Verification of multiagent systems via ordered binary decision diagrams: an algorithm and its implementation

Franco Raimondi, Alessio Lomuscio
Department of Computer Science
King's College London
London, UK
email: {franco,alessio}@dcs.kcl.ac.uk

Abstract

We investigate the problem of the verification of epistemic properties of multiagent systems via model checking. Specifically, we extend and adapt methods based on ordered binary decision diagrams, a mainstream verification technique in reactive systems. We provide an algorithm, and present a software package that implements it. We discuss the software and benchmark it by means of a standard example in the literature, the dining cryptographers.

1. Introduction

The field of multiagent systems (MAS) has seen considerable changes since its inception. In particular, a comparison of the workshops on agents of the beginning of the 90's such as MAAMAW and ATAL, through the AA and ICMAS conferences, to the current AAMAS series shows a clear shift of topics of research. Gone, for example, are the discussions on what is the *correct* definition of agency, and basic agent-based applications, and new themes such as automatic negotiation, agent communication languages, game theoretical aspects of MAS have appeared. This seems to suggest that agent-based computing is an active area of research that is getting closer in spirit to mainstream Computer Science, while still retaining its original AI inspiration.

The area of *multiagent systems theories* has always been central in this research, and it has witnessed a similar change of focus. While in the 90's the attention was firmly on the study of new logical theories that would account for crucial characteristics of agency such as knowledge, intentions, beliefs, goals, rules, etc., the focus these days seems to be more on the dynamic and temporal aspects of these logics, and on the integration of existing logical theories with software engineering aspects. The recent attention given by the community to the problem of *multiagent systems verification* can in our view be seen in this light.

The research area of verification and validation is, of course, a mainstream topic of research in traditional Software Engineering, where attention is given both to the broad topics of testing and formal verification. Formal verification encompasses two main approaches: theorem proving, and model checking. Theorem-proving-based approaches involve specifying a program by means of a formal logic system; the problem of checking whether the program in question displays a certain behaviour is translated into the problem of checking whether a particular formula, representing the property to be checked, is a theorem of the logic. One of the problems of the theorem-proving approach is that often the logics involved are computationally very demanding, hindering the feasibility of the approach.

More recently, model checking has been presented as an alternative to theorem proving in hardware and software verification. In this paradigm a system S is represented as a temporal model M_S by means of a program (representing S) written in a model-checking friendly language such as SMV. The property P to be checked is written as a formula φ_P in temporal logic. Checking whether system S satisfies property P amounts then to checking formally whether the model M_S satisfies the formula φ_P: $M \models \varphi_P$. The main problem with this approach is to manage the so called "state explosion problem", i.e., the fact that typically a system generates so many states (figures in the region of 10^{40} are possible even for relatively simple systems) that it is difficult to represent them. In attempt to solve this, *symbolic approaches* have been developed. In these, the temporal model is represented in a symbolic way by means of logical structures. The leading technique in this effort uses ordered binary decision diagrams OBDD[5]. Mainstream model checking packages such as VIS[1] and SMV[13] use this technique. Other techniques, notably SAT-based such as bounded model checking [3] and unbounded model checking [14], have proven to be very promising but have not been incorporated in most model checkers yet.

It is curious to note that while several proposal for model checking MAS have been put forward, none of them uses OBDD technology directly. In [23], M. Wooldridge et al. present the MABLE language for the specification of MAS. In this work, modalities are translated into nested data structures (in the spirit of [2]). Bordini et al. [4] use a modified version of the AgentSpeak(L) language [19] to specify agents and to exploit existing model checkers. Both the works of M. Wooldridge et al. and of Bordini et al. translate the specification into a SPIN specification to perform the verification. Effectively, the attitudes for the agents are reduced to predicates, and the verification involves only the temporal verification of those. In [18] a tool is provided to translate an interpreted system into SMV code, but the verification is limited to static epistemic properties, i.e. the temporal dimension is not present, and the approach is not fully symbolic. The works of van der Meyden and Shilov [21], and van der Meyden and Su [15], are concerned with verification of interpreted systems. They consider the verification of a particular class of interpreted systems, namely the class of synchronous distributed systems with perfect recall. An algorithm for model checking is introduced in the first paper using automata, and [15] suggests the use of OBDD's for this approach, but no algorithm or implementation is provided.

The research presented in this paper is an attempt to fill this gap by showing how it is possible to extend the mainstream verification technique in reactive systems — model checking via OBDD's — to verify key properties of MAS.

State-of-the-art MAS theories account for a variety of attitudes of the agents, such as their knowledge, beliefs, desires, as well as their temporal evolution. All these attitudes are expressed in MAS logics by means of modal operators [22]. OBDD-based techniques for reactive systems allow for the verification of properties expressed in plain temporal logic, either in its LTL or in CTL variants. This means that they cannot readily be employed for the verification of MAS, where richer formalisms are needed. In this paper we look at the case of knowledge, extend OBDD-based technology to verify temporal-epistemic properties of a MAS, treating both sets of operators on the same level. We carry out this investigation for the case of knowledge because of its traditional key importance in MAS, but clearly we would like to extend our approach to include other modalities of interest.

The paper is organised as follows. In Section 2, we introduce basic syntax and semantics of epistemic logic on interpreted systems as well as the basic definitions for model checking via ordered binary decision diagrams. In Section 3 we present a model checking algorithm for temporal epistemic logic based on OBDD. In Section 4 we present an implementation of this algorithm. In Section 5, we test the correctness and evaluate the performance of the implementation with a traditional example from the security literature — the dining cryptographers – that generates a state space in the region of 10^{14} states.

2. Preliminaries

In this section we introduce the main concepts and the notation that we are going to use in the rest of the paper. In Section 2.1 we review symbolic model checking using OBDD's. In Section 2.2 we present the formalism of interpreted systems for modelling temporal-epistemic properties of multi-agent systems.

2.1. Symbolic model checking and OBDD's

The problem of model checking can be defined as establishing whether or not a model M satisfies a formula φ ($M \models \varphi$). Though M could be a model for any logic, traditionally the problem of building tools to perform model checking automatically has been investigated almost only for *temporal* logics. This is because temporal logic has been identified for more than two decades as an adequate formalism to reason about properties of reactive systems Thus, to verify that a system complies with a certain property, one can represent the system by means of a (temporal) model M and the property by means of a (temporal) logic formula φ, and then check whether or not $M \models \varphi$.

There are various temporal logics that can be used to abstract reactive systems. Here we introduce CTL [12], a logic used to reason about the evolution of a system represented as a *branching* paths. Given a countable set of propositional variables $\mathcal{P} = \{p, q, \ldots\}$, CTL formulae are defined as follows:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\varphi \mid EG\varphi \mid E(\varphi U \varphi)$$

where the temporal operator X means in the next state, G means globally and U means until. Each temporal operator is pre-fixed by the existential quantifier E. Thus, for example, $EG(\varphi)$ means that "there exists a path in which φ is globally true". Traditionally, other operators are added to the syntax of CTL, namely AX, EF, AF, AG, AU (notice the "universal" quantifier A over paths, dual of E). These operators can be derived from the operators introduced here [12]. The semantics of CTL is given via a model $M = (S, R, \mathcal{V}, I)$ where $S = \{s_0, s_1, \ldots\}$ is a set of states, $R \subseteq S \times S$ is a binary relation, $\mathcal{V} : \mathcal{P} \to 2^S$ is an evaluation function, and $I \subseteq S$ is a set of initial states. A *path* π is a sequence of states $\pi = \{s_0, s_1, \ldots\}$ such that $s_0 \in I$ and $\forall i, (s_i, s_{i+1}) \in R$. A state s_i in a path π is denoted with π_i. Satisfaction in a state is defined inductively as follows:

$s \models p$	iff	$s \in \mathcal{V}(p)$,
$s \models EX\varphi$	iff	there exists a path π such that $\pi_i = s$ and $\pi_{i+1} \models \varphi$,
$s \models EG\varphi$	iff	there exists a path π such that $\pi_i = s$ and $\pi_{i+j} \models \varphi$ for all $j \geq 0$.
$s \models E(\varphi U \psi)$	iff	there exists a path π such that $\pi_i = s$ and a $k \geq 0$ such that $\pi_{i+k} \models \psi$ and $\pi_{i+j} \models \varphi$ for all $0 \leq j < k$.

It has been shown [8] that the problem of CTL model checking can be solved in time $O(|M| \times |\varphi|)$ where $|M| = |S| + |R|$. However, this bound assumes that the model M is given explicitly, which is not the case for real-life systems. Instead, the model M is usually built via a more succinct representation, for example using a dedicated programming language such as PROMELA[11] or SMV[13]. If this indirect description is considered, the size of the model grows exponentially with the number of variables in the program, rendering infeasible the explicit representation of the model, a difficulty often referred to as the *state explosion problem*. To try and overcome this issue, various techniques have been developed to perform *symbolic* model checking. In symbolic model checking, the algorithms operate on a symbolic representation of the model using automata [11], ordered binary decision diagrams (OBDD's, [5]), and other algebraic structures. By means of this techniques state-spaces of the region of 10^{50} have been verified.

For the purposes of this paper we consider CTL model checking using OBDD's, as presented in [9]. It has been shown that OBDD's are a compact representation for boolean functions, and that there are efficient algorithms to perform operations on OBDD's [5]. The key idea of CTL model checking using OBDD's is to represent states of the model, the temporal relation between states, and the evaluation function $\mathcal{V}$ by means of boolean formulae. This is done by encoding a state $s \in S$ as a boolean vector. Following this, set of states and relations between two states can be expressed by means of boolean formulae. All these boolean parameters are translated into OBDD's; verification is then conducted by performing operations on them (we refer to [9] for details).

This idea has been implemented successfully in a number of software tools such as SMV [13] and NuSMV [7]. Thanks to these tools, large systems have been checked, including hardware and software components.

2.2. Interpreted systems

An interpreted system [10] is a semantic structure to reason about temporal-epistemic properties of a system of agents. In this formalism, each agent i ($i = \{1, \ldots, n\}$) is characterised by a set of *local states* L_i and by a set of actions Act_i that may be performed. Actions are performed in compliance with a protocol $P_i : L_i \rightarrow 2^{Act_i}$ (notice that this definition allows for non-determinism). A tuple $g = (l_1, \ldots, l_n) \in L_1 \times \ldots \times L_n$, where $l_i \in L_i$ for each i, is called a *global state* and gives a snapshot of the system. Given a set I of *initial global states*, the evolution of the system is described by n evolution functions t_i (this definition is equivalent to the definition of a single evolution function t as in [10]): $t_i : L_1 \times \ldots \times L_n \times Act_1 \times \ldots \times Act_n \rightarrow L_i$. In this formalism, the environment in which agents "live" is usually modelled by means of a special agent E; we refer to [10] for more details. The set I, the evolution functions t_i, and the protocols P_i generate a set of *computations* (also called *runs*). A computation π is a sequence of global states $\pi = (g_0, g_1, \ldots)$ such that $g_0 \in I$ and, for each pair $(g_j, g_{j+1}) \in \pi$, there exists a set of actions a enabled by the protocols such that $t(g_j, a) = g_{j+1}$. The set $G \subseteq (L_1 \times \ldots \times L_n)$ denotes the set of *reachable* global states.

Interpreted systems semantics can be used to interpret formulae of a temporal language enriched with epistemic operators [10]. Here we assume a temporal tree structure to interpret CTLK formulae [16], an extension of CTL that includes epistemic modalities. The syntax of CTLK is defined in terms of a countable set of propositional variables $\mathcal{P} = \{p, q, \ldots\}$ and using the following modalities:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\varphi \mid EG\varphi \mid E(\varphi U \varphi) \mid K_i\varphi$$

The modalities AX, EF, AF, AG, AU are derived in the standard way. Further, given a set of agents Γ, two group modalities can be introduced: $E_\Gamma\varphi$ and $C_\Gamma\varphi$ denote, respectively, that every agent in the group knows φ, and that φ is *common knowledge* in the group (see [10] for details).

Given a valuation function $\mathcal{V} : \mathcal{P} \rightarrow 2^G$, satisfaction in a global state g is defined as follows:

$g \models p$	iff	$g \in \mathcal{V}(p)$,
$g \models \neg\varphi$	iff	$g \not\models \varphi$,
$g \models \varphi_1 \vee \varphi_2$	iff	$g \models \varphi_1$ or $g \models \varphi_2$,
$g \models EX\varphi$	iff	there exists a computation π such that $\pi_0 = g$ and $\pi_1 \models \varphi$,
$g \models EG\varphi$	iff	there exists a computation π such that $\pi_0 = g$ and $\pi_i \models \varphi$ for all $i \geq 0$,
$g \models E(\varphi U \psi)$	iff	there exists a computation π such that $\pi_0 = g$ and a $k \geq 0$ such that $\pi_k \models \psi$ and $\pi_i \models \varphi$ for all $0 \leq i < k$,
$g \models K_i\varphi$	iff	$\forall g' \in G$, $g \sim_i g'$ implies $g' \models \varphi$,
$g \models E_\Gamma\varphi$	iff	$\forall g' \in G$, $g \sim_\Gamma^E g'$ implies $g' \models \varphi$,
$g \models C_\Gamma\varphi$	iff	$\forall g' \in G$, $g \sim_\Gamma^C g'$ implies $g' \models \varphi$,

where π_j denotes the global state at place j in π. The relation $\sim_i$ is an epistemic accessibility relation for agent i defined by: $g \sim_i g'$ iff $l_i(g) = l_i(g')$, i.e. if the local state of agent i is the same in g and in g' (notice that this is an equivalence relation). The relation $g \sim_\Gamma^E g'$ between two global states holds iff $g \sim_i g'$ for some $i \in \Gamma$. The relation $\sim_\Gamma^C$ is the reflexive transitive closure of $\sim_\Gamma^E$.

3. Symbolic model checking of interpreted systems

In this section we present an algorithm based on OBDD's to verify temporal and epistemic properties of multi-agent systems, in the spirit of traditional model checking for temporal logics. Following the standard approach for model checking CTL formulae, we encode all the parameters needed by the algorithm by means of boolean functions (this is explained in Section 3.1). Verification of CTLK formulae is performed, using these parameters, by means of the algorithm presented in Section 3.2.

3.1. Translation into boolean formulae

We translate an interpreted system into a set of boolean formulae, starting with the local states of an agent; these can be encoded by means of boolean variables. Given $N = \sum_i nv(i)$, a global state can be identified by means of N boolean variables: $g = (v_1, \ldots, v_N)$. Similarly, given $M = \sum_i na(i)$, joint actions can be encoded by means of M boolean variables: $a = (a_1, \ldots, a_M)$. The evaluation function $\mathcal{V}$ associates a set of global states to each propositional atom, and so it can be translated into a boolean function. Also, the protocols, can be expressed as boolean functions relating local states and actions. The definition of t_i in Section 2.2 can be seen as specifying a list of *conditions* $c_{i,1}, \ldots, c_{i,k}$ under which agent i changes the value of its local state. Each $c_{i,j}$ has the form "if [conditions on global state and actions] then [value of "next" local state for i]". Hence, t_i is expressed as a boolean formula as follows: $t_i = (c_{i,1} \wedge \neg c_{i,2} \wedge \ldots \wedge \neg c_{i,k}) \vee (\neg c_{i,1} \wedge c_{i,2} \wedge \ldots \wedge \neg c_{i,2}) \vee \ldots \vee (\neg c_{i,1} \wedge \neg c_{i,2} \wedge \ldots \wedge c_{i,k})$, i.e. we impose that one and only one condition must hold. We assume that the last condition $c_{i,k}$ prescribes that, if none of the conditions on global states and actions in $c_{i,j} (j < k)$ is true, then the local state for i does not change. This assumption is key to keep compact the description of an interpreted system, as in this way only the conditions that are actually causing a change need to be listed.

The algorithm presented in Section 3.2 requires the definition of a boolean function $R_t(g, g')$ representing a temporal relation between g and g'. $R_t(g, g')$ can be obtained from the evolution function t_i as follows. First, we introduce a *global* evolution function t:

$$t = \bigwedge_{i \in \{1,\ldots,n\}} t_i$$

Notice that t is a boolean function involving two global states, by means of their local states components, and joint actions $a = (a_1, \ldots, a_M)$. To abstract from joint actions $a \in Act = Act_1 \times \ldots \times Act_n$, and obtain a boolean function relating two global states only, we can define R_t as follows: $R_t(g, g')$ iff $\exists a \in Act : t(g, a, g')$ is true and each local action $a_i \in a$ is enabled by the protocol of agent i in the local state $l_i(g)$. The quantification over actions can be translated into a propositional formula using a disjunction (see [13, 9] for a similar approach to boolean quantification):

$$R_t(g, g') = \bigvee_{a \in Act} [(t(g, a, g') \wedge P(g, a)]$$

where $P(g, a)$ is a boolean formula imposing that the joint action a must be consistent with the agents' protocols in global state g. The formula R_t above gives the desired boolean relation between global states.

3.2. The algorithm

In this section we present the algorithm SAT_{CTLK} to compute the set of global states in which a CTLK formula φ holds. The following are the parameters needed by the algorithm:

- the boolean variables $(v_1, \ldots, v_N)$ and $(a_1, \ldots, a_M)$ to encode global states and joint actions;
- the boolean functions $P_i(v_1, \ldots, v_N, a_1, \ldots, a_M)$ to encode the protocols of the agents;
- the boolean function R_t to encode the temporal transition;
- the function $\mathcal{V}(p)$ returning the set of global states in which the atomic proposition p holds. We assume that the global states are returned encoded as a boolean function of $(v_1, \ldots, v_N)$;
- the set of initial states I, encoded as a boolean function;
- the set of reachable states G. This can be computed as the fix-point of the operator $\tau = (I(g) \vee \exists g'(R_t(g', g) \wedge Q(g'))$ where $I(g)$ is true if g is an initial state and Q denotes a set of global states. The fix-point of τ can be computed by iterating $\tau(\emptyset)$ by standard procedure (see [13]);
- the boolean functions R_i to encode the accessibility relations $\sim_i$ (these functions are easily defined using equivalence on local states of G);
- the boolean function R^E_Γ to encode $\sim^E_\Gamma$, defined by $R^E_\Gamma = \bigwedge_{i \in \Gamma} R_i$.

The algorithm is as follows:

```
$SAT_{CTLK}(\varphi)$ {
  $\varphi$ is an atomic formula: return $\mathcal{V}(\varphi)$;
  $\varphi$ is $\neg\varphi_1$: return $G \setminus SAT_{CTLK}(\varphi_1)$;
  $\varphi$ is $\varphi_1 \wedge \varphi_2$: return $SAT_{CTLK}(\varphi_1) \cap$
    $SAT_{CTLK}(\varphi_2)$;
  $\varphi$ is $EX\varphi_1$: return $EX_{CTLK}(\varphi_1)$;
  $\varphi$ is $E(\varphi_1 U \varphi_2)$: return $EU_{CTLK}(\varphi_1, \varphi_2)$;
  $\varphi$ is $EG\varphi_1$: return $EG_{CTLK}(\varphi_1)$;
  $\varphi$ is $K_i\varphi_1$: return $K_{CTLK}(\varphi_1, i)$;
  $\varphi$ is $E_\Gamma\varphi_1$: return $E_{CTLK}(\varphi_1, \Gamma)$;
  $\varphi$ is $C_\Gamma\varphi_1$: return $C_{CTLK}(\varphi_1, \Gamma)$;
}
```

In the algorithm above, EX_{CTLK}, EG_{CTLK}, EU_{CTLK} are the standard procedures for CTL model checking [12], in which the temporal relation is R_t and, instead of temporal states, global states are considered. The procedures $K_{CTLK}(\varphi, i)$, $E_{CTLK}(\varphi, \Gamma)$ and $C_{CTLK}(\varphi, \Gamma)$ are presented below.

```
$K_{CTLK}(\varphi, i)$ {
  X = $SAT_{CTLK}(\neg\varphi)$;
  Y = $\{g \in G | R_i(g, g')$ and $g' \in X\}$
  return ¬Y;
}
```

```
$E_{CTLK}(\varphi, \Gamma)$ {
  X = $SAT_{CTLK}(\neg\varphi)$;
  Y = $\{g \in G | R^E_\Gamma(g, g')$ and $g' \in X\}$
  return ¬Y;
}
```

```
$C_{CTLK}(\varphi, \Gamma)$ {
  X = $SAT_{CTLK}(\varphi)$;
  Y = $G$;
  while ( X != Y ) {
    X = Y;
    Y = $\{g \in G | R^E_\Gamma(g, g')$ and $g' \in Y$ and $g' \in SAT_{CTLK}(\varphi)\}$
  return Y;
}
```

The procedure $C_{CTLK}(\varphi, \Gamma)$ is based on the equivalence [10] $C_\Gamma\varphi = E_\Gamma(\varphi \wedge C_\Gamma\varphi)$ which implies that the set of states satisfying $C_\Gamma\varphi$, denoted with $[[C_\Gamma\varphi]]$, is the greatest fix-point of the (monotonic) operator $\tau(Q) = [[E_\Gamma(\varphi \wedge (Q))]]$. Hence, $[[C_\Gamma\varphi]]$ can be obtained by iterating $\tau(G)$. Notice that all the parameters can be encoded as OBDD's. Moreover, all the operations inside the algorithms can be performed on OBDD's.

The algorithm presented here computes the set of states in which a formula holds, but we are usually interested in checking whether or not a formula holds in the whole model. SAT_{CTLK} can be used to verify whether or not a formula φ holds in a model by comparing two set of states: the set $SAT_{CTLK}(\varphi)$ and the set of reachable states G. As sets of states are expressed as OBDD's, verification in a model is reduced to the comparison of the two OBDD's for $SAT_{CTLK}(\varphi)$ and for G.

4. Implementation

In this section we present an implementation of the algorithm introduced in Section 3. In Section 4.1 we define a language to encode interpreted systems symbolically, while in Section 4.2 we describe how the language is translated into OBDD's as well as the structure of the algorithm.

The implementation is available for download[17].

4.1. How to define an interpreted system

To define an interpreted system it is necessary to specify all the parameters presented in Section 2.2. In other words, for each agent, we need to represent:

- a list of local states;
- a list of actions;
- a protocol for the agent;
- an evolution function for the agent.

In our implementation, the parameters listed above are provided via a text file. The formal syntax of a text file specifying a list of agents is as follows:

```
agentlist ::= agentdef |
              agentlist agentdef
agentdef ::= "Agent" ID
                 LstateDef;
                 ActionDef;
                 ProtocolDef;
                 EvolutionDef;
             "end Agent"
LstateDef ::= "Lstate = {" IDLIST "}"
ActionDef ::= "Action = {" IDLIST "}"
ProtocolDef ::= "Protocol"
                 ID ": {" IDLIST "}";
                 ...
               "end Protocol"
EvolutionDef ::= "Ev:"
                  ID "if" BOOLEANCOND;
                  ...
                 "end Ev"
IDLIST ::= ID |
           IDLIST "," ID
ID ::= [a-zA-Z][a-zA-Z0-9_]*
```

In the definition above, `BOOLEANCOND` is a string expressing a boolean condition; we omit its description here and we refer to the source code for more details. To complete the specification of an interpreted system, it is also necessary to define the following parameters:

- an evaluation function;
- a set of initial states (expressed as a boolean condition);
- optionally, a set of groups for group modalities

The syntax for this set of parameters is as follows:

```
EvaluationDef ::= "Evaluation"
                  ID "if" BOOLEANCOND;
                  ...
                 "end Evaluation"
InitstatesDef ::= "InitStates"
                  BOOLEANCOND;
                 "end InitStates"
GroupDef ::= "Groups"
```

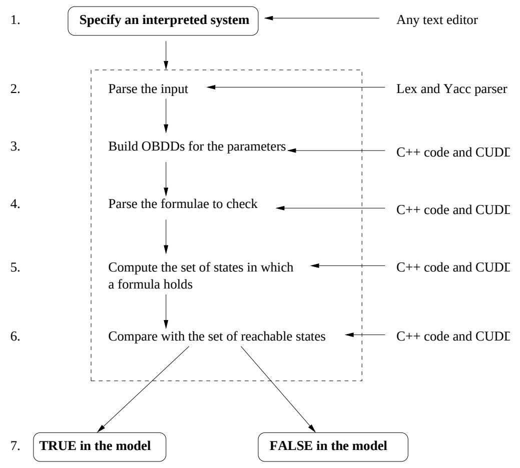
1.
Specify an interpreted system
Any text editor
2.
Parse the input
Lex and Yacc parser
3.
Build OBDDs for the parameters
C++ code and CUDD
4.
Parse the formulae to check
C++ code and CUDD
5.
Compute the set of states in which a formula holds
C++ code and CUDD
6.
Compare with the set of reachable states
C++ code and CUDD
7.
TRUE in the model
FALSE in the model

Figure 1. Software structure

```
          ID " = {" IDLIST " }";
          ...
      "end Groups"
```

Due to space limitations we refer to the files available online for a full example of specification of an interpreted system.

Formulae to be checked are specified using the following syntax

```
formula ::= ID |
            formula "AND" formula |
            "NOT" formula |
            "EX(" formula ")" |
            "EG(" formula ")" |
            "E(" formula "U" formula ")" |
            "K(" ID "," formula ")" |
            "GK(" ID "," formula ")" |
            "GCK(" ID "," formula ")"
```

In the syntax above we denote the operator for "everybody in a group knows" with `GK` (group knowledge) and the operator for common knowledge with `GCK` (group common knowledge). `K` denotes knowledge of the agent identified by the string `ID`. The remaining temporal operators are defined in a similar way. Notice that this corresponds to the full CTLK language.

4.2. Implementation of the algorithm

The steps from 2 to 6, inside the dashed box, are performed automatically upon invocation of the tool. These steps are coded mainly in C++ and can be summarised as follows:

- In step 2, the input file is parsed using the standard tools Lex and Yacc. In this step various parameters are stored in temporary lists; such parameters include agents' names, local states, actions, protocols, etc.
- In step 3, the lists obtained in step 2 are traversed to build the OBDD's for the verification algorithm. OBDD's are created and manipulated using the CUDD library [20]. In this step the number of variables needed to represent local states and actions are computed; following this, all the OBDD's are built by translating the boolean formulae for protocols, evolution functions, evaluation, etc. Also, the set of reachable states is computed using the operator presented in Section 3.2.
- In steps 4, the formulae to check are read from a text file, and parsed.
- In step 5, verification is performed by implementing the algorithm of Section 3.2. At the end step 5, an OBDD representing the set of states in which a formula holds is computed.
- In step 6, the set of reachable states is compared with the OBDD corresponding to each formula. If they are equivalent, the formula holds in the model and the tool produces a positive output. Otherwise, the tool produces a negative output.

5. Examples and experimental results

In this section we test our tool by model-checking temporal-epistemic properties of a communication scenario: the protocol of the dining cryptographers. In [6] it is shown that there exists a protocol that allows for the change in the knowledge of the participants about some global property of the system, without them being able to detect the source of this information. In Section 5.1 we describe how the example can be modelled by means of an interpreted system. In Section 5.2 we provide an evaluation of the performance of our tool on this example.

5.1. The interpreted system of the dining cryptographers

The protocol of the dining cryptographers is introduced in [6] with the following example:

"*Three cryptographers are sitting down to dinner at their favourite three-star 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 neighbour 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.*"[6]

Notice that the same protocol works for any number of cryptographers greater or equal to three (see [6]).

We introduce three agents C_i $(i = \{1, 2, 3\})$ to model the three cryptographers, and one agent E for the environment. In our representation the environment is used to select non-deterministically the identity of the payer and the results the of coin tosses. This makes a total of 32 possible local states for the environment We assume that the environment can perform only one action, the null action. Therefore, the protocol is simply mapping every local state to the null action. Also, there is no evolution of the local states for the environment. We model the local states of the cryptographers as a string containing three parameters: whether or not the coins that a cryptographer can see are equal, whether or not the cryptographer is the payer, and the number of "different" utterances. Considering that all these parameters are not initialised at the beginning of the run, there are 27 possible combinations of these, hence 27 possible local states are required. For each cryptographer, the actions allowed are "say nothing", "say equal", "say different", and these actions are performed in compliance with the protocol stated above. We refer to the code for the details of the protocol and of the evolution function.

We define the following set of atomic propositions (g is a global state):

$$\begin{array}{lll} g \models \mathbf{paid_1} & \text{if} & l_{C_1}(g) = \langle *\text{Paid}* \rangle \\ g \models \mathbf{paid_2} & \text{if} & l_{C_2}(g) = \langle *\text{Paid}* \rangle \\ g \models \mathbf{paid_3} & \text{if} & l_{C_2}(g) = \langle *\text{Paid}* \rangle \\ g \models \mathbf{even} & \text{if} & l_{C_i}(g) = \langle *\text{Even}* \rangle \text{ for every } i \\ g \models \mathbf{odd} & \text{if} & l_{C_i}(g) = \langle *\text{Odd}* \rangle \text{ for every } i \end{array}$$

$\langle *\text{Paid}* \rangle$ denotes a local state in which the string contains the value Paid (i.e. the cryptographer paid for the dinner). $\langle *\text{Even}* \rangle$ and $\langle *\text{Odd}* \rangle$ are defined similarly. We can now express formally various properties of this interpreted system. For example:

$$\begin{array}{c} \models AG(\mathbf{odd} \rightarrow (\neg \mathbf{paid_1} \rightarrow (K_{C_1}(\mathbf{paid_2} \vee \mathbf{paid_3}) \\ \wedge \\ \neg K_{C_1}(\mathbf{paid_2}) \wedge \neg K_{C_1}(\mathbf{paid_3})))) \end{array}$$

This formula expresses the claim made at the beginning of this section: if the first cryptographer did not pay for the dinner and there is an odd number of differences in the utterances, then the first cryptographer knows that either the second or the third cryptographer paid for the dinner; moreover, in this case, the first cryptographer does not know which one of the remaining cryptographers is the payer.

Analogously, it is possible to check that, if a cryptographer paid for the dinner, then there will be an odd number of "different" utterances, that is:

$$\models (\mathbf{paid_1} \vee \mathbf{paid_2} \vee \mathbf{paid_3}) \rightarrow AF(\mathbf{odd})$$

Consider now the group Γ of the three cryptographers. An interesting property is the following:

$$\models \mathbf{even} \rightarrow C_\Gamma(\neg \mathbf{paid_1} \wedge \neg \mathbf{paid_2} \wedge \neg \mathbf{paid_3})$$

This formula expresses the fact that, in presence of an even number of "different" utterances, it is common knowledge that none of the cryptographers paid for the dinner. Hence, in this protocol common knowledge can be achieved. All these formulae were correctly verified by the tool.

5.2. Experimental results

We have encoded the interpreted system introduced in the previous section by means of the language defined in Section 4.1 (a copy of the code is included in the downlodable files). In this section we evaluate some experimental results.

First, we define the size of the interpreted system. In Section 2.1 the size of a model M has been defined as $|M| = |S| + |R|$, where S is the set of states and R is the temporal relation. Here we define $|S|$ as the number all the possible combinations of local states and actions. For the example in Section 5.1, there are 32 local states for the environment, 27 local states and 3 actions for each cryptographer; hence, $|S| \approx 1.7 \cdot 10^7$. To define $|R|$ we must take into account that, besides the temporal relation, there are also the epistemic relations. Hence, we define $|R|$ as the sum of the sizes of temporal and epistemic relations. We approximate $|R|$ as $|S|^2$, hence $|M| = |S| + |R| \approx |S|^2 \approx 2.9 \cdot 10^{14}$.

To evaluate the performance of the tool, we consider the running time and the memory requirements. The running time is the sum of the time required for building all the OBDD's for the parameters and the actual running time for the verification. To quantify the memory requirements we consider the maximum number of nodes allocated for OBDD's. Notice that this figure over-estimates the number of nodes required to encode the state space and the relations. Also, we report the total memory used by the tool (in MBytes). We ran the tool on a 1.2 GHz AMD Athlon with 256 MBytes of RAM, running Debian Linux with kernel 2.4.20. The average experimental results are reported in Tables 1 and 2. We tested the formulae presented in Section—5.1 (more tests can be found in [17]); they were all correctly verified All the formulae require a similar amounts of memory. The required time for the construction of OBDD's is fixed (32 sec); the verification time ranges from 1.5 sec for small formulae to 3.5 sec for formulae involving nested modalities.

$\lvert M \rvert$	OBDD's nodes	Memory (MBytes)
$\approx 2.9 \cdot 10^{14}$	$7.6 \cdot 10^{6}$	152

Table 1. Memory requirements.

Model construction	Verification	Total
32sec	2.5sec	34.5

Table 2. Running time (for one formula).

We see these as very encouraging results. We have been able to check formulae with temporal and epistemic modalities in a few seconds on a standard PC, for a fairly large model. Moreover, our implementation does not include any optimisation technique [9]. Therefore, we estimate that our tool could perform well even in bigger scenarios. For the same reason, we estimate that it is possible to include other modal operators, besides the temporal and epistemic ones.

6. Conclusion

After years of research in the area of *specification* of MAS, interest is growing in the area of *formal verification* of MAS. In this paper we have extended what can be regarded as being the mainstream technique for formal verification of reactive systems, i.e., symbolic model checking via OBDD. In particular, the techniques and the implementation presented here allow for the verification of temporal-epistemic properties of MAS. The experimental results that we reported encourage us to optimise both the algorithm and the implementation, and to explore the feasibility of extending the framework to other operators.

References

[1] VIS - a system for verification and synthesis. *Lecture Notes in Computer Science*, 1102:428–432, 1996.

[2] M. Benerecetti, F. Giunchiglia, and L. Serafini. Model checking multiagent systems. *Journal of Logic and Computation*, 8(3):401–423, June 1998.

[3] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99), number 1579 in LNCS*, 1999.

[4] R. H. Bordini, M. Fisher, C. Pardavila, and M. Wooldridge. Model checking AgentSpeak. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'03)*, July 2003.

[5] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, pages 677–691, August 1986.

[6] D. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1:65–75, 1988.

[7] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. *Lecture Notes in Computer Science*, 1633, 1999.

[8] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[9] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

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

[11] G. J. Holzmann. The model checker spin. *IEEE transaction on software engineering*, 23(5), May 1997.

[12] M. R. A. Huth and M. D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, England, 2000.

[13] K. L. McMillan. *Symbolic model checking: An approach to the state explosion problem*. Kluwer Academic Publishers, 1993.

[14] K. L. McMillan. Applying SAT methods in unbounded symbolic model checking. *Lecture Notes in Computer Science*, 2404:250–260, 2002.

[15] R. van der Meyden and Kaile Su. Symbolic model checking the knowledge of the dining cryptographers. Submitted, 2002.

[16] W. Penczek and A. Lomuscio. Verifying epistemic properties of multi-agent systems via model checking. *Fundamenta Informaticae*, 55(2):167–185, 2003.

[17] F. Raimondi and A. Lomuscio. A tool for verification of interpreted systems. http://www.dcs.kcl.ac.uk/pg/franco/mcis/.

[18] F. Raimondi and A. Lomuscio. A tool for specification and verification of epistemic and temporal properties of multi-agent system. *Electronic Lecture Notes of Theoretical Computer Science*, 2003. To Appear.

[19] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. *Lecture Notes in Computer Science*, 1038:42–52, 1996.

[20] F. Somenzi. CU Decision Diagram Package - Release 2.3.1. http://vlsi.colorado.edu/∼fabio/CUDD/cuddIntro.html.

[21] R. van der Meyden and N. V. Shilov. Model checking knowledge and time in systems with perfect recall. *FSTTCS: Foundations of Software Technology and Theoretical Computer Science*, 19, 1999.

[22] M. Wooldridge. *Reasoning about Rational Agents*. Intelligent Robots and Autonomous Agents. The MIT Press, Cambridge, Massachusetts, 2000.

[23] M. Wooldridge, M. Fisher, M.P. Huget, and S. Parsons. Model checking multi-agent systems with MABLE. In M. Gini, T. Ishida, C. Castelfranchi, and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'02)*, pages 952–959. ACM Press, July 2002.