1

Epistemic Model Checking for Knowledge-Based Program Implementation: an Application to Anonymous Broadcast[⋆]

Omar I. Al-Bataineh and Ron van der Meyden

School of Computer Science and Engineering,
University of New South Wales

Summary. Knowledge-based programs provide an abstract level of description of protocols in which agent actions are related to their states of knowledge. The paper describes how epistemic model checking technology may be applied to discover and verify concrete implementations based on this abstract level of description. The details of the implementations depend on the specific context of use of the protocol. The knowledge-based approach enables the implementations to be optimized relative to these conditions of use. The approach is illustrated using extensions of the Dining Cryptographers protocol, a security protocol for anonymous broadcast.

1.1 Introduction

In distributed systems, we generally would like agent's actions to depend upon the information that they have. However, the way that information flows in such systems can be quite complex. It has been proposed to address this complexity by the use of formal logics of knowledge [4].

In particular, *knowledge based programs* have been proposed as a level of abstraction that directly captures the relationship between an agent's knowledge and its actions, by allowing branching statements to contain formulas of the modal logic of knowledge, expressing what the agent knows about the global state of the system. This has several advantages. By focusing on what

---

[⋆] A version of this paper appears in Proc. SecureComm'10, 6th International ICST Conference on Security and Privacy in Communication Networks, Singapore, Sept 7-9 2010 ©Spinger-Verlag. 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.

information is required, rather than how it is encoded, knowledge-based programs can be more intuitive and more easily verified to be correct. They can also provide a common description that is independent of assumptions such as the failure modes of communication channels in the system. Finally, knowledge-based programs lead us to implementations that are *optimal* in their use of information, in the sense that agents do not overlook opportunities to use relevant information that is available in their local states.

A cost of the abstraction that knowledge-based programs provide, is that they are more like specifications than concrete programs, so cannot be directly executed. To obtain an executable program, it is necessary to replace the tests for knowledge in the knowledge based program by equivalent concrete predicates of the agent's local state. Because of the complexity of information flow in distributed systems, such concrete predicates can be difficult to find. To date, this task has generally been carried out by pencil and paper reasoning. Perhaps for this reason, there remain only a handful of worked out examples of the development of concrete implementations of knowledge-based programs (e.g., [1, 3, 7, 9]).

The difficulty can be addressed through the use of model checking technology for the logic of knowledge. Model checkers are systems that take as input a formal model of a system, together with a specification, and determine whether that specification is satisfied by the model [11]. The specification language used in model checkers is generally a form of temporal logic, but in recent years work has begun on the development of model checkers based on logics of knowledge [5, 12, 19]. We describe a methodology for the use of this latter class of model checkers to the development of implementations of knowledge based programs. The methodology is partially automated. It assists users in finding a concrete predicate that is equivalent to a knowledge condition in a knowledge-based program by means of an iterative process, in which automatically computed counterexamples to a user's guess for the concrete predicate are used by the user to construct an improved concrete predicate, until one is found that is equivalent to the knowledge condition.

We illustrate the methodology by means of an example in which we use the epistemic model checker MCK [5], to develop concrete implementations of a knowledge-based program for anonymous broadcast, based on multiple rounds of Chaum's Dining Cryptographers Protocol [2].

The Dining Cryptographers Protocol enables a message to be broadcast anonymously, under the assumption that only one agent is attempting to broadcast a message. The objective of the extension that we consider is to remove this assumption, so that any number of agents may broadcast their messages anonymously. One of the main difficulties in this is that, since agents operate independently, it is possible for simultaneous broadcasts to interfere with each other, causing a failure in the transmission. Thus, a key issue is to enable the agents to detect conflicts in the transmission, and to respond appropriately when a conflict is detected.

In our analysis, we express the expected behaviour using a knowledge based program that conditions the agent's actions on whether it knows that there is a conflict. We then use our model checking supported methodology to identify exactly the concrete conditions under which an agent knows whether there is a conflict. These conditions turn out to have a surprising level of complexity. In particular, we find that these conditions can differ, depending on the assumptions that we make about the number of agents wishing to broadcast.

Our approach leads to the discovery (assisted by automation) of a number of subtleties concerning the protocol that, to our knowledge, have not been previously noticed. In particular, we find that it is possible for agents to detect conflicts (or lack of conflict) in some quite unexpected situations. Moreover, we discover situations where, even though the protocol terminates, an agent cannot be sure that its message has been successfully transmitted (although it may have a high subjective probability that this is the case). Our results both show that there are previously unnoticed opportunities to optimize the protocol, and help to clarify what should be the specification of the protocol (the previous literature generally describes the protocol without providing a formal specification beyond the statement that it is intended for anonymous broadcast.)

The structure of the paper is as follows. We give a brief introduction to the logic of knowledge and epistemic model checking in Section 1.2. In Section 1.3 we discuss knowledge-based programs and describe our methodology for the development of their implementations using epistemic model checking. The Dining Cryptographers problem and its extensions are introduced in Section 1.4. In Section 1.5, we describe the application of our methodology to this protocol. Finally, some conclusions are drawn in Section 1.6.

1.2 Model Checking Epistemic Logic

Epistemic logics are a class of modal logics that include operators whose meaning concerns the information available to agents in a distributed or multi-agent system. We describe here briefly a version of such a logic combining operators for knowledge and linear time, and its semantics in a class of structures known in the literature as *interpreted systems* [4]. We then discuss the model checker MCK [5], which is based on this semantics.

Suppose that we are interested in systems comprised of n agents and a set *Prop* of atomic propositions. The syntax of the fragment of the logic of knowledge and time relevant for this paper is given by the following grammar:

$$\phi ::= \top \mid p \mid \neg\phi \mid \phi \wedge \phi \mid K_i\phi \mid X\phi$$

where $p \in \mathit{Prop}$ is an atomic proposition and $i \in \{1 \ldots n\}$ 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, and $X\phi$ means that ϕ will be true at the next moment of time.

The semantics we use is the *interpreted systems* model for the logic of knowledge [4]. For each $i = 0 \ldots n$, let S_i be a set of states. For $i = 0$, we interpret S_i as the set of possible states of the environment within which the agents operate; for $i = 1 \ldots n$ we interpret S_i as the set of *local states* of agent i. Intuitively, a local state captures all the concrete pieces of information on the basis of which an agent determines what it knows. We define the set of *global state* based on such collection of environment and local states, to be the set $S = S_0 \times S_1 \times \ldots \times S_n$. We write s_i for the i-th component (counting from 0) of a global state s. A *run* over S is a function $r : \mathbf{N} \rightarrow S$. An *interpreted system* for n agents is a tuple $\mathcal{I} = (\mathcal{R}, \pi)$, where $\mathcal{R}$ is a set of runs over S, and $\pi : S \rightarrow \mathcal{P}(Prop)$ is an interpretation function.

A *point* of $\mathcal{I}$ is a pair (r, m) where $r \in \mathcal{R}$ and $m \in \mathbf{N}$. We say that two points $(r, m), (r', m')$ are *indistinguishable* to agent i, and write $(r, m) \sim_i (r', m')$, if $r(m)_i = r'(m')_i$, i.e., if agent i has the same local state at these two points. We define the semantics of the logic by means of a relation $\mathcal{I}, (r, m) \models \phi$, where $\mathcal{I}$ is an intepreted system, (r, m) is a point of $\mathcal{I}$ and ϕ is a formula. This relation is defined inductively as follows:

- $\mathcal{I}, (r, m) \models p$ if $p \in \pi(r(m))$,
- $\mathcal{I}, (r, m) \models \neg\phi$ if not $\mathcal{I}, (r, m) \models \phi$
- $\mathcal{I}, (r, m) \models \phi_1 \wedge \phi_2$ if $\mathcal{I}, (r, m) \models \phi_1$ and $\mathcal{I}, (r, m) \models \phi_2$
- $\mathcal{I}, (r, m) \models X\phi$ if $\mathcal{I}, (r, m + 1) \models \phi$
- $\mathcal{I}, (r, m) \models K_i\phi$ if for all points (r', m') of $\mathcal{I}$ such that $r(m) \sim_i r'(m')$ we have $\mathcal{I}, (r', m') \models \phi$

We note that the semantics of the knowledge operator depends not just on the run at which the formula is being evaluated, but also the set of all possible runs. Changing the set of runs (e.g., by making changes to the protocol), can change what an agent knows. Since knowledge-based programs change agent behaviours based on what the agent knows, this makes the semantics of knowledge-based programs somewhat subtle.

MCK is a model checker based on this semantics for the logic of knowledge. For a given interpreted system $\mathcal{I}$, and a specification ϕ in the logic of knowledge and time, MCK computes whether $\mathcal{I}, (r, 0) \models \phi$ holds for all runs r of $\mathcal{I}$.

Since interpreted systems are infinite structures, MCK allows an interpreted system to be given a finite description in the form of a program from which the interpreted system can be generated. This description is given using:

1. A list of global variables making up states of the environment, and their types.
2. A listing of the agents in the system, together with the global variables that they are able to access. For each agent, we may also introduce local variables. If v is a local variable of agent A, then we may refer to this

variable in specification formulas as $A.v$. Local variables may be aliased to global variables.
A subset of the local variables is specified as being *observable* to the agent. This means that it will be taken into account in the definition of the indistinguishability relation for the agent.
3. A statement `init_cond` ϕ, where ϕ is a boolean formula. All assignments satisfying this formula represent an initial state of the system.
4. A program that describes the protocol executed by each agent. The protocol describes how the agent chooses its actions depending on its history.

Executing the agent protocols starting at an initial state generates a set of runs, that we take to be the set of runs of the interpreted system generated by input script. (The agents operate in lock-step, each agent executing a single action in each step. Write-conflicts are syntactically prevented.) If V is the set of all local and global variables in the system, then the component $s_0 = r(n)_0$ of the global state at each point (r,n) of a run r is a well-typed assignment of values to the variables V. The local states s_i of agent i in these runs are defined using the variables declared to be local. MCK allows this to be done in a number of ways, each giving a different semantics for the knowledge operators. The construction of local states relevant to the present paper is the *perfect recall interpretation.* Writing $s_0 \restriction V_i$ for the restriction of the assignment s_0 to the variables V_i that are observable to agent $i = 1 \ldots n$, the local states are defined to be the sequence

$$r(n)_i = (r(0)_0 \restriction V_i)\,(r(1)_0 \restriction V_i)\,\ldots(r(n)_0 \restriction V_i),$$

i.e., the local state is the history of all values of the variables observable to the agent.

This perfect recall intepretation of knowledge is particularly relevant for analyses in which security or the optimal use of information are of concern. In both cases, we are interested in determining the maximal information that an agent is able to extract from what it observes. Both issues are significant in the example that we study in this paper. MCK is the only model checker currently available that supports *symbolic* model checking for the perfect recall interpretation of knowledge. (DEMO [19] uses a less scalable explicit state algorithm.)

1.3 Implementation of Knowledge-based Programs

Knowledge-based programs [4] are like standard programs, except that expressions may refer to agent's knowledge. That is, in a knowledge-based program for agent i, we may find statements of the forms if ϕ then P_1 else P_2 and $v := \phi$, where ϕ is a formula of the logic of knowledge that is a boolean combination of atomic formulas concerning the agent's local variables and formulas of the form $K_i\psi$, and P_1, P_2 are knowledge-based programs for agent i.

Unlike standard programs, knowledge-based programs cannot in general be directly executed, since, as noted above, the satisfaction of the knowledge subformulas depends on the set of all runs of the program, which depends on the actions taken, which in turn depends on the satisfaction of these knowledge subformulas.

This apparent circularity is handled by treating knowledge-based programs as specifications, and defining when a concrete standard program satisfies this specification. Suppose that we have a standard 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. In order to handle the perfect recall semantics, we also allow P to add local *history variables* v and code fragments of the form $v := e$, where e is an expression, that update these history variables, so as to make information about past states available at the current time. The predicate p_ϕ may depend on the history variables.

The concrete program P generates a set of runs that we can take to be the basis of an interpreted system $\mathcal{I}(P)$. We now say that P is an *implementation* of the knowledge-based program $\mathbf{P}$ if for each formula ϕ in a conditional, we have that in the interpreted system $\mathcal{I}(P)$, the formula $p_\phi \Leftrightarrow \phi$ is valid (at times when the condition is used). That is, the concrete condition is equivalent to the knowledge condition in the implementation. In general, knowledge-based programs may have no implementations, a behaviourally unique implementation, or many implementations. Some conditions are known under which a behaviourally unique implementation is guaranteed to exist. One of these conditions is that agents have perfect recall and all knowledge formulas in the program refer to the present time (rather than to the past or future). This case will apply to the knowledge-based programs we consider in this paper, so we are guaranteed behaviourally unique implementations.

We now describe a partially automated process, using epistemic model checking, that can be followed to find implementations of knowledge-based programs $\mathbf{P}$ (provide these terminate in a finitely bounded time: this applies to our examples) The user begins by introducing a local boolean variable v_ϕ for each knowledge formula $\phi = K_i\psi$ in the knowledge-based program, and replacing ϕ by v_ϕ. Treating v_ϕ as a history variable, the user may also add to the program statements of the form $v_\phi := e$, relying on their intuitions concerning situations under which the epistemic formula ϕ will be true. This produces a standard program P that is a candidate to be an implementation of the knowledge-based program $\mathbf{P}$. (It has, at least, the correct syntactic structure.)

To verify the correctness of P as an implementation of $\mathbf{P}$, the user must now check that the variables v_ϕ are being maintained so as to be equivalent to the knowledge formulas that they are intended to express. This can be done using epistemic model checking, where we verify formulas of the form

$$X^n(pc_i = l \Rightarrow (v_\phi \Leftrightarrow K_i\psi))$$

where n is a time at which the test containing ϕ may be executed, pc_i is the program counter of agent i and l is a label for the location of the expression containing ϕ. (This conditioning on the program counter can be dispensed with when the expression is known to always occur at particular times n, as it always is in our examples. More generally, we would write a formula that checks equivalence at *all* times for nonterminating programs, but the resulting model checking problem is undecidable with respect to the perfect recall semantics.)

In general, the user's guess concerning the concrete condition that is equivalent to the knowledge formula may be incorrect, and the model checker will report the error. In this case, the model checker can be used to generate an *error trace*, a partial run leading to a situation that falsifies the formula being checked. The next step of our process requires the user to analyse this error trace (by inspection and human reasoning) in order to understand the source of the error in their guess for the concrete condition representing the knowledge formula. As a result of this analysis, a correction of the assignment(s) to the variable v_ϕ is made by the user (this step may require some ingenuity on the part of the user.) The model checker is then invoked again to check the new guess. This process is iterated until a guess is produced for which all the formulas of interest are found to be true, at which point an implementation of the knowledge-based program has been found.

In many cases, this process can proceed monotonically. Starting from an initial assignment $v_\phi := e$, where e is a condition that the user can easily see to be *sufficient* for $K_i\psi$, the error trace leads to the identification of a situation where i may know ψ, which is not covered by the condition e. (That is, where $K_i\psi \Rightarrow e$ does not hold.) An analysis of this condition may lead to the discovery of another sufficient condition e'. In this case, the user can take as the next guess the assignment $v_\phi := e \vee e'$. Continuing in this way, we obtaining an increasing sequence of concrete lower approximations to the knowledge formula, eventually converging to the correct implementation. (We note that such a condition e' can always be found, since we may always take it to be a complete description of the run producing the counter-example. Finding a good generalization that remains a sufficient condition for the knowledge formula may be more difficult.)

In general, monotonicity is not guaranteed, but it obtains in our example in this paper. We leave the question of characterizing the situations where monotonicity applies to future work, and turn to a demonstration of the process on a particular example, introduced in the next section.

1.4 Chaum's dining cryptographers protocol

Chaum's dining cryptographers protocol [2, p. 65] is an example of a protocol for secure multiparty computation: it enables the value of a function of a group of agents to be computed while revealing nothing more than that value. Chaum introduces the protocol with the following story:

> Three cryptographers are sitting down to dinner at their favourite restaurant. Their waiter informs them that arrangements have been made with the maitre d'hotel for the bill to be paid anonymously. One of the cryptographers might be paying for the dinner, or it might have been NSA (U.S National Security Agency). The three cryptographers respect each other's right to make an anonymous payment, but they wonder if NSA is paying. They resolve their uncertainty fairly by carrying out the following protocol:
> 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.

This version of the dining cryptographers protocol has frequently been the focus of studies of verification of security protocols, but it is just one of many variants discussed in Chaum's paper. 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.

He notes that "undetected collision results only from an odd number of synchronized identical message segments". As a particular realization of this idea, he discusses grouping communication into blocks and the use of the following *2-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.

This idea has been implemented as part of the Herbivore system[6]. (Herbivore also adds mechanisms for dividing the group of participants into cliques of sufficient size to provide reasonable anonymity guarantees, as well as protocols for joining and leaving the group of participants - we will not discuss these extension here.) The Herbivore authors note that

> If an even number of nodes attempt to reserve a given slot, the collision will be evident in the reservation phase, and they will simply wait until the next round to transmit. If an odd number of nodes collide, the collission will occur during the transmission phase.

The remarks above do not constitute a concrete definition of the protocol, and leave a number of questions concerning the implementation open. 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?)

Note that each round of the DC protocol has been proved correct, but what about the way in which the rounds are combined? It is not immediately clear that there are not subtle flows of information!

Prior knowledge of the participants may also affect the flow of information. For example, suppose that the protocol is being used for the participants in a referendum to anonymously announce their votes. In this case it is known that all particpants will attempt to reseve a slot - does this information change the flow of information in any way? If so, does it affect the security of the protocol? One of the benefits of verification by epistemic model checking is that it permits such questions about variants of a protocol, and its application in a particular setting to be investigated efficiently without requiring reconstruction of possibly complex proofs.

1.5 The 2-phase Broadcast Protocol as a Knowledge-based Program

It is interesting to note that the descriptions of the 2-phase protocol above are, in their level of abstraction, more like knowledge-based programs than like concrete implementations. In this section, we explicitly study the protocol from this perspective, and apply our partially automated methodology to derive the concrete implementations. We consider a setting with 3 agents who use 3 slots for their broadcast. Each slot permits the transmission of a single-bit message.

1.5.1 The Knowledge-Based Program

Figure 1.1 represents the 2-phase protocol as a knowledge-based program. The parameters of the protocol in the first line alias certain local variables to global variables in the environment. Variable `i` is a number in the range 1..3 used to index the present instance of the protocol, and variables `keyleft` and `keyright` represent keybits (referred to as "coins", above), which are shared between by agents in the appropriate pattern. Note that since a fresh set of keybits needs to be used for each instance of the basic Dining Cryptographers protocol (which we run 6 times here), we assume that an external process generates fresh values for these keybit variables at each step; we omit the details. The final variable `said` in the parameters represent the array of public announcements by the agents at each step. All arrays are assumed to be indexed starting from 1. The local variable `slot-request` records the slot number (in the range 1..3) that this agent will attempt to reserve. If `slot-request`=0, then the agent will not attempt to reserve any slot. The variable `message` records the single bit message that the agent wishes to anonymously broadcast (if any). Variables for which an initial value is not explicitly specified can take any initial value. We write '$\oplus$' for the exclusive or operation.

```
protocol dc_agent(i:[1,3], keyleft,keyright,said[3]:Bool) {
local variables:
     slot-request:[0,3],
     message:Bool,
     rcvd0[3], rcvd1[3], dlvrd: Bool (initially false);
//reservation phase
for ($s = 1$; $s \leq 3$; $s$++)
{
  said[i] := (keyleft$\oplus$ keyright$\oplus$ (slot-request=$s$));
}
//transmission phase
for ($s = 1$; $s \leq 3$; $s$++)
{
  if (slot-request = $s \wedge \neg K_i$(conflict($s$))
     then said[i] := (keyleft$\oplus$ keyright$\oplus$ message)
     else said[i] := (keyleft$\oplus$ keyright$\oplus$ false);
  rcvd0[s] := $K_i$(sender($i, 0, s$));
  rcvd1[s] := $K_i$(sender($i, 1, s$))
};
dlvrd:= $\bigwedge_{x \in Bool, t=1..3}$((message = $x$ $\wedge$ slot-request = $t$) $\Rightarrow$
                                        $K_i(\bigwedge_{j \neq i} K_j$sender($j, x, t$)))
}
```

Figure 1.1: The knowledge-based program *CDC*

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

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

i.e., there exist two distinct agents i and j both requesting slot s.

The term $\texttt{sender}(i, x, s)$ represents that an agent other than i is sending message x in slot s; this is defined as

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

Thus, the variable `rcvd0[s]` is assigned to be true if in round s, the agent learns that someone else is trying to send the bit 0, and similarly for `rcvd1[s]`. This addresses an issue that is not explicitly mentioned in the discussion of the two-phase protocol above, viz., how does an agent know whether it has received a transmission from another? Note that this is pertinent because the knowledge-based program allows that, although an agent has declared that it wishes to reserve a slot, it may still back off from the transmission if it discovers that there is a conflict. But will the receiver always know that it has done so?

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. One of the benefits of the knowledge-based approach is that it makes explicit the difference between this and another interpretation of the protocol where, in place of the condition $\neg K_i(\texttt{conflict}(s))$, we use the condition $K_i(\neg\texttt{conflict}(s))$. In this *conservative* version, an agent would broadcast only if it is certain that there is not a conflict on its desired slot. Both versions may be appropriate depending on the circumstances, but we focus our discussion here on the speculative version.

Since an agent may attempt to reserve a slot, and then back off, or may send in a reserved slot without success, 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 (anonymous) transmission has been successful. This is the case if all other agents know that *some* agent sent the bit $i.\texttt{message}$ in slot $j.\texttt{slot-request}$. (Subtleties about the semantics of the logic of knowledge prevent simplification of this formula by substitution of these expressions for x and t.)

In order to set up the appropriate configuration of the 3 agents and to alias their parameters to variables in the environment, we use the following declaration block:

```
   agent C2 : dc_agent(1,k31,k12,said)
   agent C3 : dc_agent(2,k12,k23,said)
   agent C3 : dc_agent(3,k23,k31,said)
```

where the `k`ij are boolean variables that represent the keybit shared between agent i and agent j.

In Figure 1.2, we give the generic structure of a possible implementation of the knowledge-based program, as we seek using our partially-automated process. The lines marked with (+) indicate places of difference with CDC.

Here we have introduced some history variables `rr[s]` that record the *round results* `said[0]`$\oplus$ `said[1]`$\oplus$`said[2]` obtained from each round s of the basic Dining Cryptographers protocol. Note that, because of the pattern of sharing of the keybits between the agents, this expression contains each keybit value twice, so that the keybits cancel out, leaving just the exclusive-or of the actual content being transmitted by each of the agents (in each assignment to `said[i]`, this is the final term in the exclusive-or). In particular, under the assumption that just one agent has a genuine message x to transmit in round j, and the others transmit $false$, we obtain that `rr[j]`$=x$.

The variable `kc[s]` is used to represent the epistemic condition concerning conflict in the knowledge-based program (either $\neg K_i(\texttt{conflict}(s))$ or $K_i(\neg\texttt{conflict}(s))$, depending on whether we are dealing with the speculative or the conservative version). Thus, in verifying that we have an implementation, the key condition to be checked is whether `kc[s]` $\Leftrightarrow \neg K_i(\texttt{conflict}(s))$ (respectively, `kc[s]` $\Leftrightarrow K_i(\neg\texttt{conflict}(s))$) is valid at the times the `if` statement is executed. The main difficulty in finding an implementation is to find the appropriate concrete assignment for this variable that will make this condition valid. Similarly we seek assignments to the variables `rcvd0[s], recvd1[s]` that give these the intended meaning.

1.5.2 Verification Conditions

In order to apply our methodology, it is necessary for the user to substitute a guess for parts of the implementation marked '???', and then to use model checking to check the correctness of the guess. We now discuss the formulas that are used to verify the implementation. In general, the conditions need to be verified only at specific times n, straightforwardly determined from the structure of the program. We generally omit discussion of this.

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

Specification 1: `kc[s]` *correctly represents knowledge of the existence of a conflict in slot* $s = 1..3$*.* In case of the speculative interpretation, we use the formula

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

and in case of the conservative implementation, we use the formula

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

```
protocol dc_agent(i:[0,2], keyleft,keyright,said[3]:Bool) {
local variables:
      slot-request:[0,3],
      message:Bool,
      rcvd0[3], rcvd1[3]:Bool (initially false),
      rr[6]:Bool,                                                    (+)
      kc[3]:Bool (initially false);                                  (+)
//reservation phase
for ($s = 1$; $s \leq 3$; $s$++)
{
  said[i] := (keyleft⊕ keyright⊕ (slot-request== $s$));
  rr[s] :=said[0]⊕ said[1]⊕ said[2];                                 (+)
}
//transmission phase
for ($s = 1$; $s \leq 3$; $s$++)
{
  kc[s] :=???;                                                       (+)
  if (slot-request== s ∧ kc[s])
      then said[i] := (keyleft⊕ keyright⊕ message)
      else said[i] := (keyleft⊕ keyright⊕ false);
  rr[s+3] := said[0]⊕ said[1]⊕ said[2];                              (+)
  rcvd0[s] := ???;                                                   (+)
  rcvd1[s] := ???;                                                   (+)
}
dlvrd:= ???                                                          (+)
}
```

Figure 1.2: A generic implementation of CDC

(In both cases, the appropriate values of n are 7, 12 and 17, where we treat the `for` loops as macros and the `if` conditions as taking zero time.)

As remarked above, it has been claimed that the 2-phase protocol is guaranteed to detect a conflict either in the slot-reservation phase or else in the transmission phase. To verify this, we can use the following specification:

Specification 2: A conflict is always detected.

$$X^n(\texttt{conflict}(s) \Rightarrow K_i(\texttt{conflict}(s)))$$

where we may take time n to correspond to the final time in the protocol. We remark that the converse implication is trivial from the semantics of knowledge.

As will discuss below, Specification 2 is arguably too strong, since agents may not be able to learn about conflicts on slots they do not reserve. Thus, the following weaker specification is also of interest.

Specification 3: If there is a slot conflict involving agent i, then agent i detects it.

$$X^n((\texttt{conflict}(s) \wedge i.\texttt{slot-request} = s) \Rightarrow K_i(\texttt{conflict}(s)))$$

where again we take n to correspond to the end of the protocol.

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 by the time n immediately after the transmission in slot s is intended to be represented by the variables `rcvd0[s]` and `rcvd1[s]`. 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 4: reception variables correctly represent transmissions by others

$$X^n(i.\mathtt{rcvd0}[s] \Leftrightarrow K_i(\mathtt{sender}(i,0,s))) \qquad (4a)$$

and

$$X^n(\mathtt{rcvd1}[s] \Leftrightarrow K_i(\mathtt{sender}(i,1,s))) \qquad (4b)$$

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

Specification 5: delivery variables correctly represent knowledge about delivery

$$X^n(i.\mathtt{dlvrd} \Leftrightarrow \textstyle\bigwedge_{x\in Bool, t=1..3}(i.\mathtt{message} = x \wedge i.\mathtt{slot\text{-}request} = t \\ \Rightarrow K_i(\textstyle\bigwedge_{j\neq i} K_j \mathtt{sender}(j,x,t))))$$

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 $\bigwedge_{j\neq i}(\neg\hat{K}_i(j.\mathtt{message})$, 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 6: The protocol preserves anonymity

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

to be evaluated with n set to the final time of the protocol.

1.5.3 Finding an implemention of the knowledge-based program

We now illustrate how we find an implementation of the knowledge-based program using our methodology. We focus here on the speculative version, and consider a scenario where the number of agents that are seeking to broadcast— is initially unknown, and could be any value from the set $\{0..3\}$.

Our first task in implementing the knowledge-based program is to find an appropriate assignment for the variables $\texttt{kc}[s]$, and to verify that this assignment correctly represents knowledge about slot conflicts and validates *Specification 1.* It is plain from the discussion above that if an agent attempts to reserve slot s, but sees that the round result for that reservation attempt is not *true*, then this must be because some other agent also attempted to reserve the slot. Thus, in this case the agent detects a conflict. A reasonable guess for the assignment to $\texttt{kc}[s]$ to represent $\neg K_i(\texttt{conflict}(s))$ is therefore

$$\texttt{kc}[s] := \neg(\texttt{slot-request} = s \wedge \neg\texttt{rr}[s] = false) \ .$$

Indeed, this proves to be the correct choice: if we now model check *Specification 1s* then we find that this specification is true.[2]

The next question of interest is then whether *Specification 2* holds , as claimed. The answer obtained by model checking is that it does not, and the counter-example discovered is the following:

Example 1: (None of the agents discover conflict) Suppose that all agents (C1, C2, C3) would like to reserve slot 2 and each has message 1. The round results $\texttt{rr}[s]$ are shown in on the left in Figure 1.3, where we show for each agent the contribution other than keybits (which cancel out).

s	1	2	3	4	5	6
Agent C1	0	1	0	0	1	0
Agent C2	0	1	0	0	1	0
Agent C3	0	1	0	0	1	0
$\texttt{rr}[s]$	0	1	0	0	1	0

$\texttt{slot-request} = [2, 2, 2]$,
$\texttt{message} = [1, 1, 1]$

s	1	2	3	4	5	6
Agent C1	0	1	0	0	1	0
Agent C2	0	0	0	0	0	0
Agent C3	0	0	0	0	0	0
$\texttt{rr}[s]$	0	1	0	0	1	0

$\texttt{slot-request} = [2, 0, 0]$
$\texttt{message} = [1, 1, 1]$

Figure 1.3: Runs indistinguishable to C1

Now from agent $C1$'s perspective, this run of the protocol is indistinguishable from another run where only $C1$ attempts to reserve slot 2, and it still has message 1, shown on the right in Figure 1.3. Hence we have a situation where although there is a conflict agent $C1$ cannot know that there is a conflict, and *Specification 2* fails, contra to what one might have expected from the quote from [6] above.[3] Indeed, we see that the more liberal *Specification 3* also fails in this example.

---

[2] Strictly, in order to model check this claim, we first need to fill in the other '???' assignments. We remark that because of independencies, the outcome of model checking *Specification 1s* is the same *whatever* we choose for the other '???' assignments. We omit a detailed argument for this here.

[3] It is unclear if the authors of [6] intended to imply that all conflicts would be detected. They also state that messages are sent with an MD5 checksum, so most conflicts of messages somewhat longer than a single bit would in fact be detected

In the discussion above, we have focused on the agent's knowledge that there is a conflict. From the point of view of determining the appropriate assignments to the variables `rcvd0` and `rcvd1`, it would be helpful to determine under what circumstances an agent knows that there will be a transmission on a slot but there is *not* a conflict on that slot. Thus, it would be helpful to have a predicate $i.\texttt{conflict-free}(s)$ that is equivalent to $K_i(\bigvee_j j.\texttt{slot-request} = s \wedge \neg\texttt{conflict}(s))$. We now investigate this question, and use it to illustrate the iterative procedure to obtain local predicates that are equivalent to knowledge formulas.

Plainly, a round-result of 1 during the reservation phase implies that someone wishes to send in that slot. However, Example 1 also shows that $K_i\neg\texttt{conflict}(s)$ cannot hold in case agent i obtains round result 1 in a slot it intends to transmit in, and 0 in all other slots, since it is possible that all agents are attempting to transmit in the same slot. Hence a reasonable guess is

$$\texttt{conflict-free1}(s) = \texttt{rr}[s] = 1 \wedge \neg(\wedge_{t\in\{1,2,3\}\setminus\{s\}}\texttt{rr}[t] = 0)\ .$$

When we model check

$$X^n(i.\texttt{conflict-free1}(s) \Leftrightarrow K_i(\bigvee_j j.\texttt{slot-request} = s \wedge \neg\texttt{conflict}(s))$$

at time n after the transmission phase, we find that this formula is false. A counter-example produced by the model checker shows that this happens when $C1$ and $C3$ request slot 3, and $C2$ requests slot 1. Note that in this case the reservation round results are $(1, 0, 0)$. Here $C1$ and $C3$ detect a conflict in slot 3. Since there are only three agents, they are able to reason that the conflict must have been 2-way (else we have the scenario of Example 1). This means that they are able to deduce that there is *not* a conflict in slot 1.

This example motivates a second guess for the predicate $\texttt{conflict-free}(s)$, viz., (when all variables are local to agent i)

$$\begin{aligned}&\texttt{conflict-free2}(s) = \texttt{conflict-free1}(s) \vee \\ &\quad(\texttt{rr}[s] = 1\ \wedge\ \texttt{slot-request} \in \{1,2,3\}\setminus\{s\}\ \wedge\ \texttt{rr}[i.\texttt{slot-request}] = 0)\ .\end{aligned}$$

Model checking this predicate for equivalence to $K_i(\bigvee_j j.\texttt{slot-request} = s \wedge \neg\texttt{conflict}(s))$, we still find that the equivalence does not hold. The counter-example produced this time is the situation where agents $C1$ and $C2$ do not request a slot, but agent $C3$ requests slot s so that the round result of slot s is 1. Note that here, agents $C1$ and $C2$ know that any slot collision must be 2-way, since they cannot be a participant. Since the reservation request on slot s gave round result 1, there must be exactly one agent requesting slot s. With some reflection, we note that agent $C1$ would have been able to draw

---

with high probability through corruption of this checksum. However, even with this device, collisions of 3 identical messages would still go undetected, as noted by Chaum.

the same conclusion about slots 2 and 3 in case the round result pattern were $(0,1,1)$. Thus, we are led to the following improved guess:

$$\texttt{conflict-free}3(s) = \texttt{conflict-free}2(s) \vee (\texttt{rr}[s] = 1 \wedge \texttt{slot-request} \neq s)$$

At this point, model checking shows that we have found the predicate we seek.

Returning now to the question of when agents learn the bit that another agent is transmitting, we guess the assignment

$$\texttt{rcvd}1[s] := \texttt{rr}[s] = 1 \wedge \texttt{conflict-free}3(s) \wedge \texttt{slot-request} \neq s \ .$$

That is, the agent sees that there will be a conflict free transmission on slot s, but it is not itself using that slot. We now model check Specification 4b. Somewhat surprisingly, this specification turns out to be false! The counter example returned is one in which the agent is $C1$, all agents reserve slot 1, and the agents have messages $(1,1,0)$. Note that here, the round result obtained for the transmission is 0, so agent $C1$ detects the collision, which it knows must have been 3-way. It can also reason that the other agents cannot both have had messages 0, since this would have produced round result 0, thus, at least one must have had message 1! This observation leads to the revised guess

$$\begin{aligned}\texttt{rcvd}1[s] := (&\texttt{rr}[s] = 1 \wedge \texttt{conflict-free}3(s) \wedge \texttt{slot-request} \neq s)\vee \\ &(\texttt{slot-request} = 1 \wedge \texttt{rr}[s+3] \neq \texttt{message} \wedge \textstyle\bigwedge_{t\in\{1,2,3\}\setminus\{s\}} \texttt{rr}[t] = 0) \ .\end{aligned}$$

We now find that Specification 4b holds, so we have correctly implemented this part of the knowledge-based program. A similar assignment works for the assignment to `rcvd0` and *Specification 4a*.

This process can also be carried out also for the final specification *Specification 5*, which concerns the circumstances under which a sender knows that their message (if any) has been received by the others. One obvious situation when this is the case is when the sender i knows that the slot on which they are sending is conflict-free. Recall that this occurs only when two or more of the reservation round results equal 1, and note that this implies that all other agents also know that the slot on which i is sending is conflict-free. Thus the others will receive the message that i is sending (anonymously) on this slot. This suggests the assignment

$$\texttt{dlvrd} := \texttt{slot-request} = 0 \vee \bigvee_{s\in\{1,2,3\}} \texttt{slot-request} = s \wedge \texttt{conflict-free}3(s) \ .$$

When we model check this with respect to *Specification 5*, we find that that the specification holds, and we have a complete implementation of the knowledge-based program. Finally, we may also model check *Specification 6* and verify that the protocol preserves anonymity in the appropriate sense. This proves to be the case.

1.6 Conclusion

We have demonstrated the application of our partially automated methodology for knowledge-based program implementation on a protocol for anonymous broadcast. While, like related studies [8, 10, 18, 17, 15, 16], we verify that an anonymity property holds, the focus of our effort lies in other aspects of the protocol.

One of the main outcomes of the analysis is that the flows of information in the protocol are considerably more subtle than one might have expected. In particular, we find that there are circumstances, that go beyond those that have been identified in the literature, where agents are able to obtain knowledge of each other's bits. Significantly, we make this discovery not manually, but using automated support. We also address in our work a number of questions that have not been considered in the prior literature, viz., under what circumstances can a receiver be confident that they are receiving a transmission, and under what circumstances a sender can know that its transmission has been successful, and find complete answers to these questions in a particular scenario.

On the other hand, being based on model checking of a concrete model under very particular assumptions, our approach lacks generality: it does not yield an immediate answer to how our conclusions are affected by changing the number of agents, their topology, or the initial assumptions concerning the number of agents wishing to transmit. However, the methodology provides an efficient means to experiment with such questions. We are presently investigating further variants using our methodology, in order to obtain an empirical basis from which theoretical results may be generalized. Our present models are also starting to press the limits of the model checking technology (run times of the order of hours for some queries, for protocols of around 20 steps), so we are also investigating optimizations to increase the scale and complexity of the problems we can address. We plan to report on this in future work.

In work conducted independently, Luo et al [13] have also model checked knowledge in the 2-phase protocol, but they focus on a number of formulas concerning conflict detection, rather than attempting to implement a knowledge-based program, as we have done in this paper. They consider larger numbers of agents, but they do not consider the questions we have studied concerning reception and termination, nor do they try to find exact conditions under which knowledge properties of interest hold. They also use observational rather than perfect recall semantics, and justify this by an informal argument that what they do is equivalent to perfect recall. We believe their claim of equivalence to be correct, and it would be an interesting topic for future work to provide a more formal and systematic justification. (Some initial steps on optimizing models of the 2-phase protocol were already taken in [14].)

References

1. K. Baukus and R. van der Meyden. A knowledge based analysis of cache coherence. In *Proc. 6th Int. Conf. on Formal Engineering Methods*, volume 3308 of *LNCS*, pages 99–114. Springer, 2004.
2. D. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, pages 65–75, 1988.
3. C. Dwork and Y. Moses. Knowledge and common knowledge in a Byzantine environment: crash failures. *Information and Computation*, 88(2):156–186, 1990.
4. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge.* MIT Press, Cambridge, Mass., 1995.
5. 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)*, volume 3114 of LNCS, pages 479–483. Springer-Verlag, 2004.
6. 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.
7. V. Hadzilacos. A knowledge-theoretic analysis of atomic commitment protocols. In *PODS '87: Proc. 6th ACM Symp. on Principles of Database Systems*, pages 129–134, New York, NY, USA, 1987. ACM.
8. J. Y. Halpern and K. R. O'Neill. Anonymity and information hiding in multiagent systems. In *Proc. 16th IEEE Computer Security Foundations Workshop*, pages 75–88, 2003.
9. 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.
10. D. Hughes and V. Shmatikov. Information hiding, anonymity and privacy: a modular approach. *Journal of Computer Security*, 12 (1):3–36, 2004.
11. E. M. Clarke Jr., O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, 1999.
12. A. Lomuscio, H. Qu, and F. Raimondi. MCMAS: A model checker for the verification of multi-agent systems. In *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 682–688. Springer, 2009.
13. X. Luo, K. Su, M. Gu, L. Wu, and J. Yang. Symbolic model checking the knowledge in Herbivore protocol. In R. van der Meyden and G. Smaus, editors, *MoChArt 2010: 6th Int. Workshop on Model Checking and Artificial Intelligence*, 2010. LNCS, to appear, also AAAI Working Notes, 2010.
14. L. L. V. Nhu. Enhancing an epsitemic logic model checker for application to extensions of the dining cryptographers protocol. Honours thesis, School of-Computer Science and Engineering, University of New South Wales, Nov 2005.
15. P. Ryan and S. Schneider. *The modelling and analysis of security protocols: the CSP approach.* Addison-Wesley Professional, 2000.
16. S. Schneider and A. Sidiropoulos. CSP and anonymity. In *Proc. of the European Symposium on Research in Computer Security (ESORICS)*, pages 198–218. Springer-Verlag, 1996.
17. P. Syverson and S. Stubblebine. Group principals and the formalization of anonymity. In *FM '99: Proc. Wold Congress on Formal Methods in the Development of Computing Systems-Volume I*, pages 814–833, London, UK, 1999. Springer-Verlag.

18. R. van der Meyden and K. Su. Symbolic model checking the knowledge of the dining cryptographers. In *Proc. 17th IEEE Computer Security Foundation Workshop*, pages 280–291. IEEE Computer Society, 2004.
19. J. van Eijck. Dynamic epistemic modelling. Technical report, Centrum voor Wiskunde en Informatica, Amsterdam, 2004. CWI Report SEN-E0424.