Int J Softw Tools Technol Transfer
DOI 10.1007/s10009-015-0378-x

REGULAR PAPER

MCMAS: an open-source model checker for the verification of multi-agent systems

Alessio Lomuscio[1] · Hongyang Qu[2] · Franco Raimondi[3]

Abstract We present MCMAS, a model checker for the verification of multi-agent systems. MCMAS supports efficient symbolic techniques for the verification of multi-agent systems against specifications representing temporal, epistemic and strategic properties. We present the underlying semantics of the specification language supported and the algorithms implemented in MCMAS, including its fairness and counterexample generation features. We provide a detailed description of the implementation. We illustrate its use by discussing a number of examples and evaluate its performance by comparing it against other model checkers for multi-agent systems on a common case study.

Keywords Verification · Multi-agent systems · Model checking

1 Introduction

Model checking [15] is widely recognised as one of the leading logic-based techniques for the verification of reactive systems [54]. In this paradigm, a system S is encoded as a transition system, or model, M_S by means of a programme in a dedicated modelling language such as reactive modules [1] or NuSMV [13]. A specification P of the system is represented as a logical formula ϕ_P. Verifying whether the system S satisfies the specification P is encoded as the problem of checking whether the model M_S satisfies the logical formula ϕ_P, formally written as $M_S \models \phi_P$. Several specifications of reactive systems, including *liveness*, *safety* and several specification patterns [20] of interest, can be encoded in discrete temporal logic, either in its linear variant LTL, or in its branching version CTL. Well-known extensions to this approach include employing real-time and probabilistic specifications [40,43].

The fundamental challenge in model checking is the so-called *state-space explosion*, i.e. the fact that the state space of a system grows exponentially with the number of variables employed to describe it. Various techniques have been developed over the years to tame this difficulty including Binary Decision Diagrams, abstraction, bounded model checking, induction, and assume–guarantee reasoning, thereby resulting in systems with state spaces of 10^{25} and beyond to be verifiable.

While this approach has proven successful for a variety of systems, including reactive and embedded systems as well as hardware designs, *multi-agent systems* (MAS) are often specified using languages strictly stronger than plain temporal logic. MAS are distributed systems whose components, or *agents*, act autonomously to meet their private and joint objectives [71]. MAS are typically specified by asserting the intended evolution of high-level properties of the agents, including their knowledge [24], their beliefs [32], their intentions [18], and obligations [35]. This follows the successful tradition in MAS-based approaches that involves ascribing high-level attitudes to highly autonomous systems to better predict and specify their resulting behaviours [55].

✉ Alessio Lomuscio
A.Lomuscio@imperial.ac.uk

Hongyang Qu
h.qu@sheffield.ac.uk

Franco Raimondi
f.raimondi@mdx.ac.uk

[1] Department of Computing, Imperial College London, London, UK

[2] Department of Automatic Control and Systems Engineering, University of Sheffield, Sheffield, UK

[3] School of Science and Technology, Middlesex University, London, UK

Published online: 26 April 2015

Specifically, epistemic logic [24], or logic of knowledge, has provided a natural and intuitive but yet formally sound and computationally attractive, framework for reasoning about security protocols [30], agreement protocols [66], knowledge-bases, etc. By adopting epistemic modalities as primitives, one can naturally express both private and collective (common, or distributed) knowledge of the agents in the system. For example, a security requirement concerning privacy or secrecy in a system run can be translated into a specification stating that no agent eventually knows the fact in question [64]. Similarly, mutual authentication, is naturally expressed by stating that a principal knows that another principal knows that some key is shared [5]. Similarly, an epistemic account can provide a natural set of specifications for cache-coherence protocols [4]. These are only some examples; we refer to the specialised literature for more examples [31,64].

It is therefore compelling to extend traditional model checking approaches so that they can support specifications that include agent-based features. However, these are often sophisticated modal logics that may involve tailored fixed-points computations. So, novel labelling algorithms evaluating agent-based modalities need to be defined and integrated with those for the temporal logics of interest. In addition, since agent-based logics traditionally come equipped with a semantics that is finer-grained than plain Kripke models for temporal logic, input languages for the system description also need to be tuned to the needs of MAS. These include having an intuitive description of the states, actions, local protocols and local evolutions. These considerations suggest that adapting an existing model checking toolkit to the needs of the verification of MAS specified by epistemic and other logics inspired by MAS is not a simple exercise and may, in fact, result in more effort than building a dedicated one.

In this paper, we describe MCMAS, a model checking toolkit for the verification of MAS specified through a range of agent-based logics. MCMAS uses Ordered Binary Decision Diagrams (OBDDs, [7]) for the symbolic representation of state spaces and dedicated algorithms for the computation of a number of epistemic operators encoding private and group knowledge and Alternating-time Temporal Logic (ATL) operators. MAS are described in MCMAS by means of Interpreted Systems Programming Language (ISPL) programs, whose semantics is close to interpreted systems, a popular framework in temporal-epistemic logic [24]. MCMAS is equipped with an Eclipse-based plug-in for coding support and offers a number of features typical of advanced model checkers, including the graphical representation of counterexamples and witnesses, as well as fairness support. MCMAS is released as open source [68] and has been used in several research projects worldwide. We here present MCMAS version 1.2.2, which extends a previous version [47] by means of more efficient algorithms to compute the state space and the labelling of formulas. Also, among other new features, more sophisticated treatment of uniform strategies, counterexamples and fairness constraints is now supported.

The rest of the paper is organised as follows. In Sect. 2, we provide the formal underpinnings of the technique the model checker implements by giving the syntax and semantics of the logics employed as well as the algorithms implemented in the checker. In Sect. 3 we describe ISPL, the input to the model checker. Section 4 describes the implementation and gives a few examples. Section 5 focuses on specific applications and reports experimental results. Section 6 describes related work and concludes the paper.

2 Symbolic model checking multi-agent systems

In this section, we give the theoretical foundations of MCMAS. We succinctly describe the semantics of interpreted systems in Sect. 2.1. We give the syntax of ATLK in Sect. 2.2 and provide model checking algorithms in Sect. 2.3. We conclude in Sect. 2.4 by presenting the OBDD-based encodings for the algorithms.

2.1 Interpreted systems

At the heart of MCMAS and its modelling language is the notion of interpreted system as a formalisation of multi-agent systems. Interpreted systems were popularised by Fagin et al. [24] as a semantics for reasoning about knowledge; they can be extended to incorporate game theoretic notions such as those provided by ATL modalities. Here we loosely follow the presentation given in [47], where global transitions are given as the composition of local transitions.

We assume AP to be a set of atomic propositions and a set of agents $Ag = \{Ag_0, Ag_1, \ldots, Ag_n\}$ for the system. We often refer to Ag_0 as the environment of the system.

Definition 1 (*Interpreted systems*) Given a set of agents Ag, an interpreted system is a tuple $IS = \langle \{L_i, Act_i, P_i, \tau_i\}_{i \in Ag}, I, h \rangle$ where:

- L_i is a finite set of *possible local states* for agent i.
- Act_i is a finite set of *possible actions* for agent i.
- $P_i : L_i \to 2^{Act_i \setminus \emptyset}$ is a *local protocol* function for agent i returning possible actions at a given local state.
- $\tau_i : L_i \times Act_0 \times \cdots \times Act_n \to L_i$ is a deterministic *local transition function* returning the local state for agent i resulting from the execution of a *joint action* at a given local state; we assume that every action is protocol compliant, i.e. if $l'_i = \tau_i(l_i, a_0, \ldots, a_i, \ldots, a_n)$, then $a_i \in P_i(l_i)$ for all $i \in Ag$.

- $I \subseteq L_0 \times L_1 \times \cdots \times L_n$ is the set of *initial global states*.
- $h \subseteq L_0 \times \cdots \times L_n \times AP$ is a labelling relation encoding which atomic propositions are true in which state.

We say that $G = L_0 \times \cdots \times L_n$ is the set of *possible global states* for the system and $ACT = Act_0 \times \cdots \times Act_n$ the set of *possible joint actions*. For a global state $g = (l_0, \ldots, l_n) \in G$ and any $i \in Ag$, we consider the function $l_i : G \rightarrow L_i$ such that $l_i(g) = l_i$, returning the local state of agent i in the global state g.

Observe that interpreted systems describe finite-state systems composed of agents possibly synchronising with each other and the environment via joint actions. Also note that the local protocols implement the agents' decision making and state the conditions for the transitions in the system. We refer to [24] for more details.

Interpreted systems naturally induce Kripke models which can be used to interpret our specification language. These are defined as follows.

Definition 2 (*Induced models*) Given an interpreted system $IS = (\{L_i, Act_i, P_i, \tau_i\}_{i \in Ag}, I, h)$, the *induced model of* IS (or simply the model) is a tuple $\mathcal{M}_{\mathcal{IS}} = (Ag, ACT, S, T, \{\sim_i\}_{i \in Ag \setminus \{Ag_0\}}, h)$ where:

- Ag is the set of agents of IS;
- $ACT \subseteq Act_0 \times \cdots \times Act_n$ is the set of joint actions for the system IS;
- $S \subseteq L_0 \times \cdots \times L_n$ is the set of *global states reachable from* I *via* T;
- $T \subseteq S \times ACT \times S$ is a transition relation representing the temporal evolution of the system. We assume that T is decomposable and define the transition relation as $(s, a, s') \in T$ iff for all $i \in Ag$ we have $\tau_i(l_i(s), a)) = l_i(s')$;
- $\{\sim_i\}_{i \in Ag \setminus \{Ag_0\}} \subseteq S \times S$ is the set of equivalence relations, one for each agent but not the environment, encoding the epistemic accessibility relations. For any $i \in Ag \setminus \{Ag_0\}$, we assume that $(s, s') \in \sim_i$ iff $l_i(s) = l_i(s')$.

We use the notation $s \overset{a}{\rightarrow} s'$ as a shortcut for $(s, a, s') \in T$. We use the term *path* to denote any sequence of states $\pi = (s^0, s^1, \ldots, s^n, \ldots)$ such that, for all $i \geq 0$, we have $(s^i, a, s^{i+1}) \in T$ for some action $a \in ACT$. Given a path π, we denote with $\pi(k)$ the state at position k. Given a set of agents $\Gamma \subseteq Ag$ and a joint action $a = (a_0, a_1, \ldots, a_n)$, we denote with S_Γ the projection of S on the local states of the agents in Γ and with a_Γ the tuple consisting of the elements in a restricted to the agents in Γ. In this case, we say that a is a *completion* for a_Γ. For instance, if $a = (a_0, a_1, a_2, a_3, a_4)$ and $\Gamma = \{1, 3\}$, $a_\Gamma = (a_1, a_3)$ and $a_{Ag \setminus \Gamma} = (a_0, a_2, a_4)$. Given $\Gamma \subseteq Ag$ and ACT, the set ACT_Γ denotes the set of all tuples a_Γ as above.

We say that a joint action $a \in ACT$ is *enabled* in a state $s \in S$ if there exists a state $s' \in S$ such that $(s, a, s') \in T$. Similarly, given a group Γ, we say that a_Γ is enabled in a state if there exists a completion of a_Γ to a joint action $a \in ACT$ such that a is enabled in that state. Note that, by Definition 1, each component of an enabled action is locally protocol compliant. Further note that since a local action is always possible at any local state, the models considered are serial, i.e. there are no deadlocks.

A *strategy* for agent i is a function $\sigma_i : L_i \rightarrow 2^{ACT_i} \setminus \{\emptyset\}$ such that if $a_i \in \sigma_i(l_i)$, then $a_i \in P_i(l_i)$. Given a group of agents Γ, a *strategy* for Γ is a function $\sigma_\Gamma : S_\Gamma \rightarrow 2^{ACT_\Gamma} \setminus \{\emptyset\}$ such that $\sigma_\Gamma(l_{x_1}, \ldots, l_{x_k}) = (\sigma_{x_1}(l_{x_1}), \ldots, \sigma_{x_k}(l_{x_k}))$, where $\sigma_{x_1}, \ldots, \sigma_{x_k}$ are strategies for the agents $x_1, \ldots, x_k \in \Gamma$.

The strategies defined above are analogous to the non-uniform, incomplete information, *memoryless* strategies in [2]. Note that an agent (or a group of agents) adhering to memoryless strategies with incomplete information may perform different actions in different global states whose local component is the same. This allows for an element of action "guessing" that is not considered useful when reasoning in terms of strategic abilities. In these cases, it is more meaningful to consider, still under incomplete information and memoryless assumptions, deterministic *uniform strategies* [37] of the form $\sigma_i : S_i \rightarrow ACT_\Gamma \setminus \emptyset$, which restrict the definition of strategy above by assuming that the same action is performed in the global states in which agent i has the same local state.

The model induced by an interpreted system is said to be *non-uniform*, i.e. along its paths the agents may pick different actions, compatibly with their protocols, in the same local state at different global states. To evaluate an interpreted system under the *uniformity* assumption, we consider the various (uniform) models derived from the induced non-uniform model in which along any path the agents select the same action whenever they are in the same local state. As we will see later, MCMAS supports the verification of both uniform and non-uniform models derived from an interpreted system.

2.2 Syntax of ATLK and satisfaction

We use ATLK as the specification language for the agents in the system. ATLK combines the logic ATL [2] with modal operators to reason about the knowledge of the agents in the system. As it is known, ATL extends the logic CTL [15] by replacing CTL temporal operators with strategic cooperation modalities expressing what state of affairs a coalition of agents can bring about in a system, irrespective of the actions of the other agents. A large amount of work in Artificial Intelligence and Multi-Agent Systems routinely employs rich specifications using the concepts of what an agent, or a collection of agents, knows about the system and each other (e.g. see [24] for an introduction to the area) and what a group of agents can collectively enforce in the system (e.g. see [65]). We illustrate this through some scenarios in Sect. 5.

Definition 3 (*Syntax of ATLK*) The syntax of the logic ATLK is defined by the following BNF expression:

$$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid \langle\langle\Gamma_1\rangle\rangle X\phi \mid \langle\langle\Gamma_1\rangle\rangle[\phi U \phi] \mid \langle\langle\Gamma_1\rangle\rangle G\phi \mid K_i\phi \mid E_{\Gamma_2}\phi \mid D_{\Gamma_2}\phi \mid C_{\Gamma_2}\phi,$$

where p is an atomic proposition in AP, $i \in Ag \setminus \{Ag_0\}$, $\Gamma_1 \subseteq Ag$ denotes a set of agents and $\Gamma_2 \subseteq Ag$, $\Gamma_2 \neq \emptyset$, denotes a non-empty set of agents.

Since, differently from [2], here we work with incomplete information, the meaning of the modalities depends on the uniformity assumption made on the system. If we do not assume uniformity, the reading of the ATL modalities is as follows. The formula $\langle\langle\Gamma\rangle\rangle X\phi$ expresses that the agents in Γ can ensure that ϕ holds at the next state irrespective of the actions of the agents in $Ag \setminus \Gamma$. In other words, the agents in $Ag \setminus \Gamma$ cannot ensure that ϕ is false at the next state.

The formula $\langle\langle\Gamma\rangle\rangle G\phi$ conveys that it is *possible* that the actions of the agents in Γ result in ϕ being true forever in the future, irrespective of the actions of the agents in $Ag \setminus \Gamma$. As above, this means that the agents outside Γ cannot ensure ϕ is not uniformly realised. Similarly, the formula $\langle\langle\Gamma\rangle\rangle[\phi U \psi]$ signifies that the agents in Γ may be able to realise ψ at some point in the future and to ensure that ϕ holds till then.

While the combination of incomplete information, knowledge and ATL modalities give rises to the readings above, we are generally interested in evaluating what the agents have the power to enforce by means of ATL formulas [2]. In our setting, this is the reading of the ATL operators under the assumption of uniformity.

In this case, the formula $\langle\langle\Gamma\rangle\rangle X\phi$ is read as "group Γ has a strategy to enforce ϕ in the next state (irrespective of the actions of the agents not in Γ)"; $\langle\langle\Gamma\rangle\rangle G\phi$ represents "group Γ has a strategy to enforce ϕ forever in the future"; and $\langle\langle\Gamma\rangle\rangle[\phi U \psi]$ means "group Γ has a strategy to enforce that ψ holds at some point in the future and can ensure that ϕ holds until then".

The remaining operators are used to characterise epistemic states of agents as in [24]. In particular, $K_i\phi$ is read as "agent i knows ϕ", $E_\Gamma\phi$ as "everybody in group Γ knows ϕ", $D_\Gamma\phi$ as "ϕ is distributed knowledge in Γ", and $C_\Gamma\phi$ as "ϕ is common knowledge in Γ. It is known that the standard branching-time temporal operators EX, EG, and EU can be expressed by considering the "grand coalition of all agents", e.g. $EX\phi \equiv \langle\langle Ag\rangle\rangle X\phi$.

The satisfaction of ATLK specifications on induced models is defined recursively as follows (recall that given an action tuple a_Γ for the agents only in Γ, we write a for any of the completions of a_Γ):

Definition 4 (*Satisfaction*) Given a model $\mathcal{M} = (Ag, ACT, S, T, \{\sim_i\}_{i\in\{1\dots n\}}, h)$, a state $s^0 \in S$, and an ATLK formula ϕ, satisfaction for ϕ in $\mathcal{M}$ at state s^0, formally $(\mathcal{M}, s^0) \models \phi$, is recursively defined as follows.

- $(\mathcal{M}, s^0) \models p$ iff $p \in h(s^0)$;
- $(\mathcal{M}, s^0) \models \neg\phi$ iff it is not the case that $(\mathcal{M}, s^0) \models \phi$;
- $(\mathcal{M}, s^0) \models \phi_1 \vee \phi_2$ iff $(\mathcal{M}, s^0) \models \phi_1$ or $(\mathcal{M}, s^0) \models \phi_2$;
- $(\mathcal{M}, s^0) \models \langle\langle\Gamma\rangle\rangle X\phi$ iff there exists a strategy σ_Γ and an action $a_\Gamma \in \sigma_\Gamma(s^0_\Gamma)$ such that for all states s^1 such that $s^0 \xrightarrow{a} s^1$, we have $(\mathcal{M}, s^1) \models \phi$;
- $(\mathcal{M}, s^0) \models \langle\langle\Gamma\rangle\rangle G\phi$ iff there exists a strategy σ_Γ and an action $a^1_\Gamma \in \sigma_\Gamma(s^0_\Gamma)$ such that all states s^1 with $s^0 \xrightarrow{a^1} s^1$ are such that there is an action $a^2_\Gamma \in \sigma_\Gamma(s^1_\Gamma)$ such that all states s^2 with $s^1 \xrightarrow{a^2} s^2$ are such that, etc., and we have that $(\mathcal{M}, s^i) \models \phi$, for all $i \geq 0$.
- $(\mathcal{M}, s^0) \models \langle\langle\Gamma\rangle\rangle[\phi_1 U \phi_2]$ iff there exists a strategy σ_Γ and an action $a^1_\Gamma \in \sigma_\Gamma(s^0_\Gamma)$ such that all states s^1 with $s^0 \xrightarrow{a^1} s^1$ are such that there is an action $a^2_\Gamma \in \sigma_\Gamma(s^1_\Gamma)$ such that all states s^2 with $s^1 \xrightarrow{a^2} s^2$ are such that, etc., and we have $(\mathcal{M}, s^j) \models \phi_2$, for some $j \geq 0$, and $(\mathcal{M}, s^i) \models \phi_1$ for all $0 \leq i < j$.
- $(\mathcal{M}, s^0) \models K_i\phi$ iff for all $s^1 \in S$ we have that if $s^0 \sim_i s^1$ then $(\mathcal{M}, s^1) \models \phi$.
- $(\mathcal{M}, s^0) \models E_\Gamma\phi$ iff for all $s^1 \in S$ we have that if $s^0 \left(\bigcup_{i\in\Gamma} \sim_i\right) s^1$ then $(\mathcal{M}, s^1) \models \phi$.
- $(\mathcal{M}, s^0) \models D_\Gamma\phi$ iff for all $s^1 \in S$ we have that if $s^0 \left(\bigcap_{i\in\Gamma} \sim_i\right) s^1$ then $(\mathcal{M}, s^1) \models \phi$.
- $(\mathcal{M}, s^0) \models C_\Gamma\phi$ iff for all $s^1 \in S$ we have that if $s^0 \left(\bigcup_{i\in\Gamma} \sim_i\right)^+ s^1$, then $(\mathcal{M}, s^1) \models \phi$, where $^+$ denotes the transitive closure of the relation.

We say that an interpreted system $IS = (\{L_i, Act_i, P_i, \tau_i\}_{i\in Ag}, I, h)$ satisfies an ATLK specification ϕ if and only if $(\mathcal{M}_{IS}, s^i) \models \phi$, for all $s^i \in I$. We say that that interpreted system IS satisfies an ATLK specification ϕ under uniformity iff at least one of the models $\mathcal{M}_{IS}$ induced from IS under uniformity is such that $(\mathcal{M}_{IS}, s^i) \models \phi$, for all $s^i \in I$.

The semantics above is *observational*. In other words, the agents' local states do not necessarily encode all the local states encountered by the agent in a run. Note that a bounded form of perfect recall can still be encoded in the semantics. Observational semantics is commonly regarded as the standard treatment of epistemic modalities [24] and it does not increase the complexity of the model checking problem when combined with CTL or ATL [52]. Perfect recall semantics with ATL leads to an undecidable model checking problem [2].

2.3 Symbolic model checking ATLK

We now define model checking algorithms for the logic ATLK; these extend the corresponding ones for CTL [15]. The approach here presented is symbolic in that it uses Ordered Binary Decision Diagrams (OBDDs) [7] as basic data structures to encode sets of states and transitions.

For a given specification model, Algorithm 1 reports the high-level structure of the recursive model checking algorithm $SAT(\phi)$, returning the set of states of the model in which ϕ is true.

Algorithm 1 Model checking algorithm for ATLK

```
Input: $\phi$.
Output: $\{s \in S \mid (\mathcal{M}, s) \models \phi\}$.
if ($\phi$ is an atomic proposition) then
    return $h(\phi)$;
else if ($\phi$ is $\neg\phi_1$) then
    return $(S \setminus SAT(\phi_1))$;
else if ($\phi$ is $\phi_1 \vee \phi_2$) then
    return $(SAT(\phi_1) \cup SAT(\phi_2))$;
else if ($\phi$ is $K_i(\phi_1)$) then
    return $SAT_K(\phi_1, i)$;
else if ($\phi$ is $E_\Gamma(\phi_1)$) then
    return $SAT_E(\phi_1, \Gamma)$;
else if ($\phi$ is $D_\Gamma(\phi_1)$) then
    return $SAT_D(\phi_1, \Gamma)$;
else if ($\phi$ is $C_\Gamma(\phi_1)$) then
    return $SAT_C(\phi_1, \Gamma)$;
else if ($\phi$ is $\langle\langle\Gamma\rangle\rangle X(\phi_1)$) then
    return $SAT_{ATLX}(\phi_1, \Gamma)$;
else if ($\phi$ is $\langle\langle\Gamma\rangle\rangle G(\phi_1)$) then
    return $SAT_{ATLG}(\phi_1, \Gamma)$;
else if ($\phi$ is $\langle\langle\Gamma\rangle\rangle[\phi_1 U \phi_2]$) then
    return $SAT_{ATLU}(\phi_1, \phi_2, \Gamma)$;
end if
```

The procedures SAT_K, SAT_E, and SAT_D for the epistemic operators are described in Algorithms 2, 3, and 4. These take as input the sub-formula to be checked and return the set of states satisfying the epistemic formula.

They compute the existential pre-image of the set $SAT(\neg\phi_1)$ with respect to the appropriate epistemic relation, i.e. the set of states *not* satisfying the epistemic formula. The complement of this set with respect to the set of reachable states S is the set of states satisfying the formula in question.

Algorithm 5 iteratively calculates the set of states that can access a state not satisfying ϕ via a finite sequence of epistemic relations for the agents in Γ. The complement of this set with respect to S is equal to the set of states satisfying $C_\Gamma\phi$ [60].

The algorithms for the ATL operators depend on the auxiliary procedure $ATLPRE(\Gamma, X)$ (see Algorithm 6), which computes the set of states $Y \subseteq S$ from which there exists a joint action a_Γ for the agents in Γ such that all action completions a of a_Γ enabled at a state in Y generate a transition to a state in X. Algorithm 7 employs this procedure directly to compute SAT_{ATLX}, while Algorithms 8 and 9 implement the standard fix-point algorithms.

Given the above, to compute whether an interpreted system IS validates a formula ϕ (without the uniformity assumption) we compute the induced model $\mathcal{M}_{IS}$ and check whether $I \subseteq SAT(\phi)$. To compute whether an interpreted system IS validates a formula ϕ under uniformity, we establish whether $I \subseteq SAT(\phi)$ on some $\mathcal{M}_{IS}$ induced from IS under uniformity. The labelling for models generated under the uniformity assumption is also carried out using Algorithm 1; in this case the generation of the induced models accounts for the constraints on the actions.

Algorithm 2 $SAT_K(\phi, i)$ for $K_i\phi$.

```
$X = SAT(\neg\phi)$;
$Y = \{s \in S \mid \exists s' \in X$ such that $s \sim_i s'\}$;
return $\neg Y \cap S$
```

Algorithm 3 $SAT_E(\phi, \Gamma)$ for $E_\Gamma\phi$.

```
$X = SAT(\neg\phi)$;
$Y = \{s \in S \mid \exists s' \in X$ such that $\exists i \in \Gamma, s \sim_i s'\}$
return $\neg Y \cap S$
```

Algorithm 4 $SAT_D(\phi, \Gamma)$ for $D_\Gamma\phi$.

```
$X = SAT(\neg\phi)$;
$Y = \{s \in S \mid \exists s' \in X$ such that $\forall i \in \Gamma, s \sim_i s'\}$;
return $\neg Y \cap S$
```

Algorithm 5 $SAT_C(\phi, \Gamma)$ for $C_\Gamma\phi$.

```
$X = S; Y = SAT(\neg\phi)$;
while $X \neq Y$ do
    $X = Y$;
    $Y = \{s \in S \mid \exists s' \in X$ and $\exists i \in \Gamma$ such that $s \sim_i s'\}$;
end while
return $\neg X \cap S$
```

2.4 Symbolic model checking and OBDDs

To verify interpreted systems against specifications in ATLK we use the algorithms above in which operations on sets are implemented as operations on Boolean formulae, appropriately encoded as Ordered Binary Decision Diagrams (OBDDs). As an example, consider the Boolean formula

Algorithm 6 $ATLPRE(\Gamma, X)$.

```
$Z = \{(s, a_\Gamma) \mid s \in S$ and $a \in ACT$ and $\exists s' \in X$ such that $(s, a, s') \in T\}$;
$W = \{(s, a_\Gamma) \mid s \in S$ and $a \in ACT$ and $\exists s' \in S \setminus X$ such that $(s, a, s') \in T\}$;
$Y = \{s \mid \exists a \in ACT$ such that $(s, a_\Gamma) \in Z \setminus W\}$;
return $Y$
```

Algorithm 7 $SAT_{ATLX}(\phi, \Gamma)$ for $\langle\langle\Gamma\rangle\rangle X\phi$.

```
$X = SAT(\phi)$;
$Y = ATLPRE(\Gamma, X)$;
return $Y$
```

Algorithm 8 $SAT_{ATLG}(\phi, \Gamma)$ for $\langle\langle\Gamma\rangle\rangle G\phi$.

```
$X = SAT(\phi)$; $Z = SAT_\phi$; $Y = S$;
while $X \neq Y$ do
    $Y = X$;
    $X = Z \cap ATLPRE(\Gamma, X)$;
end while
return $Y$
```

Algorithm 9 $SAT_{ATLU}(\phi_1, \phi_2, \Gamma)$ for $\langle\langle\Gamma\rangle\rangle[\phi_1 U \phi_2]$.

```
$X = SAT(\phi_2)$; $Z = SAT(\phi_1)$; $Y = \emptyset$;
while $X \neq Y$ do
    $Y = X$;
    $X = Y \cup (Z \cap ATLPRE(\Gamma, X))$;
end while
return $Y$
```

$f_1(x_1, x_2, x_3) = \neg x_1 \vee (x_1 \wedge \neg x_2 \wedge \neg x_3)$, where x_1, x_2, x_3 are Boolean variables. The truth table of this formula is eight lines long. Alternatively, f_1 can be represented by means of a binary tree with root node x_1, as in Fig. 1, where the leaves represent the truth value of f_1. This tree can be simplified as in Fig. 2. Notice that the simplified tree only has 5 nodes instead of 15. In general, the *reduced* tree can be orders of magnitude smaller than the truth table for a given Boolean formula.

Symbolic model checking exploits the compression capabilities of OBDDs to represent large state spaces efficiently. A Boolean formula can represent a state in a model. As an example, consider a model in which $S = \{s_0, s_1, \ldots, s_7\}$. The Boolean formula $f_0(x_1, x_2, x_3) = x_1 \wedge x_2 \wedge x_3$ can be used to encode s_0, the Boolean formula $f_1(x_1, x_2, x_3) = \neg x_1 \wedge x_2 \wedge x_3$ to encode s_1, etc. The number of Boolean variables required to encode a set S grows as $O(log_2(|S|))$.

The set of states $\{s_0, s_1\}$ can be represented by the disjunction of f_0 with f_1. Boolean formulae can be encoded by means of OBDDs; Algorithm 1 and all its associated procedures can operate on OBDDs, including the strategic (ATL) and epistemic modalities. We refer to [60] for more details about the encoding process.

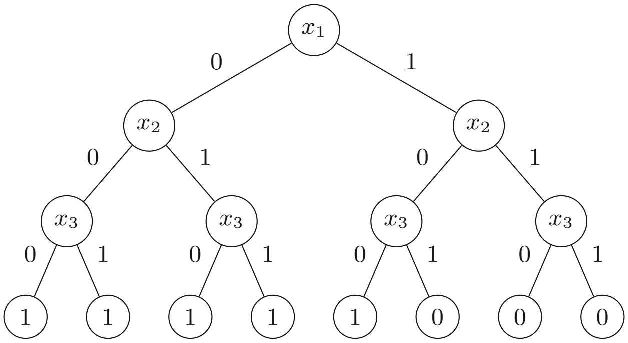


Fig. 1 The evaluation tree for $f_1(x_1, x_2, x_3) = \neg x_1 \vee (x_1 \wedge \neg x_2 \wedge \neg x_3)$

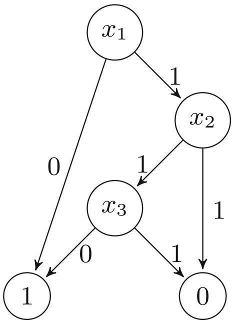


Fig. 2 The OBDD for $f_1(x_1, x_2, x_3) = \neg x_1 \vee (x_1 \wedge \neg x_2 \wedge \neg x_3)$

3 Modelling multi-agent systems in ISPL

In this section, we describe Interpreted Systems Programming Language (ISPL), the language used to model MAS within MCMAS. ISPL is strongly based on Interpreted Systems as defined in Sect. 2. In this section, we present ISPL's constructs and give its semantics.

An ISPL program describes a multi-agent system as composed of a number of agents and an environment. Agents' definitions in ISPL closely follow those of agents in Definition 1. To describe an agent in ISPL we declare the following components. *Local states* are private, internal states of the agents, declared by means of variables, and cannot be observed by the other agents. Agents interact with each other and the environment by means of publicly observable *local actions*. Actions are performed in accordance with a *local protocol* representing the agent's decision-making process. Local states change value over time following a local *evolution function*, which returns the next local state on the basis of the current local state and the joint actions performed by all the other agents at a given instant. ISPL's structure with actions and protocols is deliberately based on interpreted systems' semantics which constitute a widely used framework for describing MAS.

We describe ISPL's syntax using a simple example, the bit transmission protocol [24]. In this protocol, a Sender agent intends to deliver a message (the value of a bit) to a Receiver over an unreliable communication channel. The channel may

```
1   Agent Sender
2     Vars:
3       bit : {b0, b1}; -- The bit can be either 0 or 1
4       ack : boolean;  -- True when the ack received
5     end Vars
6     Actions = {sb0, sb1, epsilon};
7     Protocol:
8       bit=b0 and ack=false : {sb0};
9       bit=b1 and ack=false : {sb1};
10      ack=true : {epsilon};
11    end Protocol
12    Evolution:
13      ack=true if (ack=false) and
14         ( ( (Receiver.Action=sendack) and
15             (Environment.Action=sendSR) )
16          or
17         ( (Receiver.Action=sendack) and
18           (Environment.Action=sendR) )
19         );
20    end Evolution
21  end Agent
```

Fig. 3 A simple ISPL example: the Sender Agent

```
1   Agent Receiver
2     Vars:
3       state : {empty, r0, r1};
4     end Vars
5     Actions = {epsilon,sendack};
6     Protocol:
7       state=empty : {epsilon};
8       (state=r0 or state=r1): {sendack};
9     end Protocol
10    Evolution:
11      state=r0 if ( ( (Sender.Action=sb0) and (state=empty)
12                      and (Environment.Action=sendSR) ) or
13                    ( (Sender.Action=sb0) and (state=empty)
14                      and (Environment.Action=sendS) ) );
15      state=r1 if ( ( (Sender.Action=sb1) and (state=empty)
16                      and (Environment.Action=sendSR) ) or
17                    ( (Sender.Action=sb1) and (state=empty)
18                      and (Environment.Action=sendS) ) );
19    end Evolution
20  end Agent
```

Fig. 4 A simple ISPL example: the Receiver Agent

```
1   Agent Environment
2     Vars:
3       state : {S,R,SR,none};
4     end Vars
5     Actions = {sendS,sendSR,sendR,sendNone};
6     Protocol:
7       state=S: {sendS,sendSR,sendR,sendNone};
8       state=R: {sendS,sendSR,sendR,sendNone};
9       state=SR: {sendS,sendSR,sendR,sendNone};
10      state=none: {sendS,sendSR,sendR,sendNone};
11    end Protocol
12    Evolution:
13      state=S if (Action=sendS);
14      state=R if (Action=sendR);
15      state=SR if (Action=sendSR);
16      state=none if (Action=sendNone);
17    end Evolution
18  end Agent
```

Fig. 5 A simple ISPL example: the Environment Agent

randomly drop messages in either direction, but it does not modify the content of messages. To guarantee communication, the Sender keeps sending the same bit to the Receiver until it receives an acknowledgement; at this point the Sender stops sending the bit. The Receiver performs no action until a bit is received, and keeps sending acknowledgements thereafter.

Figure 3 reports the modelling of the agent Sender in ISPL. The declaration starts with the keyword Agent followed by a string identifier (the name of the agent). The variables declared in the Vars section (lines 2–5) define the local states of the agent. In this example, the Sender has four possible local states, corresponding to the possible combinations of the two variables bit and ack. ISPL also supports the declaration of bounded integers of the form x : a..b; where a and b are two integer numbers (e.g. x: 1..4;). Actions are declared as an enumeration of identifiers (line 6). The Protocol section (lines 7–11) associates actions to sets of local states; for instance, line 10 represents the fact that when the variable ack is true, the agent must perform the action epsilon. Sets of local states are characterised by Boolean conditions on local states. As an example, consider the Boolean condition ack=true on the variable ack: this corresponds to two local states, one where bit=b0 and another where bit=b1. Boolean expressions can also involve arithmetic expressions for integer variables (e.g. x>2 and x<5) and bit expressions for Boolean variables.

Non-deterministic *protocols* are encoded in ISPL either by specifying more than one target action (see lines 7–10 in Fig. 5), or by giving different actions for overlapping conditions on local states. Observe that, differently from knowledge-based programmes [25], actions in ISPL are specified on conditions on local states and not on the explicit knowledge that agents have. The ISPL section Evolution describes how the local states of an agent evolve over time (lines 12–20 of Fig. 3), using a syntax similar to the one employed in NuSMV [13]. For instance, lines 13–19 prescribe that the *next* value of the variable ack will be true if the *current* value of ack is false, agent Receiver is performing the action sendack (see Fig. 4), and the Environment agent is performing either the action sendSR or sendR (see Fig. 5).

Figure 4 reports the full ISPL code for the Receiver agent. In this example, the unreliable channel is modelled by means of the Environment. Figure 5 reports the ISPL code for the Environment; notice (lines 7–10) that its protocol is non-deterministic.

The environment is described using the keyword Environment. One of its features is that some of its local states can be observed by other agents. As an example, the section Obsvars in Fig. 6 lists the variables that are observable by *all* the agents in the system. The variables in the Vars section of the environment, instead, *can* be observed by an

```
Agent Environment
  Obsvars:
     v3: boolean;
  end Obsvars
  Vars:
    v1: boolean;
    v2: boolean;
  end Vars
 [...]
end Agent

Agent A1
   Lobsvars = {v1};
   Vars:
     [...]
end Agent

Agent A2
   Lobsvars = {v2};
   Vars:
     [...]
end Agent
```

Fig. 6 A simple ISPL example: observable and partially observable variables

```
Evaluation
  recbit if ( (Receiver.state=r0) or (Receiver.state=r1) );
  recack if ( ( Sender.ack = true ) );
  bit0 if ( (Sender.bit=b0) );
  bit1 if ( (Sender.bit=b1) );
  envworks if ( Environment.state=SR );
end Evaluation
```

Fig. 7 A simple ISPL example: atomic variables

agent only if they appear in the Lobsvars definition for that agent. For instance, agent A1 in Fig. 6 can observe variable v1 (but not v2), and agent A2 can observe variable v2 (but not v1). Both agents can observe variable v3. This feature enables faster communication and synchronisation between the agents and the environment.

Following the agents' declarations, an ISPL model contains the section Evaluation declaring the *atomic variables* for the model. Figure 7 reports the definition of five atomic variables recbit, recack, bit0, bit1, envworks. The Boolean condition appearing on the right-hand side of each line denotes the set of *global states*, where the propositions hold. The specifications to be verified (see the Formulae section in Fig. 8) are built on the propositions defined here.

The description of a system of agents is completed by providing a set of initial states, an optional set of fairness conditions, and the set of formulae to be verified. As shown in Fig. 8, the set of initial states is declared in the section InitStates by means of a Boolean function imposing conditions on local states. The Fairness section reports a list of Büchi fairness constraints, expressed as Boolean formulae. In the example of Fig. 8, it is required that the proposition envworks, which captures the fact the Environment is transmitting messages in both directions, must be true infinitely often: this means that the channel cannot block messages indefinitely. Finally, the section Formulae contains the specifications to be checked. These are formulas in the logic ATLK; since ATL subsumes CTL, formulas in the logics CTL or CTLK [59] are also supported. As an example, the first formula in Fig. 8 states that it is always true that, when recack is true and the value of the bit is 0, then the agent Sender knows that the agent Receiver knows that the value of the bit is 0. The second specification is stronger and states that when recack is true and the value of the bit is 0, it is common knowledge in the group g1 that the value of the bit is 0. The group g1 is defined above the specifications by the keyword g1, listing the groups of agents to be considered in the epistemic specifications for the model.

```
InitStates
  ( (Sender.bit=b0) or (Sender.bit=b1) ) and
  ( Receiver.state=empty) and (Sender.ack=false) and
  ( Environment.state=none);
end InitStates

Fairness
  envworks;
end Fairness

Groups
        g1 = {Sender,Receiver};
end Groups

Formulae
  AG((recack and bit0) -> K(Sender,(K(Receiver,bit0))));
  AG((recack and bit0) -> GCK(g1,bit0)));
end Formulae
```

Fig. 8 A simple ISPL example: initial states, groups, and fairness conditions

Semantics of ISPL programs Any ISPL program P uniquely denotes an interpreted system $IS_P = (\{L_i, Act_i, P_i, \tau_i\}_{i \in Ag}, I)$, obtained by instantiating the corresponding elements in IS_P by means of the corresponding declarations in P. More formally:

- The set of agents in IS_P is the set of agents declared in P, where the environment in P is mapped to Ag_0 in IS_P.
- For each agent i, the set of possible local states L_i in IS_P is defined by taking the Cartesian product of the corresponding sets defined for the local variables for agent i in P (Section Vars).
- For each agent i, the set Act_i is the corresponding set of actions agent i in the programme P (Section Actions).
- For each agent i, the protocol P_i is defined by the list of Boolean conditions in the Section Protocol for the agent i in the programme P.
- For each agent i, the function τ_i is defined by the list of Boolean conditions in the Section Evolution for the agent i in the programme P.
- The set of global initial states I is defined by evaluating the Boolean conditions in InitStates in P.

Given an ISPL program P and the interpreted system IS_P denoted by P, we construct the induced model M_{IS_P} by applying Definition 2 to IS_P and by taking the evaluation h defined by the Boolean conditions in P (Section `Evaluation`). Formally, an ISPL program P satisfies a specification ϕ (given in section `Formulae`) if $M_{IS_P} \models \phi$. Note that since the semantics of ISPL programs is defined in terms of their corresponding interpreted system, their evolution is deterministic. Also observe that the composition of the different agents is synchronous via joint actions as in interpreted systems.

As described in Sect. 2.1, under uniformity, an interpreted system induces not just one but a set of uniform models. In this case, we say that an ISPL program P satisfies ϕ under uniform semantics if there exists an induced uniform model M_{IS_P} such that $M_{IS_P} \models \phi$.

As an example, the model generated by the ISPL program for the bit transmission protocol is depicted in Fig. 9, where:

- Global states are represented by rectangles; only the local states of the Sender and Receiver are reported, for readability the Environment is not. For instance, the global state `((b0,false),empty)` in the top-left corner encodes a state in which variable `bit` for the Sender has value `r0`, variable `ack` is false, and variable `state` for the Receiver is `empty`.
- The temporal transitions are represented by solid arrows and for brevity are labelled with the Environment action only (the Sender's and Receiver's actions are derived deterministically from the protocol).
- The epistemic relations for the Sender are represented by dotted lines; dashed lines represent the epistemic relations for the Receiver. All reflexive relations are omitted.

It can be manually checked that the first specification given is satisfied on the model; so the programme satisfies it; whereas the second is false. This is in line with work in epistemic logic that establishes that common knowledge cannot be obtained in the presence of a faulty communication channel [24].

4 MCMAS: implementation and usage

MCMAS is implemented in C++ and can be compiled from its source code on most platforms (including Windows, Linux, Mac, Raspberry Pi and various other UNIX systems). The build recognises most architectures automatically. Precompiled versions of the tool are also available from the support pages [68]. In what follows, we describe MCMAS ver 1.2.2, released in March 2015.

4.1 Implementation details

To illustrate the tool, we discuss a number of implementation choices that affect the overall performance of MCMAS. In particular, we consider the following issues: (1) variable ordering in OBDDs; (2) computation of the set of reachable states; (3) construction of the temporal and epistemic transition relations; (4) consistency checks of the input model.

4.1.1 Variable ordering in OBDDs

The size of an OBDD is very sensitive to the choice of the ordering for its Boolean variables. A good ordering can use less memory and speed up OBDD operations by orders of magnitude with respect to an alternative one. Finding a static ordering that generates a compact OBDD representation

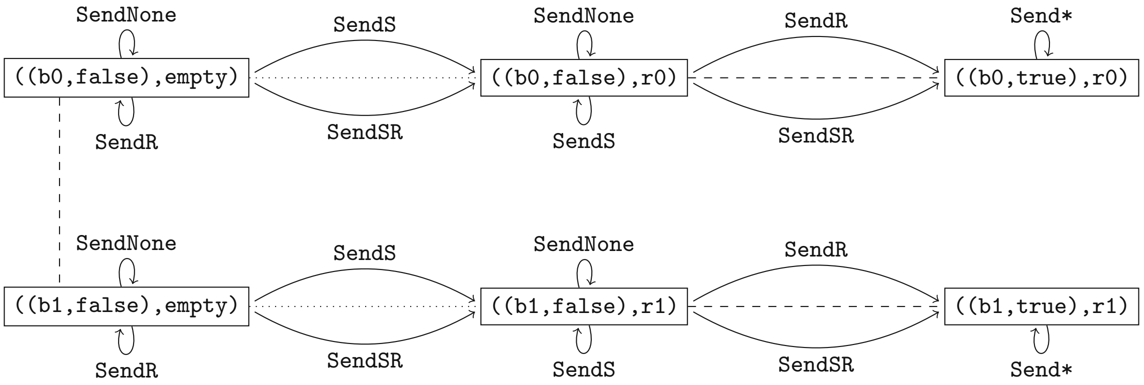


Fig. 9 The model generated from the ISPL code for the bit transmission protocol. *Solid lines* are temporal transitions labelled with actions (only the Environment action is reported); the *dotted lines* represent the epistemic accessibility relations for the Sender; the *dashed lines* represent the epistemic accessibility relations for the Receiver. Reflexive links and the local states of the environment are omitted

for representing the state space and the transition relation is challenging. Dynamic reordering is a useful technique aimed at finding a good compromise between continuous variable reordering and efficiency with the aim of reducing memory consumption during model checking.

In some cases when the overhead of dynamic reordering cannot be offset by its savings, it may be more efficient to disable it completely, a feature supported by MCMAS. It is normally advantageous to provide heuristics for dynamic reordering tailored to specific models to reduce the overhead of evaluating all the possible orderings. The CUDD library [61] used by MCMAS allows grouping OBDD variables so that the order of variables in the same group is not changed during reordering. This approach reduces the number of orderings that need to be explored. MCMAS provides four different initial OBDD grouping strategies:

1. Boolean variables for current and successor states are interleaved. The resulting ordering is: $(v_0, v'_0) \cdots (v_n, v'_n)X_0, \ldots X_n$, where $v_0, \ldots v_{i_0}$ are variables encoding the local states for agent 0 and the other agents, and primed variables encode successor states. The variables $X_0 \ldots X_n$ encode actions and are grouped at the end.
2. A variation of the above whereby variables for actions are interleaved with states: $(v_0, v'_0)X_0 \cdots \ (v_n, v'_n)X_n$.
3. For each agent, all the variables for the current state are grouped at the beginning, followed by the variables used to encode actions, followed by primed variables to encode successor states: $(v_0, \ldots, v_{i_0})X_0(v'_0, \ldots, v'_{i_0}) \ldots (v_{i_nN+1}, \ldots, v_n)X_n(v_{i_N+1}, \ldots, v_n)$,
4. A variation of the case above whereby variables for the actions follow the variables for the states: $(v_0, \ldots, v_{i_0})$ $(v'_0, \ldots, v'_{i_0})$ $X_0 \cdots (v_{i_N+1}, \ldots, v_n)$ $(v_{i_N+1}, \ldots, v_n)$ X_n.

The first two strategies are similar to those used by NuSMV and other model checkers. The default OBDD ordering strategy in MCMAS is the second one reported above, but, as we show later, some models benefit from the third and fourth heuristics.

4.1.2 Computing the set of reachable states

Let IS be an interpreted system and $\mathcal{M}_{\mathcal{IS}} = (Ag, ACT, S, T, \{\sim_i\}_{i \in \{1\ldots n\}}, h)$ its induced model. In what follows the function $image(x, T)$ returns the set of states accessible from x under the transition relation T. MCMAS uses OBDDs to represent states and functions; the corresponding operations are conducted directly on OBDDs.

Algorithm 10 Three approaches to compute reachable states

```
S ⇐ 0; q ⇐ I;
while S ≠ q do
    S ⇐ q; q ⇐ S ∪ image(S, T);
end while
```

```
S ⇐ 0; n ⇐ I; q ⇐ I
while S ≠ q do
    S ⇐ q; n = image(n, T); q ⇐ S ∪ n;
end while
```

```
S ⇐ 0; n ⇐ I; q ⇐ I
while S ≠ q do
    S ⇐ q; n = image(n, T) \ S; q ⇐ S ∪ n;
end while
```

MCMAS provides three approaches to compute the set of reachable states (see Algorithm 10). Experiments show that the time to construct OBDDs for reachable states by these approaches varies for different models and platforms (e.g. 32-bits Linux, 64-bits Linux, 32-bits Windows, etc.). No approach is consistently superior to the others. Note that MCMAS does not compute a single OBDD for T to avoid unnecessary computation time. Instead, the tool builds n transition relations $T_i \subseteq S \times ACT \times S, 0 \leq i \leq n$, one for each agent and the environment; these are defined on the basis of the local transition functions $\tau_0, \ldots, \tau_n$ (see Definition 1) as follows: $T_i(s, a, s')$ iff $l_i(s') = \tau_i(l_i(s), a)$. When the image of x with respect to T is needed, MCMAS executes a loop to construct the intersection of $T_0, \ldots, T_n$ with x, i.e.

$$y = x \cap T_0 \cap T_1 \cap \cdots \cap T_n.$$

Note that x is also encoded over the whole set of OBBD variables, among which those for actions and successor states are abstracted away.

4.1.3 Building the temporal and epistemic relations

It is often the case that the set of reachable states is only a small subset of the set of possible global states. While it may be time consuming to build a single OBDD for the *complete* transition relation T for computing the reachable states, computing the *partial* transition relation T_{reach} usually speeds up the process when checking properties on reachable states. Formally, T_{reach} is computed as follows:

$$T_{reach} = S \cap T_0 \cap T_1 \cap \cdots \cap T_n,$$

where S is the set of reachable states. In other words, T_{reach} is the projection of T on reachable states.

In Algorithm 2, the epistemic-indistinguishable relation $\sim_i$ can be pre-computed for each agent i. As in the case for the transition relation T, the computation of $\sim_i$ is usually time consuming. MCMAS employs *variable masking* to compute the OBDD encoding of the epistemic relations. For a set X of states, we compute the set Y of indistinguishable state with respect to $\sim_i$ as follows: let V be the set of all OBDD variables encoding global states, and $V_i \subseteq V$ be the set of OBDD variables encoding the local states for agent i.

1. We first compute an OBDD X' by removing the OBDD variables in $V \backslash V_i$, which are not part of agent i's encoding. X' is characterised by the following set:

   $$X' = \{s \in L_0 \times \cdots \times L_n \mid \exists s' \in X \text{ such that } s'_i = s_i\},$$

   where s_i and s'_i are the local states of agent i in s and s', respectively.
2. The OBDD Y is computed as $Y = X' \cap S$, where S is the set of reachable states.

The above two steps can be performed efficiently using the procedures available in the CUDD library, and are applicable to Algorithms 3, 4 and 5.

For efficiency purposes, MCMAS implements optimised algorithms for the verification of CTL operators, rather than employing the procedures for the ATL modalities.

4.1.4 Fairness for ATLK

In a number of circumstances, it is desirable to remove certain unwanted behaviours from the possible executions of a system. For instance, consider the code for the Environment of the bit transmission protocol reported in Fig. 5. The protocol for this agent allows the environment to block messages *forever*; yet, the designer is likely to want to describe a situation in which the communication channel randomly drops messages, but it is not continuously faulty.

The removal of unwanted behaviours is often achieved by imposing *fairness conditions*. In the case of branching logics such as ATLK, this requires the definition of constraints outside the model and the use of purpose-built verification algorithms which extend the standard labelling algorithm presented in Sect. 2.3.

Fairness conditions are declared in MCMAS using an optional set of Boolean formulae constructed using atomic propositions from AP. In line with the standard literature, an infinite path in a model is said to be *fair* if all the Boolean formulae from the set of fairness conditions are true infinitely often along the path.

MCMAS implements the standard algorithms for the verification of temporal operators under fairness [15]. When the fairness options are enabled all operators are evaluated on the set of fair paths as discussed in [9].

4.1.5 Witnesses, counterexamples and strategy synthesis

To assist with advanced validation support, MCMAS enables the user to inspect witnesses and counterexamples. These are provided as *trees*, instead of traces, adopting the algorithm described in [16]. MCMAS further extends [16] by:

1. by generating *fair* counterexamples and witnesses upon request. This is done by integrating the algorithm described in [14] for the temporal operators with the generation of the counter-example tree;
2. by including additional cases for the additional operators. Algorithm 11 generates a counterexample for the formula $K_i\phi$ by picking a state *not* satisfying ϕ (the remaining epistemic operators are treated in a similar way). A similar algorithm can be used to compute a counterexample for $K_i\phi$ under fairness by replacing SAT_ϕ is replaced by SAT^F_ϕ and S by S^F in Algorithm 11. Algorithm 12 builds a witness for a formula of the form $\langle\langle\Gamma\rangle\rangle X\phi$ by computing the set Z of states in which agents not in Γ can move to a state satisfying $\neg\phi$, and returns an element from its complement. Algorithm 13, instead, returns a witness for a formula of the form $\langle\langle\Gamma\rangle\rangle\phi_1 U \phi_2$. This algorithm and a similar one for formulae of the form $\langle\langle\Gamma\rangle\rangle G\phi$ (not reported here) follow the standard procedure for Until and Globally operators of CTL (see [15] for additional details).

Algorithm 11 Counterexample for $K_i\phi$ in state s.

```
$X = SAT_{\neg\phi}$;
$Y = \{s' \in S \mid s \sim_i s'\}$;
pick $s'$ in $Y \cap X$ and return $s'$
```

Algorithm 12 Witness for $\langle\langle\Gamma\rangle\rangle X\phi$ in state s.

```
$X = SAT_{\neg\phi}$;
$Y = \{(s, a_\Gamma) \mid \exists s' \in S \text{ such that } (s, a, s') \in T\}$;
$Z = \{(s, a_\Gamma) \mid \exists s' \in X \text{ such that } (s, a, s') \in T\}$;
pick $(s, a_\Gamma)$ in $Y \backslash Z$ and return $\{s' \in S \mid (s, a, s') \in T\}$
```

Algorithm 13 Witness for $\langle\langle\Gamma\rangle\rangle\phi_1 U \phi_2$ in state s.

```
$P = SAT_{\phi_1}$; $Q = SAT_{\phi_2}$; $X = \neg(P \cup Q)$;
$W = \{s\} \backslash Q$; $V = \{s\}$;
while $W \neq \emptyset$ do
    $Y = \{(s', a_\Gamma) \mid s' \in W \wedge \exists s'' \in S \text{ such that } (s', a, s'') \in T\}$;
    $Z = \{(s', a_\Gamma) \mid s' \in W \wedge \exists s'' \in X \text{ such that } (s', a, s'') \in T\}$;
    $Y = Y \backslash Z$;
    pick $(\overline{s}, \overline{a}_\Gamma)$ in $Y$ and $Y_1 = \{s'' \in S \mid (\overline{s}, \overline{a}, s'') \in T\}$;
    $W = Y_1 \backslash (V \cup Q)$; $V = V \cup Y_1$;
end while
return $V$
```

Notice that, in line with [1], MCMAS currently generates ATL witnesses and counterexamples *without* fairness, as this would require double-exponential algorithms and cause visualisation problems for the traces obtained.

Finally, it is customary for model checkers to compute counterexample for universal formulae that do not hold in a model, and witnesses for existential formulae that are true in a model. MCMAS extends this approach as follows (all formulae are assumed to be in negation normal form):

- MCMAS generates a counterexamples for universal formulas containing an existential formula. For instance, if the formula $AFEG\phi$ is false, MCMAS generates a counterexample consisting a loop reachable from the initial state in which $EG\phi$ does not hold.
- MCMAS generates a witness for existential formulas containing a universal formula. For instance, if the formula $EFAG\phi$ is true, MCMAS generates a witness consisting of a path leading from the initial state to a state in which $AG\phi$ holds.
- MCMAS generates witnesses or counterexamples for Boolean combinations of either universal or existential formulae (including the cases mentioned above).

MCMAS also provides limited support for *strategy synthesis* for ATLK specifications. Strategy synthesis is the problem of deducing the strategies the agents need to employ so that a given specification is satisfied. Given an interpreted system and an ATLK specification over a set of agents in Γ, if the specification is satisfied, MCMAS can often return a witness model. This witness contains the actions that every agent in Γ performs at various states thereby making the formula true. The strategies of the agents in Γ can therefore be synthesised from the output. Note that the synthesised strategies are guaranteed to be protocol compliant due to how the model is generated.

4.1.6 Consistency checking

It is known that bounded integer types may generate overflows in a reachable state. Due to the encoding strategy adopted in MCMAS, the value of a variable exceeding its upper bound is truncated to a value within its domain. This might lead to unexpected behaviours. To avoid this MCMAS allows users to verify the presence of overflows as an optional check. This is carried out by encoding the transition relation $T_{overflow}$ in such a way that an expression on the right-hand side of an assignment is assumed to be beyond the bounds of the left-hand side variable, and then by constructing the conjunction of $T_{overflow}$ and the reachable states R. A non-empty conjunction indicates the existence of an overflow.

The ISPL semantics requires that each state must have a successor state. So, checking deadlock states that do not have successor states is necessary to guarantee correct results of model checking, as deadlock states violate the premises of the model checking algorithm for EG. Checking the presence of a deadlock state is implemented by model checking the formula $EG\ true$ on the model. If the formula does not hold in all states, then there exists a deadlock state.

4.2 MCMAS usage

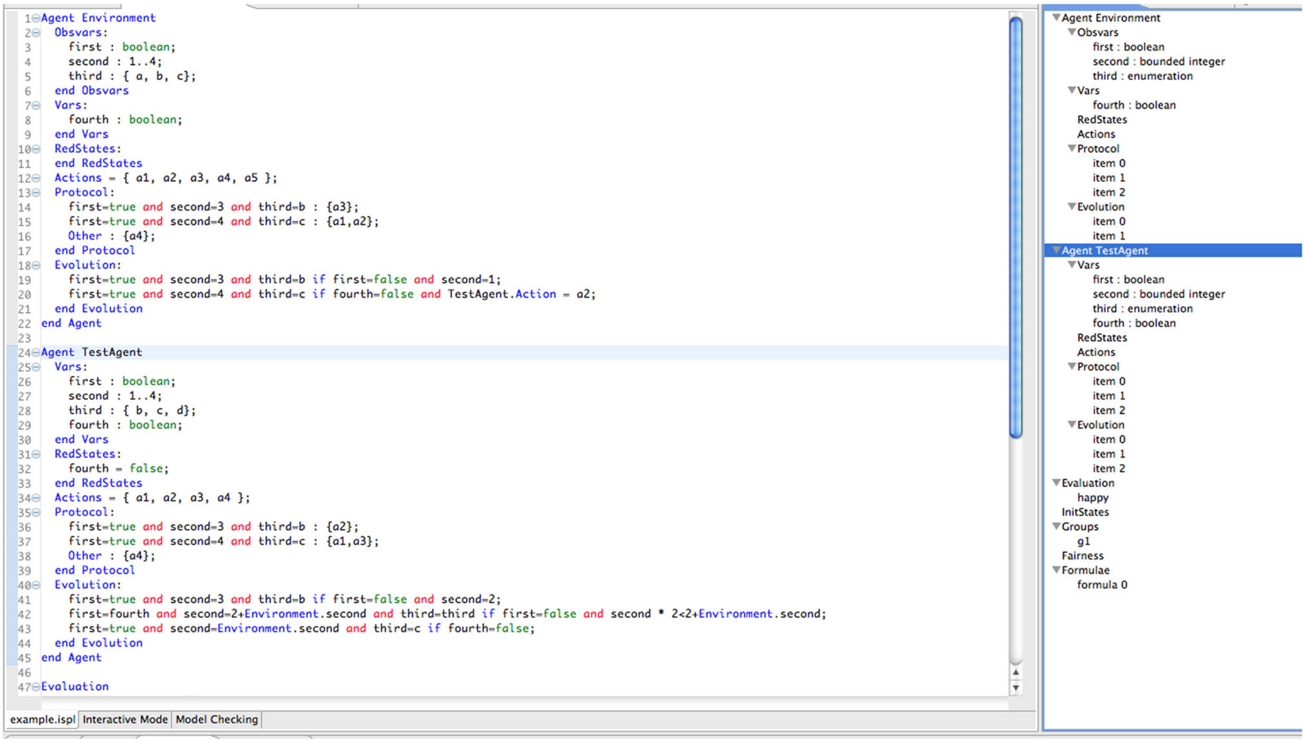
In its basic form the executable file `mcmas` is a standard command-line tool that takes as input the ISPL model to be verified. The executable accepts a number of command-line options, including:

- The options `-o`, `-g`, and `-d` are used, respectively, to select the algorithm to be used to order OBDD variables, to group OBDD variables, and to disable dynamic OBDD reordering.
- The option `-e` is used to select the algorithm to be used to generate the reachable state space (see Algorithm 10).
- The options `-k` and `-a` are used to check for deadlocks and arithmetic overflows in the model.
- The option `-c` is used to select the way in which counterexamples and witnesses are displayed. If this option is selected, the user can also tune the generation of counterexamples and witnesses using additional parameters provided using the options `-p`, `-f`, `-l`, and `-w` (we refer to the online documentation for a detailed description of these).
- The option `-uniform` is used to force the generation of uniform models as described in Sect. 2.2.

MCMAS can also be used to perform *interactive simulations*: from the command line, this is achieved by invoking the tool with the `-s` option.

To improve usability, a graphical interface is available. The GUI is implemented as an Eclipse plug-in. The plug-in creates a new menu in Eclipse that guides the interaction. Currently, this interface supports the following features:

- *ISPL program editing* This helps users create MCMAS projects and ISPL program skeletons; it implements syntax highlighting; it performs dynamic syntax checking (a separate ISPL compiler was implemented in Java + ANTLR [58] for this); it provides an outline view and the synchronisation between the outline view and the ISPL editor; it also supports text formatting and content assist.
- *Interactive execution mode* This mode implements the `-s` option described above to perform interactive simulations. It allows users to execute their model step by step by selecting an initial state first, and subsequently by choosing a successor state among those reachable via the enabled transitions. Users can backtrack an execution to the beginning at any step. Simulations can be



```
Agent Environment
  Obsvars:
    first : boolean;
    second : 1..4;
    third : { a, b, c};
  end Obsvars
  Vars:
    fourth : boolean;
  end Vars
  RedStates:
  end RedStates
  Actions = { a1, a2, a3, a4, a5 };
  Protocol:
    first=true and second=3 and third=b : {a3};
    first=true and second=4 and third=c : {a1,a2};
    Other : {a4};
  end Protocol
  Evolution:
    first=true and second=3 and third=b if first=false and second=1;
    first=true and second=4 and third=c if fourth=false and TestAgent.Action = a2;
  end Evolution
end Agent

Agent TestAgent
  Vars:
    first : boolean;
    second : 1..4;
    third : { b, c, d};
    fourth : boolean;
  end Vars
  RedStates:
    fourth = false;
  end RedStates
  Actions = { a1, a2, a3, a4 };
  Protocol:
    first=true and second=3 and third=b : {a2};
    first=true and second=4 and third=c : {a1,a3};
    Other : {a4};
  end Protocol
  Evolution:
    first=true and second=3 and third=b if first=false and second=2;
    first=fourth and second=2+Environment.second and third=third if first=false and second * 2<2+Environment.second;
    first=true and second=Environment.second and third=c if fourth=false;
  end Evolution
end Agent

Evaluation
```

Fig. 10 The ISPL Editor (Eclipse plug-in)

performed either explicitly, i.e. without OBDD encoding, or symbolically. The explicit simulation is performed by the Eclipse plug-in and does not require interaction with MCMAS. The symbolic simulation is more appropriate for large models and requires the installation of MCMAS.

- *Verification* In this modality the GUI invokes MCMAS to execute the model checking procedures. Counterexamples and witness traces are displayed in a graphic way using the `dot` utility from the Graphviz package [27]; states are shown as nodes and transitions as edges. When the mouse is rolled over a node in the graph, the corresponding state is highlighted. The executions can be projected onto a subset of agents to mask unwanted information of other agents.

A screenshot of the Eclipse-based ISPL editor with syntax highlighting is shown Fig. 10. A simple counterexample for a formula involving ATL operators is shown in Fig. 11.

5 Scenarios and applications

In the past 10 years, MCMAS has been used to verify a wide number of systems ranging from simple multi-agent systems protocols to industrial scenarios [21–23,28,29,44,48–50]. In the following, we summarise a few instructive examples, but refer to the references above for more details.

The ISPL encodings for the scenarios below were either manually given, or automatically generated by means of dedicated compilers. While other traditional model checkers could be used to model the scenarios, as we will see below, the specifications checked are based on epistemic or ATL formulas, hence normally not verifiable with traditional checkers.

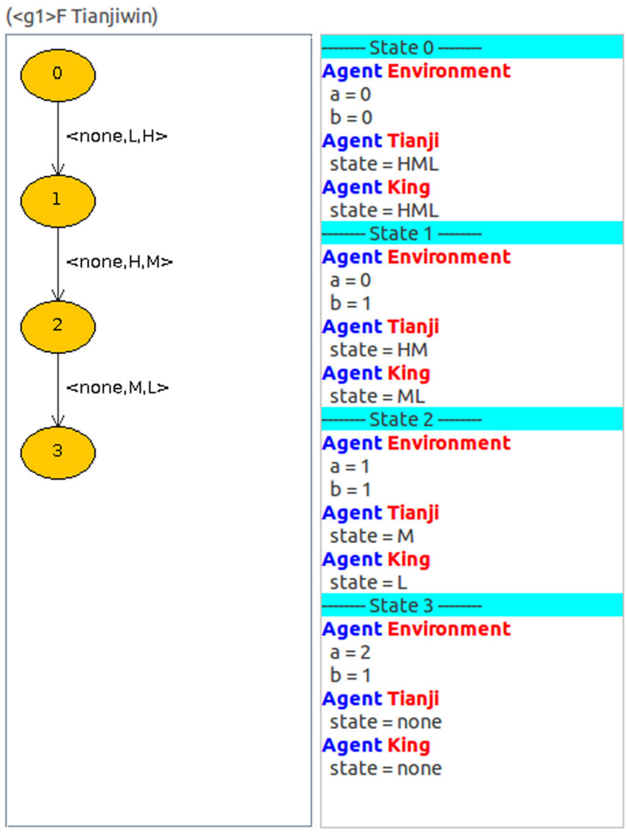
We here focus on the tool functionality; the next section covers its performance when compared to other tools.

5.1 Example scenarios

We begin with simple scenarios from the artificial intelligence and MAS literature and then move on to describe more complex use cases.

5.1.1 The muddy children puzzle

The muddy children puzzle [24] concerns a group of n children out playing in the field; k of them get mud on their forehead. They sit in a circle and see only the foreheads of other children, but not their own. An adult arrives and announces “At least one of you is muddy!”; he pauses and then asks: “Does anyone of you know whether you are muddy?”. The adult repeats this question over and over. After each question, the children, assumed to be perfect reasoners, answer “I do not know”. By reasoning about what children



```
(<g1>F Tianjiwin)
------- State 0 -------
Agent Environment
 a = 0
 b = 0
Agent Tianji
 state = HML
Agent King
 state = HML
------- State 1 -------
Agent Environment
 a = 0
 b = 1
Agent Tianji
 state = HM
Agent King
 state = ML
------- State 2 -------
Agent Environment
 a = 1
 b = 1
Agent Tianji
 state = M
Agent King
 state = L
------- State 3 -------
Agent Environment
 a = 2
 b = 1
Agent Tianji
 state = none
Agent King
 state = none
```

Fig. 11 A model witness (Eclipse plug-in)

know and what they learn from each announcement it is possible to show that at round k the muddy children say that they know they are muddy.

MCMAS can be used to verify that the children give the correct answer at each round, and have the correct knowledge at the end of the protocol. To do this we encode each child by means of an agent and use a set of initial states to generate a random assignment of muddy foreheads. We can then, for example, verify the specification stating that when a muddy child speaks, he indeed knows that he is muddy. This is captured by the formula below:

$$AG((\text{muddy}_i \wedge \text{saysknows}_i) \rightarrow K_{\text{Child}_i}\text{muddy}_i)$$

where the proposition “saysknows_i” is true after child i says that he knows whether or not he is muddy.

Given the small size of the state space per child, the analysis can be performed on a very large number of children. Keeping track of the number of round of questions is not problematic but leads to a larger state space.

To evaluate the performance of MCMAS and the speed difference among various OBDD orderings, we ran MCMAS on a sequence of models increasing the number of children. All the experiments below and later were carried out on an AMD Phenom 9600B 2.3 GHz Processor with 8 GB memory running Linux kernel version 3.11.0-18. Any run that exceeded 24 h is reported as a timeout.

Table 1 shows that MCMAS is able to handle a large number of children. The default OBDD ordering works well up to 100 children; ordering 3 and 4 are more efficient for the larger models.

5.1.2 One hundred prisoners

This puzzle concerns 100 prisoners kept in solitary confinement [19]. One day the warden gathers all the prisoners in the dining hall for dinner and announces that from the following day he will randomly choose one prisoner every day for questioning. The interrogation room has only a light governed by a toggle switch. Prisoners can observe whether the light is on when they enter the room. During their visit they are allowed to switch the light on or off as they please. While in the interrogation room, a prisoner may announce that he believes that all prisoners have already visited the interrogation room. If a prisoner makes this announcement and this corresponds to the truth, then all prisoners are set free; if the announcement is not correct, all prisoners are executed. The prisoners are granted one meeting to coordinate their actions before the interrogations begin. It is assumed that prisoners can count days and that the light is initially off.

To check whether the puzzle's solution is correct, we can model the prisoners' behaviours, corresponding to the solution of the puzzle, in ISPL and verify the formula

$$\langle\langle \text{Prisoners} \rangle\rangle F(\text{Release}) \qquad (1)$$

assuming Release to be true on the model only if all prisoners are free. MCMAS finds the formula to be true thereby confirming the correctness of the solution. Table 2 reports the performance of MCMAS on this example. As in the muddy children puzzle, we found that the default OBDD ordering is the most efficient when the model is small, while other orderings offer better performance on larger models.

5.1.3 Tian Ji racing horses

As documented above, differently from other tools, MCMAS can also be used to synthesise strategies by inspecting a model witness of an ATLK formula. In the Tian Ji puzzle, a king and a general called Tian Ji have a yearly horse racing competition. Both the king and the general have three horses, which are assumed to have *high*, *medium*, and *low* speed. The rules of the competition prescribe three rounds; in each round the king and Tian Ji should use the horse with the same speed. The overall winner is the person who wins the

Table 1 Experimental results for muddy children

Number of children	Number of OBDD vars	Time (s)			
		OBDD ordrng 1	OBDD ordrng 2	OBDD ordrng 3	OBDD ordrng 4
20	111	0.61	0.50	0.76	0.76
40	213	11.46	6.57	8.76	10.68
60	313	40.29	39.58	54.16	97.67
80	415	165.62	144.44	190.46	192.49
100	515	736.04	431.85	466.05	385.31
120	615	920.02	1036.89	784.44	787.35

Table 2 Experimental results for 100 prisoners

Number of prisoners	Number of OBDD vars	Time (s)			
		OBDD ordrng 1	OBDD ordrng 2	OBDD ordrng 3	OBDD ordrng 4
5	40	0.40	0.34	0.41	0.43
10	88	13.79	13.00	11.50	11.48
15	123	118.23	122.57	126.75	126.93
20	161	1573.96	1692.87	1636.44	913.02
25	196	6541	8682	6083	7107

highest number of runs. However, for any given speed, the king's horses always run faster than the general's. Therefore, Tian Ji is always bound to loose the race, unless he cheats and employs horses in a different way. Tian Ji knows that the king will run the horses in the three races from the fastest to the slowest. MCMAS can be used to find the winning strategy for the general: we encode the king and the general by means of two agents. We fix the protocol for the king, and we allow the general to choose any horse at any given level. We can then inspect the winning strategy for Tian Ji by analysing the witness for the formula:

$$\langle\langle \text{TianJi} \rangle\rangle F(\text{win}) \qquad (2)$$

Figure 12 illustrates the state space for the example. In each state the general's available horses are labelled with capital letters; the king's horses are labelled with lower case letters. The current score is written as a pair where the first number stands for the general's wins and the second for the king's. The horses participating in each round are labelled with transitions. The general's strategy is shown as a solid line. The leaves of the tree represent the number of victories for the general Tian Ji and the king, respectively.

The experimental results for various number of horses are reported in Table 3. On this scenario, the OBDD ordering 1 offers the best performance on large models. It is worth pointing out that the running time for the model with 40 horses is shorter than that with 35 horses under the first OBDD ordering. This is because the former model has better structural regularity, which makes the OBDD operations more efficient.

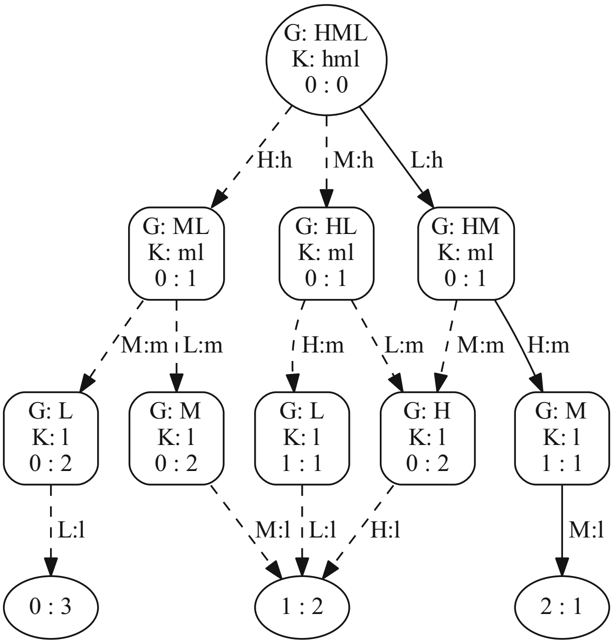


Fig. 12 The general's strategy in the Tian Ji puzzle

5.2 Applications

We now turn to larger applications and use cases outside the domain of multi-agent systems.

Table 3 Experimental results for racing horses

Number of children	Number of OBDD vars	Time (s)			
		OBDD ordrng 1	OBDD ordrng 2	OBDD ordrng 3	OBDD ordrng 4
5	39	0.09	0.03	0.03	0.02
10	65	0.47	0.34	0.35	0.47
15	85	5.66	3.17	2.86	5.62
20	111	50.22	259.95	37.78	37.67
25	131	179.48	589.59	148.47	145.36
30	151	1000.06	1612.75	1011.45	1001.54
35	177	2449.73	2307.21	525.21	2078.08
40	197	224.21	329.09	Timeout	3278.78

5.2.1 Verification of authentication protocols

Authentication protocols are a class of security protocols whereby two or more agents need to acquire knowledge of their identity, typically to initiate secure communication. Authentication protocols are notoriously difficult to analyse, due to the possible existence of subtle bugs such as man-in-the-middle attacks and impersonation. Formal models have been employed to analyse authentication protocols; however, they are often limited to reachability analysis only, thereby imposing rather severe limitations on the class of specifications that can be verified. Specifications concerning authentication are amenable to be expressed in a temporal-epistemic logic language as they concern states of knowledge of the principals in a system.

Authentication protocols expressed in CAPSL, a mainstream language for the description of security protocols, were automatically verified with MCMAS in [5]. Specifically, a compiler was built to translate protocols from the Clark–Jacobs and SPORE repository [62] into MCMAS readable input. This involved devising a translation from the SPORE protocol description into ISPL and a translation from the SPORE protocol specifications (“goals” in CAPSL) into appropriate temporal-epistemic formulas. The methodology is completely automatic and was evaluated on well-known key establishment protocols. The tool confirmed bugs already known in some key establishment protocols and verified the correctness of others. Since MCMAS also provides counterexamples when a specification is false, an attack could easily be derived by inspecting the output of the checker.

The performance of the methodology was in line with state-of-the-art model checkers for security protocols. It has been argued, see, e.g. [8,31], that epistemic specifications are considerably more intuitive for the security analyst as they refer precisely to the states of knowledge of the principals, which are the basic primitive in security analysis. This increase in expressiveness allows also the validation of other specifications, including the distributed detection of attacks at run-time [6]. Lazy intruder models are known to increase the effectiveness of model checking approaches for security protocols [3]; they can also be applied in the context of security specifications [46].

5.2.2 Verification of anonymity protocols

Anonymity protocols are a class of protocols aimed at establishing the privacy of principals during an exchange. For example, the onion routing protocol can guarantee the communication between two parties without their identities being revealed to any third party.

Onion routing and a number of other anonymity protocols have been analysed with MCMAS. One well-known example in this class is the dining cryptographers protocol [12], which can be described as follows. A group of n cryptographers shares a meal around a circular table. Either one of them paid for the meal or their employer did. They would like to discover whether one of them paid without revealing the identity of the payer (in case one of them did pay). To this end, every cryptographer tosses a coin and shows the outcome to his right-hand neighbour. Comparing his own coin to the coin shown to him, each cryptographer who did not pay for dinner announces whether the two coins agree or not. However, if a cryptographer paid for the meal, he announces the opposite of what he sees. By parity considerations, one can show that an even number of cryptographers claiming that the two coins are different entails that their employer paid for the dinner, while an odd number of “different” utterances signifies that one of the cryptographers paid for the dinner. The protocol can naturally be formalised in ISPL. The specification of the protocol can very easily be captured in an epistemic setting: after the announcements the cryptographers acquire common knowledge of the payer; but if this is one of them no one knows who this is, other than the payer itself. This analysis cannot be replicated on traditional checkers. We refer to Sect. 6.2 for more details on this scenario including the epistemic specifications checked.

5.2.3 Automatic verification of WS-BPEL services

Web services consist of distributed, networked applications exchanging messages to perform a given function. A key problem in service-oriented computing is to design and manage the service composition, whereby two or more services collaborate to achieve a certain task. Model checking has been used in this context to verify whether or not services are composed correctly according to some temporal specifications. A common approach involves modelling the services as finite-state machines and verify their composition by means of a traditional checker.

In this context, to avoid the difficulties and the error-prone task of manually encoding several services, a compiler from WS-BPEL into ISPL was built [51]. WS-BPEL [57] is the leading language and de facto industrial standard for service composition. The compiler parses the input and constructs an internal automata-based representation for the finite-state machines defined in the WS-BPEL code and encodes these in ISPL. The resulting code can then be passed to MCMAS for verification.

This methodology was evaluated in the context of a large use case of software procurement developed within a collaborative EU project. In the use case a client company places an order for software and hardware to be deployed by a number of parties. Several providers propose the equipment to the company which is given the opportunity of changing the design a number of times before deployment. Penalties are applied to the parties should they deviate from the procurement process. The interaction continues into the integration and testing phases of the hardware with additional contracts regulating the extent to which modifications can be requested by the client, compensation claims, reports from technical experts and insurance providers. The reachable state space of the scenario consists approximately of 10^6 states.

In the scenario, a number of specifications pertaining to the knowledge of the parties in the exchange can be verified. For example, it can be shown that as long as the client is not in breach of contract, he knows that either the system is installed correctly or some penalty to third parties will apply. This and other specifications can be verified in a few seconds. We refer to [51] and the source code for additional details.

Other scenarios from services and business process modelling were similarly investigated by means of MCMAS. We found that scenarios generating state spaces up to 10^{12} could be analysed with no difficulty.

6 Related work

The first version of MCMAS was first made available as open-source in 2003. In the past 10 years, the checker underwent a number of extensions and revisions that lead to a first documented release in [53] and a second in [47]. Unsupported, experimental releases continue and include a module to perform parameterised verification [41,42], and dedicated for the verification of strategy logic specifications [10,11]. This area is fast evolving; in the past few years a number of checkers have appeared that offer functionalities related to those offered by MCMAS. We compare the various functionalities and, when possible, the performance of the most prominent ones below.

6.1 Verics

Verics is a SAT-based model checker targeting real-time specifications and multi-agent systems [39]. Verics implements bounded model checking algorithms for CTL, real-time CTL, and variants of CTL that include epistemic operators. Bounded model checking and OBDDs represent radically different approaches to model checking; the former is based on a partial exploration of the model and the check made on a propositional encoding of the problem; the latter, as it was discussed here, relies on the full exploration of the model. Several comparisons were made between Verics and MCMAS, see, e.g. [38]. The conclusions drawn from them are that the techniques appear complementary with different advantages and disadvantages depending on the specific scenario considered. In terms of usability MCMAS's input language is likely to be more intuitive to researchers familiar with MAS semantics and applications. Conversely, Verics's input format may be more appealing to users familiar with real-time systems. In a related line, OBDDs and bounded model checking have recently been combined in an extension of MCMAS in [36], producing very attractive experimental results and thereby demonstrating that the two techniques can usefully be combined in a wide range of scenarios. This conclusion was also recently reached in [56] where an OBDD-based approach for bounded model checking for LTLK was developed on Verics. The results obtained cannot be compared to the ones reported here as the underlying temporal logics have different expressivity.

6.2 MCK

MCK was the first OBDD-based model checker supporting temporal-epistemic specification [26]; it has recently been re-released with improved functionalities including a graphical interface [67]. Its current version supports CTL* as the underlying temporal logic. MCK implements a variety of semantics including observational semantics, perfect recall and clock semantics. Given the high computational cost of these semantics, some are supported only in limited form; for example perfect recall is only supported for one agent. Some functionality for probabilistic reasoning was also recently added [33] and an extension to bounded model checking has

Table 4 Experimental results comparing the performance of MCMAS, MCK, and MCTK on the dining cryptographer protocol

Number of cryptographers	MCMAS		MCK		MCTK	
	OBDD vars ($9n + 3$)	Time (s)	OBDD vars ($6n + 2$)	Time (s)	OBDD vars ($6n + 2$)	Time (s)
5	48	0.017	32	1.401	32	0.024
10	93	0.091	62	74.655	62	0.128
20	183	0.667	122	47937	122	34.790
30	273	1.476	Timeout		182	2.946
40	363	5.053	Timeout		242	20.786
50	453	13.437	Timeout		302	72.444
60	543	14.180	Timeout		Timeout	

also recently been explored [34]. While the original version of MCK used a different OBDD-handler, the current one uses CUDD as MCMAS.

To compare MCK (and MCTK, as we discuss below) to MCMAS uniformly, we used the dining cryptographer benchmark [12] discussed in [38]. We instructed each model checker to verify that the protocol maintains the privacy of the payer as intended. Specifically, we verified that if cryptographer 1 did not pay the bill, then, after the announcements are made, either he knows that no cryptographers paid, or that someone paid, but in this case he does not know who did. This can be formalised by the following formula

$$
\begin{aligned}
&AG\Big(\big(\bigwedge_{i=1}^{n} c_i\_announced \wedge \neg c_1\_paid\big) \\
&\rightarrow \Big(K_{c_1}(\bigwedge_{i=1}^{n} \neg c_i\_paid) \\
&\vee \big(K_{c_1}(\bigvee_{i=2}^{n} c_i\_paid) \wedge \bigwedge_{i=2}^{n} \neg K_{c_1}(c_i\_paid)\big)\Big)\Big), \qquad (3)
\end{aligned}
$$

where n is the number of cryptographers, $c_i\_announced$ represents the announcement made by cryptographer i, and c_i_paid encodes the fact that cryptographer i paid the bill.

In the experiments, we used encodings of the scenario from both the MCMAS and MCK packages and adapted them to ensure the same number of variables were used in each model. In the experiments, we used observational semantics since several agents are present and no model checker supports perfect recall under this setting.

The experiments were carried out on the same setup as previously reported. Table 4 shows the experimental results obtained with dynamic variable reordering enabled. The results reported are consistent with other experiments we have run.

In summary, both MCK and MCMAS offer functionalities to verify epistemic logic under different semantics. They differ in the modelling language employed as well as some advanced features supported. While MCMAS also supports ATL, MCK supports a probabilistic version of epistemic logic and a very limited form of perfect recall. They are both OBDD based. Our tests appear to suggest that MCMAS is more efficient in the treatment of large state spaces; this may be due to a more effective construction of the global state space.

6.3 MCTK

MCTK is a NuSMV-based [13] model checker for knowledge and time [63,69]. In MCTK, epistemic formulas are encoded by exploiting locality of propositions and labelling of transitions. As such, it does not support interpreted systems semantics which is a feature of MCMAS. A model checker for knowledge based on NuSMV with similar characteristics was previously presented in [45]. Note that NuSMV is among the fastest and most mature OBDD-based model checker available. As above, we compared the performance of MCTK to that of MCK and MCMAS on the same example above by adapting the encoding of the dining cryptographers protocol to ensure the same state space is present. In our tests, we found MCTK to be considerably slower than MCMAS due to the efficient implementation of the model checking algorithm for epistemic logic in MCMAS. It should be also noted that MCTK's input is given in SMV; this is adequate for modelling reactive systems, but may not be suitable for MAS where actions and protocols feature prominently. As a further consequence of this, no support for ATL is offered in MCTK. In addition, none of the debugging facilities present in MCMAS, including counterexample generations for epistemic specifications, are offered by MCTK.

6.4 NuSMV

To evaluate MCMAS in a broader context, we now report the results obtained by comparing MCMAS to NuSMV when checking the dining cryptographer benchmark and a game-theoretical scenario from [17] against *plain CTL* spec-

Table 5 Experimental results comparing the performance of MCMAS and NuSMV on the dining cryptographer problem

Number of cryptos	MCMAS					NuSMV				
	Num of OBDD vars	1.2.2 (CUDD 2.5.0)		1.0 (CUDD 2.4.1)		Num of OBDD vars	Without dyn reorder		With dyn reorder	
		Time (s)	Mem (KB)	Time (s)	Mem (KB)		Time (s)	Mem (KB)	Time (s)	Mem (KB)
10	93	0.10	11,056	0.10	10,972	62	0.87	16,060	0.09	12,668
20	183	0.65	13,572	0.58	13,504	122	3151.35	4,891,100	0.27	13,932
30	273	1.45	15,332	2.01	21,764	182	Overflow		2.05	17,040
50	453	12.98	16,076	10.06	40,708	302	Overflow		8.64	21,004
100	903	185.72	60,800	284.22	49,780	602	Overflow		117.77	42,516
150	1353	1916	72,172	1619	91,976	902	Overflow		397	68,176
200	1803	841.4	58,872	3057	76,608	1202	Overflow		2560	102,840
250	2253	3040	80,680	8705	221,404	1502	Overflow		Timeout	

Table 6 Experimental results comparing the performance of MCMAS and NuSMV on the card game scenario from [17]

Number of cards	MCMAS					NuSMV				
	Num of OBDD vars	1.2.2 (CUDD 2.5.0)		1.0 (CUDD 2.4.1)		Num of OBDD vars	Without dyn reorder		With dyn reorder	
		Time (s)	Mem (KB)	Time (s)	Mem (KB)		Time (s)	Mem (KB)	Time (s)	Mem (KB)
8	59	0.37	11,592	0.41	11,552	56	0.16	17,772	1.58	15,880
10	97	7.61	38,712	12.47	38,376	94	507.63	2,441,688	1035.89	154,124
12	113	515.4	94,100	273.8	88,992	110	Overflow		Timeout	
14	129	17,783	1,191,864	10,552	539,552	126	Overflow		Timeout	

ifications. As in previous cases, to ensure the tests are robust, we inspected and compared the resulting models and state spaces.

The results are reported in Tables 5 and 6. Table 5 was constructed by checking the dining cryptographers implementations discussed earlier in Table 4 against the CTL formula:

$$AG\Big(\big(\bigwedge_{i=1}^{n} c_i\_announced \wedge \neg NSA\_paid\big) \rightarrow \bigwedge_{i=1}^{n} c_i\_paid\Big),$$

where NSA_paid stands for NSA (for which the cryptographers work) paying for the dinner.

Table 6 contains the result obtained when checking the scenario against the CTL specification

$$AG(\text{allred}_1 \rightarrow AF\ \text{win}_1),$$

which reports a winning condition for player 1. The scenario was scaled by considering the number of cards present.

Both MCMAS and NuSMV offer several features to fine-tune certain parameters in the model checking algorithms. We used the defaults for both of them and tested both with reordering enabled and disabled. Differently from NuSMV, which often offers a better performance without reordering, MCMAS's state-space construction is heavily dependent on reordering; given this MCMAS's results are reported with this feature enabled. Indeed, Table 6 shows that NuSMV is faster and uses more memory when reordering is disabled. NuSMV could not verify the model with 12 cards due to memory overflow irrespective of whether or not reordering was enabled. In contrast, MCMAS could handle the cases for 12 and 14 cards before timing out with 16 cards. To present a fair comparison, we also linked MCMAS version 1.0 to CUDD 2.4.1, which is the version used by NuSMV version 2.5.4[1] [70]. Table 6 shows the results from both versions of MCMAS.

The tables above are not intended to give a comprehensive performance evaluation of the two checkers. They are purely meant to show that MCMAS's performance is broadly in line with NuSMV, one of the most commonly used symbolic model checkers. We expect NuSMV to be faster than MCMAS on other models not tested here.

[1] MCMAS version 1.2.2 does not support CUDD 2.4.1, as the C++ interface between CUDD 2.5.0 and CUDD 2.4.1 is considerably different.

7 Conclusions

The continuous rise in the number of autonomous systems that are being deployed has made formal verification of multi-agent systems a very active area of research. In this paper, we have presented the toolkit MCMAS, a model checker supporting specifications tailoring multi-agent systems. We have discussed the details of the underlying semantics, its input language, the functionalities offered and evaluated it in the context of significant use cases and other checkers.

MCMAS is released as GNU GPL open-source software and is currently used in a number of projects worldwide [21, 28,29,44]. Several extensions are currently being developed by various groups. Many of these extensions already have in-house prototypes featuring, for example, abstraction, symmetry detection, and combinations with bounded model checking. This paper does not address these new features but instead focuses on the core, underlying technology of the MCMAS checker.

Acknowledgments The authors would like to thank Jakub Michaliszyn and the anonymous reviewers for valuable feedback on earlier versions of this paper.

References

1. Alur, R., Henzinger, T., Mang, F., Qadeer, S., Rajamani, S., Tasiran, S.: MOCHA: modularity in model checking. In: Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98), vol. 1427 of Lecture Notes in Computer Science, pp. 521–525. Springer (1998)
2. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. J. ACM **49**(5), 672–713 (2002)
3. Basin, D., Mödersheim, S., Viganò, L.: OFMC: a symbolic model checker for security protocols. Int. J. Inf. Secur. **4**(3), 181–208 (2005)
4. Baukus, K., van der Meyden, K.: A knowledge based analysis of cache coherence. In: Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM04), vol. 3308 of Lecture Notes in Computer Science, pp. 99–114. Springer (2004)
5. Boureanu, I., Cohen, M., Lomuscio, A.: A compilation method for the verification of temporal-epistemic properties of cryptographic protocols. J. Appl. Non Class. Log. **19**(4), 463–487 (2009)
6. Boureanu, I., Cohen, M., Lomuscio, A.: Model checking detectability of attacks in multiagent systems. In: Proceedings of the 9th International Conference on Autonomous Agents and Multi-Agent systems (AAMAS10), pp. 691–698. IFAAMAS Press (2010)
7. Bryant, R.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986)
8. Burrows, M., Abadi, M., Needham, R.: A logic of authentication. Proc. R. Soc. Lond. A **426**(1871), 233–271 (1989)
9. Busard, S., Pecheur, C., Qu, H., Raimondi, F.: Reasoning about strategies under partial observability and fairness constraints. In: Proceedings of the 1st International Workshop on Strategic Reasoning (SR13), vol. 112 of Electronic Proceedings in Theoretical Computer Science, pp. 71–79 (2013)
10. Čermák, P., Lomuscio, A., Mogavero, F., Murano, A.: MCMAS-SLK: a model checker for the verification of strategy logic specifications. In: Proceedings of the 26th International Conference on Computer Aided Verification (CAV14), vol. 8559 of Lecture Notes in Computer Science, pp. 525–532. Springer (2014)
11. Cermák, P., Lomuscio, A., Mogavero, F., Murano, A.: Verifying and synthesising multi-agent systems against one-goal strategy logic specifications. In: Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI15), pp. 2038–2044. AAAI Press (2015)
12. Chaum, D.: The dining cryptographers problem: unconditional sender and recipient untraceability. J. Cryptol. **1**(1), 65–75 (1988)
13. Cimatti, A., Clarke, E. M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV2: an open-source tool for symbolic model checking. In: Proceedings of the 14th International Conference on Computer Aided Verification (CAV02), vol. 2404 of Lecture Notes in Computer Science, pp. 359–364. Springer (2002)
14. Clarke, E.M., Grumberg, O., McMillan, K.L., Zhao, X.: Efficient generation of counterexamples and witnesses in symbolic model checking. In: Proceedings of the 32nd annual ACM/IEEE Design Automation Conference (DAC95), pp. 427–432. ACM Press (1995)
15. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
16. Clarke, E.M., Jha, S., Lu, Y., Veith, H.: Tree-like counterexamples in model checking. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS02), pp. 19–29. IEEE Computer Society (2002)
17. Cohen, M., Dam, M., Lomuscio, A., Russo, F.: Abstraction in model checking multi-agent systems. In: Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS09), pp. 945–952. IFAAMAS Press (2009)
18. Cohen, P.R., Levesque, H.J.: Intention is choice with commitment. Artif. Intell. **42**(2–3), 213–261 (1990)
19. Dehaye, P.O., Ford, D., Segerman, H., Vakil, R.: One hundred prisoners and a lightbulb. Math. Intell. **25**(4), 53–61 (2003)
20. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. In: Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP98), pp. 7–15. ACM Press (1998)
21. El-Menshawy, M., Bentahar, J., El Kholy, W., Dssouli, R.: Verifying conformance of multi-agent commitment-based protocols. Expert Syst. Appl. **40**(1), 122–138 (2013)
22. Ezekiel, J., Lomuscio, A.: An automated approach to verifying diagnosability in multi-agent systems. In: Proceedings of the 7th IEEE International Conference on Software Engineering and Formal Methods (SEFM09), pp. 51–60. IEEE Computer Society (2009)
23. Ezekiel, J., Lomuscio, A., Molnar, L., Veres, S.: Verifying fault tolerance and self-diagnosability of an autonomous underwater vehicle. In: Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI11), pp. 1659–1664. AAAI Press (2011)
24. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning about Knowledge. MIT Press, Cambridge (1995)
25. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Knowledge-based programs. Distrib. Comput. **10**(4), 199–225 (1997)
26. Gammie, P., van der Meyden, R.: MCK: Model checking the logic of knowledge. In: Proceedings of the 16th International Conference

on Computer Aided Verification (CAV04), vol. 3114 of Lecture Notes in Computer Science, pp. 479–483. Springer (2004)
27. Gansner, E.R., North, S.C.: An open graph visualization system and its applications. Softw. Pract. Exp. **30**, 1203–1233 (1999)
28. Gerard, S.N., Singh, M.P.: Formalizing and verifying protocol refinements. ACM Trans. Intell. Syst. Technol. **4**(2), 21 (2013)
29. De Giacomo, G., Felli, P.: Agent Composition Synthesis based on ATL. In: Proceedings of the 9th International Conference on Autonomous Agents and Multi-Agent systems (AAMAS10), pp. 499–506. IFAAMAS Press (2010)
30. Halpern, J.Y., Pucella, R.: Modeling adversaries in a logic for security protocol analysis. In: Proceedings of the Workshop on Formal Aspects of Security (FASec02), vol. 2629 of Lecture Notes in Computer Science, pp. 115–132. Springer (2002)
31. Halpern, J.Y., van der Meyden, R.: A logical reconstruction of SPKI. J. Comput. Secur. **11**(4), 581–613 (2004)
32. Hintikka, J.: Knowledge and Belief, An Introduction to the Logic of the Two Notions. Cornell University Press, Ithaca (1962)
33. Huang, X., Luo, C., van der Meyden, R.: Symbolic model checking of probabilistic knowledge. In: Proceedings of the 13th Conference on Theoretical Aspects of Rationality and Knowledge (TARK11), pp. 177–186. ACM (2011)
34. Huang, X., Luo, C., van der Meyden, R.: Improved bounded model checking for a fair branching-time temporal epistemic logic. In: Proceedings of the 6th International Workshop on Model Checking and Artificial Intelligence (MoChArt10), vol. 6572 of Lecture Notes in Computer Science, pp. 95–111. Springer (2011)
35. Jones, A.J.I., Sergot, M.J.: On the characterisation of law and computer systems: the normative systems perspective. In: Deontic Logic in Computer Science: Normative System Specification, chap 12. Wiley (1993)
36. Jones, A.V., Lomuscio, A.: Distributed bdd-based bmc for the verification of multi-agent systems. In: Proceedings of the 9th International Conference on Autonomous Agents and Multi-Agent systems (AAMAS10), pp. 675–682. IFAAMAS Press (2010)
37. Jonker, G.: Feasible strategies in alternating-time temporal epistemic logic. Master's thesis, University of Utrech, The Netherlands (2003)
38. Kacprzak, M., Lomuscio, A., Niewiadomski, A., Penczek, W., Raimondi, F., Szreter, M.: Comparing BDD and SAT based techniques for model checking Chaum's dining cryptographers protocol. Fundamenta Informaticae **63**(2,3), 221–240 (2006)
39. Kacprzak, M., Nabialek, W., Niewiadomski, A., Penczek, W., Pólrola, A., Szreter, M., Wozna, B., Zbrzezny, A.: Verics 2007—a model checker for knowledge and real-time. Fundamenta Informaticae **85**(1–4), 313–328 (2008)
40. Konrad, S., Cheng, B.H.C.: Real-time specification patterns. In: Proceedings of the 27th International Conference on Software Engineering (ICSE05), pp. 372–381. ACM Press (2005)
41. Kouvaros, P., Lomuscio, A.: Automatic verification of parametrised interleaved multi-agent systems. In: Proceedings of the 12th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS13), pp. 861–868. IFAAMAS (2013)
42. Kouvaros, P., Lomuscio, A.: A cutoff technique for the verification of parameterised interpreted systems with parameterised environments. In: Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI13), pp. 2013–2019. AAAI Press (2013)
43. Kwiatkowska, M.Z., Norman, G., Parker, D.: Prism 4.0: verification of probabilistic real-time systems. In: Proceedings of the 23rd International Conference on Computer Aided Verification (CAV11), vol. 6806 of Lecture Notes in Computer Science, pp. 585–591. Springer (2011)
44. Latif, N.A., Hassan, M.F., Hasan, M.H.: Formal verification for interaction protocol in agent-based e-learning system using model checking toolkit-mcmas. In: Proceedings of the 2nd International Conference on Software Engineering and Computer Systems (ICSECS11), vol. 180 of Communications in Computer and Information Science, pp. 412–426. Springer (2011)
45. Lomuscio, A., Pecheur, C., Raimondi, F.: Verification of knowledge and time with nusmv. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI07), pp. 1384–1389. AAAI (2007)
46. Lomuscio, A., Penczek, W.: LDYIS: a framework for model checking security protocols. Fundamenta Informaticae **85**(1–4), 359–375 (2008)
47. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: A model checker for the verification of multi-agent systems. In: Proceedings of the 21th International Conference on Computer Aided Verification (CAV09), vol. 5643 of Lecture Notes in Computer Science, pp. 682–688. Springer (2009)
48. Lomuscio, A., Qu, H., Sergot, M.J., Solanki, M.: Verifying temporal epistemic properties of web service compositions. In: Proceedings of the 5th International Conference on Service-Oriented Computing (ICSOC'07), vol. 4749 of Lecture Notes in Computer Science, pp. 456–461. Springer (2007)
49. Lomuscio, A., Qu, H., Solanki, M.: Towards verifying compliance in agent-based web service compositions. In: Proceedings of the 7th International Conference on Autonomous Agents and Multi-Agent systems (AAMAS08), pp. 265–272. IFAAMAS Press (2008)
50. Lomuscio, A., Qu, H., Solanki, M.: Towards verifying contract regulated service composition. In: Proceedings of the 8th International Conference on Web Services (ICWS08), pp. 254–261. IEEE Computer Society (2008)
51. Lomuscio, A., Qu, H., Solanki, M.: Towards verifying contract regulated service composition. Auton. Agents Multi Agent Syst. **24**(3), 345–373 (2012)
52. Lomuscio, A., Raimondi, F.: The complexity of model checking concurrent programs against CTLK specifications. In: Proceedings of the 4th International Workshop on Declarative Agent Languages and Technologies (DALT06), vol. 4327 of Lecture Notes in Computer Science, pp. 29–42. Springer (2006)
53. Lomuscio, A., Raimondi, F.: MCMAS: A model checker for multi-agent systems. In: Proceedings of the 12th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS06), vol. 3920 of Lecture Notes in Computer Science, pp. 450–454. Springer (2006)
54. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems, vol. 1. Springer, New York (1992)
55. McCarthy, J.: Ascribing mental qualities to machines. In: Philosophical Perspectives in Artificial Intelligence. Harvester Press (1979)
56. Meski, A., Penczek, W., Szreter, M., Wozna-Szczesniak, B., Zbrzezny, A.: Bdd-versus sat-based bounded model checking for the existential fragment of linear temporal logic with knowledge: algorithms and their performance. Auton. Agents Multi Agent Syst. **28**(4), 558–604 (2014)
57. Organization for the Advancement of Structured Information Standards (OASIS): Web Services Business Process Execution Language (WS-BPEL) Version 2.0 (2007)
58. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Bookshelf, Raleigh (2007)
59. Penczek, W., Lomuscio, A.: Verifying epistemic properties of multi-agent systems via bounded model checking. Fundamenta Informaticae **55**(2), 167–185 (2003)
60. Raimondi, F., Lomuscio, A.: Automatic verification of multi-agent systems by model checking via ordered binary decision diagrams. J. Appl. Log. **5**(2), 235–251 (2007)
61. Somenzi, F.: CUDD: CU decision diagram package-release 2.5.0. http://vlsi.colorado.edu/fabio/CUDD (2012)
62. SPORE: Security protocols open repository. http://www.lsv.ens-cachan.fr/spore. Accessed 1 June 2014

63. Su, K., Sattar, A., Luo, X.: Model checking temporal logics of knowledge via OBDDs. Comput. J. **50**(4), 403–420 (2007)
64. Syverson, P.F., Stubblebine, S.G.: Group principals and the formalization of anonymity. In Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM99), vol. 1708 of Lecture Notes in Computer Science, pp. 814–833 (1999)
65. van der Hoek, W., Lomuscio, A., Wooldridge, M.: On the complexity of practical ATL model checking. In: Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS06), pp. 201–208. ACM Press (2006)
66. van Oorschot, P.: Extending cryptographic logics of belief to key agreement protocols. In: Proceedings of the 1st ACM conference on Computer and communications security (CCS93), pp. 232–243. ACM press (1993)
67. MCK. http://cgi.cse.unsw.edu.au/~mck/pmck/. Accessed 1 June 2014
68. MCMAS. http://vas.doc.ic.ac.uk/software/mcmas/. Accessed 1 June 2014
69. MCTK. http://sites.google.com/site/cnxyluo/MCTK/. Accessed 1 June 2014
70. NuSMV. http://nusmv.fbk.eu/. Accessed 1 June 2014
71. Wooldridge, M.: An Introduction to MultiAgent Systems, 2nd edn. Wiley, Hoboken (2009)