Symbolic Synthesis for Epistemic Specifications with Observational Semantics

Xiaowei Huang and Ron van der Meyden

School of Computer Science and Engineering
UNSW Australia
{xiaoweih,meyden}@cse.unsw.edu.au

Abstract. The paper describes a framework for the synthesis of protocols for distributed and multi-agent systems from specifications that give a program structure that may include variables in place of conditional expressions, together with specifications in a temporal epistemic logic that constrain the values of these variables. The epistemic operators are interpreted with respect to an observational semantics. The framework generalizes the notion of *knowledge-based program* proposed by Fagin et al (Dist. Comp. 1997). An algorithmic approach to the synthesis problem is developed that computes all solutions, using a reduction to epistemic model checking, that has been implemented using symbolic techniques. An application of the approach to synthesize mutual exclusion protocols is presented.

1 Introduction

In concurrent, distributed or multi-agent systems it is typical that agents must act on the basis of local data to coordinate to ensure global properties of the system. This leads naturally to the consideration of the notion of what an agent *knows* about the global state, given the state of its local data structures. *Epistemic* logic, or the logic of knowledge [9] has been developed as a formal language within which to express reasoning about this aspect of concurrent systems. In particular, *knowledge-based programs* [10], a generalization of standard programs in which agents condition their actions on formulas expressed in a temporal-epistemic logic, have been proposed as a framework for expressing designs of distributed protocols at the knowledge level. Many of the interesting analyses of problems in distributed computing based on notions of knowledge (e.g. [13]) can be cast in the form of knowledge-based programs.

Knowledge-based programs have the advantage of abstracting from the details of how information is encoded in an agent's local state, enabling a focus on *what* an agent needs to know in order to decide between its possible actions. On the other hand, this abstraction means that knowledge-based programs do not have an operational semantics. They are more like *specifications* than like programs in this regard: obtaining an implementation of a knowledge-based program requires that concrete properties of the agent's local state be found that are equivalent to the conditions on the agent's knowledge used in the program.

This gap has meant that knowledge-based analyses have been largely conducted as pencil and paper exercises to date, and only limited automated support for knowledge-based programming has been available. One approach to automation that has emerged

in the last ten years is the development of epistemic model checking tools [11,16]. These give a partial solution to the gap, in that they allow a putative implementation of a knowledge-based program to be verified for correctness (for examples, see [2,3]). However, this leaves open the question of how such an implementation is to be obtained, which still requires human insight.

Our contribution in this paper is to develop and implement an approach that automates the construction of implementations for knowledge-based programs for the case of the *observational semantics* for knowledge-based programs. (In earlier work [14] we dealt with stronger semantics for a more limited program syntax, see Section 7 for discussion). Our approach is to reduce the problem to model checking, enabling the investment in epistemic model checking to be leveraged to *automatically synthesize* implementations of knowledge-based programs. In particular, we build on *symbolic* techniques for epistemic model checking.

We in fact generalize the notion of knowledge-based program to a more liberal notion that we call *epistemic protocol specification*, based on a protocol template together with a set of temporal-epistemic formulas that constrain how the template is to be instantiated. This enables our techniques to be applied also to cover ideas such as the *sound local proposition* generalization of knowledge-based programs [8]. We illustrate the approach through an application of the knowledge-based programming methodology to the development of protocols for *mutual exclusion*. We give an abstract knowledge-based specification of a protocol for mutual exclusion, and show how our approach can automatically extract different protocols solving this problem.

2 A Semantic model for Knowledge and Time

For brevity, we present the theory of our approach at the level of semantic structures, since the symbolic algorithms we use work at this level. However, the input to our synthesis system is given in a programming notation, and, for clarity of exposition, we use this notation to present examples. For motivation, the reader may prefer to read the example in Section 4 first. Details of the mapping from programming syntax to the semantic structures are fairly standard, and left to the reader's intuition.

Let V be a finite set of boolean variables and Ags be a finite set of agents. The language CTLK(V, Ags) has the syntax:

$$\phi ::= v \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid EX\phi \mid E(\phi_1 U \phi_2) \mid EG\phi \mid K_i\phi$$

where $v \in V$ and $i \in Ags$. This is CTL plus the construct $K_i\phi$, which says that agent i knows that ϕ holds. We freely use standard operators that are definable in terms of the above, e.g., $AF\phi = \neg EG\neg\phi$ and $AG\phi = \neg E(true\, U \neg\phi)$.

A (finite) *model* is a tuple $M = (S, I, \longrightarrow, \{\sim_i\}_{i\in Ags}, \mathcal{F}, \pi)$ where S is a (finite) set of states, $I \subseteq S$ is a set of initial states, $\rightarrow \subseteq S \times S$ is a transition relation, $\sim_i$: $S \times S \rightarrow \{0, 1\}$ is an indistinguishability relation of agent i, component $\mathcal{F}$ is a fairness condition (explained below), and $\pi : S \rightarrow \mathcal{P}(V)$ is a truth assignment (here $\mathcal{P}(V)$ denotes the powerset of V.) A *path* in M from a state $s \in S$ is a finite or infinite sequence $s = s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow \ldots$ We assume that $\longrightarrow$ is serial, i.e. for each $s \in S$ there exists $t \in S$ such that $s \longrightarrow t$. We model fairness using the condition $\mathcal{F}$ by taking this to be a *generalized*

Büchi fairness condition, expressed as a set of sets of states: $\mathcal{F} = \{\alpha_1, \ldots, \alpha_k\}$ where each $\alpha_i \subseteq S$. An infinite path $s = s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow \ldots$ is *fair* with respect to $\mathcal{F}$ if, for each $i = 1 \ldots k$, there are infinitely many indices j for which $s_j \in \alpha_i$. Let $rch(M)$ be the set of *fair reachable* states of M, i.e., the set of states s_n (for some n) such that there exists a fair path $s_0 \longrightarrow s_1 \longrightarrow \ldots \longrightarrow s_n \longrightarrow s_{n+1} \longrightarrow \ldots$ with $s_0 \in I$ an initial state. We assume that $I \subseteq rch(M)$, i.e., all initial states are the source of a fair path.

The semantics of the language is given by a satisfaction relation $M, s \models \phi$, where $s \in rch(M)$ is a fair reachable state. This relation is defined inductively as follows:

1. $M, s \models v$ if $v \in \pi(s)$,
2. $M, s \models \neg\phi$ if not $M, s \models \phi$,
3. $M, s \models \phi_1 \vee \phi_2$ if $M, s \models \phi_1$ or $M, s \models \phi_2$,
4. $M, s \models EX\phi$ if there exists $t \in rch(M)$ such that $s \longrightarrow t$ and $M, t \models \phi$,
5. $M, s \models E(\phi_1 U \phi_2)$ if there exists a fair path $s = s_0 \longrightarrow s_1 \longrightarrow \ldots \longrightarrow s_n \rightarrow \ldots$ such that $M, s_k \models \phi_1$ for $k = 0 \ldots n-1$ and $M, s_n \models \phi_2$,
6. $M, s \models EG\phi$ if for there exists a fair path $s = s_0 \longrightarrow s_1 \longrightarrow \ldots$ with $M, s_k \models \phi$ for all $k \geq 0$,
7. $M, s \models K_i\phi$ if for all $t \in rch(M)$ with $s \sim_i t$ we have $M, t \models \phi$.

We are interested in models in which each of the agents runs a protocol in which it chooses its actions based on local information. To this end, we describe how a model may be obtained from the agents running such protocols in the context of an environment, which provides shared structure through which the agents can communicate.

An *environment* for agents Ags is a tuple $E = \langle Var_e, I_e, Acts, \longrightarrow_e, \mathcal{F}_e \rangle$, where

1. Var_e is a finite set of variables, from which we derive a set of states $S_e = \mathcal{P}(Var_e)$,
2. I_e is a subset of S_e, representing the initial states,
3. $Acts = \Pi_{i \in Ags} Acts_i$ is a set of joint actions, where each $Acts_i$ is a finite set of actions that may be performed by agent i,
4. $\longrightarrow_e \subseteq S_e \times Acts \times S_e$ is a transition relation, labelled by joint actions,
5. $\mathcal{F}_e$ is a generalized Büchi fairness condition over the states S_e.

Intuitively, a joint action a represents a choice of action a_i for each agent, performed simultaneously, and the transition relation resolves this into an effect on the state. We assume that $\longrightarrow_e$ is serial in the sense that for all $s \in S_e$ and $a \in Acts$ there exists $t \in S_e$ such that $s \xrightarrow{a} t$.

Semantically, a *concrete protocol* for agent $i \in Ags$ in such an environment E may be represented by a tuple $Prot_i = \langle PVar_i, LVar_i, OVar_i, I_i, Acts_i, \longrightarrow_i \rangle$, where

1. $PVar_i \subseteq Var_e$ is a subset of the variables of E, called the *parameter* variables of the protocol,
2. $LVar_i$ is a finite set of variables, understood as the *local variables* of the agent,
3. $OVar_i \subseteq PVar_i \cup LVar_i$ is the set of variables of the above two types that are observable to the agent, and on the basis of which the agent computes what it knows,
4. I_i is a subset of $\mathcal{P}(LVar_i)$, representing the initial states of the protocol,
5. $Acts_i$ is the set of actions that the agent is able to perform (this must match the set of actions associated to this agent in the environment),
6. $\longrightarrow_i \subseteq \mathcal{P}(PVar_i \cup LVar_i) \times Acts_i \times \mathcal{P}(LVar_i)$ is a serial labelled transition relation.

We assume that the sets Var_e and $LVar_i$, for $i \in Ags$, are mutually disjoint.[1]

Note that the transition relation $\longrightarrow_i$ indicates how an agent's local variables are updated when performing an action, which may depend on the current values of the parameter variables in the environment. This transition relation does not specify a change in the value of the parameter variables: changes to these are determined in the environment on the basis of the actions that this agent, and others, perform in the given step.

Given an environment E and a collection $\{Prot_i\}_{i \in Ags}$ of concrete protocols for the agents, we may construct a model $M(E, \{Prot_i\}_{i \in Ags}) = (S, I, \longrightarrow, \{\sim_i\}_{i \in Ags}, \mathcal{F}, \pi)$ as follows. The set of states is $S = \mathcal{P}(Var_e \cup \bigcup_{i \in Ags} LVar_i)$, i.e., the set of all assignments to the environment and local variables. We represent such states in the form $s = s_e \cup \bigcup_{i \in Ags} l_i$, where $s_e \subseteq Var_e$ and each $l_i \subseteq LVar_i$. Such a state s is taken to be an initial state in I if $s_e \in I_e$ and $l_i \in I_i$ for all agents i. That is, I is the set of states where the environment and each of the agents is in an initial state. The epistemic indistinguishability relations for agent i over the states S is defined by $s \sim_i t$ iff $s \cap OVar_i = t \cap OVar_i$, i.e., the states s and t have the same values for all of agent i's observable variables. The transition relation $\longrightarrow$ is given by $s_e \cup \bigcup_{i \in Ags} l_i \longrightarrow s'_e \cup \bigcup_{i \in Ags} l'_i$ if there exists a joint action a such that $s_e \xrightarrow{a}_e s'_e$ and $(s_e \cap PVar_i) \cup l_i \xrightarrow{a_i}_i l'_i$ for each agent i. We take the fairness condition $\mathcal{F}$ to contain

$$\{s_e \cup \bigcup_{i \in Ags} l_i \mid s_e \in \alpha, l_1 \in \mathcal{P}(LVar_1), \ldots, l_n \in \mathcal{P}(LVar_n)\}$$

for each $\alpha \in \mathcal{F}_e$. That is, we impose the environment's fairness constraints on the environment portion of the state. The assignment π is given by $\pi(s) = s$.

3 Epistemic Protocol Specifications

Protocol templates generalize concrete protocols by introducing some variables that may be instantiated with a boolean expression in the observable variables in order to obtain a concrete protocol. Formally, a *protocol template* for agent $i \in Ags$ is a tuple $\mathsf{P}_i = \langle KVar_i, PVar_i, LVar_i, OVar_i, I_i, Acts_i, \longrightarrow_i \rangle$

1. $KVar_i$ is a set of variables, disjoint from all the other sets of variables, that we call the *template* variables,
2. $PVar_i, LVar_i, OVar_i, Acts_i$ are, respectively, a set of parameter variables, local variables, observable variables and actions of agent i, exactly as in a concrete protocol; as in concrete protocols, we obtain a set of local states $\mathcal{P}(LVar_i)$,
3. $I_i \subseteq \mathcal{P}(LVar_i)$ is a set of initial local states,
4. $\longrightarrow_i \subseteq \mathcal{P}(KVar_i \cup PVar_i \cup LVar_i) \times Acts_i \times \mathcal{P}(LVar_i)$ is a transition relation that describes how local states are updated, depending on the value of the template variables, parameter variables, local variables and action performed.

[1] We could also include a fairness condition, but exclude this here for brevity. We do not assume that $LVar_i \subseteq OVar_i$: this allows the impact on knowledge of particular local variables to be studied, and helps in managing the complexity of our technique, which scales exponentially in the number of observable variables.

An *epistemic protocol specification* is a tuple $\mathcal{S} = \langle Ags, E, \{\mathtt{P}_i\}_{i \in Ags}, \Phi \rangle$, consisting of a set of agents Ags, an environment E for Ags, a collection of protocol templates $\{\mathtt{P}_i\}_{i \in Ags}$ for environment E, and a collection of epistemic logic formulas Φ over the agents Ags and variables $Var_e \cup \bigcup_{i \in Ags}(KVar_i \cup LVar_i)$.

Epistemic protocol specifications generalize the notion of knowledge-based program [9,10]. Essentially, these are epistemic protocol specifications in which, for each agent $i \in Ags$ and each template variable $v \in KVar_i$, the set Φ contains a formula of the form $AG(v \Leftrightarrow K_i \psi)$. That is, each template variable is associated with a formula of the form $K_i \psi$, expressing some property of agent i's knowledge, and we require that the meaning of the template variable be equivalent to this property.

Epistemic protocol specifications also encompass the *sound local proposition* interpretation of knowledge-based programs proposed by Engelhardt et al [8]: these associate to each template variable v a formula ϕ and require that v be interpreted by a local proposition (under the observational semantics, this is just a condition on the observable variables), such that the system satisfies $AG(v \Rightarrow \psi)$. (By the assumption of locality of v, this is equivalent to satisfying $AG(v \Rightarrow K_i \psi)$.)

To implement an epistemic protocol specification with respect to the observational semantics, we need to replace each template variable v in each agent i's protocol template by an expression over the agent's observable variables, in such a way that the specification formulas are satisfied in the model resulting from executing the resulting standard program. We now formalize this semantics.

Let θ be a substitution mapping each template variable $v \in KVar_i$, for $i \in Ags$, to a boolean expression over the observable variables $OVar_i$ of agent i's protocol $\mathtt{P}_i$. If we apply this substitution to $\mathtt{P}_i$, we obtain a standard protocol $\mathtt{P}_i\theta = \langle PVar_i, LVar_i, OVar_i, I_i, Acts_i, \longrightarrow_i^\theta \rangle$, where the template variables $KVar_i$ have been removed, and all the other components are as in the protocol template, except that we derive the concrete transition relation $\longrightarrow_i^\theta \subseteq \mathcal{P}(PVar_i \cup LVar_i) \times Acts_i \times \mathcal{P}(LVar_i)$ from the transition relation $\longrightarrow_i \subseteq \mathcal{P}(KVar_i \cup PVar_i \cup LVar_i) \times Acts_i \times \mathcal{P}(LVar_i)$ in the protocol template, as follows.

Since, for each $v \in KVar_i$, the value $\theta(v)$ is an expression over the variables $OVar_i$, which is a subset of $PVar_i \cup LVar_i$, we may evaluate $\theta(v)$ on states in $\mathcal{P}(PVar_i \cup LVar_i)$. Given a state $s \in \mathcal{P}(PVar_i \cup LVar_i)$, define $s^\theta \in \mathcal{P}(KVar_i)$ by $s^\theta = \{v \in KVar_i \mid s \models \theta(v)\}$. We then define $\longrightarrow_i^\theta$ by $s \xrightarrow{a}_i^\theta l_i'$ when $s^\theta \cup s \xrightarrow{a}_i l_i'$, for $s \in \mathcal{P}(PVar_i \cup LVar_i)$ and $a \in Acts_i$ and $l_i' \in \mathcal{P}(LVar_i)$.

The substitution θ may also be applied to the specification formulas in Φ. Each $\phi \in \Phi$ is a formula over variables $Var_e \cup \bigcup_{i \in Ags}(KVar_i \cup LVar_i)$. Replacing each occurrence of a variable $v \in \bigcup_{i \in Ags} KVar_i$ by the formula $\theta(v)$ over $Var_e \cup \bigcup_{i \in Ags} LVar_i$, we obtain a formula $\phi\theta$ over $Var_e \cup \bigcup_{i \in Ags} LVar_i$. We write $\Phi\theta$ for $\{\phi\theta \mid \phi \in \Phi\}$.

We say that such a substitution θ provides an *implementation* of the epistemic protocol specification $\mathcal{S}$, provided $M(E, \{\mathtt{P}_i\theta\}_{i \in Ags}) \models \Phi\theta$. The problem we study in this paper is the following: given an environment E and an epistemic protocol specification $\mathcal{S}$, synthesize an implementation θ. This is an inherently complex problem. To provide a fair comparison with the performance of our implementation, we measure it here as a function of the size of a succint representation (by means of boolean formulas for the environment and protocol components, or programs PTIME encodable by such formulas). Since the size of the output implementation θ could be of exponential size in the

number of observable variables, we measure the complexity of determining the existence of an implementation: even this is already hard, as the following result shows:

Theorem 1. *The problem of determining the existence of an implementation of a given epistemic protocol specification is NEXPTIME-complete.*

4 Example: Mutual Exclusion

To illustrate our approach we use a running example concerned with mutual exclusion. Mutual exclusion protocols [7] are intended for settings where it is required that only one of a set of agents has access to a resource (e.g. a printer, or a write access to a file) at a given time. There exists a large literature on this topic, with many different approaches to its solution [17].

To model the structure of a mutual exclusion protocol, we suppose that each agent has three states: `waiting`, `trying`, and `critical`. Intuitively, while in the `waiting` state, the agent does not require the resource, and it idles for some period of time until it decides that it needs access to the resource. It then enters the `trying` state, where it waits for permission to use the resource. Once this permission has been obtained, it enters the `critical` state, within which it may use the resource. Once done, it exits the critical state and returns to the waiting state. The overall structure of the protocol is therefore a cycle `waiting` → `trying` → `critical` → `waiting`. To ensure fair sharing of the resource, we assume that no agent remains in its critical state forever.

To avoid the situation where two agents are using the resource at the same time, the specification requires that no two agents are in the `critical` state simultaneously. In order for a solution to the mutual exclusion problem to satisfy this specification, the agents need to share some information about their state and to place an appropriate guard on the transition from the `trying` state to the `critical` state. Mutual exclusion protocols differ in their approach to these requirements by providing different ways for agents to use shared variables to distribute and exploit information about their state.

Our application of our synthesis methodology assumes that the designer has some intuitions concerning what information needs to be distributed, and writes the protocol and environment in so as to reflect these ideas concerning information distribution. However, given a pattern of communication, it may still be a subtle matter to determine what information an agent can deduce from some particular values of its observable variables. We use the epistemic specification to relate the information distributed and the conditions used by the agent to make state transitions.

A general structure for a mutual exclusion protocol is given as a protocol template in Figure 1. The code uses a simple programming language, containing a Dijkstra style nondeterministic-if construct **if** $e_1 \to P_1 \ [] \ \ldots \ [] \ e_k \to P_p$ **fi** which nondeterministically executes one of the statements P_i for which the corresponding guard e_i evaluates to true. The final e_k may be the keyword **otherwise** which represents the negation of the disjunction of the preceding e_i. If there is no otherwise clause and none of the guards in a conditional are true then the program defaults to a skip action. Evaluation of guards in **if** and **while** statements is assumed to take zero time, and a transition occurs only once an action is encountered in the execution. This applies also to an exit from a while loop.

```
/* protocol for agent i; initially state[i] = waiting */;
  while True do
    begin
    /* waiting section: wait for some amount of time before entering the trying section */
    while state[i] = waiting do
      if True → skip [] True → EnterTry fi;
    /* trying section: wait until the condition represented by template variable x_i holds */
    while state[i] = trying do
      if x_i → EnterCrit [] otherwise → skip fi;
    /* critical section: stay critical for a random amount of time,
      return to waiting when done */
    while state[i] = critical do
      if True → skip [] True → ExitCrit fi
    end
```

Fig. 1. Protocol template for a mutual exclusion solution

Variables in the programming notation are allowed to be of finite types (these are boolean encoded in the translation to the semantic level. We assume that a vector of variables `state` indexed by agent names records the state in {`waiting`, `trying`, `critical`} of each agent. Thus, mutual exclusion can be specified by the formula

$$AG \bigwedge_{i,j \in Ags,\ i \neq j} \neg(\mathtt{state}[i] = \mathtt{critical} \land \mathtt{state}[j] = \mathtt{critical})\,. \tag{1}$$

The protocol template also uses three actions for the agent: **EnterTry**, **EnterCrit** and **ExitCrit**, which correspond to entering the trying, critical and waiting states respectively. We take the variables `state`[i] to be included in the set of environment variable Var_e. When there are n agents, with $Ags = \{0 \ldots n-1\}$, we assume the code for the environment transition always includes the following:

```
for i = 0...n − 1 do
  if i.EnterTry → state[i] := trying
  [] i.EnterCrit → state[i] := critical
  [] i.ExitCrit → state[i] := waiting
  fi
```

(Here $i.a$ is a proposition that holds during the computation of any transition in which agent i performs the action a.) Additional code describing the effect of these actions may be included, which represents the way that the agents distribute information to each other concerning their state. A number of different instantiations of this additional code for these actions are discussed below.

In our epistemic specifications, we include in Φ, for each agent i, the following constraint on the template variable $\mathbf{x}_i$ that guards entry to the critical section:

$$AG(\mathbf{x}_i \Leftrightarrow K_i(AX(\bigwedge_{j \in Ags} (j \neq i \Rightarrow \mathtt{state}[j] \neq \mathtt{critical}))) \tag{2}$$

Intuitively, this states that agent i enters its critical section when it knows that, after next transition, no other agent will be in its critical section. Note that this formula falls

within the structure of the specifications for knowledge-based programs as discussed above. We also include in Φ the formula

$$\bigwedge_{i \in Ags} AG(\mathtt{state}[i] = \mathtt{trying} \Rightarrow AF\mathtt{state}[i] = \mathtt{critical}) \tag{3}$$

which requires that the protocol synthesized ensures that whenever an agent starts trying, it is eventually able to enter its critical section.

One of the benefits of knowledge-based programs is that they enable the essential reasons for correctness of a protocol to be abstracted in a way that separates the information on the basis of which an agent acts from the way that this information is encoded in the state of the system. This, it is argued, allows for simpler correctness proofs that display the commonalities between different protocols solving the same problem.

This can be seen in the present specification: if the agents follow this specification, then they will not violate mutual exclusion. The proof of this is straightforward; we sketch it informally. Suppose that there is a violation of mutual exclusion, and let t be the earliest time that we have $\mathtt{state}[i] = \mathtt{critical} \wedge \mathtt{state}[j] = \mathtt{critical}$ for some pair of agents $i \neq j$. Then either i or j performs **EnterCrit** to enter its critical section at time $t-1$. Assuming, without loss of generality, that it is agent i, we have $\mathbf{x}_i$ at time $t-1$, so by (2), we must have $K_i(AX(\bigwedge_{k \in Ags}(k \neq i \Rightarrow \mathtt{state}[\mathrm{k}] \neq \mathtt{critical})))$ at time $t-1$. But then (since validity of $K_i\phi \Rightarrow \phi$ is immediate from the semantics of the knowledge operator), it follows that $AX(\mathtt{state}[j] \neq \mathtt{critical})$ at time $t-1$, contradicting the fact that the protocol makes a transition, in the next step, to a state where $\mathtt{state}[j] = \mathtt{critical}$.

We note that only the implication from left to right in (2) is used in this argument, and it would also be valid if we removed the knowledge operator. This is an example of a general point that led to the "sound local proposition" generalization of knowledge-based programs proposed in [8]. However, weakening (2) to only the left to right part allows the trivial implementation $\theta(\mathbf{x}_i) = False$, where no agent ever enters its critical section. The implication from right to left in (2) amounts to saying that rather than this very weak implementation, we want the strongest possible implementation where an agent enters its critical section *whenever* it has sufficient information. Here the knowledge operator is essential since, in general, the non-local condition inside the knowledge operator will not be equivalent to a local proposition implementing $\mathbf{x}_i$.

The description above is not yet a complete solution to the mutual exclusion problem: it remains to describe how agents distribute information about their state, and how the data structures encoding this information are related to a local condition of the agent's state that can be substituted for the template variable so as to satisfy the epistemic specification. We consider here two distinct patterns of information passing, based on two overall systems architectures. In both cases $KVar_i = \{\mathbf{x}_i\}$ for all agents i.

Ring Architecture: In the *ring* architecture we consider n agents $Ags = \{0, \ldots, n-1\}$ in a ring, with agent i able to communicate with agent $i + 1 \bmod n$. This communication pattern is essentially that of *token ring* protocols. In this case we assume that communication is by means of a single bit for each agent i, represented by a variable $\mathtt{bit}[i]$. We take $Var_e = \{\mathtt{bit}[i], \mathtt{state}[i] \mid i = 0 \ldots n-1\}$ and let $PVar_i = \{\mathtt{bit}[i], \mathtt{state}[i]\}$ and $LVar_i = \emptyset$ and $OVar_i = \{\mathtt{bit}[i]\}$. Agent i is able to affect its own bit as well as the

bit of agent $i+1 \bmod n$ through its actions. More precisely, we add to the above code for the environment state transitions the following semantics for the **ExitCrit** actions:

for $i = 0 \ldots n-1$ do
  if i.**ExitCrit** then begin $\texttt{bit}[i] := \neg\texttt{bit}[i]$; $\texttt{bit}[i+1 \bmod n] := \neg\texttt{bit}[i+1 \bmod n]$ end

That is, on exiting the critical section, the agent flips the value of its own bit, as well as the value of its successor's bit. To ensure fairness, we also add to the environment, for each agent i, the Büchi fairness constraint $\texttt{state}[i] \neq \texttt{waiting}$, which says that the agent does not remain forever in the waiting state, but eventually tries to go critical. This ensures that this agent takes its turn and does not forever block other agents who may be trying to enter their critical section. We also add the fairness constraints $\texttt{state}[i] \neq \texttt{critical}$ to ensure that no agent stays in its critical section forever. (However, we do not include $\texttt{state}[i] \neq \texttt{trying}$ as a fairness constraint: it is up to the protocol to ensure that an agent is eventually able to enter its critical section once it starts trying!)

Broadcast architecture: In the *broadcast architecture*, we assume that the n agents broadcast their state to all other agents. In this case, no additional variables are needed and we take $Var_e = \{\texttt{state}[j] \mid j = 0 \ldots n-1\}$. Also, for each agent i, we take $PVar_i = OVar_i = Var_e$ and $LVar_i = \emptyset$. The only code required for the actions **EnterTry**, **EnterCrit** and **ExitCrit** is that given above for updating the variables $\texttt{state}[i]$. We do not need to assume eventual progression from waiting to trying in this case (we allow an agent to wait forever, in this case) so the only fairness constraints are $\texttt{state}[i] \neq \texttt{critical}$ to ensure that no agent is forever critical.

Implementation Example: We describe an example of an implementation in the case of the ring architecture for mutual exclusion described above. We assume that initially, $\texttt{bit}[i] = 0$ for all agents i. Consider the substitution defined by $\theta(\mathbf{x}_i) = \neg\texttt{bit}[i]$ if $i = 0$ and $\theta(\mathbf{x}_i) = \texttt{bit}[i]$ if $i \neq 0$. (Note that these are boolean expressions in the observable variables $OVar_i = \{\texttt{bit}[i]\}$.) It can be shown that this yields an implementation of the epistemic protocol specification for the ring architecture (we discuss our automated synthesis of this implementation below.) Intuitively, in this implementation, agent 0 initially holds the token, represented by $\texttt{bit}[0] = 0$. After using the token to enter its critical section, it sets $\texttt{bit}[0] = 1$ to relinquish the token, and $\texttt{bit}[1] = 1$ in order to pass the token to agent 1. Thus, for agent 1, holding the token is represented by $\texttt{bit}[1]$ being true. The same holds for the remaining agents. (Obviously, there is an asymmetry in these conditions for the agents, but any solution needs to somehow break the symmetry in the initial state.) Intuitively, specification formula (2) holds because the implementation maintains the invariant that at most one of the conditions $\theta(\mathbf{x}_i)$ guarding entry to the agents' critical sections holds at any time, and when it is false, the agent is not in its critical section. Thus, the agent i for which $\theta(\mathbf{x}_i)$ is true knows that no other agent is in, or is able to enter, its critical section. Consequently, it knows that no other agent will be in its critical section at the next moment of time.

5 Reduction of Synthesis to Model Checking

We now show how the synthesis of implementations of epistemic protocol specifications $\mathcal{S}$ can be reduced to the problem of epistemic model checking. The approach

essentially constructs a model that encodes all possible guesses of the environment, and then uses model checking to determine which guesses actually yield an implementation. The consideration of all guesses is done in bulk, using symbolic techniques.

For each agent i, let O_i be the set of boolean assignments to $OVar_i$; this represents the set of possible observations that agent i can make. We may associate to each $o \in O_i$ a conjunction ψ_o of literals over variables v in $OVar_i$, containing literal v if $o(v) = 1$ and $\neg v$ otherwise.

Since an implementation $\theta(v)$ for a template variable v is a boolean condition over observable variables, we may equivalently view this as corresponding to the set of observations on which it holds. This set can in turn be represented by its characteristic mapping from O_i to boolean values. To represent the entire implementation θ, we introduce for each agent $i \in Ags$ a set of new boolean variables X_i, containing the variables $x_{i,o,v}$, where $o \in O_i$ and $v \in KVar_i$. Let $X = \bigcup_{i \in Ags} X_i$. We call X the implementation variables of the epistemic protocol specification $\mathcal{S}$.

A candidate assignment θ for an implementation of the epistemic protocol specification, can be represented by a state χ_θ over the variables X, such that for an observation $o \in O_i$ and variable $v \in KVar_i$, we have $x_{i,o,v} \in \chi_\theta$ iff $\theta(v)$ holds with respect to assignment o. Conversely, given a state χ over the variables X, we can construct an assignment θ_χ mapping, for each agent i, the variables $KVar_i$ to boolean conditions over $OVar_i$, by

$$\theta_\chi(v) = \bigvee_{o \in O_i,\ x_{i,o,v} \in \chi} \psi_o \ .$$

To reduce synthesis to model checking, we construct a system in which the state space is based on the variables X as well as a state of a model for the implementation. Given an environment $E = \langle Var_e, I_e, Acts, \longrightarrow_e \rangle$, we define an environment $E^X = \langle Var_e^X, I_e^X, Acts, \longrightarrow_e^X \rangle$ as follows. The variables making up states are defined to be $Var_e^X = Var_e \cup X$. The initial states are given by $I_e^X = \{s \cup \chi \mid s \in I_e,\ \chi \in \mathcal{P}(X)\}$, i.e., an initial state is obtained by adding any assignment to variables X to an initial state of E. The set of actions $Acts$ is the same as in the environment E. Finally, the transition relation $\longrightarrow_e^X$ is given by $s \cup \chi \longrightarrow_e^X s' \cup \chi'$ iff $s \longrightarrow_e s'$ and $\chi = \chi'$ where $s, s' \in \mathcal{P}(Var_e)$ and $\chi, \chi' \in \mathcal{P}(X)$.

Additionally, for each agent i, we transform its protocol template $\mathsf{P}_i = \langle KVar_i, PVar_i, LVar_i, OVar_i, I_i, Acts_i, \longrightarrow_i \rangle$ into a concrete protocol $\mathsf{P}_i^X = \langle PVar_i^X, LVar_i, OVar_i^X, I_i, Acts_i, \longrightarrow_i^X \rangle$ for the environment E^X. The local variables $LVar_i$ and the initial states I_i are exactly as in the protocol template. The parameter variables are given by $PVar_i^X = PVar_i \cup X$, and the observable variables are given by $OVar_i^X = OVar_i \cup X$. The transition relation $\longrightarrow_i^X \subseteq \mathcal{P}(PVar_i \cup X \cup LVar_i) \times Acts_i \times \mathcal{P}(LVar_i)$ is derived from the transition relation $\longrightarrow_i \subseteq \mathcal{P}(KVar \cup PVar_i \cup LVar_i) \times Acts_i \times \mathcal{P}(LVar_i)$ as follows. For $s \in \mathcal{P}(PVar_i \cup LVar_i)$ and $\chi \in \mathcal{P}(X)$, define $\kappa(s, \chi) \in \mathcal{P}(KVar)$ by

$$\kappa(s, \chi) = \{v \in KVar_i \mid s \cup \chi \models \bigvee_{o \in O_i} \psi_o \wedge x_{i,o,v}\} \ .$$

For $l_i' \in \mathcal{P}(LVar_i)$ and $a \in Acts_i$, we then let $s \cup \chi \xrightarrow{a}_i^X l_i'$ iff $s \cup \kappa(s, \chi) \xrightarrow{a}_i l_i'$.

Intuitively, since the assignment χ to the variables X encodes an implementation θ, we make these variables an input to the transformed protocol, which uses them to make

decisions that depend on the protocol template variables when executing the protocol template. In particular, when an observation $o = s \cap OVar_i \in O_i$ (equivalently, $s \models \psi_o$) satisfies $x_{i,o,v} \in \chi$, this corresponds to the template variable v taking the value true on state s according to the implementation $\theta(v)$. We therefore execute a transition of the protocol template in which v is taken to be true.

Note that the definition of the sets $OVar_i^X$ makes the variables X observable to all the agents: this effectively makes the particular implementation being run *common knowledge* to the agents, as it is in the system that we obtain from each concrete implementation. However, the combined transformed environment and transformed protocol templates represent not just one implementation, but *all possible implementations.* This is stated formally in the following result.

Theorem 2. *Let $\mathcal{S} = \langle Ags, E, \{\mathrm{P}_i\}_{i\in Ags}, \Phi\rangle$ be an epistemic protocol specification, and let X be the set of implementation variables of $\mathcal{S}$. For each implementation θ of $\mathcal{S}$, we have $M(E^X, \{\mathrm{P}_i^X\}_{i\in Ags}), s \models \Phi\theta$ for all initial states s of $M(E^X, \{\mathrm{P}_i^X\}_{i\in Ags})$ with $s \cap X = \chi_\theta$. Conversely, suppose that $\chi \in \mathcal{P}(X)$ is such that $M(E^X, \{\mathrm{P}_i^X\}_{i\in Ags}), s \models \Phi\theta_\chi$ for all initial states s of $M(E^X, \{\mathrm{P}_i^X\}_{i\in Ags})$ with $s \cap X = \chi$. Then θ_χ is an implementation of $\mathcal{S}$.*

This result gives a reduction from the synthesis problem to the well understood problem of model checking. Any algorithm for model checking specifications expressible in the framework can now be applied. In particular, symbolic model checking techniques apply. We have implemented the above approach as an extension of binary-decision diagram (BDD) based epistemic model checking algorithms already implemented in the epistemic model checker MCK [11], which handles formulas in $\mathrm{CTL}^*\mathrm{K}_n$ with fairness constraints using BDD based representations. The model checking techniques involved are largely standard, as in [6], with a trivial extension to handle the epistemic operators (these just require BDD's representing the set of reachable states and an equivalence on observable variables.) We make one optimization, based on the observation that the variables X encoded in the state do not actually change on any given run. We can therefore reduce the number of BDD variables required to represent the transition relation by retaining only one copy of these variables. Also, we first compute the observations $o \in O_i$ that can occur at reachable states in any putative implementation, to reduce the set X to variables $x_{i,o,v}$ where o is in fact a possible observation.

We note that the reduction does entail a blowup in the number of variables. Suppose we have n agents, with the number of observable variables of agent i being k_i. Then the size of the set X_i could be as large as $2^{k_i}|KVar_i|$, so that $|X| = \sum_{i=1\ldots n} 2^{k_i}|KVar_i|$ is the number of new variables that need to be included in the BDD computation. With BDD-based symbolic model checking currently typically viable for numbers of the BDD variables in the order of 100's, this places an inherent limit on the size of example that we can expect to handle using our technique. Evidently, the technique favours examples in which the number of observable variables per agent is kept small. This is reflected in the results obtained for our running example, which we now discuss.

6 Solutions to the mutual exclusion example

We have applied our implementation of the above reduction to the epistemic protocol specifications for mutual exclusion described in Section 4. Our technique computes the

No. of Agents	2	3	4	5	6	7	8
Ring (time)	0.3	1.7	5.5.	17.2	157.7	509.1	597
(No. BDD vars)	22	33	44	55	66	77	88
Broadcast (time)	0.2	194.2					
(No. BDD vars)	34	105	356				

Table 1. Running times (s) of Synthesis Experiments

set of all possible implementations. We now describe the implementations obtained for the two versions of this specification.

We note that, as defined above, two implementations, corresponding to substitutions θ_1 and θ_2 for the template variables, may be behaviorally equivalent, yet formally distinct. Define the equivalence relation $\sim$ on such substitutions by $\theta_1 \sim \theta_2$ if $M(E, \{\mathsf{P}_i\theta_1\}_{i\in Ags})$ and $M(E, \{\mathsf{P}_i\theta_2\}_{i\in Ags})$ have the same set of reachable states, and for all such reachable states s, and all template variables v, we have $M(E, \{\mathsf{P}_i\theta_1\}_{i\in Ags}), s \models \theta_1(v)$ iff $M(E, \{\mathsf{P}_i\theta_2\}_{i\in Ags}), s \models \theta_2(v)$. Intuitively, this means that θ_1 and θ_2 are equivalent, except on unreachable states. We treat such implementations as identical and return only one element of each equivalence class.

Ring Architecture: We have already discussed one of the possible implementations of the epistemic protocol specification for the ring architecture as the example in Section 4, viz., that in which $\theta(\mathbf{x}_0) = \neg\mathtt{bit}_0$ and $\theta(\mathbf{x}_i) = \mathtt{bit}_i$ for $i \neq 0$. Our synthesis system returns this as one of the implementations synthesized. As discussed above, this implementation essentially corresponds to a token ring protocol in which agent 0 initially holds the token. By symmetry, it is easily seen that we can take any agent k to be the one initially holding the token, and each such choice yields an implementation, with $\theta(\mathbf{x}_k) = \neg\mathtt{bit}_k$ and $\theta(\mathbf{x}_i) = \mathtt{bit}_i$ for $i \neq k$. Our synthesis system returns all these solutions, but also confirms that there are no others. Thus, up to symmetry, there is essentially just one implementation for this specification.

We note that, whatever the total number of agents n, the number of variables observable to agent i is just one, so we have $|X_i| = 2$ and we add $|X| = 2n$ variables to the underlying BDD for model checking in order to perform synthesis. This gives a slow growth rate in the number of BDD variables as we scale the number of agents, and enables us to deal with moderate size instances. Table 1 gives the performance results for our implementation as we scale the number of agents.[2] The total number of BDD variables per state (i.e., the environment variables, local protocol and program counter variables and X) is also indicated.

Broadcast Architecture: In case of the broadcast architecture, the number of variables that need to be added for synthesis increases much more rapidly. In case of n agents, we have $|X_i| = 2^{2n}$ (since we need two bits to represent each agent's state variable $\mathtt{state}[j]$), and $|X| = n2^{2n}$. Accordingly, the approach works only on modest scale examples. We describe the solutions obtained in the case of 3 agents. Our synthesis procedure computes that there exist 6 distinct solutions, which amount essentially to one solution under permutation of the roles of the agents. To understand this solution,

[2] Our experiments were conducted on a Debian Linux system, 3.3GHz Intel i5-2500 CPU, with each process allocated up to 500M memory.

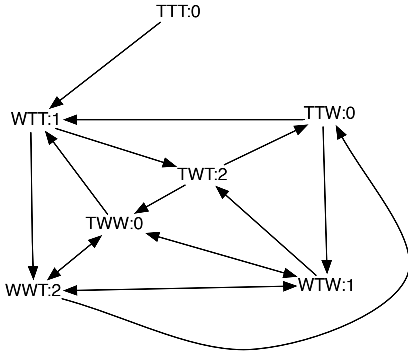

Fig. 2. Structure of a ME protocol synthesized (3 agents, broadcast architecture)

note first that if any agent is in its critical section, all others know this, but cannot know whether the agent will exit its critical section in the next step. It follows that no agent is able to enter its critical section in the next step. It therefore suffices to consider the behavior of the solution on states where no agent is in its critical section, but at least one agent is in state `trying`. We describe this by means of the graph in Figure 2. Vertices in this graph indicate the protocol state inhabited by each of the agents, as well as the agent that the protocol selects for entry to the critical state, e.g., WTT:1 indicates that agent 0 is in state `waiting`, and agents 1 and 2 are in state `trying`, and that agent 1 enters its critical state in the next step. The edges point to possible successor states reached at the time the selected agent next exits its critical state. (Note that, at this time, no other agent has had the opportunity to enter its critical state, but another agent may have moved from `waiting` to `trying`, so there is some nondeterminism in the graph.) It can be verified by inspection (a focus on the upper triangle suffices, since only one agent is trying in states in the lower triangle) that the solution is fair: there is no cycle where an agent is constantly trying but never selected for entry to the critical section.

7 Related Work

Most closely related to our work in this paper are results on the complexity of verifying and deciding the existence of knowledge-based programs [20,10], with respect to what is essentially the observational semantics. The key idea of these complexity results is similar to the one we have used in our construction: guess a *knowledge assignment* that indicates at which observations (local states, in their terminology) a knowledge formula holds, and verify that this corresponds to an implementation. However, our epistemic specifications are syntactically more expressive than knowledge-based programs, and some of the details of their work are more complex, in that a labelling of runs by subformulas of knowledge formulas is also required. In part this is because of the focus on *linear time temporal logic* in this work, compared to our use of branching time temporal logic. This work also does not consider any concrete implementation of the theoretical

results using symbolic techniques. The complexity bounds for determining the existence of an implementation of a knowledge-based program in [20,10] (NP-complete for atemporal knowledge-based programs and PSPACE-complete for programs with LTL-based knowledge conditions) are lower than our EXPTIME bound in Theorem 1 because they are based on an explicit-state rather than variable-based representation.

Our focus in this paper is on the observational semantics for knowledge. Other semantics have been studied from the point of view of synthesis. Van der Meyden and Vardi [18] consider, for the synchronous perfect recall semantics, the problem of synthesizing a protocol satisfying a formula in a linear time temporal epistemic logic in a given environment (with no limitations on the program structure of the solution). They show the problem to be decidable only in the case of a single agent. Some restrictions on environments and specifications are identified in [19] under which the problem becomes decidable. The problem can also be shown to be decidable for knowledge-based programs that run only a finitely bounded number of steps: a symbolic technique for implementing such programs with respect to synchronous semantics including perfect recall and clock based semantics is developed in [14].

A number of papers have also applied model checking of knowledge properties to synthesize distributed control strategies [4,12,15]. These works do not deal with knowledge-based programs per se, however, and it is not guaranteed that the implementing condition is equivalent to the desired knowledge property in the protocol synthesized. However, these solutions would be included in the space of solutions of specifications expressible in our more general framework.

Bonollo et al [5] have previously proposed knowledge-based specifications for distributed mutual exclusion. However, this work deals only with the specification level, and does not relate the specification developed to any concrete implementations.

Bar-David and Taubenfeld [1] considered the automated synthesis of mutual exclusion protocols. In some respects their approach is more general than ours, in that they synthesize the entire program structure, not just the implementations of conditions within a program template. However, they do not consider epistemic specifications. Also, compared to our symbolic approach, they essentially conduct a brute force search over all possible implementations up to a given size of program code, (with some optimizations to avoid redundant work) and they use explicit state model checking to verify an implementation. This limits the number of agents to which their approach can be expected to scale: they consider only two-agent systems. They mention a construction by which a two-agent solution can used to construct an n-agent solution, but this does not amount to generation of all possible solutions for the n-agent case.

8 Conclusion

Our focus in this paper has been to develop an approach that enables the space of all solutions to an epistemic protocol specification to be explored. Our implementation gives the first tool with this capability with respect to the observational semantics for knowledge, opening up the ability to more effectively explore the overall methodology of the knowledge-based approach to concurrent systems design through experimentation with examples beyond the simple mutual exclusion protocol we have considered.

Applications of the tool to the synthesis of fault-tolerant protocols, where the flow of knowledge is considerably more subtle than in the reliable setting we have considered, is one area that we intend to explore in future work. Use of alternative model checking approaches to the BDD-based algorithm we have used (e.g., SAT-based algorithms) are also worth exploring.

References

1. Y. Bar-David and G. Taubenfeld. Automatic discovery of mutual exclusion algorithms. In *Proc. Int. Conf. Distributed Computing*, Springer LNCS 2848, pages 136–150, 2003.
2. O. A. Bataineh and R. van der Meyden. Abstraction for epistemic model checking of dining-cryptographers based protocols. In *Proc. TARK*, pages 247–256, 2011.
3. K. Baukus and R. van der Meyden. A knowledge based analysis of cache coherence. In *Proc. 6th Int. Conf. on Formal Engineering Methods*, pages 99–114, 2004.
4. S. Bensalem, D. Peled, and J. Sifakis. Knowledge based scheduling of distributed systems. In *Time for Verification, Essays in Memory of Amir Pnueli*, pages 26–41. LNCS 6200, 2010.
5. U. Bonollo, R. van der Meyden, and E. Sonenberg. Knowledge-based specification: Investigating distributed mutual exclusion. In *Bar Ilan Symposium on Foundations of AI*, 2001.
6. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.
7. E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.
8. K. Engelhardt, R. van der Meyden, and Y. Moses. Knowledge and the logic of local propositions. In *Proc. Conf. Theoretical Aspects of Knowledge and Rationality*, pages 29–41, 1998.
9. R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning About Knowledge*.MIT Press, 1995.
10. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. Knowledge-based programs. *Distributed Computing*, 10(4):199–225, 1997.
11. P. Gammie and R. van der Meyden. MCK: Model checking the logic of knowledge. In *Proc. Conf. on Computer Aided Verification*, pages 479–483, 2004.
12. S. Graf, D. Peled, and S. Quinton. Achieving distributed control through model checking. *Formal Methods in System Design*, 40(2):263–281, 2012.
13. 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. *J. ACM*, 39(3):449–478, 1992.
14. X. Huang and R. van der Meyden. Symbolic synthesis of knowledge-based program implementations with synchronous semantics. In *Proc. TARK*, pages 121–130, 2013.
15. G. Katz, D. Peled, and S. Schewe. Synthesis of distributed control through knowledge accumulation. In *Proc. Int. Conf on Computer Aided Verification*, pages 510–525, 2011.
16. A. Lomuscio, H. Qu, and F. Raimondi. MCMAS: A model checker for the verification of multi-agent systems. In *Proc. CAV'09*, pages 682–688. Springer LNCS 5643, 2009.
17. P. Srimani and S. R. Das, editors. *Distributed Mutual Exclusion Algorithms*. IEEE, 1992.
18. R. van der Meyden and M. Y. Vardi. Synthesis from knowledge-based specifications. In *Proc. CONCUR'98*, Springer LNCS 1466, pages 34–49, 1998.
19. R. van der Meyden and T. Wilke. Synthesis of distributed systems from knowledge-based specifications. In *Proc. Int. Conf. on Concurrency Theory, CONCUR*, pages 562–576, 2005.
20. M. Y. Vardi. Implementing knowledge-based programs. In *Proc. Conf. on Theoretical Aspects of Rationality and Knowledge*, pages 15–30, 1996.