{cases} \{\mathsf{a}_j : (M_{\mathcal{F}}, \ell) \models t_j \wedge k_j\} & \text{if } \{j : (M_{\mathcal{F}}, \ell) \models t_j \wedge k_j\} \neq \emptyset \\ \{\Lambda\} & \text{if } \{j : (M_{\mathcal{F}}, \ell) \models t_j \wedge k_j\} = \emptyset. \end{cases}$$

Intuitively, the actions prescribed by $i$'s protocol $\mathsf{Pg}_i^{\mathcal{F}}$ are exactly those prescribed by $\mathsf{Pg}_i$ when the tests are evaluated in $M_{\mathcal{F}}$.

Consider an interpreted system $\mathcal{I} = (\mathcal{R}, \pi)$, where $\mathcal{R}$ is a system over a set $\mathcal{G}$ of global states, and $\pi$ is an interpretation for the propositions in $\Phi$ over $\mathcal{G}$. We use both $\mathcal{F}_{\mathcal{I}}$ and $\mathcal{F}_{\mathcal{R}}$ to denote the global states that occur in $\mathcal{R}$, i.e., $\mathcal{F}_{\mathcal{I}} = \mathcal{F}_{\mathcal{R}} = \{r(m) \mid r \in \mathcal{R}\}$. We can now characterize implementation of atemporal knowledge-based programs in nonrestrictive interpreted contexts.

**Proposition 3.1** *Let* $\mathsf{Pg}$ *be an atemporal knowledge-based program, let* $(\gamma, \pi)$ *be a nonrestrictive interpreted context, and let* $P$ *be a protocol. Then* $P$ *implements* $\mathsf{Pg}$ *in* $(\gamma, \pi)$ *iff* $P$ *coincides with* $\mathsf{Pg}^{\mathcal{F}_{\mathcal{R}}}$*, where* $\mathcal{R} = \mathbf{R}^{rep}(P, \gamma)$.

We can now obtain the desired complexity results for nonrestrictive finite-state interpreted contexts and atemporal knowledge-based programs.

**Theorem 3.2** *There is a polynomial-time algorithm for testing whether a given protocol implements a given atemporal knowledge-based program in a given nonrestrictive finite-state interpreted context.*

**Proof Sketch**:Let $\mathsf{Pg}$ be an atemporal finite knowledge-based program, let $(\gamma, \pi)$ be a nonrestrictive finite-state interpreted context, where $\gamma = (P_e, \mathcal{G}_0, \tau, \mathit{True})$, and let $P$ be a protocol. By Proposition 3.1, $P$ implements $\mathsf{Pg}$ in $(\gamma, \pi)$ iff $P$ coincides with $\mathsf{Pg}^{\mathcal{F}_{\mathcal{R}}}$, where $\mathcal{R} = \mathbf{R}^{rep}(P, \gamma)$.

To check that $P$ implements $\mathsf{Pg}$ in $(\gamma, \pi)$, our algorithm computes the set $\mathcal{F}_{\mathcal{R}}$ of global states and checks that it satisfies the condition of Proposition 3.1. We start with the set $\mathcal{G}_0$ of initial states and close it under the operation of the protocol $P$. That is, whenever the global state $g = (\ell_e, \ell_1, \ldots, \ell_n)$ is in $\mathcal{F}_{\mathcal{R}}$, we add to $\mathcal{F}_{\mathcal{R}}$ every state $\tau(\mathsf{a}_e, \mathsf{a}_1, \ldots, \mathsf{a}_n)(g)$, where $\mathsf{a}_e \in P_e(\ell_e)$ and $\mathsf{a}_i \in P_i(\ell_i)$. To check that $P$ coincides with $\mathsf{Pg}^{\mathcal{F}_{\mathcal{R}}}$ we have to check that

$P_i(\ell) = \mathsf{Pg}_i^{\mathcal{F}_\mathcal{R}}(\ell)$ for each agent $i$ and local state $\ell$. To compute $\mathsf{Pg}_i^{\mathcal{F}}(\ell)$ we need to evaluate the truth of knowledge tests of $\mathsf{Pg}_i$ in $M_\mathcal{F}$, but this can be done in polynomial time in the size of $\mathcal{F}$ and the size of the knowledge tests [HM92]. Thus, checking that $P$ coincides with $\mathsf{Pg}^{\mathcal{F}_\mathcal{R}}$ can be done in polynomial time. ∎

Thus, in the case of an atemporal knowledge-based program $\mathsf{Pg}$ and a nonrestrictive interpreted context $(\gamma, \pi)$, deciding whether a given protocol $P$ implements $\mathsf{Pg}$ in $(\gamma, \pi)$ can be decided in polynomial time (Proposition 3.2), whereas deciding whether this knowledge-based program is implemented by *some* protocol in this interpreted context is NP-complete ([FHMV95a]). So the first problem is tractable, while the second problem is not (assuming *P* $\neq$ *NP*).

The problem is considerably more involved when we allow temporal formulas. The difficulty stems from the fact that we can no longer collapse an interpreted system $\mathcal{I}$ to the Kripke structure $M_{\mathcal{F}_\mathcal{I}}$, while still preserving the relevant semantic information. $M_{\mathcal{F}_\mathcal{I}}$ preserves the semantics of knowledge, but does not preserve the temporal semantics. Since the knowledge tests in $\mathsf{Pg}$ may involve temporal operators, we cannot simply consider $\mathsf{Pg}^{\mathcal{F}_\mathcal{I}}$ instead of $\mathsf{Pg}^{\mathcal{I}}$.

We deal with this problem by considering *knowledge interpretations*, which tell us how to interpret knowledge tests in local states. Given a context $\gamma$ in which $L_i$ is the set of local states of agent $i$, for $i = 1, \ldots, n$, let $L_\gamma = L_1 \cup \cdots \cup L_n$. Let $\mathsf{Pg}$ be a knowledge-based program. Define $test(\mathsf{Pg})$ to be the set of subformulas of tests in $\mathsf{Pg}$ and their negations (we identify a formula $\neg\neg\psi$ with $\psi$). A knowledge interpretation $\kappa$ for $\mathsf{Pg}$ in $\gamma$ assigns to every local state $\ell \in L_\gamma$ and every formula $K_i\psi \in test(\mathsf{Pg})$ a truth value, i.e., $\kappa(\ell, K_i\psi) = \mathbf{true}$ or $\kappa(\ell, K_i\psi) = \mathbf{false}$.

Now consider a knowledge-based program $\mathsf{Pg}_i$ for agent $i$. Instead of using an interpreted system $\mathcal{I}$ to obtain a protocol $\mathsf{Pg}_i^{\mathcal{I}}$, we can associate a protocol $\mathsf{Pg}_i^{\kappa,\pi}$ with $\mathsf{Pg}_i$, with respect to a knowledge interpretation $\kappa$ and an interpretation $\pi$ that is compatible with $\mathsf{Pg}_i$. If $\varphi$ is a standard test, we define

$(\kappa, \pi, \ell) \models \varphi$ iff $(\pi, \ell) \models \varphi$.

If $\varphi$ is a knowledge test $K_i\psi$, we define

$(\kappa, \pi, \ell) \models K_i\psi$ iff $\kappa(\ell, K_i\psi) = \mathbf{true}$.

Finally, for conjunctions and negations, we follow the standard treatment.

We now define

$$\mathsf{Pg}_i^{\kappa,\pi}(\ell) = \begin{cases} \{\mathsf{a}_j : (\kappa, \pi, \ell) \models t_j \wedge k_j\} & \text{if } \{j : (\kappa, \pi, \ell) \models t_j \wedge k_j\} \neq \emptyset \\ \{\Lambda\} & \text{if } \{j : (\kappa, \pi, \ell) \models t_j \wedge k_j\} = \emptyset. \end{cases}$$

In addition to the notion of knowledge interpretation, we also need the notion of *annotated states*, which are global states tagged with sets of formulas. Let $g$ be a global state and let $\Theta$ be a subset of $test(\mathsf{Pg})$. The pair $(g, \Theta)$ is called an *annotated state*.

A set $\Theta \subseteq test(\mathsf{Pg})$ is *full* if the following holds:

1. For each $\varphi \in test(\mathsf{Pg})$, we have that $\varphi \in \Theta$ iff $\neg\varphi \notin \Theta$.

2. For each $\varphi_1 \wedge \varphi_2 \in test(\mathsf{Pg})$, we have that $\varphi_1 \wedge \varphi_2 \in \Theta$ iff $\varphi_1 \in \Theta$ and $\varphi_2 \in \Theta$.

An annotated state $(g, \Theta)$ is *consistent* with a knowledge interpretation $\kappa$ and an interpretation $\pi$ if (i) $\Theta$ is full, (ii) for each proposition $p \in \Phi$ we have that $p \in \Theta$ iff $\pi(g)(p) = \mathbf{true}$, and (iii) for each formula $K_i\psi \in test(\mathsf{Pg})$ we have that $K_i\varphi \in \Theta$ iff $\kappa(g_i, K_i\psi) = \mathbf{true}$. These conditions say that the annotations capture the standard semantics of propositions, of Boolean connectives, and of knowledge modalities. On the other hand, no constraint is imposed on the semantics of temporal operators.

To deal with the semantics of temporal operators we have to introduce the notion of *annotated runs*. An annotated run $\alpha$ over a set $\mathcal{F}$ of annotated states is a function from time to annotated states in $\mathcal{F}$ that satisfies the following condition: if $\alpha = (g^0, \Theta^0), (g^1, \Theta^1), \ldots$, then for each formula $\bigcirc\varphi \in test(\mathsf{Pg})$ or $\varphi\, U\, \psi \in test(\mathsf{Pg})$ we have:

1. $\bigcirc\varphi \in \Theta^m$ iff $\varphi \in \Theta^{m+1}$

2. $\varphi\, U\, \psi \in \Theta^m$ iff $\psi \in \Theta^{m'}$ for some $m' \geq m$ and $\varphi \in \Theta^{m''}$ for all $m''$ such that $m \leq m'' < m'$.

Thus, annotated runs have to display the "proper" temporal behavior. Given an annoted run $\alpha = (g^0, \Theta^0), (g^1, \Theta^1), \ldots$, let *run*$(\alpha)$ be the run $g^0, g^1, \ldots$ that is obtained by deleting the annotations in $\alpha$. An annotated run $\alpha$ is *consistent* with $(\kappa, \pi)$ if every annotated state in $\alpha$ is consistent with $(\kappa, \pi)$. An annotated run $\alpha$ is *consistent* with a joint protocol $P$ in a context $\gamma$ if *run*$(\alpha)$ is consistent with $P$ in $\gamma$.

We can now state a condition for implementation. We say that the knowledge interpretation $\kappa$ is *compatible with* $\mathsf{Pg}$ *in interpreted context* $(\gamma, \pi)$ if, for each local state $\ell \in L_\gamma$ and each formula $K_i\psi \in test(\mathsf{Pg})$, we have $\kappa(\ell, K_i\psi) = \mathbf{false}$ iff there is an annotated state $(g, \Theta)$ such that

- $g_i = \ell$ and $\neg\psi \in \Theta$, and

- $(g, \Theta)$ occurs in an annotated run that is consistent both with $(\kappa, \pi)$ and with $\mathsf{Pg}^{\kappa,\pi}$ in the context $\gamma$.

**Proposition 3.3** *Let* $\mathsf{Pg}$ *be a knowledge-based program, let* $(\gamma, \pi)$ *be an interpreted context, and let* $P$ *be a protocol. Then* $P$ *implements* $\mathsf{Pg}$ *in* $(\gamma, \pi)$ *iff there is a knowledge interpretation* $\kappa$ *that is compatible with* $\mathsf{Pg}$ *in* $(\gamma, \pi)$ *such that* $P$ *coincides with* $\mathsf{Pg}^{\kappa,\pi}$.

**Theorem 3.4** *Testing whether a given protocol implements a given knowledge-based program in a given finite-state interpreted context is PSPACE-complete.*

**Proof Sketch**:Let $\mathsf{Pg}$ be a finite knowledge-based program, and let $(\gamma, \pi)$ be a finite interpreted context, and let $P$ be a protocol. By Proposition 3.3, $P$ implements $\mathsf{Pg}$ in $(\gamma, \pi)$ iff there is a knowledge interpretation $\kappa$ that is compatible with $\mathsf{Pg}$ in $(\gamma, \pi)$ such that $P$ coincides with $\mathsf{Pg}^{\kappa,\pi}$. The algorithm guesses a knowledge interpretation $\kappa$ and then checks in polynomial space that it is compatible with $\mathsf{Pg}$ in $(\gamma, \pi)$ (this is done using the automata-theoretic techniques of [VW94]). Clearly, checking that $P$ coincides with $\mathsf{Pg}^{\kappa,\pi}$ can be done in polynomial time. This proves the upper bound. The lower bound is proven by a reduction from the satisfiability problem for linear temporal logic [SC85]. ∎

Thus, in contrast to the case of atemporal knowledge-based programs nonrestrictive interpreted contexts deciding whether a given protocol $P$ implements a general knowledge-based program $\mathsf{Pg}$ in a general interpreted context $(\gamma, \pi)$ is no easier than deciding whether this knowledge-based program is implemented by *some* protocol in this interpreted context: both problems are *PSPACE*-complete. This suggests that the use of temporal formulas in knowledge-based programming might be computationally quite expensive.

## References


[APP88] F. Afrati, C. H. Papadimitriou, and G. Papageorgiou. The synthesis of communication protocols. *Algorithmica*, 3(3):451–472, 1988.

[Bar81] J. Barwise. Scenes and other situations. *Journal of Philosophy*, 78(7):369–397, 1981.

[BLMS94] R. Brafman, J.-C. Latombe, Y. Moses, and Y. Shoham. Knowledge as a tool in motion planning under uncertainty. In R. Fagin, editor, *Theoretical Aspects of Reasoning about Knowledge: Proc. Fifth Conference*, pages 208–224. Morgan Kaufmann, San Francisco, Calif., 1994.

[BLS93] R. I. Brafman, J.-C. Latombe, and Y. Shoham. Towards knowledge-level analysis of motion planning. In *Proc. National Conference on Artificial Intelligence (AAAI '93)*, 1993.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, 1986.

[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.

[FHMV95a] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. Knowledge-based programming. In *Proc. 14th ACM Symp. on Principles of Distributed Computing*, pages 153–163, 1995.

[FHMV95b] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. MIT Press, Cambridge, Mass., 1995.

[GJ79] M. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. Freeman and Co., San Francisco, Calif., 1979.

[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.

[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.

[HF89] J. Y. Halpern and R. Fagin. Modelling knowledge and action in distributed systems. *Distributed Computing*, 3(4):159–179, 1989.

[HM90] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, 1990.

[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.

[HMW90] J. Y. Halpern, Y. Moses, and O. Waarts. A characterization of eventual Byzantine agreement. In *Proc. 9th ACM Symp. on Principles of Distributed Computing*, pages 333–346, 1990.

[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.

[Liu89] M.T. Liu. Protocol engineering. *Advances in Computing*, 29:79–195, 1989.

[Maz91] M. S. Mazer. Implementing distributed knowledge-based protocols. Submitted for publication, 1991.

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

[NB92] G. Neiger and R. Bazzi. Using knowledge to optimally achieve coordination in distributed systems. In Y. Moses, editor, *Theoretical Aspects of Reasoning about Knowledge: Proc. Fourth Conference*, pages 43–59. Morgan Kaufmann, San Francisco, Calif., 1992.

[NT93] G. Neiger and M. R. Tuttle. Common knowledge and consistent simultaneous coordination. *Distributed Computing*, 6(3):334–352, 1993.

[Rub89] A. Rubinstein. The electronic mail game: strategic behavior under "almost common knowledge". *American Economic Review*, 79:385–391, 1989.

[Rud87] H. Rudin. Network protocols and tools to help produce them. *Annual Review of Computer Science*, 2:291–316, 1987.

[SC85] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

[VW94] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.