Knowledge-Based Programs

Ronald Fagin*
IBM Almaden Research Center

Joseph Y. Halpern‡
IBM Almaden Research Center

Yoram Moses†
The Weizmann Institute of Science

Moshe Y. Vardi§
Rice University

Abstract: Reasoning about activities in a distributed system at the level of *knowledge* allows us to abstract away from many irrelevant details. We introduce two notions to facilitate designing and reasoning about systems in terms of knowledge. The first is *knowledge-based programs*, which improve upon Halpern and Fagin's definition of *knowledge-based protocols*. Knowledge-based programs are syntactic objects: programs with tests for knowledge. The second notion is *contexts*, which capture the setting in which a program is executed. In a given context, a standard program (one without tests for knowledge) is *represented by* (i.e., corresponds in a precise sense to) a unique system. A knowledge-based program, on the other hand, may be represented by no system, one system, or many systems. We provide a condition that is sufficient to guarantee that a knowledge-based program is represented by a unique system, and covers many of the knowledge-based programs considered in the literature. We also characterize the complexity of determining whether a given knowledge-based program has a unique representation, or any representation at all, in a *finite-state* context.

* Address: IBM Almaden Research Center, Dept. K53/802, 650 Harry Road, San Jose, CA 95120–6099, fagin@almaden.ibm.com.

† Work supported in part by a Helen and Milton A. Kimmelman career development chair. Address: Department of Applied Math. and CS, The Weizmann Institute of Science, 76100 Rehovot, Israel, yoram@cs.weizmann.ac.il.

‡ Work supported in part by the Air Force Office of Scientific Research (AFSC), under Contract F49620-91-C-0080. Address: IBM Almaden Research Center, Dept. K53/802, 650 Harry Road, San Jose, CA 95120–6099, halpern@almaden.ibm.com.

§ Part of this research was done while this author was at the IBM Almaden Research Center. Address: Department of Computer Science, Rice University, P.O. Box 1892, Houston, TX 77251-1892, vardi@cs.rice.edu.

1 Introduction

Reasoning about activities in a distributed computer system at the level of the knowledge allows us to abstract away from many irrelevant details. 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 a different communication topology or different guarantees about the reliability of various components of the system.

Motivated by these considerations, Halpern and Fagin [HF85] suggested a notion of *knowledge-based protocols*, in which an agent's actions depend explicitly on the agent's knowledge. Their goal was to provide a formal semantics for programs with tests for knowledge such as

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

where $K(x = 0)$ should be read as "you know that $x = 0$". Unfortunately, the technical definition of knowledge-based protocols given in [HF85] (later simplified somewhat in [HF89, Maz91, NT93]) had a number of deficiencies, which made it somewhat difficult to use as a tool for program design. For one thing, knowledge-based protocols were defined as functions from local states and systems to actions. Thus, the definition did not directly capture the intuition that knowledge-based programs were meant to be programs with tests for knowledge. Moreover, the approach did not provide a clean distinction between the protocol (or program) and the setting in which it is to be executed. As a result, the high-level and model-independent reasoning we wish to use knowledge for was not facilitated by the definition as much as it perhaps could have been. Nevertheless, knowledge-based protocols did provide formal justification for the technical analyses in papers such as [DM90, HZ92, MT88, NB92, HMW90].

In this paper, we attempt to overcome the deficiencies of the earlier definition. We start by introducing *knowledge-*

based programs, which are what knowledge-based protocols were intended to be: (syntactic) programs with tests for knowledge. We then introduce *contexts*, which capture the setting in which a program is to be executed. By distinguishing between programs and contexts, and by ascribing meaning to programs in different contexts in a uniform manner, we facilitate high-level and model-independent reasoning based on knowledge.

In a given context, a *standard* program (one without tests for knowledge) is *represented by* (i.e., corresponds in a precise sense to) a unique system. Intuitively, the system consists of all the possible executions of the program. A knowledge-based program, on the other hand, may be represented by no system, one system, or many systems. A knowledge-based program should be viewed as a high-level specification; a system satisfies the specification if it represents the knowledge-based program. If there are no systems representing the program, then the specification is inconsistent (at least, in the given context); if there is more than one, that simply means that the specification can be satisfied by more than one system.

Of course, if we are to program at the knowledge level, it is surely useful to be able to tell whether there is a system representing a given knowledge-based program, and if so, whether this system is unique. One of our main results is a condition that is sufficient to guarantee that a knowledge-based program is represented by a unique system, and covers many of the simple knowledge-based programs considered in the literature. We also completely characterize the complexity of determining whether a given knowledge-based program has a unique representation, or any representation at all, in a given *finite-state* context. More details on the technical results can be found in the full version of the paper; the examples and motivation are discussed in far greater detail in [FHMV95].

2 Multi-Agent Systems

We briefly review the framework of [HF89, HM90] for modeling multi-agent systems. We assume that at each point in time, each agent is in some *local state*. Informally, this local state encodes the information available to the agent. In addition, there is 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 ℓ_e and the local state ℓ_i of each agent i. 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. A *round* takes place between two time points. We define round m in run r to take place between time $m-1$ and time m. It is often convenient for us to view an agent as performing an *action* during a round. (We discuss actions in Section 3.) 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.

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. Finally, we define an *interpreted system* to be a pair $(\mathcal{R}, \pi)$ consisting of a system $\mathcal{R}$ together with a mapping π that associates a truth value with each primitive proposition (in a set Φ of primitive propositions) at each global state.

We can give semantics to formulas with respect to points in an interpreted system in a straightforward way. In particular, following [HM90, FHMV95], 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 φ precisely if φ 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; see [HV89].

3 Protocols and Programs

Intuitively, a *protocol* for agent i is a description of what actions agent i may take, as a function of her 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 to nonempty subsets of a set ACT_e. For both the agents and the environment, we allow for the possibility of a special *null* action Λ. 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 her local state. Thus, what an agent does can depend only on her 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 while we did include the environment's action in a joint action, 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 to describe the setting, or *context*, in which P is being run. Formally, a context γ 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, τ is a *transition function*, and Ψ is a set of *admissible* runs. The role of P_e and $\mathcal{G}_0$ should be clear; we now explain the role of τ and Ψ.

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 τ characterizes that. Define a *joint action* to be a tuple of the form $(\mathsf{a}_e, \mathsf{a}_1, \ldots, \mathsf{a}_n)$, where a_e is an action performed by the environment and 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. (Note that we have chosen to model the observable aspects of systems in terms of *states*; the role of *actions* is only to cause state change. Of course, in many applications one may choose to make the actions observable by reflecting them in the states; for example, the environment's state can include the history of all actions performed thus far.)

Given P_e, P, τ, 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 condition Ψ in the context. Often the condition Ψ can be characterized by one (or a collection of) formulas of temporal logic, and the runs in Ψ are those that satisfy these formulas.

When describing a context, we often have some flexibility in our choice of the components of the context. We typically think of $\mathcal{G}_0$ as describing the initial conditions, while τ and P_e describe the system's local behavior, and Ψ describes all other relevant aspects of the environment's behavior. To describe the behavior of the system we have to decide what the actions performed by the environment are (this is part of P_e) and how these actions interact with the actions of the agents (this is described by τ). There is often more than one way in which this can be done. Although we motivated the admissibility condition Ψ by the need to be able to capture global aspects of the environment's behavior, we have put no constraints on the condition Ψ. As a result, it is possible (although not advisable) to place much of the burden of determining the initial conditions and local behavior of the environment on the choice of Ψ. Thus, for example, we could do away with $\mathcal{G}_0$ altogether, and have Ψ consist only of runs r whose initial state would have been in $\mathcal{G}_0$. In a well-chosen context we would expect the components to be orthogonal. In particular, we expect that P_e would specify local aspects of the environment's protocol, and that Ψ would capture the more global properties of the environment's behavior over time (such as "all messages are eventually delivered").

It is only in a context that a joint protocol describes the behavior of a system. The combination of a context γ 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 τ from $r(m)$ by a joint action prescribed by P_e and P in $r(m)$, for $0 \leq m$, and $r \in \Psi$. (We defer the formalities of the definition to the full paper.) We define $\mathbf{R}^{rep}(P, \gamma)$ to be the system consisting of all runs of P in context γ, and call it the *system representing protocol P in context γ*.

In many cases we have a particular collection Φ of primitive propositions and a particular interpretation π for the propositions in Φ 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 (γ, π) consisting of a context γ and an interpretation π. 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 (γ, π)*.

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** 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** a_1
  if t_2 **do** a_2
  $\dots$
end case

where the t_j's are *standard tests* for agent i and the 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 introduce later.) In principle, we allow such a program to consist of an infinite number of clauses; typically, of course, we are interested in *finite* programs, i.e., programs with finitely many clauses. A standard test for agent i is simply a propositional formula over a set Φ_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 ℓ, agent i nondeterministically chooses one of the clauses in the **case** statement whose test is true at ℓ, and executes the corresponding action.

We want to use an interpretation π to tell us how to evaluate the tests. However, not just any interpretation 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 π on the global states in $\mathcal{G}$ is *compatible* with a program Pg_i for agent i (with respect to a set $\mathcal{G}$ of global states) if every proposition that appears in Pg_i is *local* to i, that is, if q appears in Pg_i, the states g and g' are in $\mathcal{G}$, and $g \sim_i g'$, then $\pi(g)(q) = \pi(g')(q)$. If φ is a propositional formula all of whose primitive propositions are local to agent i and ℓ is a local state of agent i, then we write $(\pi, \ell) \models \varphi$ if φ is satisfied by the truth assignment $\pi(g)$, where $g = (\ell_e, \ell_1, \dots, \ell_n)$ is a global state such that $\ell_i = \ell$. Since all the primitive propositions in φ are local to i, it does not matter which global state g we choose, as long as i's local state in g is ℓ. Given a program Pg_i for agent i and an interpretation π compatible with Pg_i, we define a protocol that we denote Pg_i^π by setting:

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

Intuitively, Pg_i^π selects all actions from the clauses that satisfy the test, and selects the null action Λ if no test is satisfied. In general, we get a nondeterministic protocol, since more than one test may be satisfied at a given state.

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 Pg_i is a program for agent i. We say that an interpretation π is *compatible* with Pg if π is compatible with each of the Pg_i's. From Pg and π 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 Pg in the interpreted context (γ, π) exactly if π is compatible with Pg, and $\mathcal{I}$ represents the corresponding protocol Pg^π. We denote the interpreted system representing Pg in (γ, π) by $\mathbf{I}^{rep}(\mathsf{Pg}, \gamma, \pi)$. Of course, this definition makes sense only if π is compatible with Pg. From now on we always assume that this is the case.

4 Knowledge-based programs

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 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, 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** a_1
  if $t_2 \wedge k_2$ **do** a_2
  ...
end case

where the t_j's are standard tests, the k_j's are *knowledge tests* for agent i, and the 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 φ can be an arbitrary formula that may include other modal operators, including knowledge of other agents, 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 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.

Although the idea of a knowledge-based program was implicit in the discussion in [HF85], the first formal definition seems to have been given by Kurki-Suonio [Kur86] and by Shoham [Sho93]. Kurki-Suonio and Shoham, however, did not work with interpreted systems. Rather, they assumed that an agent's knowledge was explicitly encoded in her local state (and thus, in our terminology, was independent of the interpreted system). This means that their knowledge-based programs are really more like our standard programs, although some of the tests in their programs are intuitively thought of as tests for knowledge.

Example 4.1 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 sender S can be viewed as running the following program BT_S:

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

Similarly, the receiver R can be viewed as running the following program BT_R:

if *recbit* **do** sendack.

The proposition *recbit* (resp., *recack*) is assumed to hold when the bit (resp., the acknowledgement) has been received.

It is useful to analyze the bit-transmission problem in terms of the processes' knowledge. 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 BT'_S:

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

The advantage of this program over the standard program BT_S 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.

Now consider the receiver R, which keeps sending acknowledgements after it receives the bit. Surely, if R *knew* that S received the acknowledgement, then R would stop sending it. Thus, it makes sense to replace the standard program BT_R by the knowledge-based program BT'_R:

if $\mathit{recbit} \wedge \neg K_R(\mathit{recack})$ **do** sendack.

The advantage of this program is that it allows fewer messages to be sent in many cases. For example, if messages from R to S are guaranteed to be delivered in the same round in which they are sent, then R needs to send only one 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. The reason that R keeps sending the acknowledgment to S is that R wants S to know that R knows 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 BT''_S:

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

and the receiver's behavior by the knowledge-based program BT''_R:

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

This program abstracts away the manner in which S learns that R knows the value of the bit and the manner in which R learns that S knows 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, and R need not send any acknowledgment. Furthermore, if the value of the bit is common knowledge, then BT''_S does not require S to send any messages. In contrast, BT and BT'

require S to send messages even in such contexts. 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 ℓ 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 ℓ in $\mathcal{I}_1$ and false at the local state ℓ 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 Pg according to π, and evaluate the knowledge tests in Pg according to $\mathcal{I}$. As in the case of standard programs, we require that π be compatible with Pg, that is, that every proposition appearing in a standard test in 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 ℓ of agent i. To define this, we first define when a test φ holds in a local state ℓ with respect to an interpreted system $\mathcal{I}$, denoted $(\mathcal{I}, \ell) \models \varphi$.

If φ is a standard test and $\mathcal{I} = (\mathcal{R}, \pi)$ then, in analogy to Section 3, we define $(\mathcal{I}, \ell) \models \varphi$ if $(\pi, \ell) \models \varphi$. Since φ is a standard test in Pg_i, it must be local to agent i, so this definition makes sense. If φ is a knowledge test of the form $K_i\psi$, we define

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

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

We can now define

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

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

Roughly speaking, the knowledge-based protocols of [HF89] bear the same relation to knowledge-based programs as protocols bear to standard programs. Formally, a knowledge-based protocol is defined in [HF89] to be a function from local states and interpreted systems to actions. We can associate a knowledge-based protocol with a knowledge-based program in an obvious way: Given a knowledge-based program Pg, we can associate with it the knowledge-based protocol P such that $P(\ell, \mathcal{I}) = \mathsf{Pg}^{\mathcal{I}}(\ell)$ for all local states ℓ. Knowledge-based protocols can be viewed as an intermediate step between knowledge-based programs and protocols. We find it convenient here to go directly from knowledge-based programs to (standard) protocols, skipping this intermediate step. Thus, we do not deal with knowledge-based protocols in this paper.

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* Pg in (γ, π) if π is compatible with Pg and if $\mathcal{I}$ represents $\mathsf{Pg}^{\mathcal{I}}$ in (γ, π). This means that to check if $\mathcal{I}$ represents Pg, we check if $\mathcal{I}$ represents the protocol obtained by evaluating the knowledge tests in Pg with respect to $\mathcal{I}$ itself. Because of the circularity of the definition, it is not necessarily the case that there is a unique interpreted system representing a knowledge-based program Pg in a given interpreted context. There may be more than one, or there may be none.

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 NU (for "not unique"):

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

According to NU, agent 1 sets the bit to 1 if she knows that eventually the bit is 1, and otherwise does nothing. It should be clear that there are two ways that agent 1 could be consistent with the program: either by never setting the bit to 1, or by setting the bit to 1 in the first round. Thus, this program is not represented by a unique interpreted system. By way of contrast, there can be at most one interpreted system that represents a standard program. It is not hard to show that the program

if $K_1(\neg\Diamond(bit = 1))$ **do** $bit := 1$

is not represented by *any* interpreted system.

This behavior should not be viewed as a problem. Rather, it suggests that knowledge-based programs are best thought of as specifications. Of course, it is then important to understand when the specification is satisfiable. Of special interest is the case in which there is a unique system representing the program in a given context. This is the subject of the next section.

5 Unique Representation

As we have seen, a knowledge-based program does not always have a unique representation in a given context. In many situations, however, there is a strong intuition that a knowledge-based program does completely describe the behavior of the agents, and consequently, the program ought to be represented by a unique interpreted system. For example, in the case of the muddy children puzzle, we expect the behavior of the children following the knowledge-based program MC (discussed briefly at the beginning of Section 4) to be uniquely determined. This also seems to be the case for the programs BT' and BT'' for the bit-transmission problem. When a knowledge-based program is guaranteed to have a unique representation in a given context, we can think of this program as describing a well-defined protocol. This is often desirable when we want to use knowledge-based programs to design a protocol that will solve a particular problem in the context, as in [DM90, HZ92, MT88, HMW90]. We now discuss conditions under which a unique representation is guaranteed.

Why do we feel that there should be a unique interpreted system representing MC? Intuitively, it is because, once we fix the initial set of states, we can start running the program step by step, generating the run as we go. Suppose we can generate all prefixes of runs through time m. Once we have all prefixes through time m, at any given point (r, m), the children in that situation can determine whether they do indeed know whether their own forehead is muddy, and thus can take the appropriate action at the next step. This allows us to generate all prefixes through time $m + 1$.

The key reason that this idea works is that the prefixes that we have already constructed are sufficient to determine the truth of the knowledge tests in the children's program. In general, this might not be the case. To understand why, suppose we have a knowledge-based program $\mathsf{Pg} = (\mathsf{Pg}_1, \ldots, \mathsf{Pg}_n)$, and Pg_i includes a test such as $K_i\varphi$. Suppose that we have indeed constructed all the prefixes of runs of Pg through time m. For agent i to know what actions to perform next at a point (r, m), the knowledge test $K_i\varphi$ has to be evaluated. As long as this can be done solely by considering points of the form (r', m') with $m' \leq m$—intuitively, these are the points we have already constructed—then there is no problem. If, on the other hand, φ is a temporal formula such as the formula $\Diamond(bit = 1)$ that appears in the program NU in Section 4, then we may not be able to evaluate the truth of φ in the prefixes we have constructed thus far. Even if φ is a nontemporal formula, there may be a problem. For example, suppose the time m is encoded in the environment's state and φ is the formula $m \leq 1$, which is true at all time m points with m less than or equal to 1. Then $K_1(m \leq 1)$ may be false at a point $(r, 1)$ if agent 1 does not know the time, i.e., if $(r, 1) \sim_1 (r', k)$ for some point (r', k) where $k > 1$. Note, however, that there is no point that occurs in a prefix through time 1 and "witnesses" the fact that $K_1(m \leq 1)$ fails at $(r, 1)$, since the formula $m \leq 1$ is true at all points of the form $(r', 0)$ or $(r', 1)$. This discussion suggests that to make the inductive construction work, if a test $K_i\varphi$ in the program is false, there must be a "witness" to its falsity in some prefix we have already constructed.

Subtle problems can arise in the interaction between the assumption that witnesses are provided in the sense that we have just described (informally) and the admissibility condition Ψ on runs in a context. Suppose we are interested in running the knowledge-based program Pg in the interpreted context (γ, π), where $\gamma = (P_e, \mathcal{G}_0, \tau, \Psi)$. What should the system representing Pg be? Intuitively, it should consist of all runs in Ψ whose prefixes arise in the inductive construction. Even if, at every step of the inductive construction, the prefixes constructed provide witnesses, it is possible that the admissibility condition Ψ does not include a run with a prefix ρ that arises in the construction. This means that we cannot include a run with prefix ρ in the system. This, in turn, might mean that a witness that we counted on in the course of the inductive construction may not occur in the system, thus invalidating our evaluation of the tests.

In the full paper we show that there is a unique system representing Pg if "witnesses are provided" and if the context is "reasonable". The detailed definitions of these notions are left to the full paper (see also [FHMV95]). We shall describe them here only at an intuitive level. Roughly speaking, we say that an interpreted context (γ, π) *provides witnesses* for a program Pg if, for every system of the form $\mathcal{I} = \mathbf{I}^{rep}(\mathsf{Pg}^{\mathcal{I}'}, \gamma, \pi)$ and formula $K_i\varphi$ appearing in a knowledge test in Pg_i, if $(\mathcal{I}, r, m) \models \neg K_i\varphi$, then $(\mathcal{I}, r', m') \models \neg\varphi$ for some point (r', m') in I such that $m' \leq m$ and $(r', m') \sim_i (r, m)$. One case of special interest in which (γ, π) is guaranteed to provide witnesses for a program Pg is when every system of the form $\mathcal{I} = \mathbf{I}^{rep}(\mathsf{Pg}^{\mathcal{I}'}, \gamma, \pi)$ is synchronous. (We define a system $\mathcal{I}$ to be synchronous if for every process i, if $(r, m) \sim_i (r', m')$ then $m = m'$; intuitively, in synchronous systems all processes have access to a global clock.) To capture the idea that a context is reasonable, we define the notion of a *nonexcluding* context. Roughly speaking, in a nonexcluding context γ, for every prefix that is consistent with a given protocol P, there is a run of P in γ that extends this prefix. (We think of the context as not excluding any prefix from being extended into a run; hence the term "nonexcluding".) Contexts are nonexcluding in all but pathological cases. In particular, contexts in which the condition Ψ consists of all runs are guaranteed to be nonexcluding. The same is true for contexts in which Ψ makes only standard fairness requirements, and the environment's protocol P_e does not explicitly disallow such fair behavior to take place. We get the following theorem.

Theorem 5.1 *Let* Pg *be a knowledge-based program in which the tests do not involve temporal operators, let γ be a nonexcluding context, and assume that the context (γ, π) provides witnesses for* Pg*. Then there is a unique interpreted system $\mathbf{I}^{rep}(\mathsf{Pg}, \gamma, \pi)$ representing* Pg *in (γ, π).*

While the conditions for uniqueness stated in Theorem 5.1 may seem cumbersome, they are in fact applicable to many cases of interest, and can often be verified in a fairly straightforward manner. In particular, they apply to the muddy children puzzle, as well as to the knowledge-based programs implicitly used in [DM90, HMW90, HZ92, MT88]. We remark that a similar condition was used in [HF89] to show that a knowledge-based protocol was represented by a system that was canonical in a certain sense. The intuitions behind these two results are quite similar, although the technical details of the proofs are quite different. Our result is more general than that of [HF89], and is easier to apply.

Example 5.2 We now apply our framework to an extension of the bit-transmission problem called the *sequence-transmission problem*. The sequence-transmission problem is a standard problem of data communication. Just as in the bit-transmission problem, we have a *sender* S and a *receiver* R. The sender S has an input tape with an infinite sequence $X = \langle x_0, x_1, \ldots \rangle$ of data elements. S reads these data elements and tries to transmit them to R. R must write these data elements onto an output tape. We would like a solution that satisfies (a) at any time the sequence of data elements written by R is a prefix of X and (b) every data element x_i in the sequence X is eventually written by R (or more precisely, every finite prefix of X is eventually a prefix of the data elements written by R).

The sequence-transmission problem over a faulty communication medium has been extensively studied in the literature, and solutions for several communication models have been proposed. These solutions were all designed individually, on an *ad hoc* basis [BSW69, Ste76, AUWY82]. As shown in [HZ92], we can derive all these solutions from one simple knowledge-based solution. We now recast the discussion of [HZ92] in our framework, and then compare to what was actually done there.

The idea is straightforward: S reads the i^{th} data element, and repeatedly sends it to R until S knows (a) that R has received it and (b) that R knows that it is the i^{th} element. At that point, S reads the $(i+1)^{\text{st}}$ element, and so on. R writes the data elements as it learns about them, and it then requests that S send the next data element. In Figure 1, we describe a knowledge-based program ST that captures this intuition.

In this program, we have assumed for simplicity that the input sequence consists only of 0's and 1's, although we could easily deal with any finite data domain. The variable y refers to the most recent data element read and is initialized with the first data element; i is a counter, initialized to 1, that keeps track of how many data elements have been read by S; and j is a counter, initialized to 0, that keeps track of the number of data elements written by R. The effect of the "read" action in S's program is to read the value of the current data element (where "current" is determined by the counter i). The effect of the "write(0)" and "write(1)" actions in R's program is to write an output value.

S's program ST_S:

case of
  if $\neg K_S K_R(x_{@i})$ **do** $\mathsf{send}(\langle i, y\rangle)$
  if $K_S K_R(x_{@i})$ **do** $i := i + 1$; read
end case

R's program ST_R:

case of
  if $\neg K_R(x_j)$ **do** $\mathsf{send}(j)$
  if $K_R(x_j = 0)$ **do** $\mathsf{write}(0)$; $j := j + 1$
  if $K_R(x_j = 1)$ **do** $\mathsf{write}(1)$; $j := j + 1$
end case

Figure 1: The knowledge-based program ST

Generalizing what we did in the knowledge-based program for the bit-transmission problem (Example 4.1), we take $K_R(x_i)$ to be an abbreviation for $K_R(x_i = 0) \vee K_R(x_i = 1)$. Thus, if $K_R(x_i)$ holds, then R knows the value of x_i. Recall that S sends the i^{th} data element until it knows that R knows the value of this data element. Intuitively, this suggests that S should test if $K_S K_R(x_i)$ holds. As pointed out in [HZ92], there is a subtle problem with this intuition, caused by the fact that i is a variable local to S, and, consequently, its value may not be known to R. For example, suppose we are at a point where $i = 3$. What S really wants to do is to continue sending the value of x_3 until $K_S K_R(x_3)$ holds. This is not the same as sending it the value of x_3 until $K_S K_R(x_i)$ holds. The problem is that R may know the value of x_3 without knowing the value of x_i (or, for that matter, without even knowing the value of i), since the variable i may take on different values in the global states that R considers possible. To achieve the appropriate effect, we take a clause such as "**if** $K_S K_R(x_{@i})$ **do** ..." to be an abbreviation for the infinite collection of clauses "**if** $i = k \wedge K_S K_R(x_k)$ **do** ...," for $k = 0, 1, \cdots$.

It should be clear—at least at an intuitive level—that the knowledge-based program ST does what we want: S sends $\langle i, y\rangle$, where y is the i^{th} bit, as long as S does not know that R knows the i^{th} bit. R's $\mathsf{send}(j)$ can be interpreted both as an acknowledgment for receiving the $(j-1)^{\text{st}}$ bit and as a request for the j^{th} bit.

In [HZ92] a knowledge-based protocol for the sequence-transmission problem was presented that is very similar in spirit to ST. Indeed, the protocol was initially described using notation much like that we use for ST. However, because there was no formal notion of a knowledge-based program, the description could only be informal; the informal description then had to be translated (somewhat painfully) to the corresponding knowledge-based protocol. Here, programs are first-class citizens, so we can analyze the program directly. Next, by using different contexts, we can analyze what happens when we run this program in settings where, for example, messages may be corrupted or messages may be duplicated. Finally, to do a careful analysis of the program, we want to show that in contexts of interest, the program is represented by a unique system. As we show in [FHMV95], this follows immediately from Theorem 5.1, since the contexts of interest can easily be shown to be nonexcluding and to provide witnesses for ST. Using these observations, we can then reprove the key results of [HZ92], which essentially show that ST (or, more precisely, the knowledge-based protocol corresponding to it) is correct in a variety of contexts. By working at the level of programs rather than protocols, we can do this in a more natural way. (We suggest that the reader compare the treatments of the sequence-transmission problem in [HZ92] and in [FHMV95].) ∎

6 Representation and Uniqueness in Finite-State Systems

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 Ψ 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].

We now consider the problem of checking whether a given finite knowledge-based program has some (resp. a unique) representation in a given finite-state interpreted context. It is not hard to show that the problem involves at the very least checking whether the admissibility condition Ψ is satisfiable. Satisfiability testing is known to be *PSPACE*-complete for (linear time) temporal logic [SC85], so our problem is at least *PSPACE*-hard. We can also prove the matching upper bound.

Theorem 6.1 *Testing whether there is a unique (resp. any) interpreted system representing a given finite knowledge-based program in a given finite-state interpreted context is PSPACE-complete.*

As we suggested above, the lower bound in Theorem 6.1 is proved by a straightforward reduction to the problem of testing satisfiability for linear time temporal logic. Suppose we remove all vestiges of temporal logic from the problem. Do things become any easier? As we now show, they do. A *simple* interpreted context is one where the admissibility condition on runs is $True$; a *simple* knowledge-based program is one where the tests are knowledge formulas, i.e., they do not involve temporal operators.

Theorem 6.2 *Testing whether there is any interpreted system representing a given simple finite knowledge-based program in a given simple finite-state interpreted context is NP-complete. Testing whether there is a unique interpreted system representing a given simple finite knowledge-based program in a given simple finite-state interpreted context is co-NP-complete.*

In the remainder of this section, we sketch the proof of Theorem 6.2 and discuss the modifications required to prove Theorem 6.1. Consider an interpreted system $\mathcal{I} = (\mathcal{R}, \pi)$. Let $states(\mathcal{R})$ be the set of global states that occur in $\mathcal{R}$, i.e, $states(\mathcal{R}) = \{r(m) : r \in \mathcal{R}\}$, and suppose $\mathcal{G} = states(\mathcal{R})$. Let $\mathcal{M}_{\mathcal{G}} = (\mathcal{G}, \mathcal{K}_1, \ldots, \mathcal{K}_n, \pi)$ be the Kripke structure such that $(g, g') \in \mathcal{K}_i$ iff $g_i = g'_i$, that is, if g and g' agree on their ith component. Truth of knowledge formulas in $\mathcal{M}_{\mathcal{G}}$ can now be defined in the standard way (cf. [HM92]). It is easy to prove by induction on the structure of knowledge formulas that $\mathcal{M}_{\mathcal{G}}$ completely captures the semantics of knowledge formulas in $\mathcal{I}$.

Lemma 6.3 *Let φ be a knowledge formula. Then $(\mathcal{I}, r, m) \models \varphi$ iff $(\mathcal{M}_{\mathcal{G}}, r(m)) \models \varphi$.*

Now consider a simple knowledge-based program 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 $\mathsf{Pg}_i^{M_{\mathcal{G}}}$ with Pg_i and $M_{\mathcal{G}}$, by evaluating the knowledge tests of Pg_i in $\mathcal{M}_{\mathcal{G}}$ instead of in $\mathcal{I}$. We can now state a condition for the existence of representations for simple knowledge-based programs with respect to simple interpreted contexts.

Proposition 6.4 *Let Pg be a simple knowledge-based program and let (γ, π) be a simple interpreted context. There is an interpreted system that represents Pg in (γ, π) iff there is a subset $\mathcal{F}$ of global states that contains $\mathcal{G}_0$ such that $\mathcal{F} = states(\mathcal{R})$, where $\mathcal{R} = \mathbf{R}^{rep}(\mathsf{Pg}^{\mathcal{M}_{\mathcal{F}}}, \gamma)$, i.e, $\mathcal{F}$ is precisely the set of states that occur in the system that represents $\mathsf{Pg}^{\mathcal{M}_{\mathcal{F}}}$ in γ. Furthermore, there is a unique interpreted system that represents Pg in (γ, π) iff there is unique such $\mathcal{F}$.*

With these results in hand, it is not hard to prove Theorem 6.2. Let Pg be a simple finite knowledge-based program and let (γ, π) be a simple finite-state interpreted context, where $\gamma = (P_e, \mathcal{G}_0, \tau, True)$. By Proposition 6.4, there is an interpreted system that represents Pg in (γ, π)

iff there is a subset $\mathcal{F}$ of global states that contains $\mathcal{G}_0$ such that $\mathcal{F} = states(\mathcal{R})$, where $\mathcal{R} = \mathbf{R}^{rep}(\mathsf{Pg}^{\mathcal{M}_\mathcal{F}}, \gamma)$.

Our algorithm guesses a subset $\mathcal{F}$ of global states and checks that it satisfies the condition. To that end, we need to compute the set $\mathcal{F}' = states(\mathbf{R}^{rep}(\mathsf{Pg}^{\mathcal{M}_\mathcal{F}}, \gamma))$. We start with the set $\mathcal{G}_0$ of initial states and close it under the operation of the protocol $\mathsf{Pg}^{\mathcal{M}_\mathcal{F}}$. That is, whenever the global state g is in $\mathcal{F}'$, we add to $\mathcal{F}'$ the state $\tau(\mathsf{a}_e, \mathsf{a}_1, \ldots, \mathsf{a}_n)(g)$, where $\mathsf{a}_e = P_e(g)$ and $\mathsf{a}_i = \mathsf{Pg}_i^{\mathcal{M}_\mathcal{F}}(g_i)$. To compute a_i we need to evaluate the truth of knowledge tests of Pg_i in $\mathcal{M}_\mathcal{F}$, 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 $\mathcal{F} = \mathcal{F}'$ can be done in polynomial time.

To check that there is more than one interpreted system that represents Pg in (γ, π), the algorithm simply guesses two sets $\mathcal{F}_1$ and $\mathcal{F}_2$ and check that they both satisfy the condition and that they are different. This proves that both the existence problem and the non-uniqueness problem are in NP. It follows that the uniqueness problem is co-NP. The lower bound proofs are left to the full paper.

The problem is considerably more involved for non-simple contexts and programs, when we allow temporal formulas. The first difficulty stems from the fact that we can no longer collapse an interpreted system $\mathcal{I}$ over a set $\mathcal{G}$ of global states to the Kripke structure $\mathcal{M}_\mathcal{G}$ while preserving the relevant semantic information as in Lemma 6.3, since $\mathcal{M}_\mathcal{G}$ preserves only the semantics of knowledge, but does not preserve the temporal semantics. Since the knowledge tests in Pg may involve temporal formulas, we cannot simply consider $\mathsf{Pg}^{\mathcal{M}_\mathcal{F}}$ instead of $\mathsf{Pg}^{\mathcal{I}}$. Even when all the tests in Pg are knowledge formulas, we still have to deal with the fact that Ψ is defined by a temporal formula. This means that we cannot compute the states of $\mathcal{F}' = states(\mathbf{R}^{rep}(\mathsf{Pg}^{\mathcal{M}_\mathcal{F}}, \gamma))$ by simply emulating the protocol $\mathsf{Pg}^{\mathcal{M}_\mathcal{F}}$ step by step in the proof of Theorem 6.2, since we have to make sure that these states lie along runs that satisfy Ψ.

We use two techniques to overcome these problems. First we show that by annotating states with sets of formulas we can collapse $\mathcal{I}$ to an annotated Kripke structure that does preserve all the relevant semantics. Then we show how the automata-theoretic techniques of [VW86] can help us ensure that we consider only states on runs that satisfy Ψ. Together these techniques help us to establish Theorem 6.1. We leave details to the full paper.

7 Discussion

In this paper we introduced knowledge-based programs and knowledge-based specifications. Standard programs work at the level of runs; by way of contrast, knowledge-based programs work at the knowledge level, which provides a higher level of abstraction. We believe that the idea of starting at the knowledge level, by designing a knowledge-based program that satisfies a (possibly knowledge-based) specification and then implementing it as a standard program, will give us a powerful tool for program development. We discuss this idea in detail and provide several examples in the full paper.

Two extensions of the idea of knowledge-based programs appear in the literature. One is the notion of programs based on *algorithmic knowledge* [HMV94], in which knowledge is assumed to be computed by a specific algorithm. Another is the notion of *knowledge-oriented programs* [MK93]. Recall that the actions in knowledge-based programs are standard actions, just as in standard programs. Based on an earlier draft of this paper and of [FHMV95], Moses and Kislev use the notions of contexts and programs very much in the way we do here to define a formalism in which the meaning of actions is also allowed to be knowledge-based. One example of such an action is $\mathsf{notify}(j, \varphi)$, which means that the acting agent must ensure that $K_j\varphi$ holds at some point in the past or future. Knowledge-oriented programs are shown in [MK93] to provide transparent high-level descriptions of protocols in a number of applications.

This paper also introduced the notion of a context in which a program is executed. By doing so, we are able to reason about executing the same program, or the same knowledge-based program, in different contexts. In many cases, programs solving the same problem in different contexts are implementations of the same knowledge-based programs [HZ92, MT88, MK93]. The notion of contexts therefore facilitates high-level reasoning about knowledge-based programs. We believe that contexts will find additional uses in reasoning about distributed programs and systems.

References

[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.

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

[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.

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

[FHMV95] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. MIT Press, Cambridge, Mass., 1995.

[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. (This is a preliminary version of [HF89].)

[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. 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.

[HMV94] J. Y. Halpern, Y. Moses, and M. Y. Vardi. Algorithmic knowledge. In R. Fagin, editor, *Theoretical Aspects of Reasoning about Knowledge: Proc. Fifth Conference*, pages 255–266. Morgan Kaufmann, San Francisco, Calif., 1994.

[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.

[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.

[Kur86] R. Kurki-Suonio. Towards programming with knowledge expressions. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, pages 140–149, 1986.

[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.

[MK93] Y. Moses and O. Kislev. Knowledge-oriented programming. In *Proc. 12th ACM Symp. on Principles of Distributed Computing*, pages 261–270, 1993.

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

[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.

[Sho93] Y. Shoham. Agent oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[Ste76] M. V. Stenning. A data transfer protocol. *Comput. Networks*, 1:99–110, 1976.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st IEEE Symp. on Logic in Computer Science*, pages 332–344, 1986.