Abstraction for Epistemic Model Checking of Dining Cryptographers-based Protocols*

[Extended Abstract]

Omar Al Bataineh
Computer Science and Engineering
University of New South Wales
Sydney, NSW 2052, Australia
omara@cse.unsw.edu.au

Ron van der Meyden
Computer Science and Engineering
University of New South Wales
Sydney, NSW 2052, Australia
meyden@cse.unsw.edu.au

ABSTRACT

The paper describes an abstraction for protocols that are based on multiple rounds of Chaum's Dining Cryptographers protocol. It is proved that the abstraction preserves a rich class of specifications in the logic of knowledge. This result is applied to optimize model checking of implementations of a knowledge-based program that uses the Dining Cryptographers protocol as a primitive in an anonymous broadcast system. Performance results are given for model checking knowledge-based specifications in the concrete and abstract models of this protocol, and some new conclusions about the protocol are derived.

Categories and Subject Descriptors

F.4.1 [**Mathematical Logic**]: Modal Logic; D.2.4 [**Software/ Program Verification**]: Model Checking; D.4.6 [**Security and Protection**]: Verification

General Terms

Security, Theory, Verification

*This material is based on research sponsored by the Air Force Research Laboratory, under agreement number FA2386-09-1-4156. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government. An extended version of this paper is available at `http://arxiv.org/abs/1010.2287`.

TARK 2011, July 12-14, 2011, Groningen, The Netherlands.
 ISBN 978-1-4503-0707-9/11/07,...$10.00.

Keywords

Logic of Knowledge, Model Checking, Anonymity, Knowledge-based programs, Abstraction

1. INTRODUCTION

Our contribution in this paper is to establish the correctness of an abstraction relation for abstract programs based on use a trusted third party for anonymous broadcast, which is implemented in the related concrete programs by means of the *Dining Cryptographers* protocol proposed by Chaum [3]. That Chaum's protocol implements anonymous broadcast is, of course, well-known, but we show that this statement holds in a more general sense than is usually considered in the literature, where the focus is generally on the very particular property of anonymity. Specifically, we consider a broad class of properties formulated in the logic of knowledge, including properties in which agent knowledge is nested, such as "Alice knows that Bob knows that p". We show that the abstraction relation between programs based on the trusted third party and programs based on the Dining Cryptographers protocol preserves all properties from this class.

As an application of this result, we consider a protocol from Chaum's paper [3] that uses multiple rounds of the Dining Cryptographers protocol to build a more general anonymous broadcast system. We have previously studied this protocol from the perspective of a model checking based methodology for the implementation of knowledge-based programs [2], by treating the specification of the protocol as a knowledge-based program.

Knowledge-based programs [7] are an abstract, program-like form of specification, that describe how an agent's actions are related to conditions stated in terms of the agent's knowledge. The advantage of this level of abstraction is that it provides a highly intuitive description of the intentions of the programmer, that has been argued to be easier to verify than the complex implementations one typically finds for highly optimized distributed programs [11, 7]. Knowledge-based programs cannot be directly implemented, however, so they must be *implemented* by concrete programs in which the knowledge conditions are replaced by concrete predicates of the agent's local state. The implementation relation between a knowledge-based program and a putative implementation holds when these concrete predicates are equivalent to the knowledge formulas that they replace (interpreted with respect to the system generated by running the putative implementation). Our partially-automated method-

ology for the implementation of knowledge-based programs uses a model checker for the logic of knowledge to check whether this equivalence holds, and if it does not, uses the counter-examples generated by the model checker to generate a revised putative implementation. (This process is iterated until an implementation is found.)

In our previous work on the application of this methodology, we considered model checking problems generated in this way from a knowledge-based program based on multiple rounds of the Dining Cryptographers protocol. Our experience was that the model checking problems we considered were close to the bounds of feasibility for our model checker even for instances with small numbers of agents, and we were prevented from considering instances of scale as a result. In the present paper, we apply our abstraction result in order to optimize the model checking problem, by performing model checking on the abstracted (trusted third party) version of the programs we consider rather than the concrete (Dining Cryptographers based) versions. We give performance results showing the difference: the abstraction is effective in reducing the model checking runtime by several orders of magnitude, enabling systems involving larger numbers of rounds of the Dining Cryptographers protocol and larger numbers of agents to be model checked. We use the efficiency gains to extend our previous analysis of the knowledge based program to larger numbers of agents, leading to an improved understanding of its implementations.

The structure of the paper is as follows. We begin in Section 2 by introducing the logic of knowledge, which provides the specification language for the properties that are preserved by our abstraction technique, and give its semantics in terms of a class of Kripke structures. We define a notion of bisimulation on these Kripke structures that provides the semantic basis for our program abstraction technique. In Section 3, we introduce a simple programming language used to represent our concrete and abstract programs. In Section 4, we introduce the Dining Cryptographers protocol and, in Section 5, its abstraction using a trusted third party. We then state the abstraction result. The remainder of the paper deals with our application of this result. We recall the two-phase protocol in Section 6. In Section 7 we describe knowledge-based programs and an approach to the use of model checking to identify their implementations. In Section 8 we recall our formulation of the two-phase protocol as a knowledge-based program and describe the associated verification conditions. Section 9 discusses the comparative performance of model checking in the concrete and abstract models when using the model checker MCK. We highlight some of the interesting conclusions we are able to make about implementations of the knowledge-based program for the round-based protocol in Section 10. We discuss related work in Section 11. Finally, in Section 12, we draw some conclusions and discuss future directions.

2. EPISTEMIC LOGIC

Suppose that we are interested in systems comprised of agents from a set Agt whose states are described using a set Var of boolean variables.[1] The syntax of the logic of knowledge $\mathcal{L}_{(Var,Agt)}$ is given by the following grammar:

$$\phi ::= \top \mid v \mid \neg\phi \mid \phi \wedge \phi \mid K_i\phi$$

where $v \in Var$ is a variable and $i \in Agt$ is an agent. (We freely use standard boolean operators that can be defined using the two given.) Intuitively, the meaning of $K_i\phi$ is that agent i knows that ϕ is true.

The semantics for the language is given in terms of *Kripke structures* of the form $M = (Agt, W, \{\sim_i\}_{i\in Agt}, Var, \pi)$, where Agt is the set of agents, W is a set of worlds, or situations, for each $i \in Agt$, $\sim_i$ is an equivalence relation on W, Var is a set of variables, and $\pi : W \times Var \to \{0, 1\}$ is a valuation. Intuitively, W is the set of situations that the agents consider that they could be in, and $w \sim_i w'$ if, when the actual situation is w, agent i considers it possible that they are in situation w'. The value $\pi(w, v)$ is the truth value of variable v in situation w. Such a Kripke structure M is *fit* for the language $\mathcal{L}_{(Var',Agt')}$ if $Agt' \subseteq Agt$ and $Var' \subseteq Var$. The semantics of the language is given by the relation $M, w \models \phi$, where M is a Kripke structure fit for $\mathcal{L}_{(Var,Agt)}$, w is a world of M, and ϕ is a formula, meaning intuitively that the formula ϕ holds at the world w. The definition is given inductively by

1. $M, w \models v$ if $\pi(w, v) = 1$, for $v \in Var$.
2. $M, w \models \neg\phi$ if not $M, w \models \phi$,
3. $M, w \models \phi_1 \wedge \phi_2$ if $M, w \models \phi_1$ and $M, w \models \phi_2$,
4. $M, w \models K_i\phi$ if $M, w' \models \phi$ for all $w' \in W$ with $w \sim_i w'$, for $i \in Agt$.

Intuitively, the final clause says that agent i knows ϕ if it does not consider it possible that not ϕ. We write $M \models \phi$, and say that ϕ is *valid* in M, if $M, w \models \phi$ for all $w \in W$. The *Kripke structure model checking* problem is to compute, given M and ϕ, whether $M \models \phi$. We will use this formulation of the model checking problem as the basis for another notion of model checking, to be introduced below, that concerns a way of generating M from a program.

To state our abstraction result, we work with *bisimulations* on epistemic Kripke structures. The definition is standard, but we parameterize it on a set of variables Var and a set of agents Agt. Suppose we are given a set of variables Var, a set of agents Agt, and two Kripke structures

$$M = (Agt^M, W^M, \{\sim_i^M\}_{i\in Agt^M}, Var^M, \pi^M)$$

and

$$N = (Agt^N, W^N, \{\sim_i^N\}_{i\in Agt^N}, Var^N, \pi^N)$$

such that $Agt \subseteq Agt_M \cap Agt_N$ and $Var \subseteq Var^M \cap Var^N$. (Note that these conditions imply that both M and N are fit for $\mathcal{L}_{(Var,Agt)}$.) A (Var, Agt)-bisimulation $\Re$ between M and N is defined to be a binary relation $\Re \subseteq W^M \times W^N$ such that:

1. **Atoms:** $\pi^M(w, v) = \pi^N(w', v)$ whenever $w\Re w'$ and $v \in Var$;
2. **Forth:** if $i \in Agt$, and w_1, w_2 are two worlds in M and u_1 is a world in N such that $w_1 \sim_i^M w_2$ and $w_1\Re u_1$, then there is a world $u_2 \in W_N$ such that $u_1 \sim_i^N u_2$ and $w_2\Re u_2$; and

[1] We use the term "variable" rather than "proposition" in this paper, since our atomic propositions arise as boolean variables in a program.

3. **Back:** if $i \in Agt$ and u_1, u_2 are two worlds in N and w_1 is a world in M such that $u_1 \sim_i^N u_2$ and $u_1 \Re w_1$, then there is a $w_2 \in W_M$ such that $w_1 \sim_i^M w_2$ and $u_2 \Re w_2$.

If there exists an (Var, Agt)-bisimulation $\Re$ between M and N such that $w \Re u$, then we write $(M, w) \approx_{(Var,Agt)} (N, u)$. We write $M \approx_{(Var,Agt)} N$ if there exists a bisimulation that associates every world of M with one of N, and vice versa. The point of the parameterization is to obtain the following variant of a standard result:

LEMMA 1. *If M and N are Kripke structures and u and w are worlds of M and N such that $(M, u) \approx_{(Var,Agt)} (N, w)$, then for all $\varphi \in \mathcal{L}_{(Var,Agt)}$ we have $M, u \models \varphi$ if and only if $N, w \models \varphi$. If $M \approx_{(Var,Agt)} N$ then for all $\varphi \in \mathcal{L}_{(Var,Agt)}$ we have $M \models \varphi$ if and only if $N \models \varphi$.*

3. A PROGRAMMING LANGUAGE

We use a small multi-agent programming language equipped with a notion of observability. All variables are Boolean, and expressions are formed from variables using the usual Boolean operators. The language has the following atomic actions, in which i and j are agents, x is a variable name and e is an expression:

1. $i : x := e$ — agent i evaluates e and assigns the result to x,
2. $i : rand(x)$ — agent i assigns a random (nondeterministically chosen) value to x,
3. $i : e \to j.x$ — agent i evaluates e and transmits the result across a private channel to agent j, who assigns it to its variable x,
4. $i : broadcast(x)$ – agent i broadcasts the value of the variable x to all other agents.

Note that we write $i.x$ for agent i's variable x (the variables $i.x$ and $j.x$ are considered distinct when $i \neq j$) but may omit the agent name when this is clear from the context. In particular, in an atomic action $i : a$, any variable x not explicitly associated with an agent refers to $i.x$. For example, we may write $i : x := y \oplus z$ rather than $i : i.x := i.y \oplus i.z$. Similarly, when e is an expression in which agent indices are omitted, and i is an agent, the expression $i.e$ refers to the result of replacing each occurrence of a variable name x in e that is not already associated to an agent index with $i.x$. Thus $i.(y \oplus j.z)$ represents $i.y \oplus j.z$.

Each atomic action *reads* and *writes* certain variables. Specifically, the action $i : x := e$ reads the (agent i) variables in e and writes $i.x$, the action $i : rand(x)$ reads nothing and writes $i.x$, the action $i : e \to j.x$ reads the (agent i) variables in e and writes $j.x$, and the action $i : broadcast(x)$ reads x and writes nothing. A *joint action* is a set of atomic actions in which no variable is written more than once. Intuitively, a joint action is executed by first evaluating all the expressions and then performing a simultaneous assignment to the variables.

A *program* is given by a sequence of joint actions $A_1; \ldots; A_n$. A *program for agent i* is a program in which each atomic action $j : a$ in any step has $j = i$. We permit parallelism within an agent, in the sense that we do *not* require that a joint action contains at most one atomic action for each agent. If we are given for each agent i a program $P_i = A_1^i; \ldots; A_n^i$, all of the same length n, then we may form the joint program $||_i P_i = (\cup_i A_1^i); \ldots; (\cup_i A_n^i)$.

Some well-formedness conditions are required on agent programs. An *observability mapping* is a function ov mapping each agent to a set of variables, intuitively, the set of variables that it may observe. A program runs in the context of an observability mapping, and modifies that mapping. A joint action A is *enabled* at an observability map ov if

1. no variable written to by A is in $ov(i)$ for any agent i (that is, all variables written to are *new* variables), and
2. for each atomic action $i : x := e$ and $i : e \to j.x$ in A, the expression $i.e$ contains only variables in $ov(i)$, and
3. for each action $i.broadcast(x)$ we have $i.x \in ov(i)$.

These constraints may be understood as access control constraints stating that agent i may read only the variables in $ov(i)$ and may write only new variables.

Executing the action A transforms the observability map ov to the observability map $ov[A]$ such that $ov[A](i)$ is the result of adding to $ov(i)$

1. all variables $i.x$ such that an action of the form $i : x := e$ or $i : rand(x)$ or $j : e \to i.x$ occurs in A, and
2. all variables $j.x$ such that $j : broadcast(x)$ occurs in A.

These definitions are generalised to programs: *the program $P = A_1; \ldots; A_n$ is enabled* at the observability map ov if for each $i = 1 \ldots n$, the action A_i is enabled at $ov[A_1] \ldots [A_{i-1}]$, and we define $ov[P]$ to be $ov[A_1] \ldots [A_n]$.

Example 1. Consider a two-agent system with agents i, j. The action $\{i : x := j.y\}$ is not enabled at the observability map ov given by $\{j \mapsto \{j.y\}\}$. However, the program $\{j : broadcast(y)\}; \{i : x := j.y\}$ is enabled at ov, since the action $\{j : broadcast(y)\}$ is enabled at ov, and transforms ov to $ov[\{j : broadcast(y)\}] = \{j \mapsto \{j.y\},\ i \mapsto \{j.y\}\}$, at which the action $\{i : x := j.y\}$ is enabled.

We say that an observability map ov is *consistent* with a Kripke structure $M = (Agt, W, \{\sim_i\}_{i \in Agt}, Var, \pi)$ when for all agents i, if v is a variable in $ov(i)$ then $v \in Var$, and for all worlds $w, w' \in W$ such that $w \sim_i w'$ we have $\pi(w, v) = \pi(w', v)$. Intuitively, ov is consistent with M if all variables declared to be local to agent i by ov are in fact defined and semantically local to agent i in M.

The *program P is enabled at a Kripke structure M* if there exists an observability map ov such that

1. ov is consistent with M,
2. P is enabled at ov, and
3. all variables x written by P are not defined in M (i.e., $x \notin Var$).

In particular, note that if a single joint action A is enabled at M, then for all variables x read by A, and all worlds w, the value $\pi(w, x)$ is defined. Consequently, we may also evaluate at w any expression e required to be computed by A. We write $\pi(w, e)$ for the result.

We can now give a semantics of programs, in which a program applied to a Kripke structure representing the initial

states of information of the agents, transforms the structure into another Kripke structure representing the states of information of the agents after running the program. The definition is given inductively, on an action-by-action basis. Let $M = (Agt, W, \{\sim_i\}_{i \in Agt}, Var, \pi)$ be a Kripke structure and A a joint action. We define a Kripke structure $M[A] = (Agt', W', \{\sim'_i\}_{i \in Agt'}, Var', \pi')$ as follows. Let V be the set of variables $i.x$ such that A includes the atomic action $i : rand(x)$. Intuitively, such actions increase the amount of non-determinism in the system, whereas all other actions have deterministic effects. We define $Agt' = Agt$ and take W' to be the set of states of the form (w, κ) where $w \in W$ and $\kappa : V \to \{0, 1\}$ is an assignment of boolean values to the variables in V. We may write $w + \kappa$ for the pair (w, κ). In case V is the empty set, κ is always the null function, so we may write just w for (w, κ). The set Var' of variables defined in $M[A]$ is obtained by adding to Var all variables written to by A. The assignment π' is obtained by extending π to these new variables by defining π' as follows on worlds $w + \kappa$:

1. if $v \in Var$ then $\pi'(w + \kappa, v) = \pi(w, v)$,
2. if $i : x := e$ occurs in A then $\pi'(w + \kappa, i.x) = \pi(w, i.e)$,
3. if $i : rand(x)$ occurs in A then $\pi'(w + \kappa, i.x) = \kappa(i.x)$, and
4. if $j : e \to i.x$ occurs in A then $\pi'(w+\kappa, i.x) = \pi(w, j.e)$.

Finally, the indistinguishability relations $\sim'_i$ are defined using the observability map $ov[A]$: we define $w + \kappa \sim'_i w' + \kappa'$ when $w \sim_i w$ and for all variables x in $ov[A](i) \setminus ov(i)$, we have $\pi'(w + \kappa, x) = \pi'(w' + \kappa', x)$. Intuitively, this reflects that the agent recalls any information it had in the structure M, and adds to this information that it is able to observe in the new state. Note that in fact $w + \kappa \sim'_i w' + \kappa'$ implies $\pi'(w + \kappa, x) = \pi'(w' + \kappa', x)$ for *all* variables $x \in ov[A](i)$, since we have assumed that for $x \in ov(i)$ we have that $w \sim_i w$ implies $\pi(w, x) = \pi(w', x)$. Moreover, since the set $ov[A](i) \setminus ov(i)$ is just the set of variables written to by A that are made observable to i, this observation also yields that the definition of $M[A]$ is independent of the choice of observation map ov consistent with M.

4. DINING CRYPTOGRAPHERS

Chaum's Dining Cryptographers protocol is an example of an anonymous broadcast protocol: it allows an agent to send a message without revealing its identity. Chaum introduces the protocol with a story in which three cryptographers seated at a circular dinner table try to determine whether one of the three or the NSA has anonymously paid for dinner, without compromising the anonymity of the payer if it was not the NSA. He proposes the following solution:

> Each cryptographer flips an unbiased coin behind his menu, between him and the cryptographer on his right, so that only the two of them can see the outcome. Each cryptographer then states aloud whether the two coins he can see–the one he flipped and the one his left-hand neighbor flipped–fell on the same side or on different sides. If one of the cryptographers is the payer, he states the opposite of what he sees. An odd number of differences uttered at the table indicates that a cryptographer is paying; an even number indicates that NSA is paying (assuming that the dinner was paid for only once). Yet if a cryptographer is paying, neither of the other two learns anything from the utterances about which cryptographer it is.

Chaum shows that this protocol solves the problem, and notes that it can be considered as a mechanism enabling a signal to be anonymously transmitted, under the assumption that at most one of the agents wishes to transmit. He goes on to generalize the idea to n-agent settings where, in place of the ring of coins, we have a graph representing the key-sharing arrangement.

The more general protocol can be represented in our programming language as follows. We assume that there is a set Agt of agents, who share secrets based on a (directed) key sharing graph $G = (Agt, E)$ in which the vertices are the agents in Agt and the edges $E \subseteq Agt \times Agt$ describe the keysharing arrrangement amongst the agents. We model keysharing by assuming that for each edge $e = (i, j)$, agent i generates the key corresponding to the edge, and communicates the key to j across a secure channel. For each edge $e = (i, j)$ we write e_1 for the source agent i and e_2 for the destination agent j. For each agent i we define $in(i) = \{e \in E \mid e_2 = i\}$ and $out(i) = \{e \in E \mid e_1 = i\}$. Accordingly, we use two variables for each edge $e = (i, j)$: the variable $i.k_e$ stores i's copy of the key corresponding to the edge, and the variable $j.k_e$ stores j's copy. We write $keys(i)$ for $in(i) \cup out(i)$, i.e., the set of edges incident on i. The protocol $DC_i(m)$ of an agent $i \in Agt$ (in which the message represented by the expression $i.m$ is transmitted anonymously by agent i) consists of the following five steps:

$$
\begin{aligned}
DC_i(m) = \quad & \{i : rand(k_e) \mid e \in out(i)\}; \\
& \{i : k_e \to e_2.k_e \mid e \in out(i)\}; \\
& \{i : b := m \oplus \oplus_{e \in keys(i)}\ k_e\}; \\
& \{i : broadcast(b)\}; \\
& \{i : rr := \oplus_{j \in Agt}\ j.b\}
\end{aligned}
$$

Figure 1: The protocol DC

We write $DC(m)$ for the joint program $||_{i \in Agt} DC_i(m)$.

Intuitively, the protocol DC operates by first generating keys and setting up the key sharing graph, and then having each of the agents make a public announcement encrypted using all the keys available to them. The directionality of an edge in the key sharing graph indicates who generates the key corresponding to the edge, viz, the source agent of the edge. The first step of the protocol corresponds to each agent generating the key values for which they are responsible. In the second step, these keys are shared with the other agent on the edge by transmission across a secure channel. Each agent now has the value of each of the key edges on which it is incident, and computes the xor of its message with all these key values in the 3rd step, and broadcasts the result in the 4th step. In the final step of the protocol, each agent computes the xor of the messages broadcast as the result of the protocol.

5. AN ABSTRACTION OF THE DINING CRYPTOGRAPHERS PROTOCOL

We are interested in protocols in which the DC protocol is used as a basic building block, and in model checking the agent's knowledge in the resulting protocols. In order to optimize this model checking problem, we now introduce a protocol that we will show to be an abstraction of the DC protocol that preserves epistemic properties.

The abstracted version of the protocol omits the use of keys, but adds to the set of agents a trusted third party T who computes the result of the protocol on behalf of the agents, and then broadcasts it. Here, we take $Agt^a = Agt \cup \{T\}$. The protocol $DC_i^a(m)$ for agent i is given in four steps, see Figure 2. We write $DC^a(m)$ for the joint

$$
\begin{array}{lcll}
DC_i^a(m) & = & \{i : m \to T.x_i\}; & (\text{for } i \in Agt) \\
 & & \{\}; & \\
 & & \{\}; & \\
 & & \{i : rr := y\} & \\
 & & & \\
DC_T^a(m) & = & \{\}; & \\
 & & \{T : y := \oplus_{i \in Agt}\ x_i\}; & \\
 & & \{T : broadcast(y)\}; & \\
 & & \{\} &
\end{array}
$$

Figure 2: The abstract protocol DC^a

program $||_{i \in Agt^a} DC_i^a(m)$. Intuitively, in the abstract protocol, the agents transmit their bits across a secure channel to the trusted third party, who computes the exclusive-or and broadcasts it.

We can prove the following result concerning this abstraction. (Note that we use fresh variables k_e, b, rr, x_i and y in each of the instances of DC_i and DC_i^a.)

THEOREM 1. *Let M and M^a be Kripke structures with $M \approx_{V,Agt} M^a$, and let*

$$P = Q_1; DC(m_1); Q_2; DC(m_2); \ldots DC(m_k); Q_{k+1} \quad \textit{and}$$

$$P^a = Q_1; DC^a(m_1); Q_2; DC^a(m_2); \ldots DC^a(m_k); Q_{k+1}$$

where the Q_i are programs involving agents Agt. Let V' be the set of all variables written by the programs Q_i, as well as the variables $i.rr$ introduced by the DC instances. Assume that the Q_j and m_j read only variables from $V \cup V'$. Then if P is enabled at M, and P^a writes no variable in M^a, then P^a is enabled at M^a and $M[P] \approx_{V \cup V', Agt} M^a[P^a]$.

This result states that if we have a complex protocol P, constructed by using multiple instances of the DC protocol interleaved with other actions, then we abstract P by abstracting each of the instances of DC to DC^a, while preserving the truth values of all epistemic formulas. This enables optimization of model checking epistemic formulas in $M[P]$ by applying model checking to $M[P^a]$ instead. (Note that always $M \approx M$.)

6. THE TWO-PHASE ANONYMOUS BROADCAST PROTOCOL

As noted above, the basic version of the Dining Cryptographers protocol enables a signal to be anonymously transmitted under the assumption that at most one agent wishes to transmit. One of Chaum's considerations is the use of the protocol for more general anonymous broadcast applications, and he writes:

> The cryptographers become intrigued with the ability to make messages public untraceably. They devise a way to do this at the table for a statement of arbitrary length: the basic protocol is repeated over and over; when one cryptographer wishes to make a message public, he merely begins inverting his statements in those rounds corresponding to 1's in a binary coded version of his message. If he notices that his message would collide with some other message, he may for example wait for a number of rounds chosen at random from some suitable distribution before trying to transmit again.

As a particular realization of this idea, he discusses grouping communication into blocks and the use of the following *two-phase broadcast* protocol using *slot-reservation*:

> In a network with many messages per block, a first block may be used by various anonymous senders to request a "slot reservation" in a second block. A simple scheme would be for each anonymous sender to invert one randomly selected bit in the first block for each slot they wish to reserve in the second block. After the result of the first block becomes known, the participant who caused the ith bit in the first block sends in the ith slot of the second block.

Chaum's discussion leaves open a number of questions concerning the protocol. For example, what exact test is applied to determine whether there is a collision? Which agents are able to detect a collision? Are there situations where some agent expects to receive a message, but a collision occurs that it does not detect (although some other agent may do so?) Under what exact circumstances does an agent know that some agent has sent a message? When can a sender be assured that all others have received the message?

In previous work, we have studied such questions in a 3-agent version of the protocol [2]. Our approach was to model the protocol as a knowledge-based program and to use epistemic model checking as a tool to help us identity precisely the conditions under which an agent obtains some types of knowledge of interest. The approach helped us to identify some unexpected situations in which relevant knowledge is obtained. We recap the definition of knowledge-based programs and our formulation of the 2-phase protocol as a knowledge-based program in the following sections, after which we study this knowledge-based program further using the abstraction developed above.

7. KNOWLEDGE-BASED PROGRAMS

Knowledge-based programs [7] are like standard programs, except that expressions may refer to an agent's knowledge. That is, in a knowledge-based program for agent i, we may find statements of the form "$v := \phi$", where ϕ is a formula of the logic of knowledge, i.e., a boolean combination of atomic formulas concerning the agent's observable variables and formulas of the form $K_i\psi$.

Unlike standard programs, knowledge-based programs cannot in general be directly executed, since the satisfaction of

the knowledge subformulas depends on the set of all runs of the program, which in turn depends on the satisfaction of these knowledge subformulas. This apparent circularity is handled by treating a knowledge-based program as a specification, and defining when a concrete standard program satisfies this specification. We give a formulation of the semantics of knowledge-based programs tailored to the programming language of the present paper.

Suppose that we have a concrete program P of the same syntactic structure as the knowledge-based program $\mathbf{P}$, in which each knowledge-based expression ϕ is replaced by a concrete predicate p_ϕ of the local variables of the agent. Starting at an initial Kripke Structure M_0, the concrete program P generates a set of runs that form the worlds of a Kripke Structure $M_0[P]$. We now say that P is an *implementation of the knowledge-based program* $\mathbf{P}$ *from* M_0 if for each joint action A in the program P, corresponding to a joint action $\mathbf{A}$ in the knowledge-based program, if we write $P = P_0; A; P_1$, where P_0 and P_1 are programs, then for each knowledge condition ϕ occurring in $\mathbf{A}$, we have $M_0[P_0] \models p_\phi \Leftrightarrow \phi$. That is, the concrete condition is equivalent to the knowledge condition in the implementation at each point in the program where it is used.

A partially automated process using model checkers for epistemic logic may be used to verify these equivalences, and to derive an implementation using an iterative process based on counterexamples when the equivalence fails. We refer the reader to our previous work [2] for further discussion and examples of the application of this iterative process.

8. THE TWO-PHASE PROTOCOL AS A KNOWLEDGE-BASED PROGRAM

We now give a formulation of Chaum's two-phase protocol (see Section 6) as a knowledge-based program, and discuss the associated verification conditions. (The knowledge-based program is similar to that given in our earlier work, but includes some improvements.)

We assume that there are n agents, and $Agt = \{1..n\}$. Figure 3 represents the 2-phase protocol by giving a knowledge-based program for agent i. The local variable `slot-request`, assumed to be defined in the structure from which the program is run, records the slot number (in the range 1..n) that this agent will attempt to reserve. If `slot-request`=0, then the agent will not attempt to reserve any slot. The variable `message`, also assumed to be defined, records the single bit message that the agent wishes to anonymously broadcast (if any). The program introduces the variables `rcvd0` and `rcvd1`, as well as a variable `dlvrd`. (Additional new variables are implicit in the instances of DC$_i$.)

The term `conflict(s)` in the knowledge-based program represents that there is a conflict on slot s. This is a global condition that is defined as

$$\texttt{conflict(s)} = \bigvee_{\texttt{i}\neq\texttt{j}} (\texttt{i.slot-request} = \texttt{s} = \texttt{j.slot-request}) .$$

i.e., there exist two distinct agents i and j both requesting slot s. The term `sender`(i, x), defined by

$$\texttt{sender}(i,x) = \bigvee_{j\neq i} (j.\texttt{message} = x \wedge j.\texttt{slot-request} \neq 0) ,$$

represents that some other agent is sending message x. Thus, the variable `rcvd0` is assigned *true* if the agent learns that someone else is trying to send the bit 0; similarly for `rcvd1`.

$\mathbf{P}_i$ = {
local variables:
  `slot-request`: [0..n],
  `message`: Bool,
  `rcvd0`, `rcvd1`, `dlvrd`: Bool;
//reservation phase
`for` $s = 1 \ldots n$
{
  DC$_i$(`slot-request`=s);
}
//transmission phase
`for` $s = 1 \ldots n$
{
  DC$_i$(`if` (`slot-request` $= s \wedge \neg K_i$(`conflict(s)`))
    `then message`
    `else` false));
}
`rcvd0`:= K_i(`sender`$(i, 0)$);
`rcvd1` := K_i(`sender`$(i, 1)$);
`dlvrd`:= $\bigwedge_{x\in Bool}$((`message` $= x \wedge$ `slot-request` $\neq 0) \Rightarrow K_i(\bigwedge_{j\neq i} K_j$`sender`$(j, x)))$
}

Figure 3: The knowledge-based program CDC

We note that this representation of the 2-phase protocol as a knowledge-based program is *speculative*: an agent transmits in a slot so long as it does not know that there is a conflict. This allows that a collision will occur during the transmission phase.

Since an agent may attempt to reserve a slot, and then back off, or may send in a reserved slot without success because of a collision during the transmission phase, the protocol does not guarantee that the message will be delivered. In this case, the agent is required to retry the transmission in the next run of the protocol. So that it can determine whether a retry is necessary, the final assignment to the variable `dlvrd` captures whether the agent knows that its transmission has been successful, i.e., the agent knows that the other agents know some agent is sending its message.

We interpret the knowledge-based program with respect to the assumption of perfect recall, so implementations may make use of of history variables to capture observations that the agent makes during the running of the protocol. Thus, by placing the reception and delivery assignments at the end of the program (rather than just after each DC instance), we ensure that the agents are able to behave optimally by making use of all information they gather during the running of the program. As we discuss below, this allows us to capture some subtle sources of information.

We note that each of the instances of the protocol DC_i introduces additional variables, which may be used in the concrete predicates we substitute for the knowledge conditions. In particular, they introduce round result variables, which we denote by `rr[t]` for $t \in \{1..2n\}$. Here $rr[t]$ represents the round result variable from the t-th instance of DC_i in the implementation. The implementations also introduce key variables k_e and b, which need to be separated in the different instances: we may similarly use $k_e[t]$ and $b[t]$ to denote the t-th instance of such a variable.

We now discuss the formulas that are used to verify the implementation. As discussed above, these conditions need

to be verified at specific stages of the program, viz., the step before the occurrence of the knowledge formula of interest.

The first formula of interest concerns the correctness of the guess for the knowledge condition $\neg K_i(\texttt{conflict(s)})$. In the implementation, this condition is represented by the variable `kc[s]`.

Specification 1: `kc[s]` *correctly represents knowledge about the existence of a conflict in slot* $s = 1..3$.

$$i.\texttt{kc[s]} \Leftrightarrow \neg K_i(\texttt{conflict(s)}) \qquad (1)$$

Next, the protocol has some positive goals, viz., to allow agents to broadcast some information, and to do so anonymously. Successful reception of a bit is intended to be represented by the variables `rcvd0` and `rcvd1`. To ensure that the assignments to these variables correctly implement their intended meaning in the knowledge-based program, we use specifications of the following form.

Specification 2: reception variables correctly represent transmissions by others

$$i.\texttt{rcvd0} \Leftrightarrow K_i(\texttt{sender}(i, 0) \qquad (2a)$$

and

$$i.\texttt{rcvd1} \Leftrightarrow K_i(\texttt{sender}(i, 1)) \qquad (2b)$$

Similarly, we verify correct implementation of the agent's knowledge about whether its transmission is successful.

Specification 3: delivery variables correctly represent knowledge about delivery

$$i.\texttt{dlvrd} \Leftrightarrow \textstyle\bigwedge_{x\in Bool}(i.\texttt{message} = x \wedge i.\texttt{slot-request} \neq 0 \Rightarrow K_i(\bigwedge_{j\neq i} K_j\texttt{sender}(j, x)))$$

Finally, the aim of the protocol is to ensure that when information is transmitted, this is done anonymously. An agent may know that one of the other two agents has a particular message value, but it may not know what that value is for a specific agent. We may write the fact that agent i knows the value of a boolean variable x by the notation $\hat{K}_i(x)$, defined by $\hat{K}_i(x) = K_i(x) \vee K_i(\neg x)$. Using this, we might first attempt to specify anonymity as $\bigwedge_{j\neq i}(\neg\hat{K}_i(j.\texttt{message}))$, i.e., agent i knows no other's message. Unfortunately, the protocol cannot be expected to satisfy this: suppose that all agents manage to broadcast their message and all messages have the same value x: then each knows that the other's value is x. We therefore write the following weaker specification of anonymity:

Specification 4: The protocol preserves anonymity

$$\bigvee_{x=0,1} K_i(\bigwedge_{j\neq i}(j.\texttt{message} = x)) \vee \bigwedge_{j\neq i}(\neg\hat{K}_i(j.\texttt{message})) \ .$$

This is checked at the very end of the protocol.

9. MODEL CHECKING PERFORMANCE

To verify the specifications for the knowledge-based program in a putative implementation, we have applied the epistemic model checker MCK [8]. All specifications are checked using the perfect recall interpretation of knowledge and the model checking algorithm for this semantics which is described in [13] (which is flagged by `spec_spr_xn` in MCK). To estimate individual formula timings, we deduct model construction times (estimated by the time to model check the specification True), from the actual time for model checking each specification (which includes model construction and formula verification time.) All experiments are conducted on a Linux PC with Intel(R) Xeon(R) 4 × 3 GHZ, and 16 GB memory, using MCK 0.1.1. Where the execution crashed due to a memory error, we report "x" in the tables.

For each specification x we give runtimes for model checking the specification in the concrete program and the abstract program (indicated by x^a). We count the cost of verifying all instances of the specification required to check the correctness of the implementation at different times where the knowledge condition occurs in the program. (With n agents, we need to check Specification 1 at n locations in the implementation, but specifications 2-4 just once.) To converge upon an implementation of the knowledge-based program, we use a semi-automated counter-example guided process: we refer the reader to our previous work [2] for details and examples. As we improve the approximation, the program becomes more complex, and the model checking runtimes generally increase. Table 1 gives the runtimes for the final approximation, a program that is verified as implementing the knowledge-based program.

For a more detailed indication of the impact of the abstraction, Table 2 compares the runtimes for model checking the anonymity specification (Specification 4) in the concrete and abstract programs for the final implementation after a given number of rounds of the Dining Cryptographers Protocol. Note that the maximum number of rounds of Dining Cryptographers in the 2-phase protocol is twice the number of agents.

In all these experiments, the runtimes obtained indicate that the abstraction results in a significant decrease of runtimes, (in some cases of several orders of magnitude) and helps to bring problems of larger scale (in particular, with larger numbers of agents and greater numbers of rounds of the basic Dining Cryptographers protocol) within the bounds of feasibility of model checking.

10. IMPLEMENTATIONS OF THE KNOWLEDGE-BASED PROGRAM

Using the optimization obtained from the abstraction, we have been able to extend our previous analysis of the knowledge-based program in the 3-agent case to the cases of 4 and 5 agents, gaining more insight into the n-agent case for general n. We now describe the implementations we found for the program, which demonstrate that the protocol contains some further subtle flows of information beyond those we found in the 3 agent case.

One point worth noting is that, in addition to providing an optimization of epistemic model checking, our abstraction result also provides information that is useful in the search for an implementation of the knowledge-based program. Observe that the variables k_e do not occur in the abstract version of the protocol, nor in the formulas we need to check to verify an implementation. Thus, in guessing a concrete predicate to be substituted for one of the knowledge conditions, we can confine our attention to predicates that do not contain the k_e variables. Indeed, since $i.b$ is computed from information already at agent i's disposal, we need only consider predicates based on agent i's initial information and the round result variables $rr[k]$.

The first knowledge condition we need to implement, for Specification 1, is $\neg K_i\texttt{conflict(s)}$. Plainly, one situation

			Specification							
n	Model	Model^a	1	1^a	2	2^a	3	3^a	4	4^a
3	0.45	0.4	50	16	7200	127	7350	34	7400	18
4	135	6	x	167	x	378	x	251	x	252
5	x	74	x	1096	x	1957	x	1979	x	1998

Table 1: Model Checking Runtimes (seconds)

Agents	version	Rounds									
		1	2	3	4	5	6	7	8	9	10
3	concrete	0.6	0.9	2.2	18	335	7350	-	-	-	-
3	abstract	0.5	0.6	0.7	1.6	3.1	17.8	-	-	-	-
4	concrete	340	575	587	1478	2661	x	x	x	-	-
4	abstract	9	11	11.2	11.7	32	85	86	249	-	-
5	concrete	x	x	x	x	x	x	x	x	x	x
5	abstract	91	110	133	134	183	311	752	722	950	1990

Table 2: Model Checking Runtimes (seconds) for Specification 4

where an agent knows that there is a conflict is when it attempts to reserve a slot and the round result for the reservation is not 1. (So an even number of agents attempted to reserve the slot.) Thus, one potential implementation for $\neg K_i\texttt{conflict(s)}$ is the assignment $kc[s] := \neg(\texttt{slot-request} = s \wedge rr[s] = 0)$. Model checking Specification 1 for this predicate at the point of the s-th transmission confirms in all of the cases $n = 3, 4, 5$ that this captures the knowledge condition $\neg K_i\texttt{conflict(s)}$ exactly at this point: there are no other ways that the agent can know of a conflict on a slot before transmitting on it, besides seeing a reservation clash. (In particular, previous transmissions do not contain any relevant information.)

It is interesting to consider not just the knowledge condition $\neg K_i\texttt{conflict(s)}$ that occurs in the program, but also the stronger condition $K_i\neg\texttt{conflict(s)}$. (The formula $K_i\neg p \Rightarrow \neg K_i p$ is a validity of the logic of knowledge.) For example, if an agent who is broadcasting on a slot knows that all other agents know the slot is conflict free, then it knows that its message will be delivered. Thus, we have also added a local variable $\texttt{conflict-free}(s)$ to the implementation, for $s = 1 \ldots n$, and and sought assignments to this variable that satisfy the formula $i.\texttt{conflict-free}(s) \Leftrightarrow K_i\neg\texttt{conflict(s)}$. This turns out to be quite a subtle matter.

To express this condition, it is useful to introduce a formula $C_0 = x$ where $x \in \{0, \ldots, n\}$ to express that the number of 0's obtained as round results in the reservation phase is x. We may then note the following situations in the protocol in which $K_i\neg\texttt{conflict(s)}$ holds.

- If $C_0 = 0$ or $C_0 = 1$, then the agent knows there is no conflict on any slot. Note that in this case there are at least $n-1$ agents who are requesting the at least $n-1$ distinct slots with reservation round result 1, leaving at most one further agent. If this agent had requested any of the slots with round result 1, this would have caused a 2-way reservation clash, contradicting the observed round result of 1. Hence this agent did not request any slot, and *all* slots are conflict-free.

- If $C_0 \geq 2$, then in general, an agent cannot determine whether or not there is a conflict on any of the reserved slots, since there may be a 3-way clash on one of these slots. However, in the particular case where $C_0 = 2$ and the agent itself does not request any slot ($\texttt{slot-request =0}$) then $n-2$ agents are accounted for by the $n-2$ slots on which we see a reservation round result of 1, and the remaining one agent cannot be assigned to any slot without changing the round result, and hence the count. Hence this agent cannot be requesting a slot, so the agent knows that all slots are conflict-free.

- Note that if $C_0 = 2$ or $C_0 = 3$, and the agent requests a slot but detects a collision at slot reservation time, then there must have been at least 2 agents requesting this slot, leaving at most $n-2$ agents for the $n-1$ other slots, where we see either $n-3$ or $n-4$ slots with reservation result of 1. This means at least $n-1$ or $n-2$ agents are accounted for in total, so the number of agents remaining to contribute to a further collision on the remaining $n-1$ other slots is at most 1. This agent can not be assigned to any slot without changing the round result for that slot, so it must not be requesting a slot. Thus, all the other $n-1$ slots are collision free.

- The above cases use information from the reservation phase. Agents may also be able to deduce that slots are conflict-free as a result of information they obtain during the transmission phase. If $C_0 = 2$ or $C_0 = 3$, the agent requests a slot and obtains a reservation round result of 1 for this slot, but then detects a collision at transmission time, then there must have been at least a 3-way collision on that agent's slot, and by a similar argument to the previous case, we deduce that all the other slots are collision free.

These conditions may be captured by the assignment

$$
\begin{array}{l}
i.\texttt{conflict-free}(s) := \\
C_0 = 0 \vee C_0 = 1 \vee (C_0 = 2 \wedge i.\texttt{slot-request} = 0) \vee \\
((C_0 = 2 \vee C_0 = 3) \wedge \\
\quad (\bigvee_{t=1}^{n} (s \neq t \wedge i.\texttt{slot-request} = t \wedge \texttt{rr}[t] = 0)) \vee \\
\quad (\bigvee_{t=1}^{n} (s \neq t \wedge i.\texttt{slot-request} = t \wedge \texttt{rr}[t] = 1 \\
\qquad \wedge \texttt{rr}[n+t] \neq i.\texttt{message})))
\end{array}
$$

The above formula states several concrete conditions under which the agent knows there is no conflict on a particular slot s. We have verified by model checking that for $n = 3$, 4, and 5 that, at the end of the protocol, for all slots s we have $i.\texttt{conflict-free}(s) \Leftrightarrow K_i\neg\texttt{conflict}(\texttt{s})$, and conjecture that it holds for all n.

We remark that in the case of $C_0 = 0$ or $C_0 = 1$, this information is available to all agents, and it is common knowledge[2] that all slots are conflict free. In the other cases, collision freedom on a slot may be known to some agents but not to others. For example, consider the situation with $n = 4$ and where the `slot-request` and `message` values and round results are given as in Figure 4(a). Here agent 2 sees a reservation collision and two 1's elsewhere, so knows that slots 1 and 4 are collision free. However, agent 1 does not know this, since the scenario of Figure 4(b) is consistent from its viewpoint, and here there is a collision on slot 4.

(a)

i:	1	2	3	4
i.slot-request	4	3	1	3
i.message	1	0	1	0
rr$[i]$	1	0	0	1
rr$[4+i]$	1	0	0	1

(b)

i:	1	2	3	4
i.slot-request	4	1	1	1
i.message	1	1	1	1
rr$[i]$	1	0	0	1
rr$[4+i]$	1	0	0	1

Figure 4: Collision Freedom is not Common Knowledge

As mentioned above, we consider a speculative version of the knowledge-based program, in which an agent transmits its message in its requested slot s in the transmission phase if $\neg K_i\texttt{conflict}(\texttt{s})$. One could also study a *conservative* version, where an agent only transmits if $K_i\neg\texttt{conflict}(\texttt{s})$. The analysis above shows that this would lead to a much more complicated implementation[3], where, moreover, the agent would transmit only in the low probability case when almost all other agents also have a message to send, and they happen to pick distinct slots!

Returning to the implementation of the speculative version, we need to find the appropriate assignments to the variables `rcvd0`, `rcvd1` and `dlvrd`. Reception of a bit x means that the agent knows that some other agent is sending that bit x. An obvious situation where this is the case is where the agent is not itself sending in the slot, the reservation round result is 1, and the bit x is observed as the round result in the corresponding transmission slot. Note that there may still be a collision on that slot, but since the number of agents in the collision is then odd, at least one must be sending x. As we noted in our previous work [2], there is another, less obvious, situation when an agent can know that another agent is sending a bit x in a slot, viz., when the agent is itself transmitting bit y in that slot and observes that the round result for the transmission is the compliment of y. Since the number of other agents in the conflict must be even, there must be both another agent sending 0 and another agent sending 1 in the slot. We have verified by model checking in the case of 3-5 agents that with the assignment

$$
\begin{array}{l}
i.\texttt{rcvd}x := \\
\bigvee_{s=1}^{n}((i.\texttt{slot-request} \neq s \wedge \texttt{rr}[s] = 1 \wedge \texttt{rr}[n+s] = x) \vee \\
\quad (i.\texttt{slot-request} = s \wedge \texttt{rr}[s] = 1 \wedge \texttt{rr}[n+s] \neq i.\texttt{message}))
\end{array}
$$

Specification 2 is satisfied in the strong version.

For the delivery condition, we have verified that the assignment

$$
\begin{array}{l}
\texttt{dlvrd} := (\texttt{slot-request} \neq 0 \wedge (C_0 = 0 \vee C_0 = 1)) \vee \\
(\texttt{slot-request} \neq 0 \wedge \texttt{message} = 1 \wedge \\
\quad \bigvee_{s\neq t,\ s,t=1..n}(\texttt{rr}[s]{=}\texttt{rr}[t]{=}1 \wedge \texttt{rr}[n+s]{=}\texttt{rr}[n+t]{=}1)) \vee \\
(\texttt{slot-request} \neq 0 \wedge \texttt{message} = 0 \wedge \\
\quad \bigvee_{s\neq t,\ s,t=1..n}(\texttt{rr}[s]{=}\texttt{rr}[t]{=}1 \wedge \texttt{rr}[n+s]{=}\texttt{rr}[n+t]{=}0))
\end{array}
$$

works for Specification 3 in the strong version for the cases n=3-5. The intuitions for this formula are as follows. In the case $C_0 = 0 \vee C_0 = 1$, as discussed above, it is common knowledge that all slots are conflict-free, so all transmissions are guaranteed to be delivered. As just noted, an agent who is not sending on a slot receives the value transmitted on that slot. However, an agent sending on a slot, and not noticing a clash on the transmission, considers it possible that there are other agents transmitting the very same value on that slot, and these will not know that there is another agent transmitting on the slot. However, if there are at least two distinct reserved slots where that value is transmitted, then each receives the value from some slot other than the one on which it transmits. This is expressed in the remainder of the formula.

Finally, the anonymity property, Specification 4, has been verified to hold in all the implementations obtained from the assignments discussed above, when $n = 3 \ldots 5$.

11. RELATED WORK

Abstractions of the kind we have studied, relating a protocol involving a trusted third party to a protocol that omits the trusted third party, are often used in theoretical studies to specify the objectives of a multi-party protocol. One example where this is done in a formal methods setting is work by Backes et al [1], who study the abstraction of pi- calculus programs based on multi-party computations. Where we consider a model checking approach to verification, with an expressive epistemic specification language, they use a type-checking approach. Their notion of abstraction is richer than the bisimulation-based approach we have taken, in that they also deal with probabilistic and computational concerns. However, as we have noted, we are interested in the preservation of a set of epistemic properties (nested knowledge formulas) that is richer in some dimensions than is usually considered in this literature. Our abstraction result could be easily strengthened to incorporate probability, as was done for a secure channel abstraction by van der Meyden and Wilke [14]. However computational complexity issues mesh less well with epistemic logic, and developing a satisfactory solution to this remains an open problem.

Other approaches using abstraction in the context of epistemic model checking include [5, 4]. These works are orthogonal to ours in that where we are concerned with an

[2]A fact is common knowledge [10] if all agents know it, all agents know that all other agents know it, and so on for all levels of iteration of knowledge.

[3]For a number of reasons, including the fact that we need an implementation of the knowledge condition at all transmission steps, rather than just at the end of the protocol, the above condition is not yet adequate for such an implementation.

abstraction of a particular primitive (the Dining Cryptographers protocol), that works for all formulas, they are concerned with symmetry reductions or deal with a more general class of programs than we have considered, but need to restrict the class of formulas preserved by the abstraction. An approach to abstraction in dynamic epistemic logic using 3-valued models is presented in [6].

The work of [12] also considers Chaum's 2-phase protocol, does not view the protocol as a knowledge-based program, as we have done, nor do they consider abstraction. Rather than the perfect recall semantics of knowledge, they use the observational semantics, which is less satisfactory for security and knowledge-based program analysis.

The 2-phase protocol has been implemented in the Herbivore system [9], which elaborates it with protocols allowing agents to enter and exit the system, as well as grouping agents in anonymity cliques for purposes of effciency. Variants of the protocol have also been considered by Pfitzman and Waidner [15]. These would make interesting case studies for future applications of our approach.

12. CONCLUSION

Several research directions suggest themselves as a result of our work in this paper. One is to complete the analysis of the knowledge-based program for all numbers of agents. We conjecture that our present implementation can be shown to work for all numbers of agents, and it would be interesting to have a proof of this claim: this would have to be done manually rather than by model checking, unless an induction result can be found for the model checking approach. Another direction is to consider richer extensions of the 2-phase protocol, addressing issues such as messages longer than a single bit, agent entry and exit protocols, and adversarial concerns such as collusion, cheating and disruption of the protocol. We hope to address these in future work.

Acknowledgments: Thanks to Xiaowei Huang and Kai Englehardt for comments on an earlier version of the paper.

13. REFERENCES

[1] M. Backes, M. Maffei, and E. Mohammadi. Computationally sound abstraction and verification of secure multi-party computations. In *Proc. Conf. on Foundations of Software Technology and Theoretical Computer Science*, pages 352–363, 2010.

[2] O. Al Bataineh and R. van der Meyden. Epistemic model checking for knowledge-based program implementation: an application to anonymous broadcast. In *SecureComm'10, 6th Int. ICST Conf. on Security and Privacy in Communication Networks*, 2010.

[3] D. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, pages 65–75, 1988.

[4] M. Cohen, M. Dam, A. Lomuscio, and H. Qu. A symmetry reduction technique for model checking temporal-epistemic logic. In *IJCAI*, pages 721–726, 2009.

[5] M. Cohen, M. Dam, A. Lomuscio, and F. Russo. Abstraction in model checking multi-agent systems. In *8th Int. Joint Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2009), Vol. 2*, pages 945–952, 2009.

[6] F. Dechesne, S. Orzan, and Y. Wang. Refinement of Kripke models for dynamics. In *Proc. Int. Colloq. on Theoretical Aspects of Computing - ICTAC*, pages 111–125, 2008.

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

[8] P. Gammie and R. van der Meyden. MCK: Model checking the logic of knowledge. In *Proc. 16th Int. Conf. on computer aided verification (CAV'04)*, pages 479–483, 2004.

[9] S. Goel, M. Robson, M. Polte, and E. Sirer. Herbivore: A scalable and efficient protocol for anonymous communication. Technical report, Cornell University, Ithaca, NY, February 2003.

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

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

[12] X. Luo, K. Su, M. Gu, L. Wu, and J. Yang. Symbolic model checking the knowledge in herbivore protocol. In *Proc. Workshop on Model Checking and Artificial Intelligence, AAAI-2010*, 2010.

[13] R. van der Meyden and K. Su. Symbolic model checking the knowledge of the dining cryptographers. In *Proc. 17th IEEE Computer Security Foundations Workshop*, pages 280–291, 2004.

[14] R. van der Meyden and T. Wilke. Preservation of epistemic properties in security protocol implementations. In *Proc. Conf. on Theoretical Aspects of Rationality and Knowledge*, pages 212–221, 2007.

[15] M. Waidner and B. Pfitzmann. The Dining Cryptographers in the disco: unconditional sender and recipient untraceability with computationally secure serviceability. In *Proc. Eurocrypt'89*, page 690, 1989.