Knowledge Based Programs: On the Complexity of Perfect Recall in Finite Environments*

Extended Abstract

Ron van der Meyden
Computing Science
University of Technology, Sydney
PO Box 123, Broadway NSW 2007
Australia
email: ron@socs.uts.edu.au

Jan 30, 1996

Abstract

Knowledge based programs have been proposed as an abstract formalism for the design of multi-agent protocols, based on the idea that an agent's actions are a function of its state of knowledge. The key questions in this approach concern the relationship between knowledge based programs and their concrete implementations. We present a variant of the framework of Fagin et al. that facilitates the study of a certain sort of optimization of these implementations. Within this framework, we investigate the inherent complexity of the implementations of atemporal knowledge based programs under the assumptions that the environment is finite state, and that agents operate synchronously and with perfect recall. We provide a simple example showing that one cannot expect to always obtain finite state implementations under this assumption. In fact, we show there exist environments in which knowledge based programs may generate behaviour of PSPACE-complete complexity. This is the most complex behaviour possible given our assumptions.

1 Introduction

In the analysis of distributed systems, it has been found useful to view processors as having states of knowledge [HM90, CM86, DM90, Had87, Hal87, HZ92, Maz86]. A common element of these analyses is the identification of states of knowledge that are necessary and sufficient conditions for the performance of certain actions. This has lead to the idea of *knowledge based programs* [FHMV95b, FHMV95a], in which formulae expressing states of knowledge are used as preconditions for the actions. Concrete protocols may be derived

*This paper appears in the *Proceedings of the 6th Conference on Theoretical Aspects of Rationality and Knowledge (TARK 1996)*, Morgan Kaufmann, pp. 31-50, 1996. Work begun while the author was with the Information Sciences Laboratory, NTT Basic Research Laboratories, Kanagawa, Japan, and continued at the Department of Applied Math and Computer Science, Weizmann Institute of Science. Thanks to Yoram Moses and Joe Halpern for helpful discussions on this paper.

from these abstract descriptions by translating the knowledge preconditions into standard computations based on the processor's state. Often, these concrete protocols are able to exploit avenues for optimization that may not have been readily apparent had the protocol been designed without the use of the knowledge abstraction. Additionally, the knowledge level protocol description is independent of specific assumptions concerning behaviour and structure of the communication substrate on which the protocol is to run, and this makes it possible to capture common intuitions underlying what are at face value different protocols.

A central question in this area is how the relationship between between knowledge level programs and their implementations may be realized in practice: in a given environment how does one compile a knowledge based program into a standard implementation? As part of an effort to understand this problem we address it in the present paper from the point of view of complexity: we ask what is the inherent complexity of the computations performed by an implementation of a knowledge based program.

We work with a specific set of assumptions regarding knowledge based programs and their semantic interpretation. First, we deal with *atemporal* programs, in which the tests for knowledge concern only the current state. Second, we assume that the environment is finite state. Finally, we assume that processors operate *synchronously,* and have *perfect recall,* i.e., operate as if they maintain a complete history of their local states. The perfect recall assumption is frequently made in the literature: it enables the development of protocols in which agents make optimal use of the information to which they are exposed, and allows design of the data structures that agents must maintain to be deferred until the information theoretic aspects of the problem to be solved have been clarified.

Our contributions are twofold. First, we introduce in Sections 2-4 a modification of the framework of [FHMV95b, FHMV95a] that highlights certain aspects of the relationship between, on the one hand, the semantic model used in interpreting a knowledge based program, and on the other, its implementations, that are obscured in the earlier framework. In particular, our modified framework facilitates the study of a certain sort of optimization of implementations.

Secondly, we present two data points relating to the complexity of implementations of knowledge based programs with respect to the assumptions of synchrony and perfect recall. One is a simple and natural example, analyzed in Section 5, that shows that finite state implementations do not in general exist, but which remains within the realm of the feasible. The second data point is a general result, presented in Section 6, showing that the behaviour generated by knowledge based programs interpreted with respect to the perfect recall semantics may have very high (PSPACE-complete) complexity in some (somewhat unnatural) finite environments. While these results are negative in spirit, they leave considerable room for further analysis and identification of tractable cases.

Section 7 compares our framework to that of [FHMV95b], and relates our results to others concerning the complexity of knowledge based programs. Section 8 concludes.

2 Knowledge Based Programs

In this section we introduce knowledge based programs and the environments in which they run. Our definitions are a variant of the definitions in [FHMV95a], to which the reader is referred for additional motivation. To illustrate the definitions, we use the following example at various points in the paper.

Example 2.1: Consider a system in which agent 1, after waiting an arbitrary amount of time, sends a message to agent 2. The communication channel delivers the message either immediately or with a delay of one second. However, the channel may fail at any time, and once it has failed does not recover. Both agents monitor the channel, and can detect a failure as soon as it happens. If the channel fails, then if agent 2 has received the message it keeps a copy of the message as long as it knows that agent 1 knows that it has a copy of the message. However, if it has any doubt about whether agent 1 knows that it has a copy, it immediately discards its copy. (Question: if the channel fails, how long does agent 2 keep its copy? We will provide an answer in Section 5.)□

The formal description of an example like the above will consist of two components: a set of *knowledge based programs,* describing the behaviour of the agents in terms of the relationship between their states of knowledge and their actions, and a description of the behaviour of the *environment* in which the agents operate, which includes an account of how their actions affect this environment.

To describe the agents' states of knowledge we work with the propositional multi-modal language for knowledge $\mathcal{L}_n^C$ generated from some set of basic propositions $Prop$ by means of the usual boolean operators, the monadic modal operators K_i, where $i = 1 \ldots n$ is an agent, and the monadic operators C_G, where G is a set of two or more agents. Intuitively, $K_i\varphi$ expresses that agent i knows φ, and $C_G\varphi$ expresses that φ is common knowledge amongst the group of agents G.

This language is interpreted in the standard way in a class of $S5_n$ Kripke structures of the form $M = \langle W, \sim_1, \ldots, \sim_n, V\rangle$, where W is a set of worlds, for each $i = 1 \ldots n$ the accessibility relation $\sim_i$ is an equivalence relation on W, and $V : W \times Prop \to \{0, 1\}$ is a valuation on W. If G is a group of agents then we may define the equivalence relation $\sim_G$ on W by $u \sim_G v$ if there exists a sequence $u_0, \ldots, u_k$ of worlds such that $u_0 = u$, $u_k = v$ and for each $j = 0 \ldots k-1$ there exists an agent $i \in G$ with $u_j \sim_i u_{j+1}$. The crucial clauses of the truth definition are given by

1. $(M, u) \models p$, where p is an atomic proposition, if $V(u, p) = 1$.
2. $(M, u) \models K_i\varphi$ if $M, v \models \varphi$ for all worlds v with $u \sim_i v$.
3. $(M, u) \models C_G\varphi$ if $M, v \models \varphi$ for all worlds v with $u \sim_G v$.

We now describe the structure of knowledge based programs. For each agent $i = 1 \ldots n$ let ACT_i be the set of actions that may be performed by agent i. Similarly, let ACT_e be the set of actions that may be performed by the environment in which the agents operate. If $a_e \in ACT_e$ and $a_i \in ACT_i$ for each $i = 1 \ldots n$, we say that the tuple $\mathbf{a} = \langle a_e, a_1, \ldots, a_n\rangle$ is a *joint action,* and we write ACT for the set of all joint actions.

Call a formula of $\mathcal{L}_n^C$ *i-subjective* if it is a boolean combination of formulae of the form $K_i\varphi$. A *knowledge based program* for agent i is a finite statement $\mathbf{Pg}_i$ of the form

$$\begin{array}{l}\textbf{case of}\\ \quad \textbf{if } \varphi_1 \textbf{ do } a_1\\ \quad\quad \vdots\\ \quad \textbf{if } \varphi_m \textbf{ do } a_m\\ \textbf{end case}\end{array}$$

where the φ_j are i-subjective formulae of $\mathcal{L}_n^C$ and the $a_j \in ACT_i$ are actions of agent i. Intuitively, a program of this form is executed by repeatedly evaluating the case statement, ad infinitum. At each step of the computation, the agent determines which of the formulae φ_j accurately describe its current state of knowledge. It non-deterministically executes one of the actions a_j for which the formula φ_j is true, updates its state of knowledge according to any new input it receives, and then repeats the process. We will give a more precise semantics below. We will write

$$\begin{array}{l}\textbf{case of}\\ \quad \textbf{if } \varphi_1 \textbf{ do } a_1\\ \quad\quad \vdots\\ \quad \textbf{if } \varphi_m \textbf{ do } a_m\\ \quad \textbf{else } a_{m+1}\\ \textbf{end case}\end{array} \qquad \text{for} \qquad \begin{array}{l}\textbf{case of}\\ \quad \textbf{if } \varphi_1 \textbf{ do } a_1\\ \quad\quad \vdots\\ \quad \textbf{if } \varphi_m \textbf{ do } a_m\\ \quad \textbf{if } \neg(\varphi_1 \vee \ldots \vee \varphi_m) \textbf{ do } a_{m+1}\\ \textbf{end case}\end{array}$$

To describe the behaviour of the world in which agents operate, we define an *interpreted environment* to be a tuple of the form $E = \langle S_e, I_e, P_e, \tau, O_1, \ldots, O_n, V_e\rangle$ where the components are as follows:

1. S_e is a set of *states of the environment.* Intuitively, states of the environment may encode such information as messages in transit, failure of components, etc. and possibly the values of certain local variables maintained by the agents.

2. I_e is a subset of S_e, representing the possible *initial states* of the environment.

3. $P_e : S_e \to \mathcal{P}(ACT_e)$ is a function, called the *protocol of the environment,* mapping states to subsets of the set ACT_e of actions performable by the environment. Intuitively, $P_e(s)$ represents the set of actions that may be performed by the environment when the system is in state s.

4. τ is a function mapping joint actions $\mathbf{a} \in ACT$ to state transition functions $\tau(\mathbf{a}) : S_e \to S_e$. Intuitively, when the joint action $\mathbf{a}$ is performed in the state s, the resulting state of the environment is $\tau(\mathbf{a})(s)$.

5. For each $i = 1..n$, the component O_i is a function, called the *observation function of agent i,* mapping the set of states S_e to some set $\mathcal{O}$. If $s \in S_e$ is a state then $O_i(s)$ will be called the *observation* of agent i in the state s.

6. $V_e : S_e \times \mathit{Prop} \to \{0, 1\}$ is a valuation, assigning a truth value $V(s, p)$ in each state s to each atomic proposition $p \in \mathit{Prop}$.

A *run* of an environment E is a *finite* sequence $s_0 \dots s_m$ of states such that for all $i = 0 \dots m-1$ there exists a joint action $\mathbf{a} = \langle a_e, a_1, \dots, a_n \rangle$ such that $s_{i+1} = \tau(\mathbf{a})(s_i)$ and $a_e \in P_e(s_i)$. In general, environments may have an infinite set of states. An environment is *finite* if its set of states is finite. We will be concerned in this paper primarily with finite environments.

Our definition of environments resembles in many respects the definition of *contexts* in [FHMV95a], so we warn that our notion is intended to capture only part of what is meant by the latter term. In particular, we typically do not capture within an environment all the information a context would contain about the agents' local states. We will give a more detailed comparison in Section 7.

Example 2.2: We model the system described in Example 2.1 as a set of knowledge based programs $\mathbf{Pg}_1, \mathbf{Pg}_2$ and an interpreted environment

$$\langle S_e, I_e, P_e, \tau, O_1, O_2, V_e \rangle.$$

Since the behaviour we wish to capture does not depend on the message contents, we do not model this. The propositions of interest are the following:

- *failed*: the channel has failed,
- *sent*: agent 1 has sent the message at some time in the past, and the channel did not simultaneously fail,
- *rcvd*: agent 2 has received the message at some time in the past,
- *dlyd*: the message has been sent, but is currently being delayed for one second, and
- *copy*: agent 2 has a copy of the message.

We may take the set of states S_e of the environment to consist of tuples

$$\langle \mathit{failed}, \mathit{sent}, \mathit{rcvd}, \mathit{dlyd}, \mathit{copy} \rangle$$

where the components correspond directly to the boolean variables described above. The valuation V_e is that obtained from this correspondence. There is a single initial state, that in which all of these variables take the value 0.

The set of actions of agent 1 is $ACT_1 = \{\mathit{wait},\ \mathit{send}\}$, with the obvious interpretations. In the case of agent 2, we have the set of actions $ACT_2 = \{\mathit{wait}, \mathit{discard}\}$, again with the obvious interpretations. The environment models the communication channel, so here we take $ACT_e = \{\mathit{delay}, \mathit{deliver}, \mathit{fail}\}$, where

- *delay* represents the environment's delaying of the message if one is currently being sent,
- *deliver* represents the environment's preparedness to immediately deliver any message that is currently being sent, and
- *fail* represents the failure of channel.

We describe the environment's protocol P_e by means of a nested if statement. If $s = \langle \mathit{failed}, \mathit{sent}, \mathit{rcvd}, \mathit{dlyd}, \mathit{copy} \rangle$ then

if $\mathit{failed} = 1$ then $P_e(s) = \{\mathit{fail}\}$, else if $dlyd = 1$ then $P_e(s) = \{deliver, fail\}$

else $P_e(s) = \{delay, deliver, fail\}$,

To describe the action interpretation function τ, suppose we are given a joint action $\mathbf{a} = \langle a_e, a_1, a_2 \rangle$ and a state $s = \langle failed, sent, rcvd, dlyd, copy \rangle$. Then the result $\tau(\mathbf{a})(s)$ of the agents performing the joint action $\mathbf{a}$ in state s is the state $s' = \langle failed', sent', rcvd', dlyd', copy' \rangle$ obtained by letting the corresponding components of s and s' be equal, with the following exceptions, describing the effects of the actions:

- if $a_e = fail$ then $failed' = 1$
- if $a_1 = send$ and $failed \neq 1$ then
  - if $a_e \neq fail$ then $sent' = 1$, and
  - if $a_e = deliver$ then $rcvd' = 1$ and $copy' = 1$, and
  - if $a_e = delay$ then $dlyd' = 1$
- if $dlyd = 1$ and $a_e = deliver$ then $rcvd' = 1$ and $copy' = 1$ and $dlyd' = 0$
- if $a_2 = discard$ then $copy' = 0$

Finally, to describe the observations of the agents, note that agent 1 is able to determine from local information whether not it has sent the message, and also detects failure, so if $s = \langle failed, sent, rcvd, dlyd, copy \rangle$ then $O_1(s) = \langle failed, sent \rangle$. Agent 2 detects failure of the channel, and can determine whether it has received or discarded the message, so $O_2(s) = \langle failed, rcvd, copy \rangle$.

The behaviour of the agents in the system is modelled by associating with each agent a knowledge based program. For agent 1 we take the program

Pg$_1$ = **case of**
  if $K_1(\neg failed \wedge \neg sent)$ **do** *wait*
  if $K_1(\neg failed \wedge \neg sent)$ **do** *send*
  else *wait*
end case

That is, if it knows that the channel has not yet failed, and that it has not previously successfully sent the message, then agent 1 may choose to either send the message or perform the null action wait. Otherwise the agent must wait. For agent 2, we have the program

Pg$_2$ = **case of**
  if $K_2(failed \wedge copy) \wedge \neg K_2 K_1 copy$ **do** *discard*
  else *wait*
end case

This completes the description of the system. □

3 Protocols

Next, we introduce the standard protocols which will be used as implementations of knowledge based programs, and describe the set of runs produced by executing such a protocol. Define a *protocol* for agent i to be a tuple $P_i = \langle S_i, q_i, \alpha_i, \mu_i \rangle$ consisting of

1. a set S_i of *protocol states*,
2. an element $q_i \in S_i$, the protocol's *initial state*,
3. a function $\alpha_i : S_i \times \mathcal{O} \rightarrow \mathcal{P}(ACT_i)$, such that $\alpha_i(s, o)$ is a nonempty set representing the possible next actions that agent i may take when it is in state s and is making observation o.
4. a function $\mu_i : S_i \times \mathcal{O} \rightarrow S_i$, such that $\mu_i(s, o)$ represents the next protocol state the agent assumes after it has been in state s making observation o.

In general, we allow the set of protocol states to be infinite, indeed that the functions α and μ be non-computable. A *joint protocol* is a tuple $\mathbf{P} = \langle P_1, \ldots, P_n \rangle$ such that each P_i is a protocol of agent i.

Intuitively, protocol states represent the memory that agents maintain about their sequence of observations for the purpose of implementing their knowledge based programs. For each run r of an environment E, a protocol P_i determines a protocol state $P_i(r)$ for agent i by means of the following recursive definition. If r is a run of length one, consisting of an initial state of E, then $P_i(r) = q_i$, the initial state of the protocol. If r is the run $r's$, where r' is a run and s is a state of the environment, then $P_i(r) = \mu_i(P_i(r'), O_i(\mathit{fin}(r')))$, where $\mathit{fin}(r')$ is the final state of r'. That is, $P_i(r's)$ is a function of the sequence of observations made by agent i when the environment goes through the sequence of states r'. In other words, $P_i(r)$ represents the agent's memory of its *past* observations in the run r, excluding the current observation.

In executing a protocol, an agent's next action in a given run is selected from a set determined by its memory of past observations together with its current observation. Define the *set of actions of agent i enabled by the protocol P_i at a run r* with final state s to be the set $act_i(P_i, r) = \alpha_i(P_i(r), O_i(s))$. Similarly, the set of joint actions $\mathbf{act}(\mathbf{P}, r)$ enabled at r by a joint protocol $\mathbf{P}$ contains precisely the joint actions $\langle a_e, a_1, \ldots, a_n \rangle$ such that $a_e \in P_e(s_e)$ and each $a_i \in act_i(P_i, r)$.

We may now describe the set of runs that result when a particular protocol is executed in a given environment. Given a joint protocol $\mathbf{P}$ and an environment E, we define the *set of runs generated by* $\mathbf{P}$ *and* E, to be the smallest set of runs $\mathcal{R}$ such that

1. for each initial state $s_e \in I_e$ of the environment $\mathcal{R}$ contains the run of length one composed just of the state s_e, and
2. if r is a run in $\mathcal{R}$ with final state s, and if $\mathbf{a} \in \mathbf{act}(\mathbf{P}, r)$ is a joint action enabled by $\mathbf{P}$ at r then the run $r \cdot \tau(\mathbf{a})(s)$ is in $\mathcal{R}$.

Intuitively, this is the set of runs generated when the agents incrementally maintain their memory using the update functions μ_i, and select their next action at each step using the functions α_i.

4 Implementation of Knowledge Based Programs

Clearly, in order to execute a knowledge based program according to the informal description above, we need some means to interpret the knowledge formulae it contains. To do so, we introduce a particular class of Kripke structures.

These structures will be obtained from a set of runs by means of a generalization of observations. Define a *joint view* of an environment E to be a function $\{.\}$ mapping runs of E to X^n for some set X. We write $\{r\}_i$ for the i-th component of the result of applying $\{.\}$ to the run r. Intuitively, a view captures the information available to the agents in each run. One particular view that will be of central importance to us is the *synchronous perfect recall view* $\{.\}^{pr}$. If r is the run $s_1 \ldots s_k$ then this view is defined by $\{r\}_i^{pr} = O_i(s_1) \ldots O_i(s_k)$. That is, the perfect recall view provides the agent with a complete record of all the observations it has made in a run. The term "synchronous" here refers to the fact that an agent may use this to determine the "time", simply by counting the number of observations. Clearly, the synchronous perfect recall view captures the maximal information that an agent may have in an environment, so we place the following additional requirement on views $\{.\}$: there must exist a function f such that $\{.\} = f \circ \{.\}^{pr}$.

Given a set $\mathcal{R}$ of runs of environment E and a joint view $\{.\}$ of E, we may define the Kripke structure $M(\mathcal{R}, \{.\}) = (W, \sim_1, \ldots, \sim_n, V)$, where

- the set of worlds $W = \mathcal{R}$, and
- for all $r, r' \in \mathcal{R}$ we have $r \sim_i r'$ iff $\{r\}_i = \{r\}_i$, and
- the valuation $V : \mathcal{R} \times Prop \rightarrow \{0, 1\}$ is defined by $V(r, p) = V_e(\mathit{fin}(r), p)$.

That is, the accessibility relations are derived from the view, and truth of basic propositions at runs is determined from their final states according to the valuation V_e provided by the environment. Intuitively, $r \sim_i r'$ if, based on the information provided by the view $\{.\}_i$, the agent cannot distinguish between the runs r and r'. We will call $M(\mathcal{R}, \{.\})$ the *interpreted system* obtained from $\mathcal{R}$ and $\{.\}$.[1]

We may now define the *set of actions enabled by the program* $\mathbf{Pg}_i$ *at a run* r *of system* M to be the set $\mathbf{Pg}_i(M, r)$ consisting of all the a_i such that $\mathbf{Pg}_i$ contains a line "**if** φ **do** a_i" where $(M, r) \models \varphi$. Similarly, the set of joint actions $\mathbf{Pg}(M, r)$ enabled by a joint knowledge based program in a run r with final state having environment component s_e contains precisely the joint actions $\langle a_e, a_1, \ldots, a_n \rangle$ such that $a_e \in P_e(s_e)$ and each $a_i \in \mathbf{Pg}_i(M, r)$.

This definition leaves open the question of what system we are to use when executing a knowledge based program. The answer to this is that we should use a system that is itself generated by running the program. To avoid the circularity, we first consider a system generated by some standard protocol, and then state when such a system is equivalent to that generated by the knowledge based program. Say that *a joint protocol* $\mathbf{P}$ *implements a knowledge based program* $\mathbf{Pg}$ *in an environment* E *with respect to the joint view* $\{.\}$ if for all r in the set of runs $\mathcal{R}(\mathbf{P}, E)$ generated by $\mathbf{P}$ and E, we have $\mathbf{Pg}(M, r) = \mathbf{act}(\mathbf{P}, r)$, where $M = M(\mathcal{R}(\mathbf{P}, E), \{.\})$ is the system obtained from $\mathcal{R}(\mathbf{P}, E)$ and $\{.\}$. That is, the joint actions prescribed by the knowledge based program, when the tests for knowledge are interpreted according to the system M, are precisely those prescribed by the standard protocol $\mathbf{P}$.

The framework for the implementation of knowledge based programs just developed differs in some regards from that of [FHMV95a]: we will make a more detailed comparison

[1] Although much of the literature has dealt with Kripke structures in which the accessibility relations are derived from the local states appearing in runs, a semantics based on views appears already in [HM90].

below. The benefit we derive from the changes is that our approach allows us to more naturally investigate the inherent complexity of the behaviours generated by knowledge based programs, and to more cleanly characterize a certain type of optimization.

Specifically, suppose that $\mathbf{P}$ is a joint protocol implementing a knowledge based program $\mathbf{Pg}$. We take it that what is most of interest about an implementation is not the particular set of protocol states it exploits, but the *behaviour* it generates. Thus, if some other protocol $\mathbf{P}'$ has the same effects on the environment as $\mathbf{P}$ but uses a smaller state space or computes the functions α_i or μ_i more efficiently, it is reasonable to substitute the protocol $\mathbf{P}'$ for $\mathbf{P}$. This type of optimization is already to be found in the literature. For example, Halpern and Zuck [HZ92] first derive a perfect recall implementation of a knowledge based protocol (a precursor of the knowledge based programs we have discussed here) but then show that the behaviour of this implementation may be captured by means of a finite state implementation.

To formalize this, define two joint protocols $\mathbf{P}$ and $\mathbf{P}'$ to be *behaviourally equivalent* with respect to an environment E if they generate precisely the same set of runs by means of the same sets of action at each step. That is, $\mathbf{P}$ and $\mathbf{P}'$ are behaviourally equivalent when $\mathcal{R}(\mathbf{P}, E) = \mathcal{R}(\mathbf{P}', E)$, and for all $r \in \mathcal{R}(\mathbf{P}, E)$ we have $\mathbf{act}(\mathbf{P}, r) = \mathbf{act}(\mathbf{P}', r)$.[2] As with any account of optimization, we must identify precisely what properties of the original program are preserved by the optimization. The following result shows that behavioural equivalence preserves the implementation relation.

Proposition 4.1: Suppose that the joint protocols $\mathbf{P}$ and $\mathbf{P}'$ are behaviourally equivalent with respect to an environment E, and that $\mathbf{P}$ implements the knowledge based program $\mathbf{Pg}$ with respect to the view $\{.\}$. Then $\mathbf{P}'$ also implements $\mathbf{Pg}$ with respect to the view $\{.\}$.

In the framework of [FHMV95a] knowledge based programs, even atemporal ones, may have many different implementations. Our notion of implementation is more liberal than theirs, so this will also be the case for our definitions. However, the notion of implementation is better behaved when we consider implementations with respect to the synchronous perfect recall view. Say that P_i is a *perfect recall protocol* if q_i is the empty string and $\mu_i(s, o) = s \cdot o$. That is, in a perfect recall protocol the agent retains its sequence of all its prior observations as its protocol state.

Proposition 4.2: For every environment E and joint knowledge based program $\mathbf{Pg}$, all implementations of $\mathbf{Pg}$ in E with respect to $\{.\}^{pr}$ are behaviourally equivalent. Thus, there exists a unique[3] joint synchronous perfect recall protocol $\mathbf{P}$ that implements $\mathbf{Pg}$ in E with respect to $\{.\}^{pr}$.

Although our formulation is slightly different, this result can be viewed as a special case of Theorem 7.2.4 of [FHMV95b]. Intuitively, the set of runs of length one generated by any implementation must be the set of initial states of the environment. Because of

[2] This definition is appropriate given the level of generality at which we work, but it does not capture all aspects of the optimizations that have been considered in the literature. For example, in message passing environments, one may optimize not just the set of protocol states, but also the messages that a protocol exploits.

[3] Strictly, the implementation is unique only with respect to "reachable" protocol states.

the synchrony assumption, this set of states may be used to uniquely determine the actions enabled at the runs of length one in any perfect recall implementation. These actions in turn uniquely determine the set of runs of length two in any implementation. By the synchrony assumption we then find that the set of actions enabled at a run of length two in any perfect recall implementation is uniquely determined. This again uniquely determines the runs of length three, and so on. We will see an example of this incremental construction of the system generated by the synchronous perfect recall implementation in Section 5.

Proposition 4.2 guarantees the existence of an implementation, the unique perfect recall implementation, but it is clear that this implementation is very inefficient in its usage of space, since the size of the protocol states equal the length of the run. This, however, does not preclude the possibility that there exists a behaviourally equivalent implementation that makes more efficient use of space resources. This brings us to the central question we study in this paper: what is the inherent complexity of the behaviour generated by the perfect recall implementation of a knowledge based program? That is, what is the most efficient that an implementation of the program with respect to the perfect recall view can be?

The most promising answer we could hope to obtain to this question is that there exists a *finite state implementation*, in which for each each agent i, the set of protocol states S_i is finite, so that the implementation consumes constant space, and both α_i or μ_i may be computed in constant time independent of the length of the run. Before addressing the complexity question in its full generality, we first show by means of an example that finite state implementations do not always exist, even for remarkably simple environments and programs.

5 A Non-Finite State Example

We now analyze the system M generated by the perfect recall implementation of the programs and environment introduced in Example 2.2. It will turn out that there is no finite state protocol that is behaviourally equivalent to this implementation.

For the analysis, it is convenient to view the combination of the system and environment as a labelled transition diagram, in which the nodes represent states of the environment, and an edge from state s to state t labelled by a formula φ represents that if r is a run of M ending in state s, then rt is also a run of M just in case $(M, r) \models \varphi$.

Note first that although the environment contains 32 states, not all of these are reachable from the initial state by means of a sequence of joint actions. Moreover, many state transitions can be seen by means of a simple consideration to occur in no run of M. For example, consider a state s in which $\mathit{failed} = 0$. Because $O_2(s) = \langle \mathit{failed},\ \mathit{rcvd},\ \mathit{copy}\rangle$, we must have $K_2(\neg \mathit{failed})$ in any run r ending in state s. But this formula implies that $K_2(\mathit{failed} \land \mathit{copy})$ is false, so that the only action agent 2 can perform is *wait*. Similarly, if $\mathit{copy} = 0$ then agent 2's action must be *wait*, and agent 1's action must be *wait* if either $\mathit{failed} = 1$ or $\mathit{sent} = 1$.

Applying these considerations shows that at most 7 states can occur. These states are depicted in Figure 1, and may be interpreted as follows. In states a to c, the channel has not yet failed, and

- $a = \langle \mathit{failed} : 0,\ \mathit{sent} : 0,\ \mathit{rcvd} : 0,\ \mathit{dlyd} : 0,\ \mathit{copy} : 0\rangle$ is the initial state, in which agent 1 has not yet sent the message;

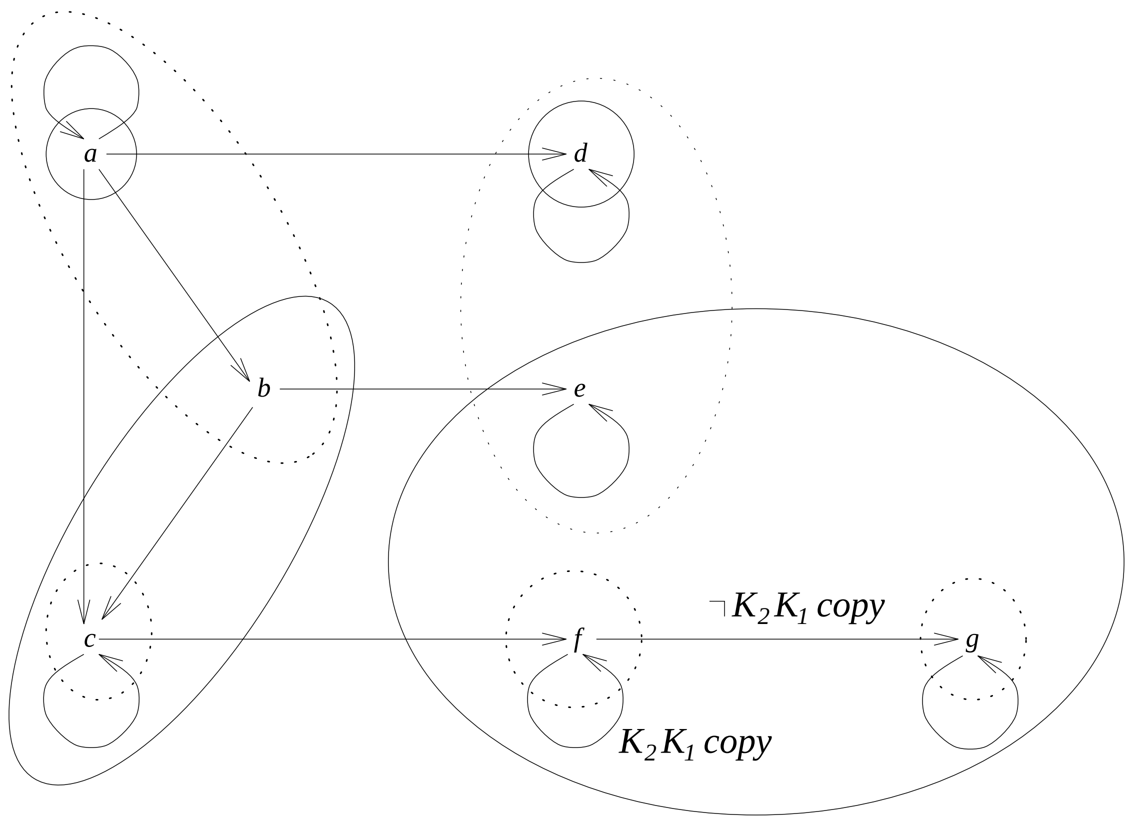

Figure 1: A knowledge-based environment

- $b = \langle \mathit{failed} : 0,\ \mathit{sent} : 1,\ \mathit{rcvd} : 0,\ \mathit{dlyd} : 1,\ \mathit{copy} : 0 \rangle$ is the state in which agent 1 has successfully sent the message, but in which it is being delayed for one second.

- $c = \langle \mathit{failed} : 0,\ \mathit{sent} : 1,\ \mathit{rcvd} : 1,\ \mathit{dlyd} : 0,\ \mathit{copy} : 1 \rangle$ is the state in which agent 2 has received the message.

In states d to g, the channel has failed:

- $d = \langle \mathit{failed} : 1,\ \mathit{sent} : 0,\ \mathit{rcvd} : 0,\ \mathit{dlyd} : 0,\ \mathit{copy} : 0 \rangle$ is the state in which the failure occurred before the message was sent (or simultaneously with a send operation),

- $e = \langle \mathit{failed} : 1,\ \mathit{sent} : 1,\ \mathit{rcvd} : 0,\ \mathit{dlyd} : 0,\ \mathit{copy} : 0 \rangle$ is the state in which the failure occurred while the message was in transit,

- $f = \langle \mathit{failed} : 1,\ \mathit{sent} : 1,\ \mathit{rcvd} : 1,\ \mathit{dlyd} : 0,\ \mathit{copy} : 1 \rangle$ is the state in which the failure occurred after the message was received, and agent 2 has not discarded its copy of the message

- $g = \langle \mathit{failed} : 1,\ \mathit{sent} : 1,\ \mathit{rcvd} : 1,\ \mathit{dlyd} : 0,\ \mathit{copy} : 0 \rangle$ is the state in which the failure occurred after the message was received, and agent 2 has discarded its copy of the message

We have also used the considerations noted above to simplify the edge labels. For example, consider the transition from state f to state g, which corresponds to the joint action $\langle \mathit{fail}, \mathit{wait}, \mathit{discard} \rangle$. It can be seen from the knowledge based programs that this action is enabled at a run r ending in state f only if the formula

$$\neg K_1(\neg \mathit{failed} \wedge \neg \mathit{sent}) \wedge K_2(\mathit{failed} \wedge \mathit{copy}) \wedge \neg K_2 K_1 \mathit{copy}$$

holds at that run. By the same considerations as above, the first two of these conjuncts must hold at any such run. Thus the transition will be enabled just when the formula $\neg K_2 K_1 \mathit{copy}$ holds at the run, and we have labelled the transition by this formula. Applying a similar simplification shows that all but one other edge (that from f to f) is always enabled, in which case we have omitted the label *true* from the diagram.

The ellipses in the figure represent the equivalence classes of states associated with the equivalence relations on states defined by $s \approx_i t$ if $O_i(s) = O_i(t)$. The ellipses drawn with a solid line represent the observation equivalence classes for agent 1; those drawn with broken lines represent the equivalence classes for agent 2. It may be verified that these classes are precisely those obtained from the observation functions described in Example 2.2.

Suppose that rs and $r't$ are runs of M, where r and r' are runs and s and t are states. Then it follows from the definition of the relation $\sim_i$ that $rs \sim_i r't$ if and only if $r \sim_i r'$ and $s \approx_i t$. Thus, we may use the equivalence relations $\approx_i$ in an incremental calculation of the relations $\sim_i$.

We are now in a position to describe the structure of M. We will do so by computing the connected components of this structure, i.e., the minimal sets of runs C such that if $r \in C$ and $r \sim_i r'$ for some agent i then $r' \in C$. Note that, because of synchrony, all runs in a connected component of M are of the same length. Second, if r is a run in which the channel first fails at the k-th step, and r' is a run in which the channel either does not fail, or fails at the m-th step for some $m \neq k$, then for all agents i we have $r \not\sim_i r'$, because agent i's view records the time of failure. Thus, all runs in a connected component of M must also have failure occur at the same time, or not at all.

We consider first the structure of M on runs of length n in which the channel has not yet failed, and derive from this the structure of M on runs containing a failure. The states in which a failure has not yet occurred are a, b and c. Since the transitions between these states are always enabled, the runs comprised of these states are strings of the form $a^k b^l c^m$, where $k \geq 1$ and $0 \leq l \leq 1$ and $m \geq 0$ are natural numbers. It may be verified that a single component of M_E contains all the runs of length n of this form, described by the sequence

$$ac^{n-1} \sim_1 abc^{n-2} \sim_2 a^2c^{n-2} \sim_1 \ldots$$

$$\ldots \sim_2 a^{n-2}c^2 \sim_1 a^{n-2}bc \sim_2 a^{n-1}c \sim_1 a^{n-1}b \sim_2 a^n$$

where in addition to the relationships $\sim_i$ shown explicitly we also have the derived relations obtained by forming the symmetric reflexive closure. (We considered this subenvironment in [Mey94], to which we refer the reader for a more detailed discussion of this claim.) Thus, one equivalence class of $\sim_1$ has states ac^{n-1} and abc^{n-2}. Intuitively, in these runs agent 1's view records that it sent the message at time 1, but although agent 1 has enough information to know that a failure has not yet occurred, it does not know whether the message was delivered immediately or delayed.

Consider now the runs of length $n+1$ in which a failure occurred at time n. Notice that if we have $r \not\sim_i r'$ for two runs r and r' then for any extensions of these runs rs and $r't$

we must also have $rs \not\sim_i r't$. Thus, to obtain the connected components of M comprised of runs of length $n+1$ in which a failure occurs at time n, it suffices to consider extensions of the runs listed above, and determine which of the above relationships are preserved under extension. Observing that the failure transitions are always enabled, it may be verified that we obtain a single connected component comprised of runs of this form, in which the linear sequence above is preserved:

$$ac^{n-1}f \;\sim_1\; abc^{n-2}f \;\sim_2\; a^2c^{n-2}f \;\sim_1\; \ldots$$

$$\ldots \sim_2\; a^{n-2}c^2f \;\sim_1\; a^{n-2}bcf \sim_2\; a^{n-1}cf \;\sim_1\; a^{n-1}be \;\sim_2 a^nd$$

In order to obtain the connected component containing runs of length $n+2$ in which a failure occurs at time n, we now need to consider the satisfaction at the above runs of the formulae labelling the transitions from state f. Since f and c are the only states satisfying the proposition *copy*, we see that the formula $K_1 copy$ holds at the run $a^{n-2}bcf$ and all runs to its left, but not at runs to its right. Thus, the formula $K_2K_1 copy$ holds at the run $a^{n-2}c^2f$ and all runs to its left, but not at runs to its right. This means that at the run $a^{n-2}c^2f$, and to its left, the transition from f to f is enabled, and the transition from f to g is disabled. On the other hand, for the runs ending in f to the right $a^{n-2}c^2f$, the transition from f to g is enabled, but the transition from f to f is disabled. Computing the resulting equivalence relations on the runs of length $n+3$ extending the above, we obtain the linear sequence

$$ac^{n-1}f^2 \;\sim_1\; abc^{n-2}f^2 \;\sim_2\; a^2c^{n-2}f^2 \;\sim_1\; \ldots$$

$$\ldots \sim_2\; a^{n-2}c^2f^2 \;\sim_1\; a^{n-2}bcfg \sim_2\; a^{n-1}cfg \;\sim_1\; a^{n-1}be^2 \;\sim_2 a^nd^2$$

Notice that in this structure the formula $K_1K_2 copy$ holds at the run $a^{n-2}c^2f$ and to its left, but not to its right. Continuing this argument, we see that the runs of length $n+k$ in which a failure occurs at time n form a single connected component of the form

$$ac^{n-1}f^k \;\sim_1\; abc^{n-2}f^k \;\sim_2\; a^2c^{n-2}f^k \;\sim_1\; \ldots$$

$$\ldots \sim_1 a^{n-m}bc^mf^mg^{k-m-2} \sim_2\; a^{n-m-2}c^mf^mg^{k-m-2} \sim_1 \ldots$$

$$\ldots \sim_1 a^{n-3}bc^2f^2g^{k-2} \sim_2\; a^{n-2}c^2f^2g^{k-2} \;\sim_1$$

$$a^{n-2}bcfg^{k-1} \sim_2\; a^{n-1}cfg^{k-1} \;\sim_1\; a^{n-1}be^k \;\sim_2 a^nd^k$$

where there are no runs ending in f if $k > n-1$.

We are now in a position to answer the question raised in Example 2.1: how long does agent 2 keeps its copy if the channel fails? Note that the second line in the last sequence above may be understood as expressing the following: if a failure occurs m seconds after agent 2 receives the message, then agent 2 discards its copy precisely m seconds after the failure. In particular, we find that ac^Nf^M is a run if and only if $M \leq N$. We note one important consequence of this fact: the set of runs generated by the perfect recall implementation of **Pg** is not regular. It immediately follows from this that there is no finite state joint protocol that is behaviourally equivalent to the perfect recall implementation.

6 Complexity

The result of the previous section indicates that in general, we cannot expect to find finite state protocols that are behaviourally equivalent to the prefect recall implementation of a knowledge based program. In the case of this example, a comparatively efficient implementation is still possible, that maintains just a single counter, thus using space $O(\log |r|)$, functions μ_i running in time $O(\log |r|)$ and functions α_i running in constant time. We now seek to answer the following question: just how complex an implementation is needed in general?

First, we note that a variety of ways of measuring the complexity of an implementation suggest themselves, depending on what we seek to compute about the implementation. Suppose that **Pg** is a set of knowledge based programs. Let $M = (\mathcal{R}, \sim_1, \ldots, \sim_n, V)$ be the system obtained from the perfect recall implementation of the knowledge based program **Pg** in environment E. We consider the following computational problems.

P1. Given a run $r \in \mathcal{R}$, compute $\mathbf{Pg}(M, r)$.

P2. Given a run $r \in \mathcal{R}$ and a formula φ occurring in **Pg**, decide $(M, r) \models \varphi$.

P3. Given a run r, determine if $r \in \mathcal{R}$.

P4. Given an agent i and $\{r\}_i^{pr}$ for some $r \in \mathcal{R}$, compute $\mathbf{Pg}_i(M, r)$. (Note that whether $(M, r) \models K_i\varphi$ depends only upon $\{r\}_i^{pr}$, rather than upon r itself.)

P5. Given an agent i and $\{r\}_i^{pr}$ for some run $r \in \mathcal{R}$, and a formula φ occurring in $\mathbf{Pg}_i$, decide $(M, r) \models \varphi$.

Of these problems, P1-P3 take the perspective of an external observer of the implementation, and measure the complexity for such an observer of a number of questions related to this implementation. The problems P4 and P5 measure the complexity of the implementation from the point of view of the agents themselves. The following result provides an upper bound on the complexity of all these problems. (PSPACE is the set of all computational problems that can be solved by a computation using space (i.e. memory resources) no more than some polynomial of the input size.)

Theorem 6.1: For all environments E and knowledge based programs **Pg**, the problems P1-P5 are in PSPACE.

In particular, this result shows that to compute the functions α_i, the perfect recall implementation requires no more than a polynomial amount of space. (Note that in this implementation the computation of the functions μ_i is trivial.) However, the following result indicates that it is not possible to do with less than a polynomial amount of space. (A problem is PSPACE hard if it is as least as hard as any other problem in PSPACE.)

Theorem 6.2: There exists an environment E and a joint knowledge based program **Pg** for two agents in which all knowledge tests are of the form $K_i\varphi$ and $\neg K_i\varphi$, where φ contains no knowledge operators, such that the problems P1-P5 are PSPACE hard.

Proof: (Sketch) The proof is based on problem P2, together with reductions showing this problem is as easy as any of the five problems. The proof proceeds by simulation of polynomial time bounded alternating computations [CKS81], using the fact that PSPACE = APTIME. Given a PTIME ATM $\mathcal{M}$, the proof shows that there exists a knowledge based program **Pg** and an environment E such that if $\mathcal{R}$ is the set of runs generated by the perfect recall implementation of **Pg** in E, then for each configuration C of the machine $\mathcal{M}$ there exists a run $f(C)$ of length polynomial in $|C|$ such that $f(C) \in \mathcal{R}$ iff $\mathcal{M}$ accepts when started from initial configuration C.

The simulation resembles the behaviour described in Section 5, inasmuch it consists of two phases. The first phase, which does not essentially involve the knowledge based program, constructs a connected component containing runs of the form $r_1\#_c$, where r_1 is a run of length m encoding a configuration of an alternating Turing machine, and $\#_c$ is a special "configuration termination" state that also encodes whether the configuration is universal or existential, and the outcome of the computation if the configuration is terminal. The structure of the alternating computation tree is reflected in this component by the relations $\sim_1$ and $\sim_2$ as follows: If $r_1\#_c$ is run representing a configuration that has a child in the computation tree that is represented by the configuration $r_2\#_d$, then there exists a run r such that $r_1\#_c \sim_1 r$ and $r \sim_2 r_2\#_d$.

The second phase of the simulation consists of certain runs of the form $r_1\#_c r'$ extending the runs of the first phase, where r' is a run of length bounded by the height of the computation tree. Much in the way that the linear structure of connected components was preserved under extension of runs in the analysis of Section 5, these extensions preserve the embedding of the computation tree within the relations $\sim_1, \sim_2$. At each step (increasing the length of the runs in the component by one) the knowledge based program is used to propagate information concerning acceptance from the leaves of the tree towards the root, one level at a time, much like the progressive flow of information discussed in Section 5. If $r_1\#_c$ is a run representing the initial configuration C of the computation, the decision about acceptance of the computation tree is finally reflected in the final state of a run of the form $r_1\#_c r'$ where the length of r' is twice the height of the computation tree starting at configuration $r_1\#_c$. Thus, we take $f(C) = r_1\#_c r'$.$\Box$

This result shows that the PSPACE upper bound to be tight for certain combinations of environment and knowledge based program. The amount of time required by PSPACE hard problems is not known, but is widely believed to be non-polynomial: all known approaches to the solution of PSPACE hard problems require exponential time. Thus, Theorem 6.2 indicates that we cannot always expect to find efficient implementations with respect to the perfect recall view.

The problem P2 is a special case of the more general problem of *model checking* a formula φ at a run r in a given system M, i.e., determining $(M, r) \models \varphi$. It is instructive to compare Theorem 6.2 with the results in [Mey94] on the complexity of model checking when M is the system obtained from the set of *all* runs of an environment E and the view $\{.\}^{pr}$. In this case, if φ is an arbitrary formula in $\mathcal{L}_n^C$ then model checking is also in PSPACE. However, for fixed formulae φ containing the operators K_i, but not the operators C_G, the set $\{\{r\}^{pr} \mid (M, r) \models \varphi\}$ is a regular set, so such formulae may be model checked very efficiently. Thus, we find that compared to the system containing all runs of the environment, the systems generated by implementations of knowledge based programs with respect to the perfect recall view do not increase the complexity of model checking for arbitrary formulae in $\mathcal{L}_n^C$, but they do increase the complexity of model checking formulae containing only knowledge operators. This is the case even when the knowledge based program contains only tests for a single level of knowledge.

We have focussed above on the complexity of the perfect recall implementation, but our results place immediate limitations on all other implementations. The perfect recall implementation is space inefficient, but allows rapid calculation of the update functions μ_i. Alternate implementations afford the opportunity to make trade offs between the size of the protocol states and the complexity of the functions μ_i and α_i. While little is known about the limitations of such tradeoffs, we do obtain the following corollary of Theorem 6.2.

Corollary 6.3: There exists an environment E and a knowledge based program **Pg** and an agent i such that for all implementations **P** of **Pg** in E with respect to the perfect recall view, there exists a constant $\epsilon > 0$ such that the size of the state $P_i(r)$ is infinitely often greater than $|r|^\epsilon$.

Thus, there are strict limits on the ability of alternate implementations to reduce the amount of memory they must maintain.

7 Comparison

We now explain how our notion of an implementation of a knowledge based program differs from that of [FHMV95b], and relate our results to other work on the complexity of knowledge based programs.

The relationship between the framework of the present paper and that of [FHMV95b] is open to varying interpretations, but we will sketch here what seems to be the most natural mapping of our framework to theirs. We note that in a number of cases the two frameworks use the same terminology with different meanings (e.g. "runs", "protocols") so the following should be understood as an attempt to explicate their framework using the meaning of the terms from the present paper, rather than a precise translation.

One obvious difference is that instead of dealing with *environments*, [FHMV95b] define a notion of *context*. Under the proposed mapping, a context corresponds to our notion of environment packaged together with, for each agent i, a set of local protocol states S_i, the initial protocol state q_i, and the protocol state update function μ_i from some protocol $\langle S_i, q_i, \alpha_i, \mu_i \rangle$. When combined with the tuple $\langle \alpha_1, \ldots, \alpha_n \rangle$, which we will call a *joint action determination function*, a context determines a particular set of runs, in exactly the same way as an environment together with a protocol generates a set of runs.[4] Together with a view, this set of runs determines (in a way to be described below) a Kripke structure, which may be used to determine the actions enabled by the knowledge based program at a given run. Then, whereas we define joint *protocols* to be implementations of a knowledge based program with respect to an *environment*, they essentially define a joint *action determination function* $\langle \alpha_1, \ldots, \alpha_n \rangle$ to be an implementation of a knowledge based program with respect to a *context*. Thus, we may characterize one difference to be that our definition models the

[4]Strictly, our notion of run corresponds to their notion of *point*: we are able to avoid consideration of infinite runs because we do not deal with temporal formulae.

boundary between the "environment" and the "implementation" in such a way as to allow for a more liberal notion of implementation. In this respect, the difference between the two approaches amounts to a difference in modelling style.

However, our approach appears to be more general in certain other respects. Note that a joint protocol $\mathbf{P}$ for an environment E naturally generates a view $\{.\}^{\mathbf{P}}$, defined by $\{r\}_i^{\mathbf{P}} = (P_i(r), O_i(s))$. That is, according to $\{.\}^{\mathbf{P}}$ an agent's view in a given run consists of its protocol state (representing the agent's memory of past observations) together with its current observation. The perfect recall view may be obtained as a special case of this construction: it is readily seen that if $\mathbf{P}$ is a perfect recall protocol then $\{.\}^{\mathbf{P}}$ is equivalent to the perfect recall view.

Note that under the proposed correspondence, the notion of context of [FHMV95b] contains the initial protocol states q_i and update functions μ_i necessary to define the view $\{.\}^{\mathbf{P}}$. We believe that in a run r the view $\{r\}_i^{\mathbf{P}}$ is the most natural correspondent for the *local state* of the agent i, that they use to determine the accessibility relations. If so, whereas we have defined implementations with respect to an arbitrary view, their framework allows only implementations with respect to the view $\{.\}^{\mathbf{P}}$. In other words, their approach derives the view from the protocol states in the implementation.

Now, Proposition 4.1 does not hold for the framework of [FHMV95b] under these assumptions. That is, if the joint protocols $\mathbf{P}$ and $\mathbf{P}'$ are behaviourally equivalent with respect to an environment E, and $\mathbf{P}$ implements the knowledge based program $\mathbf{Pg}$ with respect to the view $\{.\}^{\mathbf{P}}$, then it is not necessarily the case that $\mathbf{P}'$ implements $\mathbf{Pg}$ with respect to the view $\{.\}^{\mathbf{P}'}$.

Example 7.1: Consider for Example 2.2 the joint protocol $\mathbf{P}'$ which for agent 2 is the perfect recall implementation, but for agent 1 has exactly one state. Noting that agent 1's observations contain the variables *sent, failed*, this protocol ignores the protocol state, and performs a *send* or a *wait* if these variables are both false, and a *wait* otherwise. It is easy to see that this is an implementation of $\mathbf{Pg}$ with respect to the perfect recall view, and $\mathcal{R} = \mathcal{R}(\mathbf{P}', E)$ is exactly the set of runs of the perfect recall implementation. However, in the system obtained from $\mathcal{R}$ and $\{.\}^{\mathbf{P}'}$, the formula $K_1 copy$ is false at all runs, hence so is $K_2 K_1 copy$. Thus, agent 2 should discard the copy immediately upon receipt, not after waiting a certain period. This shows that $\mathbf{P}'$ is not an implementation of $\mathbf{Pg}$ with respect to the view $\{.\}^{\mathbf{P}'}$.

Thus, at least under the proposed correspondence, the framework in [FHMV95b] does not naturally model the type of optimization we have studied in this paper. On the other hand, it appears that some of the optimizations of implementations with respect to the perfect recall view considered in the literature in fact are implementations of the original knowledge based program both with respect to the perfect recall view and with respect to the view generated by the optimization.[5] We feel that the nature of optimizations of implementations of knowledge based programs needs further study, and that this would lead to a clearer understanding of precisely what is being specified by a knowledge based program.

Let us now briefly comment on two other papers [FHMV95a, Var96] that concern the complexity of problems related to the implementation of knowledge based programs. Both

[5]Apparently, the optimization in [DM90] has this property, but the optimization in [HZ92] does not.

papers deal with *finite* contexts. Under the correspondence sketched above, this means that they deal also with finite state environments. However, while we consider implementations with respect to the perfect recall view, which takes an infinite number of values, they deal with implementations with respect to views that take only a finite number of values. Moreover, as noted above, their notion of implementation identifies the view with that derived from the implementation. The problems they consider are also orthogonal to ours. While the perfect recall view guarantees the existence and uniqueness (up to behavioural equivalence) of an implementation, but leaves the complexity of implementations as an issue, in finite state contexts the complexity of implementations is trivial, but the existence and uniqueness of implementations remains at issue, as is the question of whether a given protocol is an implementation. It is these problems that are addressed in [FHMV95a, Var96].

8 Conclusion

Theorem 6.2 indicates that, in general, we cannot expect to obtain efficient implementations of knowledge based programs with respect to the synchronous perfect recall view. This interpretation needs to be made cautiously, however, since the environment used in the proof is somewhat unnatural. Note that this result does not state that the perfect recall view yields complex implementations in *all* environments. In the light of our analysis of Example 2.1, it is plausible that tractable (if not finite state) implementations may still be available with respect to many environments of practical interest. We believe that there is a great deal of middle ground to be explored here.

If it does turn out that the complexity of perfect recall implementations persists for programs and environments of practical interest, then one must compromise, and seek implementations that use less information. How best to pursue such compromises remains a relatively unexplored topic.

Finally, while the framework we have developed has enabled us to model a certain sort of optimization, it may be that we have generalized too far. Arguably, Example 7.1 is artificial. Note that in **Pg**, agent 2 acts on the basis of information about agent 1's knowledge about the proposition *copy*, but agent 1's behaviour is completely independent of this proposition. It may be that most "reasonable" knowledge based programs have the property that all implementations are in fact implementations with respect to their own view. It seems that further work in this area is required, and that the methodology of knowledge based design of protocols still needs further elucidation.

References

[CKS81] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.

[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. *Information and Computation*, 88:156–186, 1990.

[FHMV95a] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. Knowledge-based programs. In *Proc. ACM Symposium on Principles of Distributed Computing*, 1995.

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

[Had87] V. Hadzilacos. A knowledge theoretic analysis of atomic commitment protocols. In *Proc. 6th ACM SIGACT-SIGMOD-SIGART Conf. on Principles of Database Systems*, pages 129–134, 1987.

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

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

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

[Maz86] M. Mazer. A knowledge theoretic account of recovery in distributed systems: The case of negotiated commitment. In M.Y. Vardi, editor, *Proc. 2nd Conf. on Theoretical Aspects of Reasoning about Knowledge*, pages 207–222, San Mateo, California, 1986. Morgan Kaufmann.

[Mey94] R. van der Meyden. Common knowledge and update in finite environments I. In *Proc. of the Conf. on Theoretical Aspects of Reasoning about Knowledge*, pages 225–242, 1994.

[Var96] M. Vardi. Implementing knowledge-based programs. In *Proc. of the Conf. on Theoretical Aspects of Rationality and Knowledge*, pages 15–30, 1996.