Automatic verification of deontic properties of multi-agent systems

Franco Raimondi and Alessio Lomuscio

Department of Computer Science
King's College London
London, UK
{franco,alessio}@dcs.kcl.ac.uk

Abstract. We present an algorithm and its implementation for the verification of correct behaviour and epistemic states in multiagent systems. The verification is performed via model checking techniques based on OBDD's. We test our implementation by means of a communication example: the bit transmission problem with faults.

1 Introduction

In the last two decades, the paradigm of multiagent systems (MAS) has been employed successfully in several fields, including, for example, philosophy, economics, and software engineering. One of the reasons for the use of MAS formalism in such different fields is the usefulness of ascribing autonomous and social behaviour to the components of a system of agents. This allows to *abstract* from the details of the components, and to focus on the *interaction* among the various agents.

Besides *abstracting* and *specifying* the behaviour of a complex system by means of MAS formalisms based on logic, recently researchers have been concerned with the problem of *verifying* MAS, i.e., with the problem of certifying formally that a MAS satisfies its specification.

Formal verification has its roots in software engineering, where it is used to verify whether or not a system behaves as it is supposed to. One of the most successful formal approaches to verification is *model checking*. In this approach, the system S to be verified is represented by means of a logical model M_S representing the computational traces of the system, and the property P to be checked is expressed via a logical formula φ_P. Verification via model checking is defined as the problem of establishing whether or not $M_S \models \varphi_P$. Various tools have been built to perform this task automatically, and many real-life scenarios have been tested.

Unfortunately, extending model checking techniques for the verification of MAS does not seem to be an easy task. This is because model checking tools consider standard reactive systems, and do not allow for the representation of the social interaction and the autonomous behaviour of agents. Specifically, traditional model checking tools assume that M is "simply" a *temporal* model, while MAS need more complex formalisms. Typically, in MAS we want to reason about epistemic, deontic, and doxastic properties of agents, and their temporal evolution. Hence, the logical models required are richer than the temporal model used in traditional model checking.

Various ideas have been put forward to verify MAS. In [20], M. Wooldridge et al. present the MABLE language for the specification of MAS. In this work, non-temporal modalities are translated into nested data structures (in the spirit of [1]). Bordini et al. [2] use a modified version of the AgentSpeak(L) language [18] to specify agents and to exploit existing model checkers. Both the works of M. Wooldridge et al. and of Bordini et al. translate the MAS specification into a SPIN specification to perform the verification. In this line, the attitudes for the agents are reduced to predicates, and the verification involves only the temporal verification of those. In [8] a methodology is provided to translate a deontic interpreted system into SMV code, but the verification is limited to static deontic and 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 [12], and van der Meyden and Su [13], are concerned with the verification of temporal and epistemic properties of MAS. They consider a particular class of interpreted systems: synchronous distributed systems with perfect recall. An automata-based algorithm for model checking is introduced in the first paper using automata. In [13] an example is presented, and [13] suggests the use of OBDD's for this approach, but no algorithm or implementation is provided.

In this paper we introduce an algorithm to model check MAS via OBDD's. In particular, in this work we investigate the verification of epistemic properties of MAS, and the verification of the "correct" behaviour of agents.

Knowledge is a fundamental property of the agents, and it has been used for decades as key concept to reason about systems[5]. In complex systems, reasoning about the "correct" behaviour is also crucial. As an example, consider a client-server interaction in which a server fails to respond as quickly as it is supposed to a client's requests. This is an unwanted behaviour that may, in certain circumstances, crash the client. It has been shown[14] that correct behaviour can be represented by means of deontic concepts: as we show in this paper, model checking deontic properties can help in establishing the extent in which a system can cope with failures. We give an example of this in Section 5.2, where two possible "faulty" behaviours are considered in the bit transmission problem[5], and key properties of the agents are analysed under these assumptions. In one case, the incorrect behaviour does not cause the whole system to fail; in the second case, the incorrect behaviour invalidates required properties of the system. We use this as a test example, but we feel that similar situations can arise in many areas, including database management, distributed applications, communication scenarios, etc.

The rest of the paper is organised as follows. In Section 2 we review the formalism of deontic interpreted systems and model checking via OBDD's. In Section 3 we introduce an algorithm for the verification of deontic interpreted systems. An implementation of the algorithm is then discussed in Section 4. In Section 5 we test our implementation by means of an example: the bit transmission problem with faults. We conclude in Section 6.

2 Preliminaries

In this section we introduce the formalisms and the notation used in the rest of the paper. In Section 2.1 we review briefly the formalism of interpreted systems as presented in [5] to model a MAS, and its extension to reason about the correct behaviour of some of the agents as presented in [9]. In Section 2.2 we review some model checking methodologies.

2.1 Deontic interpreted systems and their temporal extension

An interpreted system [5] is a semantic structure representing a system of agents. Each agent in the system i ($i \in \{1, \ldots, n\}$) 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 $g = (l_1, \ldots, l_n) \in L_1 \times \ldots, L_n$, where $l_i \in L_i$ for each i, is called a *global state* and gives a description of the system at a particular instance of time. 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 [5]): $t_i : L_1 \times \ldots \times L_n \times Act_1 \times \ldots \times Act_n \rightarrow L_i$. In this formalism, the environment in which agents "live" is usually modelled by means of a special agent E; we refer to [5] for more details. The set I, the functions t_i, and the protocols P_i generate a set of *computations* (also called *runs*). Formally, 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 $t(g_j, a) = g_{j+1}$. $G \subseteq (L_1 \times \ldots \times L_n)$ denotes the set of *reachable* global states.

In [9] the notion of *correct behaviour* of the agents is incorporated in this formalism. This is done by dividing the set of local states into two disjoint sets: a non-empty set G_i of allowed (or "green") states, and a set R_i of disallowed (or faulty, or "red") states, such that $L_i = G_i \cup R_i$, and $G_i \cap R_i = \emptyset$. Given a countable set of propositional variables $\mathcal{P} = \{p, q, \ldots\}$ and a valuation function for the atoms $\mathcal{V} : \mathcal{P} \rightarrow 2^G$, a deontic interpreted systems is a tuple $DIS = (G, \{\sim_i\}_{i \in \{1,\ldots,n\}}, \{\prec_i^O\}_{i \in \{1,\ldots,n\}}, \mathcal{V})$. The relations $\sim_i$ are epistemic accessibility relations defined for each agent i 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 relations $\prec_i^O$ are accessibility relations defined by $g \prec_i^O g'$ iff $l_i(g') \in G_i$, i.e. if the local state of i in g' is a "green" state. We refer to [9] for more details. Deontic interpreted systems can be used to evaluate formulae involving various modal operators. Besides the standard boolean connectives, the language considered in [9] includes:

- A deontic operator $O_i \varphi$, denoting the fact that *under all the correct alternatives for agent i, φ holds*.
- An epistemic operator $K_i \varphi$, whose meaning is *agent i knows φ*.
- A particular form of knowledge $\widehat{K}_i^j \varphi$ denoting the knowledge about a fact φ that an agent i has *on the assumption that agent j is functioning correctly*.

We extend this language by introducing the following temporal operators: $EX(\varphi), EG(\varphi), E(\varphi U \psi)$. Formally, the language we use is defined as follows:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\varphi \mid EG\varphi \mid E(\varphi U \varphi) \mid K_i(\varphi) \mid O_i(\varphi) \mid \widehat{K}_i^j(\varphi)$$

We now define the semantics for this language. Given a deontic interpreted system DIS, a global state g, and a formula φ, satisfaction is defined as follows:

$DIS, g \models p$ iff $g \in \mathcal{V}(p)$,
$DIS, g \models \neg\varphi$ iff $g \not\models \varphi$,
$DIS, g \models \varphi_1 \vee \varphi_2$ iff $g \models \varphi_1$ or $g \models \varphi_2$,
$DIS, g \models EX\varphi$ iff there exists a computation π such that $\pi_0 = g$ and $\pi_1 \models \varphi$,
$DIS, g \models EG\varphi$ iff there exists a computation π such that $\pi_0 = g$ and $\pi_i \models \varphi$ for all $i \geq 0$.
$DIS, g \models E(\varphi U \psi)$ 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 $0 \leq i < k$,
$DIS, g \models K_i\varphi$ iff $\forall g' \in G$, $g \sim_i g'$ implies $g' \models \varphi$
$DIS, g \models O_i\varphi$ iff $\forall g' \in G$, $g \prec_i^O g'$ implies $g' \models \varphi$
$DIS, g \models \widehat{K}_i^j \varphi$ iff $\forall g' \in G$, $g \sim_i g'$ and $g \prec_j^O g'$ implies $g' \models \varphi$

In the definition above, π_j denotes the global state at place j in computation π. Other temporal modalities can be derived, namely AX, EF, AF, AG, AU. We refer to [5, 9, 15] for more details.

2.2 Model checking techniques

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 [4, 7].

The model M is usually represented by means of a dedicated programming language, such as PROMELA[6] or SMV[11]. The verification step avoids building the model M explicitly from the program; instead, various techniques have been investigated to perform a *symbolic* representation of the model and the parameters needed by verification algorithms. Such techniques are based on automata [6], *ordered binary decision diagrams* (OBDD's, [3]), or other algebraic structures. These approaches are often referred to as *symbolic model checking* techniques. For the purpose of this paper, we review briefly symbolic model checking using OBDD's.

It has been shown that OBDD's offer a compact representation of boolean functions. As an example, consider the boolean function $a \wedge (b \vee c)$. The truth table of this function would be 8 lines long. Equivalently, one can evaluate the truth value of this function by representing the function as a directed graph, as exemplified on the left-hand side of Figure 1 . As it is clear from the picture, under certain assumptions, this graph can be simplified into the graph pictured on the right-hand side of Figure 1. This "reduced" representation is called the OBDD of the boolean function. Besides offering a compact representation of boolean functions, OBDD's of different functions can be composed efficiently. We refer to [3, 11] for more details.

The key idea of model checking temporal logics using OBDD's is to represent the model M and all the parameters needed by the algorithms by means of boolean functions. These boolean functions can then be encoded as OBDD's, and the verification step can operate directly on these. The verification is performed using fix-point characterisation of the temporal logics operators. We refer to [7] for more details. Using this technique, systems with a state space in the region of 10^{40} have been verified.

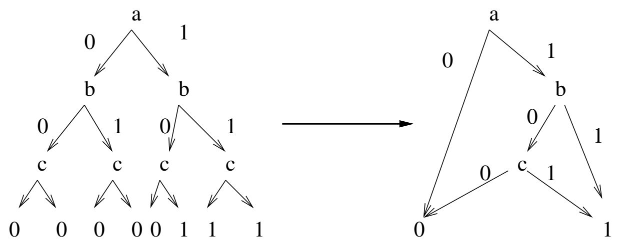

Fig. 1. OBDD representation for $a \wedge (b \vee c)$.

3 Model checking deontic properties of interpreted systems

In this section we present an algorithm for the verification of deontic, epistemic, and temporal modalities of MAS, extending with deontic modalities the work that appeared in [17]. Our approach is similar, in spirit, to the traditional model checking techniques for the logic CTL. Indeed, we start in Section 3.1 by representing the various parameters of the system by means of boolean formulae. In Section 3.2, we provide and algorithm based on this representation for the verification step. The whole technique uses deontic interpreted systems as its underlying semantics.

3.1 From deontic interpreted systems to boolean formulae

In this section we translate a deontic interpreted system into boolean formulae. As boolean formulae are built using boolean variables, we begin by computing the required number of boolean variables. To encode local states of an agent, the number of boolean variables required is $nv(i) = \lceil log_2 |L_i| \rceil$. To encode actions, the number of variables required is $na(i) = \lceil log_2 |Act_i| \rceil$. Hence, given $N = \sum_i nv(i)$, a global state can be encoded by means N boolean variables: $g = (v_1, \ldots, v_N)$. Similarly, given $M = \sum_i na(i)$, a joint action can be encoded as $a = (a_1, \ldots, a_M)$.

Having encoded local states, global states, and actions by means of boolean variables, all the remaining parameters can be expressed as boolean functions as follows. The protocols relate local states to set of actions, and can be expressed as boolean formulae. The evolution functions can be translated into boolean formulae, too. Indeed, the definition of t_i in Section 2.1 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: $t_i = c_{i,1} \oplus \ldots \oplus c_{i,k}$ where $\oplus$ denotes exclusive-or. We assume that the last condition $c_{i,k}$ of t_i 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 the system, so that only the conditions causing a change in the configuration of the system need to be listed. 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.

In addition to these parameters, 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 functions t_i as follows. First, we introduce a *global* evolution function t:

$$t = \bigwedge_{i \in \{1,\dots,n\}} t_i = \bigwedge_{i \in \{1,\dots,n\}} (c_{i,1} \oplus \ldots \oplus c_{i,k_i})$$

Notice that t is a boolean function involving two global states and a joint action $a = (a_1, \dots, a_M)$. To abstract from the joint action 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 above can be translated into a propositional formula using a disjunction (see [11, 4] 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 relation R_t gives the desired boolean relation between global states.

3.2 The algorithm

In this section we present the algorithm $SAT(\varphi)$ to compute the set of global states in which a formula φ holds. The following are the parameters needed by the algorithm:

- the boolean variables $(v_1, \dots, v_N)$ and $(a_1, \dots, a_M)$ encoding global states and joint actions;
- n boolean functions $P_i(v_1, \dots, v_N, a_1, \dots, a_M)$ encoding the protocols of the agents;
- 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 $(v_1, \dots, v_N)$;
- 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 [11]);
- the boolean function R_t encoding the temporal transition;
- n boolean functions R_i encoding the accessibility relations $\sim_i$ (these functions are defined using equivalence on local states of G);
- n boolean functions R_i^O encoding the deontic accessibility relations $\prec_i^O$.

The algorithm is as follows:

```
SAT(φ) {
  φ is an atomic formula: return 𝒱(φ);
  φ is ¬φ1: return G \ SAT(φ1);
  φ is φ1 ∧ φ2: return SAT(φ1) ∩ SAT(φ2);
  φ is EXφ1: return SAT_EX(φ1);
  φ is E(φ1 U φ2): return SAT_EU(φ1, φ2);
  φ is EGφ1: return SAT_EG(φ1);
  φ is K_i φ1: return SAT_K(φ1, i);
  φ is O_i φ1: return SAT_O(φ1, i);
  φ is K̂_i^j φ1: return SAT_KH(φ1, i, j);
}
```

In the algorithm above, SAT_{EX}, SAT_{EG}, SAT_{EU} are the standard procedures for CTL model checking [7], in which the temporal relation is R_t and, instead of temporal states, global states are considered. The procedures $SAT_K(\varphi, i)$, $SAT_O(\varphi, i)$ and $SAT_{KH}(\varphi, i, j)$ return a set of states in which the formulae $K_i\varphi$, $O_i\varphi$ and $\widehat{K}_i^j\,\varphi$ are true. Their implementation is presented below.

```
SAT_K(φ, i) {
  X = SAT(¬φ);
  Y = {g ∈ G | ∃g' ∈ X and R_i(g, g')}
  return ¬Y;
}
```

```
SAT_O(φ, i) {
  X = SAT(¬φ);
  Y = {g ∈ G | ∃g' ∈ X and R_i^O(g, g')}
  return ¬Y;
}
```

```
SAT_KH(φ, Γ) {
  X = SAT(φ);
  Y = {g ∈ G | ∃g' ∈ X and R_i(g, g') and R_j^O(g, g')}
  return ¬Y;
}
```

Notice that all the parameters can be encoded as OBDD's. Moreover, all the operations in 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. $SAT(\varphi)$ can be used to verify whether or not a formula φ holds in a model by comparing two set of states: the set $SAT(\varphi)$ 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 $SAT(\varphi)$ 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 deontic interpreted systems symbolically, while in Section 4.2 we describe how the language is translated into OBDD's and how the algorithm is implemented. The implementation is available for download[16].

4.1 How to define a deontic interpreted system

To define a deontic interpreted system it is necessary to specify all the parameters presented in Section 2.1. In other words, for each agent, we need to represent:

- a list of local states, and a list of "green" 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;
                 LgreenDef;
                 ActionDef;
                 ProtocolDef;
                 EvolutionDef;
             "end Agent"
LstateDef ::= "Lstate = {" IDLIST "}"
LgreenDef ::= "Lgreen = {" 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 a deontic interpreted system, it is also necessary to define the following parameters:

- an evaluation function;

– a set of initial states (expressed as a boolean condition);
– a list of subsets of the set of agents to be used for particular 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"
                    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 ")" |
            "O(" ID "," formula ")" |
            "KH(" ID ", " ID ", " formula ")"
```

Above, `K` denotes knowledge of the agent identified by the string `ID`; `O` is the deontic operator for the agent identified by `ID`. To represent the knowledge of an agent under the assumption of correct behaviour of another agent we use the operator `KH` followed by an identifier for the first agent, followed by another identifier for the second agent, and a formula.

4.2 Implementation of the algorithm

Figure 2 lists the main components of the software tool that we have implemented. Steps 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 the 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. These OBDD's are created and manipulated using the CUDD library [19]. 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

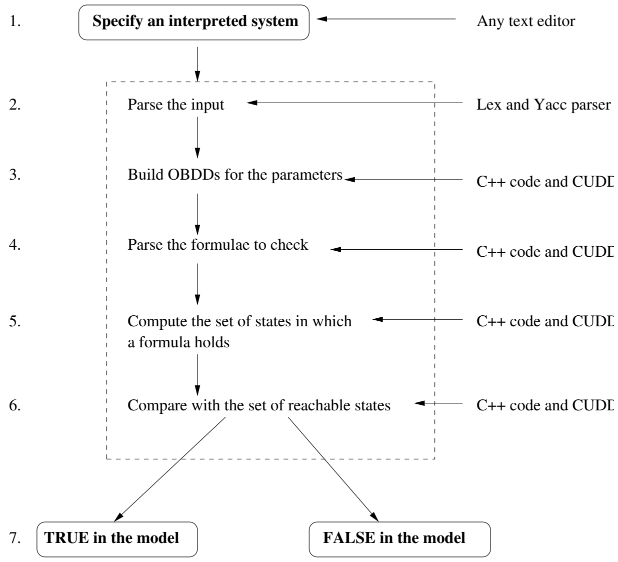

Fig. 2. Software structure

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 be checked are read from a text file, and parsed.
– In step 5 the 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 the two sets are equivalent, the formula holds in the model and the tools produce a positive output. If the two sets are not equivalent, the tool produces a negative output.

5 An example: the bit transmission problem with faults

In this section we test our implementation by verifying temporal, epistemic and deontic properties of a communication example: the bit transmission problem [5].

The bit-transmission problem involves two agents, a *sender* S, and a *receiver* R, communicating over a faulty communication channel. The channel may drop messages but will not flip the value of a bit being sent. S wants to communicate some information (the value of a bit) to R. One protocol for achieving this is as follows. S immediately starts sending the bit to R, and continues to do so until it receives an acknowledgement from R. R does nothing until it receives the bit; from then on it sends acknowledgements of receipt to S. S stops sending the bit to R when it receives an acknowledgement.

This scenario is extended in [10] to deal with failures. In particular, here we assume that R may not behave as intended perhaps as a consequence of a failure. There are different kind of faults that we may consider for R. Following [10], we discuss two examples; in the first, R may fail to send acknowledgements when it receives a message. In the second, R may send acknowledgements even if it has not received any message.

In Section 5.1, we give an overview of how these scenarios can be encoded in the formalism of deontic interpreted systems. This section is taken from [10]. In Section 5.2 we verify some properties of this scenario with our tool, and we give some quantitative results about its performance.

5.1 Deontic interpreted systems for the bit transmission problem

It is possible to represent the scenario described above by means of the formalism of deontic interpreted systems, as presented in [10, 8]. To this end, a third agent called E (environment) is introduced, to model the unreliable communication channel. The local states of the environment record the possible combinations of messages that have been sent in a round, either by S or R. Hence, four possible local states L_E are taken for the environment: $L_E = \{(.,.), (sendbit, .), (., sendack), (sendbit, sendack)\}$, where '.' represents configurations in which no message has been sent by the corresponding agent. The actions Act_E for the environment correspond to the transmission of messages between S and R on the unreliable communication channel. It is assumed that the communication channel can transmit messages in both directions simultaneously, and that a message travelling in one direction can get through while a message travelling in the opposite direction is lost. The set of actions Act_E for the environment is: $Act_E = \{S{-}R,\ S{\rightarrow},\ {\leftarrow}R,\ -\}$. "$S{-}R$" represents the action in which the channel transmits any message successfully in both directions, "$S{\rightarrow}$" that it transmits successfully from S to R but loses any message from R to S, "${\leftarrow}R$" that it transmits successfully from R to S but loses any message from S to R, and "$-$" that it loses any messages sent in either direction. We assume the following constant function for the protocol of the environment, P_E:

$$P_E(l_E) = Act_E = \{S{-}R,\ S{\rightarrow},\ {\leftarrow}R,\ -\}, \quad \text{for all } l_E \in L_E.$$

The evolution function for E is reported in Table 1.

Final state	**Transition condition**
$(.,.)$	$Act_S = \lambda$ and $Act_R = \lambda$
$(sendbit, .)$	$Act_S = sendbit(0)$ and $Act_R = \lambda$ or $Act_S = sendbit(1)$ and $Act_R = \lambda$
$(., sendack)$	$Act_R = \lambda$ and $Act_R = sendack$
$(sendbit, sendack)$	$Act_S = sendbit(0)$ and $Act_R = sendack$ or $Act_S = sendbit(1)$ and $Act_R = sendack$

Table 1. Transition conditions for E.

We model sender S by considering four possible local states. They represent the value of the bit S is attempting to transmit, and whether or not S has received an acknowledgement from R: $L_S = \{0, 1, (0, ack), (1, ack)\}$. The set of actions Act_S for S is: $Act_S = \{sendbit(0), sendbit(1), \lambda\}$, where λ denotes a null action.. The protocol for S is defined as follows:

$$P_S(0) = sendbit(0), \quad P_S(1) = sendbit(1),$$
$$P_S((0, ack)) = P_S((1, ack)) = \lambda.$$

The transition conditions for S are listed in Table 2.

Final state	**Transition condition**
$(0, ack)$	$L_S = 0$ and $Act_R = sendack$ and $Act_E = S{-}R$ or $L_S = 0$ and $Act_R = sendack$ and $Act_E = \leftarrow R$
$(1, ack)$	$L_S = 1$ and $Act_R = sendack$ and $Act_E = S{-}R$ or $L_S = 1$ and $Act_R = sendack$ and $Act_E = \leftarrow R$

Table 2. Transition conditions for S.

We now consider two possible faulty behaviours for R, that we model below.

Faulty receiver – 1 In this case we assume that R may fail to send acknowledgements when it is supposed to. To this end, we introduce the following local states for R: $L'_R = \{0, 1, \epsilon, (0, f), (1, f)\}$. The state "$\epsilon$" is used to denote the fact that R did not receive any message from S; "0" and "1" denote the value of the received bit. The states "(i, f)" ($i = \{0, 1\}$) are *faulty* or *red* states denoting that, at some point in the past, R received a bit but failed to send an acknowledgement. The set of allowed actions for R is: $Act_R = \{sendack, \lambda\}$. The protocol for R is the following:

$$P'_R(\epsilon) = \lambda, P'_R(0) = P'_R(1) = \{sendack, \lambda\}, P'_R((0, f)) = P'_R((1, f)) = \{sendack, \lambda\}.$$

The transition conditions for R are listed in Table 3.

Final state	**Transition condition**
0	$Act_S = sendbit(0)$ and $L_R = \epsilon$ and $Act_E = S{-}R$ or $Act_S = sendbit(0)$ and $L_R = \epsilon$ and $Act_E = S{\rightarrow}$
1	$Act_S = sendbit(1)$ and $L_R = \epsilon$ and $Act_E = S{-}R$ or $Act_S = sendbit(1)$ and $L_R = \epsilon$ and $Act_E = S{\rightarrow}$
$(0, f)$	$L_R = 0$ and $Act_R = \epsilon$
$(1, f)$	$L_R = 1$ and $Act_R = \epsilon$

Table 3. Transition conditions for R.

Faulty receiver – 2 In this second case we assume that R may send acknowledgements without having received a bit first. We model this scenario with the following set of local states L''_R for R:

$$L''_R = \{0, 1, \epsilon, (0, f), (1, f), (\epsilon, f)\}.$$

The local states "ϵ", "0", "1", "$(0, f)$" and "$(1, f)$" are as above; "(ϵ, f)" is a further *faulty* state corresponding to the fact that, at some point in the past, R sent an acknowledgement without having received a bit. The actions allowed are the same as in the previous example. The protocol is defined as follows:

$$P''_R(\epsilon) = \lambda,$$
$$P''_R(0) = P''_R(1) = sendack,$$
$$P''_R((0,f)) = P''_R((1,f)) = P''_R((\epsilon,f)) = \{sendack, \lambda\}.$$

The evolution function is reported in Table 4.

Final state	**Transition condition**
0	$Act_S = sendbit(0)$ and $L_R = \epsilon$ and $Act_E = S{-}R$ or $Act_S = sendbit(0)$ and $L_R = \epsilon$ and $Act_E = S{\rightarrow}$
1	$Act_S = sendbit(1)$ and $L_R = \epsilon$ and $Act_E = S{-}R$ or $Act_S = sendbit(1)$ and $L_R = \epsilon$ and $Act_E = S{\rightarrow}$
(ϵ, f)	$L_R = \epsilon$ and $Act_R = sendack$
$(0, f)$	$Act_S = sendbit(0)$ and $L_R = (\epsilon, f)$ and $Act_E = S{-}R$ or $Act_S = sendbit(0)$ and $L_R = (\epsilon, f)$ and $Act_E = S{\rightarrow}$
$(1, f)$	$Act_S = sendbit(1)$ and $L_R = (\epsilon, f)$ and $Act_E = S{-}R$ or $Act_S = sendbit(1)$ and $L_R = (\epsilon, f)$ and $Act_E = S{\rightarrow}$

Table 4. Transition conditions for R.

For both examples, we introduce the following evaluation function:

$$\mathcal{V}(\mathbf{bit} = \mathbf{0}) = \{g \in G | l_S(g) = 0 \text{ or } l_S(g) = (0, ack)\}$$
$$\mathcal{V}(\mathbf{bit} = \mathbf{1}) = \{g \in G | l_S(g) = 1 \text{ or } l_S(g) = (1, ack)\}$$
$$\mathcal{V}(\mathbf{recbit}) = \{g \in G | l_R(g) = 1 \text{ or } l_R(g) = 0\}$$
$$\mathcal{V}(\mathbf{recack}) = \{g \in G | l_S(g) = (1, ack) \text{ or } l_S(g) = (0, ack)\}$$

The evaluation function $\mathcal{V}$ and the parameters above generate two deontic interpreted systems, one for each faulty behaviour of R; we refer to these deontic interpreted systems as DIS_1 and DIS_2.

It is now possible to express formally properties of these scenarios by means of the language of Section 2.1.

$$A(\neg(K_S(K_R(\mathbf{bit} = \mathbf{0}) \vee K_R(\mathbf{bit} = \mathbf{1}))) \; U \; recack) \tag{1}$$

$$A(\neg(\widehat{K}_S^R(K_R(\mathbf{bit} = \mathbf{0}) \vee K_R(\mathbf{bit} = \mathbf{1}))) \; U \; recack) \tag{2}$$

Formula 1 above captures the fact that S will not know that R knows the value of the bit, until S receives an acknowledgement. Formula 2 expresses the same idea but by using knowledge under the assumption of correct behaviour. In the next section we will verify in an automatic way that Formula 1 holds in DIS_1 but not in DIS_2. This means that the faulty behaviour of R in DIS_1 does not affect the key property of the system. On the contrary, Formula 2 holds in both DIS_1 and DIS_2; hence, a particular form of knowledge is retained irrespective of the fault.

5.2 Experimental results

We have encoded the deontic interpreted system and the formulae 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). The two formulae were correctly verified by the tool for DIS_1, while Formula 1 failed in DIS_2 as expected.

To evaluate the performance of our tool, we first analyse the space requirements. Following the standard conventions, we define the size of a deontic interpreted system as $|DIS| = |S| + |R|$, where $|S|$ is the size of the state space and $|R|$ is the size of the relations. In our case, we define $|S|$ as the number all the possible combinations of local states and actions. In the example above, there are 4 local states and 3 actions for S, 5 (or 6) local states and 2 actions for R, and 4 local states and 4 actions for E. In total we have $|S| \approx 2 \cdot 10^3$. To define $|R|$ we must take into account that, in addition to the temporal relation, there are also the epistemic and deontic relations. Hence, we define $|R|$ as the sum of the sizes of temporal, epistemic, and deontic relations. We approximate $|R|$ as $|S|^2$, hence $|M| = |S| + |R| \approx |S|^2 \approx 4 \cdot 10^6$.

To quantify the memory requirements we consider the maximum number of nodes allocated for the OBDD's. Notice that this figure over-estimates the number of nodes required to encode the state space and the relations. Further, we report the total memory used by the tool (in MBytes). The formulae of both examples required a similar amount of memory and nodes. The average experimental results are reported in Table 5.

$\|M\|$	OBDD**'s nodes**	**Memory (MBytes)**
$\approx 4 \cdot 10^6$	$\approx 10^3$	≈ 4.5

Table 5. Memory requirements.

In addition to space requirements, we carried out some test on time 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. 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 results are listed in Table 6.

Model construction	**Verification**	**Total**
0.045sec	<0.01sec	0.05sec

Table 6. Running time (for one formula).

We see these as very encouraging results. We have been able to check formulae with nested temporal, epistemic and deontic modalities in less than 0.1 seconds on a standard PC, for a non-trivial model. Also, the number of OBDD's nodes is orders of magnitude smaller than the size of the model. Therefore, we believe that our tool could perform reasonably well even in much bigger scenarios.

6 Conclusion

In this paper we have extended a major verification technique for reactive systems — symbolic model checking via OBDD's — to verify temporal, epistemic, and deontic properties of multiagent systems. We provided an algorithm and its implementation, and we tested our implementation by means of an example: the bit transmission problem with faults. The results obtained are very encouraging, and we estimate that our tool could be used in bigger examples. For the same reason, we see as feasible an extension of the tool to include other modal operators.

References

1. M. Benerecetti, F. Giunchiglia, and L. Serafini. Model checking multiagent systems. *Journal of Logic and Computation*, 8(3):401–423, June 1998.
2. 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.
3. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, pages 677–691, August 1986.
4. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
5. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. The MIT Press, Cambridge, Massachusetts, 1995.
6. G. J. Holzmann. The model checker spin. *IEEE transaction on software engineering*, 23(5), May 1997.
7. M. R. A. Huth and M. D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, England, 2000.
8. A. Lomuscio, F. Raimondi, and M. Sergot. Towards model checking interpreted systems. In *Proceedings of MoChArt*, Lyon, France, August 2002.
9. A. Lomuscio and M. Sergot. On multi-agent systems specification via deontic logic. In J.-J Meyer, editor, *Proceedings of ATAL 2001*, volume 2333. Springer Verlag, July 2001.
10. A. Lomuscio and M. Sergot. Violation, error recovery, and enforcement in the bit transmission problem. In *Proceedings of DEON'02*, London, May 2002.
11. K. L. McMillan. *Symbolic model checking: An approach to the state explosion problem*. Kluwer Academic Publishers, 1993.
12. 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.
13. R. van der Meyden and K. Su. Symbolic model checking the knowledge of the dining cryptographers. Submitted, 2002.
14. J.-J. Meyer and R. Wieringa, editors. *Deontic Logic in Computer Science*, Chichester, 1993.
15. W. Penczek and A. Lomuscio. Verifying epistemic properties of multi-agent systems via model checking. *Fundamenta Informaticae*, 55(2):167–185, 2003.
16. F. Raimondi and A. Lomuscio. A tool for verification of deontic interpreted systems. http://www.dcs.kcl.ac.uk/pg/franco/mcdis-0.1.tar.gz.
17. F. Raimondi and A. Lomuscio. Verification of multiagent systems via ordered binary decision diagrams: an algorithm and its implementation. Submitted, 2004.
18. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. *Lecture Notes in Computer Science*, 1038:42–52, 1996.
19. F. Somenzi. CU Decision Diagram Package - Release 2.3.1. http://vlsi.colorado.edu/∼fabio/CUDD/cuddIntro.html.
20. 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.