Model Checking a Knowledge Exchange Scenario

Sieuwert van Otterloo   **Wiebe van der Hoek**   **Michael Wooldridge**
Department of Computer Science
University of Liverpool
Liverpool L69 7ZF, UK
{sieuwert,wiebe,mjw}@csc.liv.ac.uk

Abstract

We are interested in applying model checking techniques to the verification of communication protocols which require safe communication. Typically, in such scenarios, one desires to demonstrate that one party can reliably communicate information to another party without a third party being able to determine this information. Our approach involves using the modal logic of knowledge, which has only relatively recently been studied in the context of secure protocols. We demonstrate our approach by means of a detailed case study: The Russian cards problem. This is an example of a security protocol with nontrivial requirements on the knowledge of the agents involved. Using the Russian cards problem as an example it is shown how the satisfaction of properties involving knowledge can be verified in a standard model checker — in our case, SPIN.

1 Introduction

A typical requirement in security protocols is that two agents should be able to exchange some information without a third agent being able to eavesdrop and learn some of the information. One conventional way to achieve this is to let the two parties use a key to encrypt the information, making the decryption of the encoded message computationally intractable for third parties. Another approach is to equip the two parties with sufficient knowledge to publicly announce some messages, after which they know both that the information is obtained and that no other agent can have inferred this from the message. Rather than relying on what the agents can(not) computationally achieve, the success of the latter approach completely depends on what the agents *know*. In other words, rather than the more traditional *capability based protocol*, there is currently also a growing interest in *knowledge based protocols*. A nice and conceptually clear example of such a protocol is given in [van Ditmarsch, 2003], where it is referred to as the "Russian cards problem". This problem will serve as a case study for the work presented in this paper.

To specify epistemic properties — properties pertaining to the knowledge of communication participants — one can make use of a by now well-accepted modal logic for knowledge [Meyer and van der Hoek, 1995; Fagin *et al.*, 1995], which provides modal operators K_i to represent what a specific agent i knows, and group modalities E_G and C_G to represent what everybody in a group G knows, or what is common knowledge in group G, respectively. Where logics such as that proposed by Burrows, Abadi and Needham (BAN, [Burrows *et al.*, 1990]) provide a suitable tool to reason about capability-based protocols, we believe that *Dynamic Epistemic Logic* (DEL, see [Baltag *et al.*, 2002; van Ditmarsch *et al.*, 2003]) is an appropriate framework to deal with knowledge-based protocol verification. DEL provides a framework within which to reason about what agents learn, and is an extension of epistemic logic. Since a security protocol is robust when it is possible to prove that certain properties hold, even when all agents intercept all messages, we will here restrict ourselves to a version of DEL in which all learning is the result of a *public announcement*.

Checking that a particular protocol satisfies certain desirable properties is a *verification* problem. One of the most successful general approaches to verification is known as *model checking* [Clarke *et al.*, 2000]. Model checking approaches to the analysis of capability based security protocols have proved rather successful (for an early example, see the work of Dolev and Yao [Dolev and Yao, 1998]). The basic idea here is to examine all possible execution traces by the protocol under investigation, in the presence of a malicious intruder with well-defined capabilities. One then either determines that the protocol enforces its security guarantees, or else obtains a sample trace of an attack on the protocol. This has led to the discovery of a number of attacks against existing protocols (for example, Lowe [Lowe, 1996] used a model checker for CSP to find a previously unknown error in the Needham-Schroeder Public-Key Authentication protocol [Needham and Schroeder, 1978]).

One of the problems with using model checking to verify properties involving knowledge is that existing model checkers are designed to verify temporal, rather than epistemic properties. In this paper, we use and further develop ideas from [van der Hoek and Wooldridge, 2002] to deal with this problem. We do this by considering the "Russian cards problem" [van Ditmarsch, 2003], which will be explained in the next section. The setting differs from problems such as the bit transmission problem [Meyer and van der Hoek, 1995;

van der Hoek and Wooldridge, 2002] or the dining cryptographers [van der Meyden and Su, 2002], in the sense that we also give an account of an *adversarial* agent, as opposed to a *co-operative* agent. In Section 3 we describe how we find solutions to this problem, and in Section 4 we present some conclusions.

2 The Russian Cards Problem

In the Russian Cards Problem, seven cards are distributed among three players. Alice and Bob receive each three cards, and Caroline receives the remaining card. All of them know the number of cards everyone has, they know their own cards, but they do not know how the remaining cards are distributed among the remaining players. Alice and Bob want to (commonly) know the deal of cards, while preventing Caroline from finding out who has any of the cards not in Caroline's possession.

As an attempt to a solution of the Russian Cards problem, one might devise a complicated dialogue, in which Alice and Bob make many cryptic comments which are helpful to them but not really to Caroline. However, instead, we are looking for a *direct exchange solution*, in which the actual dialogue is short: Alice makes a single (true) statement from which Bob learns which cards she has. Bob can then deduce which card Caroline has and (truthfully) announce this. If Alice learns Caroline's card she can deduce which cards Bob must have. In all such solutions, it is sufficient and necessary that Bob's reply to Alice is to announce which card Caroline holds. A direct exchange is thus completely characterised by Alice's initial announcement. Informally a statement is a *solution* if, after Alice announces this statement, Caroline does not know any of the cards. An example of a solution in a situation where Alice has cards 0,1 and 2 is that Alice says "I have one of the following five sets of three cards: 0, 1 and 2, 0,3 and 4, 1, 3 and 5, 2,3 and 6 or 4,5 and 6".

A logical description of this problem needs propositional logic to describe a deal of cards, epistemic operators to express knowledge, and some way of dealing with the dynamics of the problem. From now on A, B and C will be used to denote the three agents. Moreover, i is a variable over the agents, and the variables x and y range over the deck of cards $\mathcal{D} = \{0, 1, 2, 3, 4, 5, 6\}$. The atoms a_x are used to denote that fact that A holds card x. Similarly for b_x and c_x with respect to B and C, respectively.

The fact that these propositions must represent a card deal puts constraints on these propositions. For instance, every card i must be held by exactly one person ($\triangledown$ denotes exclusive or): $\bigwedge_{x\in\mathcal{D}}(a_x \triangledown b_x \triangledown c_x)$. Let us now denote the state in which A has cards 0,1,2, B has 3,4,5 and C has 6 by $s = 012|345|6$. In the sequel, we thus consider states s of the form $x_0x_1x_2|x_3x_4x_5|x_6$ where $x_0x_1x_2x_3x_4x_5x_6$ is a permutation of $\{0, \ldots, 6\}$ and where $x_0 < x_1 < x_2$ and $x_3 < x_4 < x_5$. Let S denote the set of all such states. A valuation function π then translates the basic facts to our propositional atoms: for instance, we have $\pi(s)(a_0) = \mathit{true}$ and $\pi(s)(a_3) = \mathit{false}$.

For reasoning about knowledge, we add knowledge operators K_A, K_B and K_C. The meaning of $K_i\phi$ is that agent i knows that ϕ is true. An agent knows something if it is true in each world that the agent considers possible. This leads us to consider Kripke models of the form

$$M = \langle S, \sim_A, \sim_B, \sim_C, \pi\rangle$$

where S and π are as defined above, and $\sim_i$ is an equivalence relation, modelling what agent i considers to be indistinguishable states. We then can interpret formulas on pairs (M, s), where s is a state from S. If φ is true in such a pair, we write $(M, s) \models \varphi$. Interpretation is according to the next definitions.

$$\begin{array}{ll}
(M,s) \models p & \text{iff } \pi(p) = \mathit{true} \\
(M,s) \models \neg\phi & \text{iff not } (M,s) \models \phi \\
(M,s) \models \phi \wedge \psi & \text{iff } (M,s) \models \phi \text{ and } (M,s) \models \psi \\
(M,s) \models K_i\varphi & \text{iff for all } t : s \sim_i t \text{ implies } (M,t) \models \varphi
\end{array}$$

We say that a formula φ is true in a model M, and write $M \models \varphi$, if $(M, s) \models \varphi$ for every state s in M.

Initially, agents cannot differentiate worlds in which they have the same cards. Thus, our initial Kripke model is the tuple $M^0 = \langle S, \sim^0_A, \sim^0_B, \sim^0_C, \pi\rangle$, where S and π are as defined above, and $x_0x_1x_2|x_3x_4x_5|x_6 \sim^0_A y_0y_1y_2|y_3y_4y_5|y_6$ iff $\{x_0, x_1, x_2\}$ equals $\{y_0, y_1, y_2\}$. Similarly for the other agents. Now let $M = \langle S, \sim_A, \sim_B, \sim_C, \pi\rangle$ be one of our Kripke models. The model M^ϕ should denote the model that is obtained when making a public announcement in M. More precisely, $M^\phi = \langle S, \sim^\phi_A, \sim^\phi_B, \sim^\phi_C, \pi\rangle$ is defined by refining the accessibility relations in such a way that the agents learn to distinguish ϕ-states from non-ϕ-states. In the next formula the refined relation $\sim^\phi$ is defined in terms of the previous epistemic relation $\sim$.

$$s \sim^\phi_i t \text{ iff } (s \sim_i t \text{ and } M, s \models \phi \Leftrightarrow M, t \models \phi)$$

We can now define the effect of announcements. Following recent work on dynamic epistemic logic [Baltag *et al.*, 2002; van Ditmarsch *et al.*, 2003] we represent announcements in the object language by using an update operator $[\phi]\psi$, which is read as "after ϕ becomes common knowledge between all agents, ψ holds".

Definition 1 *Suppose $M, s \models \phi$. Then $M, s \models [\phi]\psi$ is defined as $(M^\phi, s) \models \psi$.*

In states where the formula ϕ is not true, the update $[\phi]\psi$ is not defined. The reason this condition is posed is that an update with false "knowledge" does not make sense in our model of knowledge. According to the problem description, B must learn the deal of cards and C should not learn any card. These two learning goals for B and C are expressed in the epistemic formulas bknows ('Bob knows the deal of cards'— note that in order to know this it is sufficient for B to learn A's cards) and cig ('Caroline is, concerning any card that she does not hold, ignorant about who holds it').

$$\begin{array}{lcl}
\mathsf{bknows} & = & \bigwedge_{i\in\mathcal{D}} a_i \rightarrow K_B a_i \\
\mathsf{cig} & = & \bigwedge_{i\in\mathcal{D}} (\neg K_C a_i \wedge \neg K_C b_i)
\end{array}$$

We assume that an agent making an announcement φ knows it to be true. The requirement $\mathsf{req}(\varphi)$ of the protocol thus stipulates that A knows φ, and after the public announcement that φ we have both bknows and cig.

$$\mathsf{req}(\varphi) = \varphi \wedge [\varphi]\mathsf{bknows} \wedge [\varphi]\mathsf{cig} \qquad (1)$$

The above definition is the one we use in our model checking approach. In this definition, the information obtained by the announcement is exactly the truth of that announcement. This is not the only available choice, and in a security setting this will not always be a realistic assumption. In the following paragraphs it will be shown why this is a correct choice under our definition of a solution. The argument given is not just applicable to this specific example but for a wide class of multi-agent protocols with announcements.

The reason that agents can deduce more from an announcement than just the truth of the proposition used in the announcement, is that the agents also know what kind of game they are involved in. For instance, C may deduce more information from the fact that A knows $\varphi = a_0 \vee b_0$: if an outsider would announce φ, it would not be informative for C, but, on the other hand, if A would say he knew φ, agent C can deduce that A must hold a_0. But there is more to it: C even knows what A wants to achieve with his statement, namely $\mathsf{req}(\varphi)$, and we require that, even if we take this into account, C remains ignorant. One could say that the truth of φ is the *literal meaning* of the utterance. The *performative meaning* is the stronger fact that the agent utters that formula. In the context of a specific protocol, where all agents are aware of the precise protocol they are involved in, the performative meaning is very close to the fact that the uttering agent knows that the statement meets the exact knowledge goals that form the requirement. For the example the literal meaning is just $a_0 \vee b_0$. The performative meaning would depend on the context but it would at least imply the stronger statement a_0. For the Russian Cards problem, the fact that A uses φ has the performative meaning that A knows that $\mathsf{req}(\varphi)$ is true. This difference in meaning is especially important when considering adversarial agents. It is not a priori clear that, even when $\mathsf{req}(\varphi)$ is true in the actual world, C cannot guess the deal of cards after it comes to believe that $\mathsf{req}(\varphi)$ is true. To ensure C is ignorant even when C is aware of the performative meaning one would like to use, rather than (1), the following 'recursive' definition:

$$\mathsf{req}_2(\varphi) = \varphi \wedge [\varphi]\mathsf{bknows} \wedge [K_A\mathsf{req}_2(\varphi)]\mathsf{cig} \qquad (2)$$

Unfortunately $\mathsf{req}_2(\varphi)$ is not well-defined for every φ. The idea is that C can deduce from A uttering φ that A thinks that φ is a solution for the problem. A condition in the semantics of the update operator is that one can only meaningfully update with true formulas. However, if φ is not a solution, this assumption is violated. This can be resolved if one first defines what it means for a statement to be a solution. One cannot say that φ is a solution if $\mathsf{req}(\varphi)$ is true in all worlds. For no non-tautological formula φ it is the case that $M \models \mathsf{req}(\varphi)$ or $M \models \mathsf{req}_2(\varphi)$, since φ will not be true in every state of the model. One could argue that φ is a solution if $\mathsf{req}(\phi)$ is true in at least one state s. This is problematic because if A is restricted in its use of φ then it is giving C information that it is in the specific state in which it can use φ. Therefore we define a formula to be a *solution* if it can be used whenever true.

Definition 2 *Let φ be a formula for which $M \models \varphi \leftrightarrow K_A\varphi$. This formula is a solution if $M \models \phi \rightarrow \mathsf{req}(\phi)$.*

This definition uses our first, intuitively too weak, proposal. The good thing however is that if φ is a solution, then φ and $\mathsf{req}(\varphi)$ are equivalent. If this is the case then $\mathsf{req}_2(\varphi)$ is true and well-defined in all states where $\mathsf{req}(\varphi)$ holds, so we know that under this definition of a solution, considering performative meaning will not give adversarial agents more information.

To formally prove the equivalence and welldefinedness claims from the previous paragraph, we use the fact that all formulas true on the entire model are known by all agents. This gives $M \models K_A(\varphi \rightarrow \mathsf{req}(\varphi))$. We assumed that $M \models \varphi \leftrightarrow K_A\varphi$. From this we derive $M \models \varphi \rightarrow K_A\mathsf{req}(\varphi)$. Thus, if φ is a solution, req_2 is well-defined and true. The statement req_2 is the strongest possible conclusion C can draw from the announcement without risking to draw an incorrect conclusion.

An alternative approach is to define a safe communication using *common knowledge* [van Ditmarsch, 2003]. A requirement using a common knowledge operator C is

$$\mathsf{req}_C(\varphi) = \varphi \wedge [\varphi]C\mathsf{bknows} \wedge [K_A\varphi][C\mathsf{cig}]C\mathsf{cig} \qquad (3)$$

The use of common knowledge is a sufficient but not necessary step to avoid the problems around literal and performative meaning. This approach generates the same answers: Each safe communication according to Van Ditmarsch is also what we call a solution. An advantage of his formalisation is that this formula has only to be checked in a single state. A disadvantage is that the formula contains a more complicated nesting of knowledge operators.

3 Model Checking and the Russian Cards Problem

We now turn to the key contribution of this paper: the use of model checking to automatically verify properties of the Russian Cards problem. Model checking was developed as a technique for automatically verifying that finite state systems satisfy requirements expressed in the language of temporal logic [Clarke *et al.*, 2000]. The term 'model checking' arises from the fact that the state transition system of a finite state system can be understood as a model (in fact, a Kripke structure) for temporal logic. Verifying that a system satisfies certain requirements, where these requirements are expressed in temporal logic, can then be understood as a problem of checking that the formula representing the requirements is realised in the model. In our work, we make use of the SPIN model checker for Linear-time Temporal Logic [Holzmann, 1997]. SPIN takes as input a system S described using the PROMELA modelling language, and a formula ϕ of LTL, and checks whether or not the formula is realised in the system; if the answer is "no", then it produces an execution trace of S that falsifies ϕ. While SPIN has been widely used for the verification of LTL properties, it is not currently equipped to deal with LTL formulas that contain knowledge modalities.

In previous work [van der Hoek and Wooldridge, 2002], we explored the use of SPIN for model checking temporal knowledge properties of systems, by finding *local propositions* to stand for knowledge modalities in formulas.

LTL is a modal propositional logic which allows to specify properties of linear sequences of states — *runs*. Each PROMELA specification defines a set of runs, and SPIN checks whether the LTL formula holds for all runs. The PROMELA specification itself does not have to be linear (two different states can be reachable from the same state) but properties that must refer to two different runs containing the current state cannot be expressed in LTL. One can use any boolean expression in the PROMELA specification as propositions in the LTL formula. It is custom to define the propositions one wants to use as macro's and this style has been adopted in our code. Two additional operators extend the language to talk about runs: $\Box$ (always) and $\diamond$ (sometime) . Let r be a run of the system and s a state in r: then $(r, s) \models \Box\phi$ if ϕ is true in all states of r following s. Finally, $(r, s) \models \diamond\phi$ if there is some state in r after s that satisfies ϕ.

Before one can apply modelchecking to prove properties of any protocol, one must obtain a formal specification implementing the protocol. This is difficult because the formal model specified in Section 2 defines the epistemic relations directly, while this is not possible in most specification languages. In PROMELA one can only define the dynamic transitions directly. Epistemic relations can be derived by using the *interpreted systems* model of knowledge [Fagin *et al.*, 1995].

For each process i, we can define an equivalence relation $\sim_i$, as follows: $s \sim_i t$ iff all variables accessible to i have the same value in s and t. For the purpose of knowledge, the program counter of each process, which is used to remember which is the next computation step, is also considered an accessible variable for the corresponding process.

The definition of knowledge states that something is known if it is true in any reachable state that cannot be distinguished by the agent. In the next definition $\mathcal{I}$ is the set of runs defined by a PROMELA model, r is a run and s a state in r.

Definition 3 $(\mathcal{I}, r, s) \models K_i\phi$ *if and only if* $\forall t \in \mathcal{I}$ *with* $t \sim_i s : (\mathcal{I}, r, t) \models \phi$.

A very important question is how one can be sure that the epistemic relations in the PROMELA specification are exactly the same as in our formalisation of the Russian Cards Problem. We have constructed the PROMELA specification as a message passing system in which all messages are stored in variables accessible to all agents that have received the message [Fagin *et al.*, 1995]. A standard binary representation for every formula has been used, ensuring that no information is contained in the way formulas are written. The decisions regarding the task of each module, the representation of cards and the representation of statements are explained below. Once these decisions have been made the implementation is fairly straightforward, and this gives us good reason to believe that the specification we use indeed implements the protocol we want to check.

Modelling the Russian Cards Problem in PROMELA The specification that models the Russian cards problem can be retrieved by appending all code fragments in this paper. Nearly identical lines are sometimes replaced by "...". Alternatively, the specification can be downloaded from `www.csc.liv.ac.uk/~sieuwert/`. First the general approach and design decisions are discussed, then the detailed code is presented with comments.

A specification in SPIN consists of multiple processes that operate in parallel. Each agent in a multi agent system can be modelled as a separate process and this leads to the definition of three processes: `A()`, `B()` and `C()`. Every process `X()` has access to a variable x denoting its hands of cards. Based on their respective cards `A()` must make a statement and `B()` and `C()` must listen to and interpret the statement. That statement is stored in the variable `disj`, accessible to all three processes. Process `B()` moreover has a variable `ba` in which he stores his calculated hand of `A()`.

Another process is needed to generate a deal of cards, since no single agent can be said to be in control of the deal of cards. The process responsible for determining the deal of cards is called `deal()`. The next table shows all processes with their accessible variables.

process	**variables**
deal()	a,b,c
A()	a,disj
B()	b, ba, disj
C()	c,disj

In each run all processes perform the same steps but on a different deal of cards. The next table sums up what each process does. The labels `END` indicate the points in the execution of `B` and `C` where their knowledge must be tested. Naturally these labels are placed at the end of their specification since we are interested in the knowledge state after the announcement.

process	**actions**
`deal()`	set up a deal in `a`, `b` and `c` signal `A()`
`A()`	wait for signal from `deal()` choose ϕ and place in `disj` wait until ϕ is true signal `B()`, signal `C()`
`B()`	wait for signal from `A()` let `ba` be a hand from `disj` for which no card is both in `b` and `ba` `END:` do nothing
`C()`	wait for signal from `A()` `END:` do nothing

When choosing a representation or a data structure for a certain kind of information there is often a trade-off in readability of the program code and computational efficiency. With current model checkers only relatively small models can be checked. In most examples of protocols one abstracts away from the actual value of variables to achieve a model with a suitable number of states [Dams, 2002]. If one is interested in the exact contents of messages, for instance to see whether specific statements are safe in the Russian Cards problem, one cannot make this abstraction. Therefore we have opted for a binary representation that allows us to store hands of cards in single bytes. This keeps memory consumption low and also allows for a single instruction overlap tests by means of bitwise operators. The next macro's make use of the binary shift operator $\ll$ to generate the bit pattern of each card. The macro `overlap` returns true if its arguments have any card in common. `noverlap` is the negation of `overlap`.

```
#define card(j)       (1<<(j))
#define overlap(i,j)  (((i)&(j))!=0)
#define noverlap(x,y) (((x)&(y))==0)
```

All variables are declared to be global, since local variables are not sharable between processes and it is not possible to refer to them in LTL formulas.

```
byte disj[7];
byte a=0,b=0,c=0; ba=0;
```

The bytes `a,b` and `c` are used to store cards the cards of the players in. Only seven bits of these bytes are used. Synchronisation between the processes is done by the use of synchronous channels.

```
chan dealtoa=[0] of{byte};
chan atoc=[0] of{byte};
chan atob=[0] of{byte};
```

These channels are suitable for passing bytes. The values that are broadcast are not significant. Channel $x$to$y$ is used for synchronising between process x and process y: x sends a number and y will wait for that number. `Deal()` is the only process that does not have to wait for other processes so it is the first one to do something. It puts cards in the variables `a`, `b` and `c`. In the code below a card `x` is chosen and assigned to any player that has not received its maximal number of cards yet. The counters `na`,`nb` and `nc` are used to keep track of the number of cards each player has.

```
active proctype deal(){
 byte na=0,nb=0,nc=0; byte x;
 do
 ::na+nb+nc==7 ->break
 ::na+nb+nc<7 ->
  if
  ::noverlap(a|b|c,card(0))->x=card(0)
   ...
  ::noverlap(a|b|c,card(6))->x=card(6)
  fi;
  //select someone to give the card to.
  if
  :: na<3 -> a=a+x; na++
  :: nb<3 -> b=b+x; nb++
  :: nc<1 -> c=c+x; nc++
  fi
 od;
 dealtoa!a;
}
```

Process `A()` waits for a signal from process `deal()` and then it sets a statement in `disj`. The code shown writes the formula $((a_0 \wedge a_1 \wedge a_2) \vee (a_0 \wedge a_3 \wedge a_4) \vee (a_1 \wedge a_3 \wedge a_5) \vee (a_4 \wedge a_5 \wedge a_6) \vee (a_2 \wedge a_3 \wedge a_6))$. Only one formula ϕ can be tested at the same time. After writing the formula process `A()` waits until its cards are equal to one of the possibilities in the formula. The process will either continue immediately, or block forever. The reason for this is that the other processes do not alter `disj` (they are not doing anything because they waiting for signals). If the run ends here both `END` labels will never be reached, and because we refer to these labels in the properties later it means that the properties are only tested for runs in which `disj` is true, as intended. The reason that `A()` is only allowed to make one statement, and does not have statements covering all situations, is that the goal of this implementation is to verify statements. This verification is more efficient if the statements are tested separately.

```
active proctype A(){
 dealtoa?a;
 disj[0]=card(0)+card(1)+card(2);
 disj[1]=card(0)+card(3)+card(4);
 disj[2]=card(1)+card(3)+card(5);
 disj[3]=card(4)+card(5)+card(6);
 disj[4]=card(2)+card(3)+card(6);
 //only proceed if formula is true
 a==disj[0]||a==disj[1]||a==disj[2]||
 a==disj[3]||a==disj[4]||a==disj[5]||
 a==disj[6];
 atob!0;
 atoc!0;
}
```

Agent B waits until A signals that it has made a statement. Then, B will non-deterministically choose a hand of cards for agent A that is consistent with its own cards. After this choice it reaches the statement skip with the label END attached to it.

```
active proctype B(){
 atob?0;
 if
 ::disj[0]!=0&&noverlap(disj[0],b)
  -> ba=disj[0]
  ...
 ::disj[6]!=0&&noverlap(disj[6],b)
  -> ba=disj[6]
 fi;
 END:skip
}
```

Agent C waits for a signal from agent A and reaches its `END` state. There is nothing to do for this agent because the knowledge properties on this agent refer to the knowledge of this agent according to the given definition. The knowledge of for instance A's cards does not have to be explicitly available inside this process.

```
active proctype C(){
 atoc?0; //check if A is ready
 END:skip //here the properties must hold
}
```

The LTL formulas that are evaluated use propositions to express basic facts. These propositions are defined in the following macro's and correspond to the propositions in the logical formalisation of section 2. Two propositions about the labels in the program will be used to make the step from dynamic epistemic to temporal epistemic logic.

```
#define c_end C[3]@END
#define b_end B[2]@END
#define a_0 ((a&1)!=0)
#define a_1 ((a&2)!=0)
```

```
...
#define c_5 ((c&32)!=0)
#define c_6 ((c&64)!=0)
```

Capturing the Protocol Requirements We now consider the requirements that we wish to check. They are captured in Linear Temporal Logic, enriched by the knowledge modalities that we introduced earlier. The update operator used in the requirement is not available in the enriched LTL, but one can refer to the knowledge after the update by using temporal operators. The requirement 1 thus breaks down in three parts, φ, bknows and cig , which have to be tested at the end label of the corresponding process. On a system where a formula ϕ is passed as a message, one can replace an update formula $[\phi]\psi$ by an epistemic LTL formula $\Box(L \rightarrow \psi)$ where L is a label only reached after ϕ has been announced. This provides a translation from dynamic epistemic logic to epistemic LTL.

The truth of φ, appearing in equation 1, has been coded explicitly in the specification of process `A()`. Since it is clear from the specification that only runs in which ϕ is true terminate successfully we have not provided a separate LTL property to test this.

In the logical analysis much attention has been given to the question of what we should update with. Because of the definition of knowledge we are using, which is defined on the existing runs, this is much easier to handle in a model checking situation. Agent C learns that it is in one of the situations in which A can utter the formula, which seems a good way of modelling that A knows that φ meets A's intention to communicate using a solution: $K_A req_2(\varphi)$. In our implementation agent A uses a strategy in which it uses φ whenever true, so there is no actual difference between learning A's use of φ and just the announcement φ by an outsider.

The formula bknows is an example of a positive knowledge requirement, which often occurs between adversarial agents. Because an explicit procedure for obtaining the value has been supplied in the specification code, it is sufficient to test that this procedure assures that `B()`'s guess of `A()`'s card is correct. This is expressed in the following LTL formula, which we have used.

$$\Box(\text{at}_B(\text{END}) \rightarrow ba = a) \tag{4}$$

An interesting question is how close the correspondence between checking bknows using formula 4 and the general method [van der Hoek and Wooldridge, 2002] available for model checking is. The general method asks the user to find *local propositions* which correspond to the knowledge formula. The method can then prove the suitability of these local formulas. Instead of coming up with a local formula for each card x, describing the local states in which a_x is true, we have introduced a procedure for determining the value of all propositions. We check the suitability of that procedure instead of checking local propositions. The method is thus similar except that we do more of the work in the specification instead of in the LTL formulas. This variant on the local propositions method is in our opinion acceptable in situations where one does not start with a given formal specification, but only with an informal protocol description. The advantage of the above formula is that by working in the specification one can avoid having to mention all the cards explicitly. Checking one procedure is much more elegant than checking a local proposition for each card.

To see that the truth of bknows indeed follows from the truth of formula 4, assume that the model checker cannot find any counterexample to the formula. This shows that the variables `a` and `ab` are equal in each end state of process `B()`. We have not specified any propositions referring to the variable `ba`, but it is not unreasonable to assume a set similar to the propositions referring to A's cards: $ba_0, ba_1, \ldots ba_6$. Using these propositions we can express the model checking result in LTL logic extended with knowledge operators. The model checking result can be expressed as $\mathcal{I} \models \Box(\text{at}_B(\text{END}) \rightarrow (a_x \leftrightarrow ba_x))$.

Everything that is true in all states of every run in the system I is commonly known by all agents, so in particular it is known by process `B()` in all states in all runs: $\mathcal{I} \models \Box K_B(\text{at}_B(\text{END}) \rightarrow K_B(a_x \leftrightarrow ba_x))$. Because knowledge distributes over implication this gives $\mathcal{I} \models \Box(K_B\text{at}_B(\text{END}) \rightarrow (K_B a_x \leftrightarrow K_B ba_x))$. The variable ba is part of `B()`'s state, giving $\mathcal{I} \models \Box(ba_x \leftrightarrow K_B ba_x))$, and `B()` knows when it is in its end state, giving $\mathcal{I} \models \Box(\text{at}_B(\text{END}) \leftrightarrow K_B\text{at}_B(\text{END}))$. Since x was chosen arbitrarily, we have proven

$$\mathcal{I} \models \Box(\text{at}_B(\text{END}) \rightarrow \mathsf{bknows})$$

Checking the Absence of Knowledge So far, we have been concerned with establishing the *presence* of knowledge — that after the public announcement by A is made, B knows the deal. But we also need to show the *absence* of knowledge, i.e., that C does not know the deal after the announcement. This is more problematic than establishing the presence of knowledge. One can show that a certain piece of knowledge is not present by attempting to verify that the knowledge is present [van der Hoek and Wooldridge, 2002]. This verification will fail, and result in a counterexample. The counterexample shows that in a certain local state the statement ϕ is not necessarily true, so it cannot be known ($K_x\phi$ does not hold in state s).

But more often, one wants to prove the absence of knowledge in a set of states S instead of just a single state s. All states in the set can for example correspond to the same time point for a certain agent in the protocol, but have different values for the variables. A suitable definition of truth in a set of states is that the formula must be true in all members of the set: $(\mathcal{I}, S) \models \phi \quad \Leftrightarrow \quad \forall s \in S \; (\mathcal{I}, s) \models \phi$. The set S can for instance correspond to all global states in which agent x is at a certain label in its specification. In the above notation we implicitly quantify over all runs, using:

Definition 4 $(\mathcal{I}, s) \models \phi \quad \Leftrightarrow \quad \forall r (s \in r \Rightarrow \; (\mathcal{I}, r, s) \models \phi)$

One example of an absence of knowledge formula is $\neg K_C a_0$. This formula states that C does not know that A has card 0. The entire statement cig is a conjunction of 14 statements like this. The statement must hold in the end state of any run. Let S be the set of states in which C has reached their end state. One way to prove absence of knowledge of a formula is to show that there is at least one indistinguishable run in which the formula is not true. In the example, one can do this by letting SPIN verify that a_0 is true in all states in S. If the verification fails, a counterexample run r will result, containing a state s which is in S but in which a_0 is not true. This tells us $\neg\forall s \in S\ (\mathcal{I}, s) \models a_0$ One might call this *weak* absence of knowledge, because it shows one $s \in S$ in which a_0 is not true, i.e., in the local state of C corresponding to s the formula is thus not known. If agent C has a unique local state in all states of set S this is indeed a proof that C does not know a_0. But if C has more than one local state this is not a complete proof. It is not the same thing as $(\mathcal{I}, s) \models \neg K_C a_0$, which we might call *strong* absence of knowledge.

From the definition of knowledge in a set of states one might get the impression that one needs a counterexample for each member of the set. We obtain the following.

$$\begin{array}{c}(\mathcal{I}, S) \models \neg K_x\phi \quad \Leftrightarrow \\ \forall s \in S\ \forall r(s \in r \Rightarrow\ (\mathcal{I}, r, s) \models \neg K_x\phi)\end{array} \tag{5}$$

In formula 5 containing universal quantification one can do an expansion based on formula 6, which is derived from the definition of knowledge and negation.

$$\begin{array}{c}(\mathcal{I}, r, s) \models \neg K_x\phi \quad \Leftrightarrow \\ \exists s', r'(s' \in r' \wedge s' \sim_x s \wedge (\mathcal{I}, r', s') \models \neg\phi)\end{array} \tag{6}$$

We see that proving $\neg K_x\phi$ on a set of states S involves finding a counterexample (r', s') for each state s in S. However, if $s \sim_x t$ and both are in S, then the same r', s' is also a counterexample for t. This is true because $s' \sim_x s$ and $s \sim_x t$ implies $s' \sim_x t$. It is sufficient to find one counterexample (r', s') for each different local state of the agent x. The number of local states that agent x might have when the system state is in S will often be much smaller than S itself. Agent C for instance has only 7 different end states, one for each card.

The agent for which we want to prove absence of knowledge is agent C. The set of states S consists all the states in which agent C reaches its end state. On this set C is required to have no knowledge of all propositions a_i and b_i.

$$\mathsf{cig} = \bigwedge_{i \in \mathcal{D}} (\neg K_C a_i \wedge \neg K_C b_i)$$

SPIN is much slower at verifying large formulas, therefore we verify this requirement by verifying each of the fourteen conjuncts separately. In the following paragraphs the steps needed for verification of $\neg K_C a_0$ are explained.

The property must hold at the end of each run, when process `C()` is at the label `END`. At this label, C has seven different states, because it can have seven different cards. In each of these states a different proposition from the set $T = \{c_0, c_1, c_2, c_3, c_4, c_5, c_6\}$ is true. It could be the case that some of these local states are unreachable. For instance, if $\varphi = a_0$ then c_0 will always be false. If checking the formula $\Box\neg c_x$ does not result in a counterexample run, the state c_x should be removed from set T. In most cases, and also for the formula ϕ that is present in the program listing all of these local states are reachable.

For all reachable local states one must find a counterexample run, in which the agent is in the local state but the formula is not true. These counterexamples can be found by using SPIN to check and disprove the following formulas, for all $i \in \mathcal{D}$:

$$\Box((\text{at}_C(\text{END}) \wedge c_i) \rightarrow a_0)$$

Finding these seven counterexamples takes seven runs of SPIN and proves C's ignorance of the fact that A holds card 0. A complete proof of all fourteen conjuncts would require 98 runs of SPIN. The number of runs depends on the number of local states and one needs a set of propositions enumerating all local states in order to write down the requirement.

4 Conclusions

It is possible to use model checking to prove knowledge properties of protocol specification, even using a standard model checker such as SPIN. A translation from dynamic epistemic logic to LTL formulas can be made and the transformation is efficient when proving presence of knowledge. It is less efficient when proving absence of knowledge for a set of states, since the number of times one needs to run SPIN depends on the number of local states of the ignorant agent; and the number of states will of course be exponential in the number of bits that make up that agent's local state.

One important consequence is that *adversarial* multi agent systems are more difficult to model check, because establishing the absence of knowledge typically arises in the adversarial case. In *co-operative* agent systems one can sometimes get away with the easier to prove notion of *weak absence of knowledge*.

In future work, we intend to refine our techniques to develop a general methodology for model checking knowledge, and we intend to apply this methodology to larger and more realistic scenarios.

References

[Baltag *et al.*, 2002] A. Baltag, L.S. Moss, and S. Solecki. The logic of public announcements, common knowledge and private suspicions. Originally presented at TARK 98, accepted for publication in Annals of Pure and Applied Logic, 2002.

[Burrows *et al.*, 1990] M. Burrows, M. Abadi, and R. M. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8:18–36, 1990.

[Clarke *et al.*, 2000] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press: Cambridge, MA, 2000.

[Dams, 2002] Dennis Dams. Abstraction in software model checking: Principles and practice. In Dragan Bosnacki and Stefan Leue, editors, *SPIN*, volume 2318 of *Lecture Notes in Computer Science*. Springer, 2002.

[Dolev and Yao, 1998] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1998.

[Fagin *et al.*, 1995] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning about knowledge*. The MIT Press: Cambridge, MA, 1995.

[Holzmann, 1997] Gerard J Holzmann. The model checker Spin. *IEEE Trans. on Software Engineering*, 23:279–295, May 1997.

[Lowe, 1996] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FKR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACS)*, volume 1055 of *LNCS*, pages 147–166. Springer-Verlag, 1996.

[Meyer and van der Hoek, 1995] J.-J. Ch. Meyer and W. van der Hoek. *Epistemic Logic for AI and Computer Science*. Cambridge University Press: Cambridge, England, 1995.

[Needham and Schroeder, 1978] R. Needham and M. Schroeder. Using encryption for authenticationin large networks of computers. *Communcations of the ACM*, 21(12):993–999, 1978.

[van der Hoek and Wooldridge, 2002] W. van der Hoek and M. Wooldridge. Model checking knowledge and time. In D. Bošnački and S. Leue, editors, *Model Checking Software, Proceedings of SPIN 2002 (LNCS Volume 2318)*, pages 95–111. Springer-Verlag: Berlin, Germany, 2002.

[van der Meyden and Su, 2002] Ron van der Meyden and Kaile Su. Symbolic model checking the knowledge of the dining cryptographers. submitted, 2002.

[van Ditmarsch *et al.*, 2003] H. P. van Ditmarsch, W. van der Hoek, and B. Kooi. Concurrent epistemic dynamic logic. Accepted for Autonomous Agents and Multi Agent Systems, 2003.

[van Ditmarsch, 2003] H. P. van Ditmarsch. The russian cards problem: a case study in cryptography with public announcements, 2003. Accepted for Studia Logica.