Advance Access publication on May 15, 2007 doi:10.1093/comjnl/bxm009

Model Checking Temporal Logics of Knowledge Via OBDDs[1]

Kaile Su[2,3], Abdul Sattar[3] and Xiangyu Luo[4,*]

[2]*School of Electronics Engineering and Computer Science, Peking University, Beijing, China*
[3]*Institute for Integrated and Intelligent Systems, Griffith University, Brisbane, Qld 4111, Australia*
[4]*Department of Computer Science, Guilin University of Electronic Technology, Guilin 541004, P.R. China*
**Corresponding author: luo.xiangyu@yahoo.com*

Model checking is a promising approach to automatic verification, which has concentrated on specification expressed in temporal logics. Comparatively little attention has been given to temporal logics of knowledge, although such logics have been proven to be very useful in the specifications of protocols for distributed systems. In this paper, we addressed the model checking problem for a temporal logic of knowledge (Halpern and Vardi's logic of *CKL_n*). Based on the semantics of *interpreted systems* with *local propositions*, we developed an approach to symbolic *CKL_n* model checking via Ordered Binary decision diagrams and implemented the corresponding symbolic model checker MCTK. In our approach to model checking specifications involving agents' knowledge, the knowledge modalities are eliminated via quantifiers over agents' non-observable variables. We then modelled the Dining Cryptographers protocol and the five-hands protocol for Russian Cards problem in MCTK. Via these two examples, we compare MCTK's empirical performance with two different state-of-the-art epistemic model checkers, MCK and MCMAS.

Keywords: model checking, temporal reasoning, reasoning about knowledge, temporal logics of knowledge, distributed AI

Received 14 June 2006; revised 4 December 2006

Handling editors: Zhi-Hua Zhou and Ruqian Lu

1. INTRODUCTION

Model checking is most widely understood as a technique for automatically verifying that finite state systems satisfy formal specifications. The success of model checking in mainstream computer science has led to a recent growth of interest in the use of the technology in fields of AI such as planning and multi-agent systems. However, the formal specifications for finite state systems are most commonly expressed as formulae of temporal logics such as linear temporal logic (LTL) in the case of SPIN [2] and FORSPEC [3] and Computation Tree Logic (CTL) in the case of SMV [4], while the specifications for multi-agent systems involve agents' knowledge, belief and other notions of agents' mental states. In this paper, we address ourselves to the model checking problem for Halpern and Vardi's logic of CKL_n [5], a temporal **l**ogic of **k**nowledge and **c**ommon knowledge with n agents.

The application of model checking within the context of the logic of knowledge was first mooted by Halpern and Vardi [6]. A number of algorithms for model checking epistemic specifications and the computational complexity of the related problems were studied in [7]. However, they did not investigate 'practical' model checking for knowledge and time.

Rao and Georgeff [8] investigated the model checking problem for situated reasoning systems, but they did not consider S5 logics of knowledge and they did not implement any of the techniques they developed. Benerecetti *et al.* [9, 10] developed techniques for some temporal modal logics, but these logics have an unusual (non-Kripke) semantics.

van der Meyden and Su [11] took a promising first step towards model checking of anonymity properties in formulas involving knowledge. Nevertheless, they took the assumptions that agents are of perfect recall and considered only a small class of epistemic formulas without any nesting of epistemic modalities.

van der Hoek and Wooldridge [12] developed an approach to reduce CKL_n model checking to LTL [13] model checking. However, the verification process of their approach still requires an input from a human verifier (to obtain the so-called local propositions when reducing the CKL_n specification to

[1]An earlier version of this paper appeared in *Proceedings of AAAI 2004* [1].

LTL). A 'direct' implementation of CKL_n model checking would thus be desirable.

Our approach presents a methodology for symbolic model checking, based CKL_n on the semantics of *interpreted systems* with *local propositions* [14], which leads to a 'direct' implementation of CKL_n model checking.

Moreover, by the results presented, we implement a symbolic *CKL_n* model checker that we call MCTK, which can also provide via Ordered Binary Decision Diagram (OBDD) [15] an approach to symbolic verifying CTL*, the combination of LTL and CTL. This is interesting because LTL and CTL have been well studied and implemented efficiently into a number of tools [2, 16] and the community of model checking expects such a tool that can verify specifications in full CTL* efficiently.

The present paper follows similar lines to [12], which is based on the idea of local propositions as described in [14, 17]. The following are the main advantages of the present paper over [12]:

(i) We explicitly introduce the notion of finite-state program with n agents (which is a symbolic representation of the well-known interpreted systems) and present some interesting results on the theoretical foundations of [12].

(ii) In order to determine whether $K_i\varphi$ holds at some point of an interpreted system, van der Hoek and Wooldridge [12] attempt to find an i-local proposition ψ, which is equivalent to $K_i\varphi$ at that point; whereas, we try to get an i-local proposition ψ, which is equivalent to $K_i\varphi$ at any point (see Remark 1).

The structure of the paper is as follows. In the next section, we shortly introduce the well-known interpreted system [18] and a temporal logic of knowledge, Halpern and Vardi's CKL_n [5]. Then, we define a class of interpreted systems that are generated by finite-state programs with n agents. In Section 3, the most exciting result is to show how to use OBDDs to implement symbolic CKL_n model checking, based on those interpreted systems generated by finite-state programs with n agents. Next, we introduce the implementation of MCTK in Section 4 and present two case studies for the Dining Cryptographers protocol and the Russian Cards problem, respectively, in Sections 5 and 6.

2. KNOWLEDGE IN AN INTERPRETED SYSTEM WITH LOCAL VARIABLES

In this section, we define the semantic framework within which we study the model checking of specifications in the logic of knowledge. First, we introduce interpreted systems [18] and a temporal logic of knowledge CKL_n (Halpern and Vardi's CKL_n [5]). Then, we present the notion of finite-state program with n agents, a finite-state transition representation for those interpreted systems with local variables.

2.1. Interpreted systems

The systems we are modelling are composed of multiple agents, each of which is in some state at any point of time. We refer to this as the agent's *local* state, in order to distinguish it from the system's state, the *global* state. Without loss of too much generality, we make the system's state a tuple $(s_1, \ldots, s_n)$, where s_i is agent i's state.

Let L_i be a set of possible local states for agent i, for $i = 1, \ldots, n$. We take $G \subseteq L_1 \times \ldots \times L_n$ to be the set of *reachable global* states of the system. A *run* over G is a function from the time domain—the natural numbers in our case—to G. Thus, a run over G can be identified with a sequence of global states in G. We refer to a pair (r, m) consisting of a run r and time m as a *point*. We denote the ith component of the tuple $r(m)$ by $r_i(m)$. Thus, $r_i(m)$ is the local state of agent i in run r at 'time' m.

The idea of the interpreted system semantics is that a run represents one possible computation of a system and a system may have a number of possible runs, so we say a *system* is a set of runs.

Assume that we have a set Φ of primitive propositions, which we can think of as describing basic facts about the system. An *interpreted system* $\mathcal{I}$ consists of a pair $(\mathcal{R}, \pi)$, where $\mathcal{R}$ is a set of runs over a set of global states and π is a valuation function, which gives the set of primitive propositions true at each point in $\mathcal{R}$ [18].

To define knowledge in interpreted systems, we associate with every agent i, an equivalence relation $\sim_i$ over the set of points [18]: $(r, u) \sim_i (r', v)$ iff $r_i(u) = r_i'(v)$.

If $(r, u) \sim_i (r', v)$, then we say that (r, u) and (r', v) are indistinguishable to agent i, or, alternatively, that agent i carries exactly the same information in (r, u) and (r', v).

To give a semantics to the 'common knowledge' among a group Γ of agents, two further relations, $\sim_\Gamma^E$ and $\sim_\Gamma^C$, are introduced [18]. We define the relation $\sim_\Gamma^E$ as $\cup_{i\in\Gamma} \sim_i$ and the relation $\sim_\Gamma^C$ as the transitive closure of $\sim_\Gamma^E$.

Notice that a system as a set of infinite runs seems not well suited to model checking directly as it is generally applied to the finite-state systems. In fact, we can represent an interpreted system as a *finite-state program* (G, G_0, R, V), where G_0 is a set of initial states, R a total 'next time' relation and V associates each state with a truth assignment function. A set $\mathcal{R}$ of infinite runs is then obtained by 'unwinding' the relation R starting from initial states in G_0. More specifically,

$$\mathcal{R} = \{r | r(0) \in G_0 \text{ and } r(m)Rr(m+1) \text{ for all } m\}$$

2.2. Semantics

Given a set Φ of primitive propositions, we use *Prop* to denote the set of all propositional formulas over Φ.

The LTL [19] is propositional logic augmented by the future-time connectives $\bigcirc$ (next) and $\boldsymbol{U}$ (until). Formally, the syntax of *linear temporal logic LTL* is defined as follows.

$$
\begin{aligned}
\text{LTL} ::= \;& \langle\Phi\rangle \qquad /*\text{ Propositional variables }*/\\
& \mid \bigcirc\langle\text{LTL}\rangle \mid \langle\text{LTL}\rangle\, \boldsymbol{U}\, \langle\text{LTL}\rangle\\
& \qquad\qquad /*\text{ Temporal connectives }*/\\
& \mid \neg\langle\text{LTL}\rangle \mid \langle\text{LTL}\rangle \vee \langle\text{LTL}\rangle\\
& /*\text{ Propositional connectives }*/
\end{aligned}
$$

We take the set of future-time connectives $\bigcirc$ (next) and $\boldsymbol{U}$ (until). The other future-time connectives $\Diamond$ (sometime or eventually) and $\Box$ (always) can be introduced as abbreviations.

The language of CKL_n is *LTL* augmented by a modal operator K_i for each agent i, and common knowledge operators C_Γ, where Γ is a group of agents. The formal definition is as follows:

$$
\begin{aligned}
\text{CKL}_n ::= \;& \langle\Phi\rangle\\
& \mid \bigcirc\langle\text{CKL}_n\rangle \mid \langle\text{CKL}_n\rangle\boldsymbol{U}\langle\text{CKL}_n\rangle\\
& \mid \neg\langle\text{CKL}_n\rangle \mid \langle\text{CKL}_n\rangle \vee \langle\text{CKL}_n\rangle\\
& \mid K_i\langle\text{CKL}_n\rangle \;/*\text{ Agenti } i \text{ knows }*/\\
& \mid C_\Gamma\langle\text{CKL}_n\rangle\\
& /*\text{ It is common knowledge in } \Gamma \text{ that }*/
\end{aligned}
$$

The semantics of CKL_n is given via the satisfaction relation '$\models_{CKL_n}$'. Given an interpreted system $\mathcal{I} = (\mathcal{R}, \pi)$ and a point (r, u) in $\mathcal{I}$, we define $(\mathcal{I}, r, u) \models \psi$ by the induction on the structure ψ.

(i) $(\mathcal{I}, r, u) \models_{CKL_n} p$ for primitive proposition p iff $p \in \pi(r(u))$.
(ii) $(\mathcal{I}, r, u) \models_{CKL_n} \neg\varphi$ iff it is not $(\mathcal{I}, r, u) \models_{CKL_n} \varphi$.
(iii) $(\mathcal{I}, r, u) \models_{CKL_n} \varphi_1 \wedge \varphi_2$ iff $(\mathcal{I}, r, u) \models_{CKL_n} \varphi_1$ and $(\mathcal{I}, r, u) \models_{CKL_n} \varphi_2$.
(iv) $(\mathcal{I}, r, u) \models_{CKL_n} K_i\varphi$ iff $(\mathcal{I}, r', v) \models_{CKL_n} \varphi$ for all (r', v) such that $(r, u) \sim_i (r', v)$.
(v) $(\mathcal{I}, r, u) \models_{CKL_n} C_\Gamma\varphi$ iff $(\mathcal{I}, r', v) \models_{CKL_n} \varphi$ for all (r', v) such that $(r, u) \sim_\Gamma^C (r', v)$.
(vi) $(\mathcal{I}, r, u) \models_{CKL_n} \bigcirc\varphi$ iff $(\mathcal{I}, r\,(u + 1)) \models_{CKL_n} \varphi$.
(vii) $(\mathcal{I}, r, u) \models_{CKL_n} \varphi\boldsymbol{U}\varphi'$ iff $(\mathcal{I}, r, u') \models_{CKL_n} \varphi'$ for some $u' \geq u$ and $(\mathcal{I}, r, u'') \models_{CKL_n} \varphi$ for all u'' with $u \leq u'' < u'$.

We say that φ is valid in $\mathcal{I}$, denoted by $\mathcal{I} \models_{\text{CKL}_n} \varphi$, if $(\mathcal{I}, r, u) \models_{\text{CKL}_n} \varphi$ for every point (r, u) in $\mathcal{I}$. We also write $(\mathcal{I}, r, u) \models_{\text{LTL}}\varphi$ for $(\mathcal{I}, r, u) \models_{\text{CKL}_n} \varphi$ when φ is an *LTL* formula. For a propositional formula φ, we use $\models\varphi$ to express that φ is a valid formula or tautology.

2.3. Finite-state program with *n* agents

A *finite-state program with n agents* is a finite-state program associated with a set O_i of observable variables for each agent i. To get a symbolic representation of a finite-state program with n agents, we present a symbolic representation of a finite-state program (G, G_0, R, V) in what follows.

(i) We use a tuple of boolean variables $\boldsymbol{x} = \{x_1, \ldots, x_k\}$ and encode a state as an assignment for $\boldsymbol{x}$, or a subset of $\boldsymbol{x}$. (For convenience, we sometimes do not distinguish a set and its characteristic function.) Thus, G_0 and any set of states can be represented as a propositional formula over $\boldsymbol{x}$.
(ii) Further, we use another tuple of boolean variables $\boldsymbol{x}' = \{x_1{}', \ldots, x_k{}'\}$ and represent the 'next time' relation R between two states as a propositional formula τ over $\boldsymbol{x} \cup \boldsymbol{x}'$. In other words, for two assignments s and s' for $\boldsymbol{x}$, sRs' holds iff $\tau\,(\boldsymbol{x}, \boldsymbol{x}')$ is satisfied by the assignment $s \cup N(s')$, where $N(s')$ denotes $\{x_j' \mid x_j \in s'$ and $0 < j \leq k\}$.
(iii) We assume that for each s, $V\,(s)$ equals s, that is, for each variable x_j $(1 \leq j \leq k)$, $V\,(s)(x_j) = 1$ iff $s(x_j) = 1$.

Omitting the component V, we represent the finite-state program (G, G_0, R, V) just as $(\boldsymbol{x}, \theta(\boldsymbol{x}), \tau(\boldsymbol{x}, \boldsymbol{x}'))$.

Hence, we formally define a (symbolic) *finite-state program with n agents* as a tuple $\mathcal{P} = (\boldsymbol{x}, \theta(\boldsymbol{x}), \tau(\boldsymbol{x}, \boldsymbol{x}'), O_1, \ldots, O_n)$, where

(i) $\boldsymbol{x}$ is a set of system variables;
(ii) θ is a boolean formula over $\boldsymbol{x}$, called the *initial condition*;
(iii) τ is a boolean formula over $\boldsymbol{x} \cup \boldsymbol{x}'$, called the *transition relation*; and
(iv) for each i, $O_i \subseteq \boldsymbol{x}$, containing agent i's *local variables*, or *observable variables*.

Given a state s, we define agent i's local state at state s to be $s \cap O_i$. For convenience, we denote $(s \cap O_1, \ldots, s \cap O_n)$ by $g(s)$. We associate with $\mathcal{P}$ the interpreted system $\mathcal{I}_\mathcal{P} = (\mathcal{R}, \pi)$, where $\mathcal{R}$ is a set of those runs r satisfying that

(i) for each m, $r(m)$ is of the form $g(s) = (s \cap O_1, \ldots, s \cap O_n)$ where s is a state in $\mathcal{P}$ and the assignment $\pi\,(g(s))$ is the same as s;
(ii) $r(0)$ is $g(s)$ for some assignment s that satisfies θ;
(iii) for each natural number m, if $r(m) = g(s)$ and $r(m + 1) = g(s')$ for some assignments s and s' for $\boldsymbol{x}$, then $s \cup N(s')$ is an assignment satisfying $\tau\,(\boldsymbol{x}, \boldsymbol{x}')$.

The interpreted system $\mathcal{I}_\mathcal{P}$ is called *the generated interpreted system of* $\mathcal{P}$.

The interpreted systems generated by 'finite-state programs with n agents' represent rather a restricted class of interpreted systems. Nevertheless, this class is reasonably general for distributed computer systems since finite-state programs can be regarded as a general form of programs for computer systems from at least the theoretical point of views.

For convenience, we fix throughout this paper $\mathcal{P} = (\boldsymbol{x}, \theta(\boldsymbol{x}), \tau\,(\boldsymbol{x}, \boldsymbol{x}'), O_1, \ldots, O_n)$ to be a finite-state program with n agents.

Moreover, all variables in $\boldsymbol{x}$ are atomic in the two languages LTL and CKL$_n$ considered. In other words, we assume that $\boldsymbol{x} \subseteq \Phi$ in the definitions of LTL and CKL$_n$.

Example 1. To illustrate the concept of finite-state programs as mentioned above, we consider a scenario in which agent 1 sends some message to agent 2. We introduce two atomic propositions p and q. Proposition p stands for that agent 1 has sent the message. Proposition q indicates that agent 2 has received the message. This can be specified as the following finite-state program:

$$\mathcal{P}_0 = (\boldsymbol{x}, \theta(\boldsymbol{x}), \tau(\boldsymbol{x}, \boldsymbol{x}'), O_1, O_2)$$

where $\boldsymbol{x} = \{p, g\}$, $\theta = (\neg p \wedge \neg q)$, $\tau = (p \Rightarrow p' \wedge q \Rightarrow q' \wedge q' \Rightarrow p)$, $O_1 = \{p\}$ and $O_2 = \{q\}$.

In the generated interpreted system $\mathcal{I}_{\mathcal{P}_0}$, agent 1 has only two distinctive local states { } and $\{p\}$, while all the local states that agent 2 has are { } and $\{q\}$.

We have that

$$\models_{\mathrm{CKL}_2} \quad q \Rightarrow K_1 p$$

which means that if agent 2 has received the message then agent 1 must have sent it. Moreover, we have that

$$\models_{\mathrm{CKL}_2} (\neg \bigcirc q) \boldsymbol{U} p.$$

2.4. Local propositions

We now introduce the notion of a *local* proposition [14]. An i-local proposition is a formula whose interpretation is the same in each of the points in each equivalence class induced by the $\sim_i$ relation. Formally, given an interpreted system $\mathcal{I}$ and an agent i, a formula φ is *i-local* iff for each point (r, u) in $\mathcal{I}$, if $(\mathcal{I}, r, u) \models_{\mathrm{CKL}_n} \varphi$, then $(\mathcal{I}, r', u') \models_{\mathrm{CKL}_n} \varphi$ for all points (r', u') such that $(r, u) \sim_i (r', u')$. Thus, whether an i-local formula holds depends only on i's local state.

Further, for a set $\Gamma \subseteq \{1, \ldots, n\}$, we say a formula φ is Γ-*local* if φ is *i-local* for each $i \in \Gamma$.

The *model checking* problem for CKL$_n$ we are concerned is the problem of determining whether, given an interpreted system $\mathcal{I} = (\mathcal{R}, \pi)$ and a formula φ, the formula φ is true in the initial state of every run in $\mathcal{R}$. More concisely, given an interpreted system $\mathcal{I}$ and a formula φ, we say that $\mathcal{I}$ realizes φ, denoted by $mc_{\mathrm{CKL}_n}(\mathcal{I}, \varphi)$, if for all runs r in $\mathcal{I}$, we have $(\mathcal{I}, r, 0) \models_{\mathrm{CKL}_n} \varphi$.

If φ is an *LTL* formula in the above definition, we also write $mc_{\mathrm{LTL}}(\mathcal{I}, \varphi)$ to stand for $mc_{\mathrm{CKL}_n}(\mathcal{I}, \varphi)$. We use $l_i(\mathcal{I}_{\mathcal{P}}, r, u)$ to denote the above formula $(\wedge_{x \in r_i(u)} x \wedge \wedge_{x \in (O_i - r_i(u))} \neg x)$.

Proposition 1. *A formula φ is i-local in the generated interpreted system $\mathcal{I}_{\mathcal{P}}$ iff there is a propositional formula ψ containing only variables in O_i such that $mc_{CKL_n}(\mathcal{I}_{\mathcal{P}}, \Box(\varphi \Leftrightarrow \psi))$:*

Proof. Given an i-local formula φ in the generated interpreted system $\mathcal{I}_{\mathcal{P}}$. Let

$$\psi = \bigvee_{(\mathcal{I}_{\mathcal{P}}, r, u) \models_{\mathrm{CKL}_n} \varphi} l_i(\mathcal{I}_{\mathcal{P}} r, u)$$

Clearly ψ is a formula that contains only variables in O_i. We will prove that $mc_{CKL_n}(\mathcal{I}_{\mathcal{P}}, \Box(\varphi \Leftrightarrow \psi))$. It suffices to show that, for arbitrary (r', v), $(\mathcal{I}_{\mathcal{P}}\ r', v) \models_{CKL_n} \varphi$ iff $(\mathcal{I}_{\mathcal{P}}, r', v) \models_{\mathrm{CKL}_n} \psi$. If $(\mathcal{I}_{\mathcal{P}}, r', v) \models_{\mathrm{CKL}_n} \varphi$, then ψ is, by the definition of ψ, the disjunction of $l_i(\mathcal{I}_{\mathcal{P}}, r', v)$ and others. As a result, $l_i(\mathcal{I}_{\mathcal{P}}, r', v) \Rightarrow \psi$ is a tautology. However, $(\mathcal{I}_{\mathcal{P}}, r', v) \models_{\mathrm{CKL}_n} l_i(\mathcal{I}_{\mathcal{P}}, r', v)$; therefore, $(\mathcal{I}_{\mathcal{P}}, r', v) \models_{\mathrm{CKL}_n} \psi$. On the other hand, suppose $(\mathcal{I}_{\mathcal{P}}, r', v) \models_{\mathrm{CKL}_n} \psi$. Then, there is a point (r, u) such that $(\mathcal{I}_{\mathcal{P}}, r', v) \models_{\mathrm{CKL}_n} l_i(\mathcal{I}_{\mathcal{P}}, r, u)$ and $(\mathcal{I}_{\mathcal{P}}, r, u) \models_{\mathrm{CKL}_n} \varphi$. By $(\mathcal{I}_{\mathcal{P}}, r', v) \models_{\mathrm{CKL}_n} l_i(\mathcal{I}_{\mathcal{P}}, r, u)$, we have that $(r, u) \sim_i (r', v)$. By i-locality of φ, we have $(\mathcal{I}_{\mathcal{P}}, r', v) \models_{\mathrm{CKL}_n} \varphi$. This completes the 'only if' part of the proof for the proposition.

To show the 'if' part, given an arbitrary formula φ, assume that there is a propositional formula ψ containing only variables in O_i such that $mc_{\mathrm{CKL}_n}(\mathcal{I}_{\mathcal{P}}, \Box(\varphi \Leftrightarrow \psi))$. We want to prove that φ is i-local. Assume that $(\mathcal{I}_{\mathcal{P}}, r, u) \models_{\mathrm{CKL}_n} \varphi$. Then, $mc_{\mathrm{CKL}_n}(\mathcal{I}_{\mathcal{P}}, \Box(\varphi, \Leftrightarrow \psi))$, we have $(\mathcal{I}_{\mathcal{P}}, r, u) \models_{\mathrm{CKL}_n} \psi$. Clearly, ψ is i-local. Thus, for every $(r', v) \sim_i (r, u)$, we have $(\mathcal{I}_{\mathcal{P}}, r', v) \models_{\mathrm{CKL}_n} \varphi$. Therefore, for every $(r', v) \sim_i (r, u)$, we have $(\mathcal{I}_{\mathcal{P}}, r', v) \models_{\mathrm{CKL}_n} \varphi$. This proves the i-locality of φ. □

Proposition 2. *Let Γ be a set of agents. Then, a formula φ is Γ-local in the generated interpreted system $\mathcal{I}_{\mathcal{P}}$ iff for each agent i in Γ, there is a propositional formula ψ_i containing only variables over O_i such that $mc_{CKL_n}(\mathcal{I}_{\mathcal{P}}, \Box(\varphi \Leftrightarrow \psi_i))$.*

Proof. By the definition of Γ-locality and Proposition 1. □

We remark that the two propositions above present both necessary and sufficient conditions for i-locality and Γ-locality, respectively, whereas Propositions 1 and 2 in [12] give only sufficient conditions.

2.5. Reachable global states

Let ξ be an operator from the set of boolean formulas over $\boldsymbol{x}$ to the set of boolean formulas over $\boldsymbol{x}$. We say ψ is a *fixed point* of ξ, if $\models \xi(\psi) \Leftrightarrow \psi$. We say a ψ_0 is a *greatest fixed point* of ξ, if ψ_0 is a fixed point of ξ and for every fixed point ψ of ξ, we have that $\models \psi \Rightarrow \psi_0$. Clearly, any two greatest fixed points are logically equivalent to each other. Thus, we denote a greatest fixed point of ξ by $\boldsymbol{gfp} Z \xi(Z)$. Similarly, we say a ψ_0 is a *least fixed point* of ξ, if ψ_0 is a fixed point of ξ and for every fixed point ψ of ξ, we have that $\models \psi_0 \Rightarrow \psi$. A least fixed point of ξ is denoted by $\boldsymbol{lfp} Z \xi(Z)$. We say ξ is *monotonic*, if for every two formulas ψ_1 and ψ_2 such that $\models \psi_1 \Rightarrow \psi_2$, we have $\models \xi(\psi_1) \Rightarrow (\psi_2)$: For a finite set $\boldsymbol{x}$ of boolean formulas if ξ is monotonic, then there exist a least fixed point and a greatest fixed point [20].

As usual, for a set of boolean variables $\mathbf{v} = \{v_1, \ldots, v_m\}$, $\exists \boldsymbol{v}\varphi$, $(\forall \boldsymbol{v}\varphi)$ stands for $\exists v_1 \ldots \exists v_m \varphi$ $(\forall v_1 \ldots \forall v_m \varphi)$, and $\psi(\boldsymbol{x}'/\boldsymbol{x})$ is the result of renaming variables in $\boldsymbol{x}'$ by those in $\boldsymbol{x}$, respectively.

Let

$$G(\mathcal{P}) = \boldsymbol{lfp} Z\left[\theta(\boldsymbol{x}) \vee (\exists \boldsymbol{x}(Z \wedge \tau(\boldsymbol{x}, \boldsymbol{x}')))\left(\frac{\boldsymbol{x}'}{\boldsymbol{x}}\right)\right].$$

The following lemma says that the (quantified) boolean formula $G(\mathcal{P})$ expresses the set of *reachable global states*.

LEMMA 1. *The following holds:*

(i) $\mathcal{I}_{\mathcal{P}} \vDash_{\mathrm{CKL}_n} G(\mathcal{P})$.
(ii) *For a boolean formula* φ, $\mathcal{I}_{\mathcal{P}} \vDash_{\mathrm{CKL}_n} \varphi$ *iff* $\vDash G(\mathcal{P}) \rightarrow \varphi$.

3. SYMBOLIC MODEL CHECKING CKL_n

The intuition of our approach to symbolic model checking CKL$_n$ is to replace a formula of the form $K_i\varphi$ by some i-local formula ψ. There are two cases depending on whether φ is a pure propositional formula or an LTL formula contain modalities $\boldsymbol{U}$ or $\bigcirc$.

3.1. Model checking knowledge of state properties

First, we consider the case that φ does not contain temporal modalities, this is, φ represents a state property.

PROPOSITION 3. *Let* φ *be a formula that does not contain any temporal modalities. Then*

$$\mathcal{I}_{\mathcal{P}} \vDash_{\mathrm{CKL}_n} K_i\varphi \Leftrightarrow \forall(\boldsymbol{x} - O_i)(G(\mathcal{P}) \Rightarrow \varphi).$$

Proof. Because $K_i\varphi$ is i-local, by Proposition 1, there is proposition formula ψ containing no variable in $(\boldsymbol{x} - O_i)$ such that for every point (r, u) in $\mathcal{I}(\mathcal{P})$,

$$(\mathcal{I}_{\mathcal{P}}, r, u) \vDash_{\mathrm{CKL}_n} K_i\varphi \Leftrightarrow \psi,$$

thus,

$$(\mathcal{I}_{\mathcal{P}}, r, u) \vDash_{\mathrm{CKL}_n} \psi \Leftrightarrow \varphi.$$

By Lemma 1, we have $\vDash G(\mathcal{P}) \Rightarrow (\psi \rightarrow \varphi)$, and hence $\vDash \forall(\mathbf{x} - O_i)$ $(\psi \Rightarrow (G(\mathcal{P}) \Rightarrow \varphi))$. Thus, $\vDash \forall(\boldsymbol{x} - O_i)\psi \Rightarrow \forall$ $(\boldsymbol{x} - O_i)(G(\mathcal{P}) \Rightarrow \varphi)$. It follows that $\vDash \psi \Rightarrow \forall$ $(\boldsymbol{x} - O_i)$ $(G(\mathcal{P}) \Rightarrow \varphi)$. Thus, for every point (r, u) in $\mathcal{I}(\mathcal{P})$,

$$(\mathcal{I}_{\mathcal{P}}, r, u) \vDash_{\mathrm{CKL}_n} K_i\varphi \Rightarrow \forall(\boldsymbol{x} - O_i)(G(\mathcal{P}) \Rightarrow \varphi).$$

On the other hand, assume that

$$(\mathcal{I}_{\mathcal{P}}, r, u) \vDash_{\mathrm{CKL}_n} \forall(\boldsymbol{x} - O_i)(G(\mathcal{P}) \Rightarrow)\varphi).$$

We want to prove $(\mathcal{I}_{\mathcal{P}}, r, u) \vDash_{\mathrm{CKL}_n} K_i\varphi$. By Proposition 1, the formula $\forall(\boldsymbol{x} - O_i)$ $(G(\mathcal{P}) \Rightarrow \varphi)$ is i-local, thus, for every point (r', u') such that $(r, u) \sim_i (r, u)$, we have $(\mathcal{I}_{\mathcal{P}}, r', u') \vDash_{\mathrm{CKL}_n}$ $\forall(\boldsymbol{x} - O_i)$ $G(\mathcal{P}) \Rightarrow \varphi)$. It follows that $(\mathcal{I}_{\mathcal{P}}, r', u') \vDash_{\mathrm{CKL}_n} G(\mathcal{P})$ $\Rightarrow \varphi$. But by Lemma 1, we have $(\mathcal{I}_{\mathcal{P}}, r', u') \vDash_{\mathrm{CKL}_n} G(\mathcal{P})$. So, we have $(\mathcal{I}_{\mathcal{P}}, r'; u') \vDash_{\mathrm{CKL}_n} \varphi$. It follows that $(\mathcal{I}_{\mathcal{P}}, r, u) \vDash_{\mathrm{CKL}_n}$ $K_i\varphi$. This completes the proof. □

PROPOSITION 4. *Let* φ *be a formula that does not contain any temporal modalities*, Γ *a set of agents and* Λ *an operator such that*

$$\Lambda(Z) = \bigwedge_{i \in \Gamma} \forall(\boldsymbol{x} - O_i)(G(\mathcal{P}) \Rightarrow Z).$$

Then

$$\mathcal{I}_{\mathcal{P}} \vDash_{\mathrm{CKL}_n} C_\Gamma\varphi \Leftrightarrow \boldsymbol{gfp}\, Z(G(\mathcal{P}) \wedge \varphi \wedge \Lambda(Z)).$$

Proof. Let ξ be the operator $G(\mathcal{P}) \wedge \varphi \wedge \Lambda(Z)$. It is easy to see that ξ is monotonic. Thus, there is a greatest fixed point of ξ. We will prove that

$$(\mathcal{I}_{\mathcal{P}}, r, u) \vDash_{\mathrm{CKL}_n} C_\Gamma\varphi \Rightarrow \boldsymbol{gfp}\, Z(G(\mathcal{P}) \wedge \varphi \wedge \Lambda(Z))$$

by showing

(i) $(\mathcal{I}_{\mathcal{P}}, r, u) \vDash_{\mathrm{CKL}_n} C_\Gamma\varphi \Rightarrow G(\mathcal{P}) \wedge \varphi$.
(ii) For every formula ψ, if $(\mathcal{I}_{\mathcal{P}}, r, u) \vDash_{\mathrm{CKL}_n} C_\Gamma\varphi \Rightarrow \psi$, then $(\mathcal{I}_{\mathcal{P}}, r, u) \vDash_{\mathrm{CKL}_n} C_\Gamma\varphi \Rightarrow \xi(\psi)$.

The former is trivially true by Lemma 1 and the fact that

$$(\mathcal{I}_{\mathcal{P}}, r, u) \vDash_{\mathrm{CKL}_n} C_\Gamma\varphi \Rightarrow \varphi.$$

As for the latter, assume $(\mathcal{I}_{\mathcal{P}}, r, u) \vDash_{\mathrm{CKL}_n} C_\Gamma\varphi \Rightarrow \psi$. We want to prove $(\mathcal{I}_{\mathcal{P}}, r, u) \vDash_{\mathrm{CKL}_n} C_\Gamma\varphi \Rightarrow \xi(\psi)$: Since $(\mathcal{I}_{\mathcal{P}}, r, u) \vDash_{\mathrm{CKL}_n}$ $C_\Gamma\varphi \Rightarrow K_iC_\Gamma\varphi$, we have that

$$(\mathcal{I}_{\mathcal{P}}, r, u) \vDash_{\mathrm{CKL}_n} C_\Gamma\varphi \Rightarrow \bigwedge_{i \in \Gamma} K_i\psi.$$

By Proposition 3, we have

$$(\mathcal{I}_{\mathcal{P}}, r, u) \vDash_{\mathrm{CKL}_n} \bigwedge_{i \in \Gamma} K_i\psi \Rightarrow \Lambda(\psi).$$

It follows that $(\mathcal{I}_{\mathcal{P}}, r, u) \vDash_{\mathrm{CKL}_n} C_\Gamma\varphi \Rightarrow \xi(\psi)$.

We now show that

$$(\mathcal{I}_{\mathcal{P}}, r, u) \models_{\mathrm{CKL}_n} \textbf{\textit{gfp}}\, Z(G(\mathcal{P}) \wedge \varphi \wedge \Lambda(Z)) \Rightarrow C_\Gamma \varphi.$$

Let α be the formula $\textbf{\textit{gfp}}\ Z(G(\mathcal{P}) \wedge \varphi \wedge \Lambda\ (Z))$: It suffices to show

(i) $(\mathcal{I}_{\mathcal{P}}, r, u) \models_{\mathrm{CKL}_n} \alpha \Rightarrow \varphi$, and
(ii) for each $i \in \Gamma$ and formula ψ, if $(\mathcal{I}_{\mathcal{P}}, r, u) \models_{\mathrm{CKL}_n} \alpha \Rightarrow \psi$, then $(\mathcal{I}_{\mathcal{P}}, r, u) \models_{\mathrm{CKL}_n} \alpha \to K_i\psi$.

The first assertion is trivially. The second follows by Proposition 3 and the fact $(\mathcal{I}_{\mathcal{P}}, r, u) \models_{\mathrm{CKL}_n} \alpha \Rightarrow \Lambda(\alpha)$. □

By Propositions 3 and 4, when we do the task of model checking CKL_n formula, we can replace formulas of the form $K_i\varphi$ $(C_\Gamma\varphi)$ by some i-local (Γ-local) proposition, where φ does not contain any temporal modalities.

3.2. Model checking knowledge of temporal properties

Now, we deal with the case that φ may contain some temporal modalities. We use the idea of the so-called tableau construction as descried in [21, 22]. For a formula, ψ, we write $\psi \in \varphi$ to denote that ψ is a sub-formula of (possibly equals to) φ. Formula ψ is called *principally temporal* if its main operator is temporal operator, i.e. ψ is of the form $\bigcirc\ \alpha$ or $\alpha \boldsymbol{U} \beta$.

Given a formula φ, we define a finite-state program $\mathcal{P}_\varphi = (\boldsymbol{x}_\varphi, \theta_\varphi, \tau_\varphi, O_1, \ldots, O_n)$ as follows.

System variables: The set $\boldsymbol{x}_\varphi$ of system variables of $\mathcal{P}_\varphi$ consists of $\boldsymbol{x}$ plus a set of auxiliary boolean variables

$$X_\varphi : \{x_\psi \mid \psi \text{is a principally temporal sub-formula of } \varphi\}.$$

The auxiliary variable x_ψ is intended to be true in a state of a computation iff the temporal formula ψ holds at the state.

For convenience, we define a function χ, which maps every sub-formula of φ into a boolean formula over $\boldsymbol{x} \cup X_\varphi$.

$$\chi(\psi) = \begin{cases} \psi, & \text{for } \psi \text{ a variable in } \boldsymbol{x} \\ \neg\chi(\alpha), & \text{for } \psi = \neg\alpha \\ \chi(\alpha) \wedge \chi(\beta), & \text{for } \psi = \alpha \wedge \beta \\ x_\psi, & \text{for principally temporal } \psi \end{cases}$$

Let X'_φ be the primed version of X_φ. For a formula ψ over $\boldsymbol{x} \cup X_\varphi$, we use $\chi'\ (\psi)$ to denote the formula $\psi\ ((\boldsymbol{x} \cup X_\varphi)/(\boldsymbol{x}' \cup X_\varphi'))$, i.e. the primed version of ψ.

Initial condition: The initial condition of $\mathcal{P}_\varphi$ is the same as for $\mathcal{I}_{\mathcal{P}}$.

Transition relation: The transition relation τ_φ of $\mathcal{P}_\varphi$ is the conjunction of the transition relation τ and the following formula

$$\bigwedge_{\bigcirc\psi\in\varphi} (x_{\bigcirc\psi} \Leftrightarrow \chi'(\psi)) \wedge \bigwedge_{\alpha\boldsymbol{U}\beta\in\varphi} (x_{\alpha\boldsymbol{U}\beta} \Leftrightarrow (\chi(\beta) \wedge (\chi(\alpha) \wedge x'_{\alpha\boldsymbol{U}\beta})))$$

For convenience, we introduce now some more notations from the CTL logic [16]. Let $\boldsymbol{EX}$ be the operator such that for a boolean formula ψ over $\boldsymbol{x} \cup X_\varphi$,

$$\boldsymbol{EX}\psi(\boldsymbol{x}, X\varphi) = \exists(\boldsymbol{x}' \cup X'_\varphi)(\psi(\boldsymbol{x}', X'_\varphi) \wedge \tau_\varphi)$$

In other words, the set of those states satisfying $\boldsymbol{EX}\psi(\boldsymbol{x}, X_\varphi)$ is the image of the set of those states satisfying ψ under the transition relation τ_φ.

The operators $\boldsymbol{EF}$ and $\boldsymbol{EU}$ are defined by the least fixed point of some monotonic operators: $\boldsymbol{EF}f = \textbf{\textit{lfp}}\ Z(f \vee \boldsymbol{EX}Z)$, and $\boldsymbol{EU}(f, g) = \textbf{\textit{lfp}}\ Z(g \vee (f \wedge\ \boldsymbol{EX}Z))$.

Let $\mathcal{J}_\varphi$ be the set of all formulas $\neg x_{\alpha\boldsymbol{U}\beta} \wedge \chi(\beta)$, where $\alpha\boldsymbol{U}\beta$ is a sub-formula of φ. To give the knowledge of agent i at some state, we consider the following fairness constraints:

C^1_φ: There is a computational path for which each formula in $\mathcal{J}_\varphi$ holds for infinite times.
C^2_φ: There is a finite computational path such that each formula in $\mathcal{J}_\varphi$ holds at the last state of the computational path, and the last state does not have any next state in the system $\mathcal{I}_{\mathcal{P}}$.

Clearly, if C^1_φ holds with $\mathcal{J}_\varphi \neq 0$, then there is a computational path, which is infinitely long. If C^2_φ holds, then there is a finite computational path at which the last state does not have a next state in the system $\mathcal{I}_{\mathcal{P}}$.

We suppose that $\mathcal{J}_\varphi$ is not an empty set. This assumption does not lose any generality because we can put ***true*** in $\mathcal{J}_\varphi$.

The constrain C^2_φ can be expressed as $\boldsymbol{EF}(\mathrm{End}(\mathcal{P}) \wedge \bigwedge_{\psi\in\mathcal{J}\varphi}\psi)$ in the standard CTL logic, where $\mathrm{End}(\mathcal{P})$ is the formula related to the set of dead states in the system $\mathcal{I}_{\mathcal{P}}$, it can be represented as $\neg\exists\boldsymbol{x}'\tau(\boldsymbol{x}, \boldsymbol{x}')$.

The constrain C^1_φ can be defined as:

$$C^1_\varphi = \textbf{\textit{gfp}}\, Z\left[\bigwedge_{J\in\mathcal{J}\varphi} \boldsymbol{EX}(\boldsymbol{EU}(\textbf{\textit{true}}, Z \wedge J))\right].$$

It is not difficult to see that a state satisfies the condition C^1_φ iff the state is at some run where each $J \in \mathcal{J}_\varphi$ holds for infinite times along the run [16].

We say a run r_φ in $\mathcal{I}_{\mathcal{P}_\varphi}$ is *fair*, if either r_φ is infinitely long and each J in $\mathcal{J}_\varphi$ is satisfied by infinitely many states at r_φ, or there is a state, say s_{end}, such that $s_{\mathrm{end}} \cap \boldsymbol{x}$ has no successor in $\mathcal{I}_{\mathcal{P}}$ and each $J \in \mathcal{J}_\varphi$ is satisfied by s_{end}. We obtain the following assertion.

LEMMA 2. *Let φ be an LTL formula, s_φ a state of $\mathcal{I}_{\mathcal{P}_\varphi}$. Then, s_φ satisfies $C^1_\varphi \vee C^2_\varphi$ iff s_φ is at some fair run of $\mathcal{I}_{\mathcal{P}_\varphi}$.*

LEMMA 3. *Let φ be an LTL formula. Then for each run r in $\mathcal{I}_\mathcal{P}$, there is a fair run r_φ in $\mathcal{I}_{\mathcal{P}_\varphi}$ such that for every natural number u and for every subformula ψ of φ.*

(i) $r(u) = r_\varphi(u) \cap \boldsymbol{x}$,
(ii) $(\mathcal{I}_\mathcal{P}, r, u) \vDash_{LTL} \psi$ iff $\chi(\psi)$ *is satisfied by* $r_\varphi(u)$.

Proof. Let r be a run in $\mathcal{I}_\mathcal{P}$. We define a fair run r_φ in $\mathcal{I}_{\mathcal{P}_\varphi}$ as follows. For each point $r(u)$, let $r_\varphi(u)$ be the variable set

$$r(u) \cup \left\{ x_\alpha \,\middle|\, \begin{array}{l} \alpha \text{ is principally temporal subformula of } \varphi \\ \text{and } (\mathcal{I}_\mathcal{P}, r, u) \vDash_{LTL} \alpha \end{array} \right\}$$

It follows immediately that $r(u) = r_\varphi(u) \cap \boldsymbol{x}$ and, for a principally temporal subformula ψ of φ, we have that $(\mathcal{I}_\mathcal{P}, r, u) \vDash_{\text{LTL}} \psi$ iff $\chi(\psi)$ is satisfied by $r_\varphi(u)$. For other sub-formulas ψ of φ, we can prove the above assertion holds by induction on ψ. It is also easy to see that r_φ is a run in $\mathcal{I}_{\mathcal{P}_\varphi}$ and r_φ is fair. □

LEMMA 4. *Let φ be as in Lemma 3. Then, for each fair run r_φ in $\mathcal{I}_{\mathcal{P}_\varphi}$, there is a run r in $\mathcal{I}_\mathcal{P}$ such that for every natural number u and for every subformula ψ of φ,*

(i) $r(u) = r_\varphi(u) \cap \boldsymbol{x}$,
(ii) $(\mathcal{I}_\mathcal{P}, r, u) \vDash_{LTL} \psi$ *iff* $\chi(\varphi)$ *is satisfied by* $r_\varphi(u)$.

Proof. Let r_φ be a fair run in $\mathcal{I}(\mathcal{P}_\varphi)$. We define a run r in $\mathcal{I}_\mathcal{P}$ simply by

$$r(u) = r\varphi(u) \cap \boldsymbol{x}$$

for each state $r_\varphi(u)$.

We assume that r_φ is infinitely long, the other case, where r_φ is finite, can be dealt with in the same way.

Given a subformula ψ of φ, we show, by induction on the structure of ψ, that $(\mathcal{I}_\mathcal{P}, r, u) \vDash_{\text{LTL}} \psi$ iff $\chi(\psi)$ is satisfied by $r_\varphi(u)$.

(i) Case $\psi \in \boldsymbol{x}$. By the fact $r(u) = r_\varphi(u) \cap \boldsymbol{x}$, the result follows immediately.
(ii) Case $\psi = \neg\psi_1$ or $\psi = \psi_1 \wedge \psi_2$. By the induction hypothesis and the definition of $\neg$ and $\wedge$, it is easy to prove these cases.
(iii) Case $\psi = \bigcirc\psi_1$. By the semantic definition of $\bigcirc$, we have that

$$(\mathcal{I}_\mathcal{P}, r, u) \vDash_{\text{LTL}} \psi \text{ iff } (\mathcal{I}_\mathcal{P}, r, u+1) \vDash_{\text{LTL}} \psi_1.$$

By the induction hypothesis, we can see that

$$(\mathcal{I}_\mathcal{P}, r, u+1) \vDash_{\text{LTL}} \psi_1 \text{ iff } r_\varphi(u+1) \text{ satisfies } \chi(\psi_1).$$

By the semantic definition of $\bigcirc$ again,

$$r_\varphi(u+1) \text{ satisfies } \chi(\psi_1)$$

iff

$$(\mathcal{I}(\mathcal{P}_\varphi), r_\varphi, u) \vDash_{\text{LTL}} \bigcirc\chi(\psi_1).$$

By the definition of the transition relation of $\mathcal{P}_\varphi$, we get that

$$(\mathcal{I}(\mathcal{P}_\varphi), r_\varphi, u) \vDash_{\text{LTL}} \bigcirc\chi(\psi_1) \text{ iff } r_\varphi(u) \text{ satisfies } x_\psi.$$

(iv) Case $\psi = \psi_1 \boldsymbol{U} \psi_2$. ($\Rightarrow$) Assume that $(\mathcal{I}(\mathcal{P}_\varphi), r_\varphi, u) \vDash_{\text{LTL}} \psi_1 \boldsymbol{U} \psi_2$. Then, for some $v \geq u$, $(\mathcal{I}(\mathcal{P}_\varphi), r_\varphi, v) \vDash_{\text{LTL}} \psi_2$, and for all $u \leq m \leq v$, $\mathcal{I}(\mathcal{P}_\varphi), r_\varphi, m) \vDash_{\text{LTL}} \psi_1$. Thus, by the induction hypothesis, $r_\varphi(v)$ satisfies ψ_2, and for all $u \leq m \leq v$, $r_\varphi(m)$ satisfies ψ_1. Now we prove, for all $u \leq m \leq v$, that $r_\varphi(m)$ satisfies x_ψ by induction on the degression of m. First, by the transition relation of $\mathcal{P}_\varphi$ and the fact $r_\varphi(v)$ satisfies ψ_2, we get $r_\varphi(v)$ satisfies x_ψ. Assume that for $u < m \leq v$, $r_\varphi(m)$ satisfies x_ψ. Then, by the transition relation of $\mathcal{P}_\varphi$ and the fact $r_\varphi(m-1)$ satisfies x_φ, we have $r_\varphi(m-1)$ satisfies x_ψ. This completes the proof of part ($\Rightarrow$).
($\Leftarrow$) Suppose that $r_\varphi(u)$ satisfies x_ψ but not $(\mathcal{I}(\mathcal{P}), r, u) \vDash_{\text{LTL}} \psi_1 \boldsymbol{U} \psi_2$. There are two cases:
(a) There is a natural number $v \geq u$ such that $(\mathcal{I}(\mathcal{P}), r, v) \vDash_{\text{LTL}} \neg\psi_1$, and for all $u \leq m \leq v$, we have $(\mathcal{I}(\mathcal{P}), r, m) \vDash_{\text{LTL}} \neg\psi_2$.
(b) For all $m \geq u$, we have $(\mathcal{I}(\mathcal{P}), r, u) \vDash_{\text{LTL}} \psi_1 \wedge \neg(\psi_2)$.

In case (a), by the induction hypothesis, we have that $r_\varphi(v)$ satisfies $\chi(\psi_2)$, and for all $u \leq m \leq v$, $r_\varphi(v)$ does not satisfy $\chi(\psi_1)$. By the transition relation of $\mathcal{P}_\varphi$, we have that for all m, $r_\varphi(m)$ satisfies x_ψ iff $r_\varphi(m)$ satisfies $\chi(\psi_2)$, or $r_\varphi(m)$ satisfies $\chi(\psi_1)$ and $r_\varphi(m+1)$ satisfies x_ψ. Thus, for all $u \leq m \leq v$, we have that $r_\varphi(m)$ satisfies x_ψ iff $r_\varphi(m)$ satisfies $\chi(\psi_1)$ and $r_\varphi(m+1)$ satisfies x_ψ. Since $r_\varphi(v)$ does not satisfy $\chi(\psi_1)$, we can infer that $r_\varphi(v)$ does not satisfy x_ψ. Because for all $u \leq m \leq v$, from $r_\varphi(m)$ satisfying x_ψ, we can infer $r_\varphi(m+1)$ satisfying x_ψ, we can see that $r_\varphi(u)$ does not satisfy x_ψ. This is a contradiction.

In case (b), by the induction hypothesis, we have that for all $m \geq u$, $r_\varphi(m)$ does not satisfy $\chi(\psi_2)$ and $r_\varphi(m)$ satisfies $\chi(\psi_1)$. Thus, by the transition relation again, we have that, for all $m \geq u$, $r_\varphi(m)$ satisfies x_ψ iff $r_\varphi(m+1)$ satisfies x_ψ. Thus, for all $m \geq u$, $r_\varphi(m)$ satisfies x_ψ and $r_\varphi(m)$ does not satisfy $\chi(\psi_2)$.

This leads to a contradiction, because the fairness of r_φ guarantees, if r_φ is infinitely long, that there are infinitely many m such that r_φ satisfies $\neg x_\psi \wedge \psi_2$. □

We now extend the logic CKL$_n$ by introducing two path quantifiers $\boldsymbol{A}$ and $\boldsymbol{E}$. The resulting language is denoted by ECKL$_n$. For a finite-state program $\mathcal{P}$ with n agents, a run r in $\mathcal{I}_\mathcal{P}$, a formula ψ and a natural number u, we have $(\mathcal{I}_\mathcal{P}, r, u) \models_{\mathrm{CKL}_n} \boldsymbol{E}\psi$ iff there is a run r' such that, for some natural number v, $r(u) = r'(v)$ and $(\mathcal{I}_\mathcal{P}, r', v) \models_{\mathrm{CKL}_n} \psi$. We define $\boldsymbol{A}\psi$ as $\neg\boldsymbol{E}\neg\psi$.

Clearly, if we remove knowledge modalities from *ECKL$_n$*, we get the well-known logic CTL*. The following proposition presents a methodology of implementing symbolic verifying CTL* via OBDDs.

Proposition 5. *Let φ be an LTL formula. Then*

$$\mathcal{I}\mathcal{P} \models_{CKL_n} \boldsymbol{E}\varphi \Leftrightarrow \exists X\varphi((C^1_\varphi \vee C^2_\varphi) \wedge \chi(\varphi)).$$

Proof. ($\Rightarrow$) Assume that $(\mathcal{I}_\mathcal{P}, r, u) \models_{\mathrm{CKL}_n} \boldsymbol{E}\varphi$. There is a run r' and a natural number v such that $r'(v) = r(u)$ and $(\mathcal{I}_\mathcal{P}, r', v) \models_{\mathrm{CKL}_n} \varphi$. By Lemma 3, there is a fair run r_φ in $\mathcal{I}_{\mathcal{P}_\varphi}$, such that $r_\varphi(v)$ satisfies $\chi(\varphi)$ iff $(\mathcal{I}_\mathcal{P}, r, v) \models_{\mathrm{CKL}_n} \varphi$. Thus, $r_\varphi(v)$ satisfies $\chi(\varphi)$. Moreover, because r_φ is a fair run, every state at run r_φ must satisfy $C^1_\varphi \vee C^2_\varphi$. So, $r_\varphi(v)$ satisfies $(C^1_\varphi \vee C^2_\varphi) \wedge \chi(\varphi)$. Because $r(u) = r'(v) = r_\varphi(v) \cap \boldsymbol{x}$, we have that $r(u) \cup (r_\varphi(v) \cap X_\varphi) = r_\varphi(v)$ satisfies $(C^1_\varphi \vee C^2_\varphi) \wedge \chi(\varphi)$. This is, $(\mathcal{I}_\mathcal{P}, r, u) \models_{CKL_n} \exists X_\varphi((C^1_\varphi \vee C^2_\varphi \wedge \chi(\varphi))$.

($\Leftarrow$) Suppose that $(\mathcal{I}_\mathcal{P}, r, u) \models_{CKL_n} \exists X_\varphi((C^1_\varphi \vee C^2_\varphi) \wedge \chi(\varphi))$. Then, $r(u)$ satisfies $\exists X_\varphi((C^1_\varphi \vee C^2_\varphi) \wedge \chi(\varphi))$, and there is a state s_φ in $\mathcal{I}_{\mathcal{P}_\varphi}$ such that $r(u) = s_\varphi \cap \boldsymbol{x}$, and s_φ satisfies $((C^1_\varphi \vee C^2_\varphi) \wedge \chi(\varphi))$. By the fact that s_φ satisfies $C^1_\varphi \vee C^2_\varphi$ and Lemma 2, we get that s_φ is at some fair run r_φ in $\mathcal{I}_{\mathcal{P}_\varphi}$, and there is a natural number v such that $s_\varphi = r_\varphi(v)$. By Lemma 4, there is a run r' in $\mathcal{I}_\mathcal{P}$ such that $r'(v) = r_\varphi(v) \cap \boldsymbol{x}$ and $(\mathcal{I}_\mathcal{P}, r', v) \models_{\mathrm{CKL}_n} \varphi$ iff $r_\varphi(v)$ satisfies $\chi(\varphi)$. Recalling $r_\varphi(v) = s_\varphi$ and $r(u) = s_\varphi \cap \boldsymbol{x}$, we have $r'(v) = r(u)$. Moreover, by the fact that $r_\varphi(v)$ satisfies $\chi(\varphi)$, we have that $(\mathcal{I}_\mathcal{P}, r', v) \models_{CKL_n} \varphi$. Hence, $(\mathcal{I}_\mathcal{P}, r, u) \models_{CKL_n} \boldsymbol{E}\varphi$. □

Now follow the main results in this section.

Proposition 6. *Let φ be an LTL formula. Then, the following formula*

$$K_i\varphi \Leftrightarrow \forall(X_\varphi \cup \boldsymbol{x} - O_i)((C^1_\varphi \vee C^2_\varphi) \wedge G(\mathcal{P}) \Rightarrow \chi(\varphi))$$

is valid in $\mathcal{I}_\mathcal{P}$.

Proof. We first notice that the formula $K_i\varphi \Leftrightarrow K_i\boldsymbol{A}\varphi$ is valid in $\mathcal{I}_\mathcal{P}$. Proposition 5 says that the formula $\boldsymbol{A}\varphi\forall X_\varphi\ ((C^1_\varphi \vee C^2_\varphi) \Rightarrow \chi(\varphi))$ is valid. Hence,

$$\mathcal{I}_\mathcal{P} \models_{CKL_n} K_i\varphi \Leftrightarrow K_i(\forall X_\varphi((C^1_\varphi \vee C^2_\varphi) \Rightarrow \chi(\varphi))).$$

By Proposition 3, the following formula

$$K_i\varphi \Leftrightarrow \forall(\boldsymbol{x} - O_i)(G(\mathcal{P}) \Rightarrow \forall X_\varphi((C^1_\varphi \vee C^2_\varphi) \Rightarrow \chi(\varphi)))$$

must be valid in $\mathcal{I}_\mathcal{P}$. Because variables in X_φ do not appear in $G(\mathcal{P})$, the formula

$$K_i\varphi \Leftrightarrow \forall(X_\varphi \cup \boldsymbol{x} - O_i)((C^1_\varphi \vee C^2_\varphi) \wedge G(\mathcal{P}) \Rightarrow \chi(\varphi))$$

is thus valid in $\mathcal{I}_\mathcal{P}$. □

Remark 1. In order to determine whether $K_i\varphi$ holds at some point of an interpreted system, van der Hoek and Wooldridge [12] attempt to find an i-local proposition ψ such that $K_i\varphi$ holds iff ψ holds at that point. However, they did not provide a general methodology to obtain such an i-local proposition ψ automatically. In addition, the local-proposition formula ψ may depend on the point at which we check $K_i\varphi$ (see Proposition 5 in [12]). Thus, when faced with the problem of determining whether some point satisfies a formula α with a sub-formula of the form $K_i\varphi$, we could not reduce the problem to determining whether the point satisfies the formula $\alpha((K_i\varphi)/\psi)$ (which results from α by replacing $K_i\varphi$ with ψ.) The main advantage of Proposition 6 over van der Hoek and Wooldridge's results is that the i-local proposition ψ is given out (i.e. $\forall(X_\varphi \cup \boldsymbol{x} - O_i)((C^1_\varphi \vee C^2_\varphi) \wedge G(\mathcal{P}) \Rightarrow \chi(\varphi)))$ and the proposition ψ does not depend on the point (r, u).

We also remark that Proposition 6 provides a reduction of CKL$_n$ to LTL while Proposition 5 gives an OBDD-based method of model checking LTL formulas. The complexity of our reduction of CKL$_n$ to LTL is PSPACE-complete. Nevertheless, because quantifications of boolean functions and fixed-point operators can be dealt with in any OBDD package, the reduction can be based on OBDDs. Thus, the CKL$_n$ model checking algorithm with the above results might be practically implementable.

As for model checking common knowledge of temporal properties, we can see the following proposition holds.

Proposition 7. *Let φ be a formula that may contain some temporal modalities, Λ an operator such that*

$$\Lambda(Z) = \bigwedge_{i\in\Gamma} \forall(x - O_i)(G(\mathcal{P}) \Rightarrow Z).$$

Then, the following formula is valid in $\mathcal{I}_\mathcal{P}$:

$$C_\Gamma\varphi \Leftrightarrow \boldsymbol{gfp}\, Z[G(\mathcal{P}) \wedge \forall X_\varphi((C^1_\varphi \vee C^2_\varphi) \Rightarrow \chi(\varphi)) \wedge \Lambda(Z)].$$

Proof. Notice that for every point (r, u) in $\mathcal{I}_\mathcal{P}$, we have that

$$(\mathcal{I}_\mathcal{P}, r, u) \models_{\mathrm{CKL}_n} C_\Gamma\varphi \Leftrightarrow C_\Gamma\boldsymbol{A}\varphi.$$

By Proposition 5, for every point (r, u) in $\mathcal{I}_{\mathcal{P}}$,

$$(\mathcal{I}_{\mathcal{P}}, r, u) \models_{\mathrm{CKL}_n} \boldsymbol{A}\varphi \Leftrightarrow \forall X_{\varphi}((C^1_{\varphi} \vee C^2_{\varphi}) \Rightarrow \chi(\varphi)).$$

Thus,

$$(\mathcal{I}_{\mathcal{P}}, r, u) \models_{\mathrm{CKL}_n} C_{\Gamma}\varphi \Leftrightarrow C_{\Gamma}(\forall X_{\varphi}((C^1_{\varphi} \vee C^2_{\varphi}) \Rightarrow \chi(\varphi))).$$

By Proposition 4, we have that $(\mathcal{I}_{\mathcal{P}}, r, u) \models_{\mathrm{CKL}_n} C_{\Gamma}\varphi \Leftrightarrow$ ***gfp*** $Z[G(\mathcal{P}) \wedge \forall X_{\varphi}((C^1_{\varphi} \vee C^2_{\varphi}) \Rightarrow \chi(\varphi)) \wedge \Lambda(Z)]$. □

4. THE IMPLEMENTATION OF MCTK

According to the present approach, we have developed a symbolic model checker for ECKL$_n$ logic, called MCTK, based on NuSMV v2.1.2 [23] under the Open Source License that allows free academic and non-commercial usage. The MCTK allows for the representation of finite-state program with n agents, and for the analysis of specifications expressed in ECKL$_n$ logic, using BDD-based model checking techniques. The BDD-package exploited in MCTK is the CUDD library developed by Fabio Somenzi at Colorado University.

4.1. Software architecture

A multi-agent system contains an *environment* and a finite number of *agents*. The basic form of an agent is ‘agent A:M’, where A is the *name* of the agent and M is the agent’s *program module* (‘module’ for short). Each agent in a multi-agent system is assumed to have a unique name. The program module M is the main part of an agent, which determines its observable variables and its behaviour.

Each agent is able to perform its actions according to its own local state, which is encoded by its observable variables. The change of the *state* of the system occurs as a result of *joint actions* performed by the agents and the environment. In the environment and each agent’s module, we thus include the descriptions of how the actions change the state of the system. Note that agent’s actions do not need to be represented explicitly for each system.

The architecture of MCTK is an extension of NuSMV v2.1.2. Before model checking an ECKL$_n$ specification, the following modules should be invoked in turn to obtain the symbolic representation of a finite-state program with n agents:

Parser builds a parse tree representing the internal format of a MCTK input file using Lex and Yacc.

Instantiation processes the parse tree, and performs the instantiation of the declared modules, building a description of the finite-state machine (FSM) representing the model (e.g. the transition relation, the initial states and the fairness).

Encoder performs the encoding of data types and finite ranges into boolean domains. The state variables $\boldsymbol{x}$ of the system and the observable variables O_i of each agent $i(1 \leq i \leq n)$ can be retrieved in this phase.

FSM Compiler provides the routines for constructing and manipulating FSMs at the BDD level. We can get the BDDs that represents the initial condition θ and the transition relation τ of the system in this phase.

4.2. Input language

The MCTK input language is extended from the NuSMV input language and its syntax is described as follows. We refer to NuSMV v2.1 user manual [24] for more details of its input language.

```
MCTK_program : := EnvDef AgentDefList
EnvDef : := "MODULE" "main" "(" ")"
  EVarDef    ; ; environmental variables
  AgentList
  [EVarInitDef]
  [EVarTransDef]
  [ECKLnSpecDef]
  ...
AgentList : := atom ":" AgentType ";" |
  AgentList atom ":" AgentType ";"
AgentType : := atom [ "("AParaList ")" ] |
  "array" number ".." number "of" AgentType
AParaList : : =  simple_f |
  AParaList "," simple_f
AgentDefList : := AgentDef |
  AgentDefList AgentDef
AgentDef : :=
  "MODULE" atom ["(" FParaList ")" ]
    [LVarDef]     ; ; local variables
    [ActDef]      ; ; action variables
    [ActInitDef]
    [ActTransDef]
    [OVarInitDef]
    [OVarTransDef]
    ...
FParaList : := ["Observable"] atom |
  FParaList "," ["Observable"] atom
ActDef : := atom ": {"AtomList "};"
AtomList : := atom | AtomList "," atom
atom : := [A-Za-z_][A-Za-z0-9_\$#-]*
```

4.2.1. Environment declaration

We use the `main` module to define the environment. `EVarDef` declares some state variables of the environment, which may be used by some agents as shared variables for the purpose of inter-agent communication. `AgentList` defines a number of agents. Note that, for conveniently describing a set of agents with the same module definition, we allow `AgentType` to be an array of `AgentType` itself. The set of the initial states and the next states relate current or next states, encoded by the environmental variables in `EVarDef`, are defined in `EVarInitDef` and `EVarTransDef`, respectively. `ECKLnSpecDef` gives some specifications in ECKL$_n$ logic to be checked. Its syntax is listed as follows

```
ECKLnSpecDef : := "ECKLNSPEC" f [";"]
f: :=
  simple_f
  | "(" f ")"
  | "!"f           ; ; logical not
  | f "&"          ; ; logical and
  | f "|" f        ; ; logical or
  | f "xor" f      ; ; logical exclusive or
  | f "->" f       ; ; logical implication
  | f "<-> "f      ; ; logical equivalence
  | "A" f          ; ; for all paths
  | "E" f          ; ; for some path(s)
  | "X" f          ; ; next state
  | "G" f          ; ; globally
  | "F" f          ; ; finally
  | f "U" f        ; ; until

  | name "K" f     ; ; agent name knows f

  | "(" NameList ")" "C" f
  ...
name : : = var_id    ; ; the name of an agent
NameList : := name | NameList "," name
```

Simple expressions `simple_f` are constructed from variables, constants and a collection of operators, including boolean connectives, integer arithmetic operators, case expressions and set expressions. We have $\boldsymbol{F}f \equiv true\ \boldsymbol{U}f$ and $\boldsymbol{G}f \equiv \neg\boldsymbol{F}\neg f$. `"(" NameList ")" "C" f` express that formula f is the common knowledge among the agents in `NameList`. Note that each agent's `"Name"` must be an *instance* of an agent's module.

4.2.2. *Agent declaration*

As is shown in the syntax of `AgentDef`, each agent is defined as a MODULE. We explain some important parts of an agent's module below.

Formal parameters: The formal parameters of an agent's module are defined in `FParaList`. Whenever these parameters occur in expressions within the module, they are replaced by the actual parameters, which are supplied when the module is instantiated. The difference between the syntax of the formal parameters of an agent's module and that of a NuSMV module is that the former allows some formal parameters to be specified *observable*. In the current implementation, we only allow the type of the observable actual parameter, `simple_f` declared in `AParaList`, to be *variable*. Thus, an agent is able to observe or modify these observable actual parameters, i.e. some variables of environment and even some local variables of other agents.

Observable variables: The set O_i of observable variables of agent i is defined to be the set of agent i's local variables and observable actual parameters. In our framework, an agent's local state is identified by its observable variables. If a local variable defined in `LVarDef` is a *module instance*, then all of the local variables except those instances of other modules in the module instance are recursively inserted into the set of local variables of the agent. So the variable defined in `ActDef` is also in the set of the agent's observable variables.

Protocol: The protocol for agent i is a description of what actions agent i may take. It can be formally defined as a function from the set of its local states to non-empty sets of its actions. If at a given step of the protocol there is more than one action that may be chosen, then only one of them can be actually performed. The choice of the action to perform is non-deterministic. All of the allowable actions can be encoded by a variable of an enumerated type, which is defined in `ActDef`. The evolution of the variable that represents action is defined in `ActInitDef` and `ActTransDef`, which define an agent's initial actions and the next actions relate current or next local states, respectively. The variables appearing in `ActInitDef` and `ActTransDef` are restricted observable. Note that the protocol is not necessary because we are mainly concerned about the evolution of an agent's observable variables.

Evolution of observable variables: Agent i's evolution function for observable variables is the description of how the values of agent i's observable variables change. It is a function from the set of its local states to the set of its local states. The evolution function is defined in `OVarInitDef` and `OVarTransDef`.

5. CASE STUDY I: THE DINING CRYPTOGRAPHERS PROTOCOL

In this section, we apply our model checker MCTK to the verification of the Dining Cryptographers Protocol [25], a protocol for anonymous broadcast. Chaum introduces this protocol by the following story:

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

Chaum shows that the following protocol can be used to solve the dining cryptographers' quandary. The protocol assumes that at most one cryptographer is paying.

(i) Each cryptographer flips an unbiased coin behind his menu, between him and the cryptographer to his right, so that only the two of them can see the outcome.
(ii) Each cryptographer then states aloud whether the two coins that he can see—the one he flipped and the one his left-hand neighbour flipped—fell on the same side or different sides.
(iii) As an exception to the previous step, if one of the cryptographers is the payer, he states the opposite of what he sees.

It can be shown that after running this protocol, all the cryptographers are able to determine whether it was the NSA or one of the cryptographers who paid for dinner. Specifically, an even number of differences uttered at the table indicates that NSA is paying, an odd number of differences indicates that a cryptographer is paying. Significantly, if a cryptographer is paying neither of the other two learns anything from the utterances about which cryptographer it is.

5.1. Model description in MCTK

To formalize this protocol with three cryptographers in MCTK input language, we should first define the environment of this protocol as follows:

```
         Environment Declaration
MODULE main()
VAR
  coin1 : boolean;   --cryptographer C1
  coin2 : boolean;   --cryptographer C2
  coin3 : boolean;   --cryptographer C3
  C1: DC(coin1, coin2, C2.said, C3.said);
  C2: DC(coin2, coin3, C1.said, C3.said);
  C3: DC(coin3, coin1, C1.said, C2.said);
INIT (C1.paid + C2.paid + C3.paid) <= 1;
TRANS next(coin1)=coin1 &
      next(coin2)=coin2 &
      next(coin3)=coin3
```

In the environment declaration, we define three boolean variables, `coin1`, `coin2` and `coin3`, to, respectively, indicate the coins the cryptographers `C1`, `C2` and `C3` flipped. For example, if the value of variable `coin1` is *true*, then it means that the coin the cryptographer `C1` flipped is on its obverse side, otherwise it is on its reverse side. Next expressions under the 'TRANS' keyword relate current and next state variables to express transition relation of the model. For example, expression `next(coin1)=coin1` in line 12 expresses that the next value of variable `coin1` is equal to its current value. So lines 12–14 has kept variables `coin1`, `coin2` and `coin3` unchangeable since their initial values are assigned non-deterministically.

Then, we define the module of the dining cryptographers as the following source code.

```
              Agent's Module
MODULE DC(
  Observable coin_self,
  Observable coin_left,
  Observable said1,
  Observable said2)
VAR
  paid : boolean; --true=pay for the dinner
  said : boolean; --true=say "different side"
ASSIGN
  init(said): = paid xor
               (coin_self xor coin_left);
  next(said): = said;
  next(paid): = paid;
```

In the declaration of agent's module, 'DC' is the name of the module used by the cryptographers. The variables defined under the 'VAR' keyword are viewed as the local ones in the agent's module, where boolean variable `paid` represents whether the cryptographer pays for the dinner. The set of initial states of the model is determined by a boolean expression under the 'INIT' keyword, so line 11 of the environment declaration initially restricts that at most one cryptographer pays for the dinner. Another local boolean variable `said` represents the cryptographer's utterance.

Lines 12–13 of agent's module simulates Steps (ii) and (iii) of this protocol. Note that we assign the value of variable `said` in the initial state of the model, so that we can determine the agents' knowledge not considering the temporal dimension. Lines 14–15 of agent's module have kept variables `said` and `paid` unchangeable since their initial values are determined.

Besides the local variables `paid` and `said`, an agent's observable variables also include his own coin and the coin his left-hand neighbour flipped and the other two agents' utterances. So, we should define four observable formal parameters, `coin_self`, `coin_left`, `said1` and `said2`, respectively, for the four observable variables in lines 2–5 of agent's module. In the environment declaration, for example, line 7 define a cryptographer `C1` with the actual parameters `coin1`, `coin2`, `C2.said` and `C3.said`. So the observable variables of `C1` are `C1.paid`, `coin1`, `coin2`, `C1.said`, `C2.said` and `C3.said`.

5.2. *ECKL$_n$* specification

Now let us consider the specification of the problem. We see that the following *ECKL$_n$* formula expresses Chaum's requirements in the case of cryptographer `C1`:

```
ECKLNSPEC
!C1.paid -> (
  (C1 K (!C1.paid & !C2.paid & !C3.paid)) |
  ( (C1 K (C2.paid | C3.paid)) &
    !(C1 K C2.paid) & !(C1 K C3.paid))
)
```

That is, if cryptographer `C1` does not pay, then he knows either that no cryptographer pays, or knows that one of the other two pays, but does not know which. The specifications for the other cryptographers are similar. To verify the protocol, it suffices to consider only the specification for cryptographer `C1` above, because of symmetry considerations.

5.3. The protocol in two other epistemic model checkers MCK and MCMAS

In this section, we consider the model construction and specification checking of the protocol in two other epistemic model checkers MCK and MCMAS.

5.3.1. The protocol in MCK

An MCK [26] is a model checker for the logic of knowledge, developed at the School of Computer Science and Engineering at the University of New South Wales. The novelty of this model checker is that it supports several different ways of defining knowledge given a description of a multi-agent system and the observations made by the agents: observation alone; observation and clock; and perfect recall of all observations. Both linear and branching time temporal operators are supported. Currently, the system is primarily on BDD-based model checking algorithms.

We directly use the MCK input file of this protocol in the MCK user manual available at http://www.cse.unsw.edu.au/~mck/mck.ps.gz. In this MCK input file, there are three boolean arrays `paid[3]`, `chan[3]` and `said[3]` defined in the environment. `paid[0]`, `paid[1]` and `paid[2]`, respectively, represent whether cryptographer `C1`, `C2` and `C3` pay for the dinner; `said[0]`, `said[1]` and `said[2]`, respectively, represent these cryptographers' utterances. The elements of array `chan[3]` are used as some shared variables so that each cryptographer is able to access the coin he flipped and the coin his left-hand neighbour flipped. In an agent's protocol, the observable formal parameters are `paid` and array `said[]`, which means that one agent is able to observe all cryptographers' utterances. The observable local variables are the two coins he can see: `coin_left` and `coin_right`. Therefore, the observable variables of an agent is identified with those defined in our MCTK input file.

The specification checked in the MCK input file of this protocol with three cryptographers is as follows:

```
spec_spr_xn = X 6 (neg paid[0]) =>
  ((Knows C1 (neg paid[0]) /\
             (neg paid[1]) /\
             (neg paid[2])) \/
  ((Knows C1 (paid[1] \/ paid[2])) /\
  (neg (Knows C1 paid[1])) /\
  (neg (Knows C1 paid[2]))))
```

The part `spec_spr_xn` means that we are using the perfect recall module of MCK, and `X 6` is the abbreviation of 6 'next time' temporal operator `X`.

5.3.2. The protocol in MCMAS

An MCMAS is also a model checker that handles knowledge and branching time using BDD-based algorithms. This model checker has been developed by Raimondi and Lomuscio [27].

We directly use the input files of the protocol in the package of MCMAS v0.7. In the case of three cryptographers, we introduce the input file as [28]: three agents $C_i (i = \{1, 2, 3\})$ are introduced to model the three cryptographers, and one agent E for the environment. The environment is used to select non-deterministically the identity of the payer and the results of the coin tosses. This makes a total of 32 possible local states for the environment. The environment can perform only one action, the null action. Therefore, the protocol is simply mapping every local state to the null action. Also, there is no evolution of the local states for the environment. The local states of the cryptographers are modelled as a string containing three parameters representing, respectively, whether or not the coins that a cryptographer can see are equal, whether or not the cryptographer is the payer, and the number of 'different' utterances reported. Considering that all these parameters are not initialized at the beginning of the run, there are 27 possible combinations of these, hence 27 possible local states are required for every agent. For each cryptographer, the actions allowed are 'say nothing', 'say equal', 'say different' and these actions are performed in compliance with the protocol stated above. We refer to the input file available in the package of MCMAS v0.7 for the details of the protocol and of the evolution function.

We define the following set of atomic propositions to reason about this scenario: $\mathrm{AP} = \{\mathtt{paid}_1, \mathtt{paid}_2, \mathtt{paid}_3, \mathtt{even}, \mathtt{odd}\}$ and we have (g is a global state):

$$\begin{array}{rll} g \models \mathtt{paid}_1 & \text{if} & l_{C_1}(g) = \langle *\text{Paid}* \rangle \\ g \models \mathtt{paid}_2 & \text{if} & l_{C_2}(g) = \langle *\text{Paid}* \rangle \\ g \models \mathtt{paid}_3 & \text{if} & l_{C_3}(g) = \langle *\text{Paid}* \rangle \\ g \models \mathtt{even} & \text{if} & l_{C_i}(g) = \langle *\text{Even}* \rangle \text{ for every } i \\ g \models \mathtt{odd} & \text{if} & l_{C_i}(g) = \langle *\text{Odd}* \rangle \text{ for every } i \end{array}$$

Function $l_i(g)$ returns the local state of agent i in the global state g. <*Paid*> denotes a local state in which the string contains the value `Paid` (i.e. the cryptographer paid for dinner). <*Even*> and <*Odd*> are defined similarly. We can now express formally some MCMAS specifications similar to our MCTK specification.

As for the input file with three cryptographers, the first cryptographer may not follow the protocol, so we design the following two MCMAS specifications in the case of three cryptographers:

```
(even and !c2paid) ->
  KH(DinCrypt2, DinCrypt1,
       (!c1paid and !c2paid and !c3paid));
(odd and !c2paid) ->
  (   KH(DinCrypt2, DinCrypt1,
        (c1paid or c3paid)) and
    !(KH(DinCrypt2,DinCrypt1,c1paid)) and
    !(KH(DinCrypt2,DinCrypt1,c3paid)));
```

where formula KH(i, Γ, φ) expresses the knowledge φ that agent i has on the assumption that all agents in Γ are functioning correctly.

In the input file with four cryptographers, all cryptographers are modelled as honest ones, then in this case, we can design the following MCMAS specification that is formally equal to our MCTK specification:

```
(odd and !c1paid) ->
  (K(DinCrypt1,(c2paid or c3paid or c4paid))
   and !K(DinCrypt1,c2paid)
   and !K(DinCrypt1,c3paid)
   and !K(DinCrypt1,c4paid));
```

5.4. Comparison of the experimental results

In this section, we conduct a comparative study for the three epistemic model checkers MCK, MCMAS and our MCTK, by verifying the specification given above in these model checkers. We directly use the binary (version 0.1.0) of MCK, which can be found on the MCK homepage. The version of MCMAS we use is 0.7. The performance results for the input files described above are based on a laptop configuration Ubuntu 2.6.10-5-386 Linux system, Intel(R) Pentium(R) M 1.60 GHz processor and 512 Mb RAM.

The experimental results for MCTK, MCK and MCMAS are for the whole model checking process, i.e. both model construction and specification checking. All of the specifications described above are checked true, respectively, in these model checkers. When checking the protocol, we use the BDD dynamic reordering functions of MCTK and MCK, respectively, by adding command line parameter '-dynamic' for MCTK and by adding command line parameter '-rs' for MCK. The BDD dynamic reordering function is inherently activated in MCMAS.

We list the running time for the three model checkers in Table 1. Notice that the same protocol works for any number of cryptographers greater or equal to three, we try to individually verify the protocols with 3, 4, 20 and 80 agents in the three model checkers. Because the MCK running time for the protocol with 80 agents is too long to wait, we halt the execution 1 h later. Besides, because the current version 0.7 of MCMAS does not support variables in the description of agents' local states and all of the possible local states should be enumerated in each agent's protocol, it is difficult to extend the description of the protocol in MCMAS to a number of cryptographers greater than four, we have to check only the two input files `dc-3.ispl` and `dc-4.ispl`, respectively, for three and for four cryptographers, that are available in the package of MCMAS v0.7.

TABLE 1: experimental results for the dining cryptographers protocol

Agents	Model checker	Running time	BDD variables	Number of nodes
	MCTK	0.292 s	19	773
3	MCMAS	1.160 s	49	50 180
	MCK	0.815 s	109	8285
	MCTK	0.276 s	25	1701
4	MCMAS	2.537 s	63	93 212
	MCK	1.914 s	143	7453
	MCTK	1.109 s	121	5445
20	MCMAS	–	–	–
	MCK	3 m 23.888 s	687	78 789
	MCTK	1 m 6.610 s	481	125 274
80	MCMAS	–	–	–
	MCK	>60 m	–	–

From Table 1, we can conclude that for this protocol and its modellings stated above, the running efficiency of our model checker MCTK is much better than the other two. As far as we are concerned, there are two reasons. First, the modelling for this protocol in MCTK is ore succinct than that in MCK and MCMAS. In the case of three cryptographers, only 19 BDD variables and 773 BDD nodes are created in MCTK, whereas in MCK, 109 BDD variables and 8285 BDD nodes are created, because MCK need extra BDD variables `chan[0]`, `chan[1]` and `chan[2]` as shared variables and more transitions to specify read and write operations on the shared variables. In addition, from Table 1 we can see that MCMAS create 49 BDD variables and 50 180 BDD nodes for this protocol, which states that the modelling in MCMAS are much larger than that in MCTK. Second, the running efficiency of the CUDD library, the BDD package used in MCTK and MCMAS, is better than that of David Long's BDD library, which is used by the binary of MCK that we get from the MCK homepage.

To show the expressive power of $ECKL_n$ in temporal dimension, we also check the following $ECKL_n$ specification in MCTK:

```
ECKLNSPEC
!C1.paid -> G (
  (C1 K (!C1.paid & !C2.paid & !C3.paid)) |
  ( (C1 K (C2.paid | C3.paid)) &
    !(C1 K C2.paid) & !(C1 K C3.paid))
)
```

That is, if cryptographer `C1` did not pay, then it is *always* holds that he knows either that no cryptographer paid, or knows that one of the other two paid, but does not know which. This specification is also true in MCTK.

6. CASE STUDY II: RUSSIAN CARDS

From a pack of seven known cards two players draw three cards each and a third player gets the remaining card. How can the players with three cards openly inform each other about their cards, without the third player learning from any of their cards who holds it?

The 'Russian Cards' problem [29] was originally presented at the Moscow Math Olympiad 2000. The problem is to find a solution that allow the sender and receiver to learn each other's hand of cards, without revealing this information to the eavesdropper.

Let us call the players Anne, Bill and Cath, and the cards 0, ..., 6. Suppose Anne holds {0, 1, 2}, Bill holds {3; 4; 5} and Cath holds card 6. For the hand of cards {0, 1, 2}, we write 012 instead. For the card deal, we write 012.345.6, etc. Assume from now on that 012.345.6 is the actual card deal. Anne and Bill exchange each other's information by some announcements. All announcements must be public and truthful. Obviously, the first requirement for a solution to the problem is that Cath's ignorance of Anne and Bill's hands of cards is always commonly known to the three players after any announcement. Such announcements are called safe.

A solution to the Russian Cards problem is a sequence of safe announcements. The second requirement for a solution is that after these safe announcements, it is commonly known to Anne and Bill (not necessarily including Cath) that Anne knows Bill's hand and Bill knows Anne's hand. The following five-hands protocol is a solution to the problem:

Anne says 'my hand of cards is one of 012, 034, 056, 135, 246', after which Bill says 'Cath has card 6' (in other words, Bill announces that his hand of cards is one of 345, 125, 024).

6.1. Russian Cards in MCTK

In this section, we specify the five-hands protocol in MCTK and verify the two requirements above, which are expressed in ECKL$_n$ logic. We first define the environment of this protocol as follows.

```
          Environment Declaration
MODULE main
VAR
  stage:0..3;
  a0:0..1;  a1:0..1;  ... a6:0..1;
  b0:0..1;  b1:0..1;  ... b3:0..1;
  c0:0..1;  c1:0..1;  ... c3:0..1;
  a_ann:0..1;
  b_ann:0..1;
  PA: Anne(a0,...,a6,a_ann,b_ann,stage);
  PB: Bill(b0,...,b6,a_ann,b_ann,stage);
  PC: Cath(c0,...,c6,a_ann,b_ann,stage);
TRANS
  (stage=0 ->
    (a0+a1+a2+a3+a4+a5+a6+
    b0+b1+b2+b3+b4+b5+b6+
    c0+c1+c2+c3+c4+c5+c6)=0) &
  (stage=1 ->
    ((a0+a1+a2+a3+a4+a5+a6)=3&
    (b0+b1+b2+b3+b4+b5+b6)=3&
    (c0+c1+c2+c3+c4+c5+c6)=1&
    (a0+b0+c0)=1&(a1+b1+c1)=1&
    (a2+b2+c2)=1&(a3+b3+c3)=1&
    (a4+b4+c4)=1&(a5+b5+c5)=1&
    (a6+b6+c6)=1)) &
  (stage>=1 ->
    (next(a0)=a0 & ... & next(a6)=a6 &
    next(b0)=b0 & ... & next(b6)=b6 &
    next(c0)=c0 & ... & next(c6)=c6))
ASSIGN
  init(stage):=0;
  init(a_ann):=0;
  init(b_ann):=0;
  next(stage):=case
    stage < 3: stage+1;
    stage=3: stage;
  esac;
  next(a_ann):=case
    next(PA.announce)=1: 1;
                      1: a_ann;
  esac;
  next(b_ann):=case
    next(PB.announce)=1&c6=1: 1;
                            1: b_ann;
  esac;
```

In the environment declaration, variable `stage` is the 'clock tick', which stands for the execution stage of this protocol. The initial value of `stage` is 0. `stage=3` indicates that the execution of the protocol finishes. Boolean variables `a0,...,a6`, `b0,...,b6` and `c0,...,c6` stand for Anne, Bill and Cath's hands of cards, respectively. For example, `b3=1` means that Bill holds card 3, `b3=0` otherwise. Lines 17–31 describe how the hands of cards of the three players change in each stage. Lines 17–20 express that card are not dealt to those players in stage 0. Lines 21–27 restrict that Anne and Bill each can receive three cards and Cath can receive one in stage 1. But the values of `a0,...,a6`, `b0,...,b6` and `c0,...,c6` are uncertain, so by this restriction, we can model $\binom{7}{3}\binom{4}{3}\binom{1}{1} = 140$ deals. Lines 28–31 keep all players' hands of cards invariant after stage 1. Boolean variable `a_ann` denotes whether Anne announce some information or not. The value of `a_ann` is initially assigned 0 in line 35. Lines 43–46 assign 1 to `a_ann` after Anne's announcement `(PA.announce=1)`. Similarly, boolean variable b_ann stand for Bill's announcement. Lines 48–51 assign 1 to b_ann after Bill announces that Cath holds card 6 (PB.announce=1 and c6=1).

Agent PA, for Anne, is declared by Line 12. The name of the agent is PA. It uses module 'Anne'. It can observe the variables between parentheses. So, the set of Anne's observable variables is {a0, a1, a2, a3, a4, a5, a6, a_ann, b_ann, stage} plus the local variables of module 'Anne'. Variables a_ann, b_ann, stage are also the actual parameters of agents PB and PC, as they are publicly observable.

Anne's protocol is defined as the following module.

```
                Anne's Module
MODULE Anne(Observable a0,...,Observable a6,
        Observable a_ann,Observable b_ann,
        Observable stage)
VAR
  announce:0..1; - announcement action,
ASSIGN
    init(announce):=0;
    next(announce):=case
      stage=1 &
      ((a0 & a1 & a2) | (a0 & a3 & a4) |
        (a0 & a5 & a6) | (a1 & a3 & a5)|
        (a2 & a4 & a6)): 1;
      1: announce;
  esac;
```

All of the formal parameters in Anne's module are observable. Boolean variable announce is used to denote whether Anne takes the action of announcement or not. announce=1 means that Anne has taken the action of announcement, announce=0 otherwise. Lines 9–14 mean that Anne announces 'My hand is one of 012, 034, 056, 135, and 246' in stage 1.

Bill and Cath's protocols are defined as the following modules. Lines 9–12 mean that if in stage 2, Anne has already announced that her hand is one of the five hands, then Bill will take the action of announcement. Note that there is not any code in the body of Cath's module, as Cath does not act.

```
           Bill and Cath's Modules
MODULE Bill(Observable b0,...,Observable b6,
         Observable a_ann,Observable b_ann,
         Observable stage)
VAR
  announce:0..1; - announcement action
ASSIGN
  init(announce):=0;
  next(announce):=case
    stage=2 & a_ann=1: 1;
                1: announce;
  esac;
MODULE Cath(Observable c0,...,Observable c6,
         Observable a_ann,Observable b_ann,
         Observable stage)
```

6.2. *ECKL$_n$* specifications

Now let's formalize the two requirements for the five-hands protocol in *ECKL$_n$* logic.

We use a_know_b, b_know_a and c_ignorant to denote 'Anne knows Bill's hand', 'Bill knows Anne's hand' and 'Cath does not know any of Anne's or Bill's hand', respectively. We say that an agent knows whether formula ψ holds if it either knows ψ or its negation. c_ignorant also means that 'If Cath does not hold a card, Cath can imagine both Anne and Bill to have it', it follows that Cath does not know Anne to have it, and does not know Bill to have it. Therefore, we have the following *ECKL$_n$* formulas

```
a_know_b := ((PA K b0)|(PA K !b0))
                & ... &
            ((PA K b6)|(PA K !b6));
b_know_a := ((PB K a0)|(PB K !a0))
                & ... &
            ((PB K a6)|(PB K !a6));
c_ignorant := (!(PC K a0) & !(PC K b0))
                & ... &
            (!(PC K a6) & !(PC K b6)).
```

Thus, we can formalize the first and the second requirements as the following two *ECKL$_n$* specifications:

```
ECKLNSPEC A G((PA,PB,PC) C c_ignorant)
ECKLNSPEC A G((a_ann & b_ann) ->
          ((PA,PB) C (a_know_b & b_know_a)))
```

6.3. Russian Cards in MCK and MCMAS

van Ditmarsch *et al.* [30] have implemented the five-hands protocol for Russian Cards in MCK and MCMAS. These MCK and MCMAS input scripts can be found in http://www.cs.otago.ac.nz/staffpriv/hans/aoard/. We directly use the MCK and MCMAS input scripts on this hyperlink to specify the five-hands protocol, just properly modify existing specifications so that they are semantically close to the above two ECKL$_n$ ones. So we omit these MCK and MCMAS input scripts here and refer to this hyperlink for details. In fact, our MCTK input script is similar to the MCK input script given in [30], as the former is derived from the latter.

6.3.1. *MCK specifications for Russian Cards*

Because the current version 0.2.0 of MCK does not support common knowledge operators for specification in the perfect recall module, we create the following two MCK specifications for the first requirement of the five-hands protocol:

```
  spec_spr_xn = X 2 (a_announce =>
    (neg((Knows   C   a_hand[0])\/(Knows   C
b_hand[0]))
```

```
              /\.../\
   neg((Knows   C   a_hand[6])\/(Knows   C
b_hand[6]))))
  spec_spr_xn=X 3 ((a_announce/\b_announce) =>
   (neg((Knows   C   a_hand[0])\/(Knows   C
b_hand[0]))
              /\.../\
   neg((Knows   C   a_hand[6])\/(Knows   C
b_hand[6]))))
```

The first spec_spr_xn specification says that after Anne's announcement, Cath remains ignorant, whenever the announcement could be made, so this is public knowledge. The second one says that after Bill's announcement, it is public that Cath remains ignorant. Therefore, we can believe that Cath remains ignorant of Anne and Bill's hands of cards after any announcement.

We also create the following two spec_spr_xn specifications for the second requirement:

```
  spec_spr_xn =
  X 3 ((a_announce/\b_announce) =>
   (((Knows  A  b_hand[0])  \/  (Knows  A  neg
b_hand[0]))
              /\.../\
     ((Knows  A  b_hand[6])  \/  (Knows  A  neg
b_hand[6]))))
  spec_spr_xn =
  X 3 ((a_announce/\b_announce) =>
  (((Knows  B  a_hand[0])  \/  (Knows  B  neg
a_hand[0]))
              /\.../\
  ((Knows  B  a_hand[6])  \/  (Knows  B  neg
a_hand[6]))))
```

The two spec_spr_xn specifications say that after Bill's announcement, it is public that 'Anne knows Bill's cards' and 'Bill knows Anne's cards', respectively. Therefore, they meet the second requirement for the five hands protocol.

6.3.2. *MCMAS specifications for Russian Cards*

The MCMAS specification can be expressed in an extension of branching-time temporal logic CTL that includes epistemic modalities. Therefore, we can create the following two MCMAS specifications that are semantically equivalent to the MCTK specifications presented in Section 6.2

```
AG( GCK( ABC,
        ((!K(Cath,a0) and !K(Cath,b0))
              and ... and
        (!K(Cath,a6) and !K(Cath,b6)))));
AG( ab_d0123456 ->
    GCK( AB,
      -- a_know_b
      ((K(Anne,b0) or K(Anne,!b0))
              and ... and
       (K(Anne,b6) or K(Anne,!b6))) and
      -- b_know_a
      ((K(Bill,a0) or K(Bill,!a0))
              and ... and
       (K(Bill,a6) or K(Bill,!a6)))));
```

where formula K(*agt*, φ) expresses that agent *agt* knows φ, formula GCK(Γ, φ) expresses that φ is the common knowledge of the agents in Γ, and when ab_d0123456 is true, it means that both Anne and Bill's announcements have been made.

6.4. Comparison of the experimental results

In the same experimental environment for the Dining Cryptographers protocol, we implement the five-hands protocol, respectively, in MCK, MCMAS and our MCTK. Because the BDD reordering function is inherently activated in MCMAS, in order to make the experimental results more convincible, we use BDD reordering functions of MCK and MCTK, respectively, by adding command line parameter "-rw" and "-dynamic". The experimental results are listed in Table 2, in which each data are also for the whole model checking process, i.e. both model construction and specification checking. All of the specifications given above are verified true in these model checkers. Table 2 illustrates again that MCTK is an efficient epistemic model checker.

7. CONCLUSIONS

In this paper, we have considered the model checking problem for Halpern and Vardi's well-known temporal epistemic logic CKL_n. We have introduced the notion of a finite-state program with *n* agents, which can be thought of as a symbolic representation of interpreted systems. We have developed an approach to symbolic CKL_n model checking using OBDDs. In our approach to model checking specifications involving agents' knowledge, the knowledge modalities are eliminated via quantifiers over agents' non-observable variables. As a by-product, we have presented a methodology for implementing symbolic verifying CTL* via OBDDs.

TABLE 2: experimental results for Russian Cards

Requirement	Model checker	Running time	BDD variables	Number of nodes
	MCTK	0.693 s	57	35 256
1	MCMAS	1 m 13.502 s	68	681 989
	MCK	1 m 6.412 s	140	337 874
	MCTK	0.936 s	57	22 659
2	MCMAS	1 m 13.528 s	68	738 530
	MCK	1 m 10.545 s	168	281 603

Based on the results in this paper, we have implemented a symbolic CKL$_n$ model checker MCTK by functionally extending the open source of NuSMV v2.1.2 (an efficient symbolic model checker for linear and branching-time temporal logics). Our experimental results are encouraging. In particular, modelling in MCTK is more succinct than those in MCK and MCMAS. The run-time efficiency of MCTK turns out to be better at least for the two case studies. The three model checkers are based on different modelings, and their run-time efficiency can be quite sensitive to the modelling approach they use. We believe that the encouraging experimental results demonstrate competitive advantage of MCTK.

As for future work, we are interested in providing automated support for the analysis of knowledge in distributed system protocols and game theoretic examples, and the verification and compilation of knowledge-based programs [18].

ACKNOWLEDGEMENTS

Thanks to Qingliang Chen and the reviewers of The Computer Journal for their valuable comments. This work was partially supported by the Australian Research Council grant DP0452628, National Basic Research 973 Program of China under grant 2005CB321902, National Natural Science Foundation of China grants 60496327, 10410638 and 60473004, and Guangdong Provincial Natural Science Foundation grants 04205407 and 06023195.

REFERENCES

[1] Su, K. (2004) Model checking temporal logics of knowledge in distributed systems. *Proc. Nineteenth National Conf. Artificial Intelligence, Sixteenth Conf. Innovative Applications of Artificial Intelligence*, pp. 98–103. AAAI Press / The MIT Press.

[2] Holzmann, G. (1997) The SPIN model checker. *IEEE Trans. Soft. Eng.*, **23**, 279–295.

[3] Vardi, M. (2001) Branching vs. linear time. In Margaria, T. and W. Yi (eds.), *Proc. 7th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, 1–22. Springer.

[4] McMillan, K. (1993) *Symbolic Model Checking*. Kluwer Academic Publisher, Boston.

[5] Halpern, J. and Vardi, M. (1989) The complexity of reasoning about knowledge and time, I: lower bounds. *J. Comput. Syst. Sci.*, **38**, 195–237.

[6] Halpern, J. and Vardi, M. Y. (1991) Model checking vs. theorem proving: a manifesto. Technical report. IBM Almaden Research Center. *Proc. 2nd Int. Conf. Principles of Knowledge Representation and Reasoning, 1991*.

[7] vander Meyden, R. (1998) Common knowledge and update in finite environments. *Inf. Comput.*, **140**, 115–157.

[8] Rao, A. and Georgeff, M. (1993) A model theoretic approach to the verification of situated reasoning systems. *Proc. 13th Int. Joint Conf. Artificial Intelligence*, pp. 318–324.

[9] Benerecetti, M., Giunchiglia, F. and Serafini, L. (1999) A model checking algorithm for multi-agent systems. In Muller, J., Singh, M. and Rao, A. (eds.), *Intelligent Agents V*. Springer-Verlag, Berlin.

[10] Benerecetti, M. and Giunchiglia, F. (2000) Model checking security protocols using a logic of belief. In Graf, S. and Schwartzbach, M. (eds.), *Proc. 6th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, pp. 519–534. Springer.

[11] vander Meyden, R. and Su, K. (2004) Symbolic model checking the knowledge of the dining cryptographers. *Proc. 17th IEEE Computer Security Foundations Workshop (CSFW'04)*, Washington, DC, USA, P. 280. IEEE Computer Society.

[12] vander Hoek, W. and Wooldridge, M. (2002) Model checking knowledge and time. *Proc. 9th Int. SPIN Workshop on Model Checking of Software*, London, UK, 95–111. Springer-Verlag.

[13] Pnueli, A. (1977) The temporal logic of programs. *Proc. 18th IEEE Symp. Foundations of Computer Science*, pp. 46–57.

[14] Engelhardt, K., van der Meyden, R. and Moses, Y. (1998) Knowledge and the logic of local propositions. *TARK'98: Proc. 7th Conf. on Theoretical Aspects of Rationality and Knowledge*, San Francisco, CA, USA, pp. 29–41. Morgan Kaufmann Publishers Inc.

[15] Bryant, R. E. (1986) Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, **35**, 677–691.

[16] Clarke, E. M., Grumberg, O. and Peled, D. A. (2000) *Model Checking*. The MIT Press, Cambridge, Massachusetts.

[17] Engelhardt, K., van der Meyden, R. and Su, K. (2002) Modal logics with a linear hierarchy of local propositional quantifiers. In Balbiani, P., Suzuki, N.-Y., Wolter, F. and Zakharyaschev, M. (eds.), *Advances in Modal Logic*, pp. 9–30. King's College Publications.

[18] Fagin, R., Halpern, J., Moses, Y. and Vardi, M. (1995) *Reasoning About Knowledge*. MIT Press, Cambridge, MA.

[19] Manna, Z. and Pnueli, A. (1995) *Temporal Verification of Reactive Systems*. Springer-verlag, Berlin, Germany.

[20] Tarski, A. (1955) A lattice-theoretical fixpoint theorem and its applications. *Pac. J. Math.*, **5**, 285–309.

[21] Lichtenstein, O. and Pnueli, A. (1985) Checking that finite state concurrent programs satisfy their linear specification. *POPL '85: Proc. 12th ACM SIGACT-SIGPLAN Symp. Principles of programming languages*, New York, NY, USA, pp. 97–107. ACM Press.

[22] Clarke, E. M., Grumberg, O. and Hamaguchi, K. (1994) Another look at LTL model checking. In Dill, D. L. (ed.), *CAV*, Vol. 818, Lecture Notes in Computer Science, pp. 415–427. Springer.

[23] Cimatti, A. *et al*. (2002) Nusmv 2: an opensource tool for symbolic model checking. In Brinksma, E. and Larsen, K. G. (eds.), *CAV-2002*, Vol. 2404, Lecture Notes in Computer Science, pp. 359–364. Springer.

[24] Cavada, R. *et al*. (2002) *NuSMV 2.1 User Manual*. ITCIRST: Istituto Trentino di Cultura/Istituto Ricerca Scientifica e Tecnologica, Trento, Italy. http://nusmv.irst.itc.it/NuSMV/userman/v21/nusmv.pdf.

[25] Chaum, D. (1988) The dining cryptographers problem: unconditional sender and recipient untraceability. *J. Cryptolo.*, **1**, 65–75.

[26] Gammie, P. and van der Meyden, R.. (2004) MCK: model checking the logic of knowledge. In Alur, R. and Peled, D. (eds.), *Proc. CAV-2004*, Vol. 3114, Lecture Notes in Computer Science, pp. 479–483. Springer.

[27] Lomuscio, A. and Raimondi, F. (2006) MCMAS: a model checker for multi-agent systems. In Hermanns, H. and Palsberg, J. (eds.), *Proc. TACAS-2006*, Vol. 3920, Lecture Notes in Computer Science, pp. 450–454. Springer.

[28] Raimondi, F. and Lomuscio, A. (2004) Verification of multiagent systems via ordered binary decision diagrams: an algorithm and its implementation. *Proc. AAMAS-2004*, pp. 630–637.

[29] van Ditmarsch, H. P. (2003) The russian cards problem. *Stud. Log.*, **75**, 31–62.

[30] van Ditmarsch, H. P. *et al*. (2006) Model checking russian cards. *Electron. Notes Theor. Comput. Sci.*, 149, 105–123.