Model Checking Knowledge and Time via Local Propositions: Cooperative and Adversarial Systems

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

Abstract. Model checking as an approach to the automatic verification of finite state systems has focused predominantly on system specifications expressed in temporal logic. In the distributed systems community, *logics of knowledge* (epistemic logics) have been advocated for expressing desirable properties of protocols and systems. A range of logics combining temporal and epistemic components have been developed for this purpose. However, the model checking problem for temporal logics of knowledge has received (comparatively) little attention. In this paper, we address ourselves to this problem. Following a brief survey of the relevant issues and literature, we introduce a temporal logic of knowledge (Halpern and Vardi's logic CKL_n). We then develop an approach to CKL_n model checking that combines ideas from the *interpreted systems* semantics for knowledge with the *logic of local propositions* developed by Engelhardt et al. With our approach, local propositions provide a means to reduce CKL_n model checking to linear temporal logic model checking. After introducing and exploring the ideas underpinning our approach, we present three case studies (the bit transmission problem, leadership election and the Dining Cryptographers) in which SPIN was used to establish temporal epistemic properties of a system implemented in PROMELA.

Keywords: model, checking, verification, knowledge, time, epistemic, temporal, logic, bit, transmission, leadership, election, dining, cryptographer, SPIN

Keywords: model, checking, verification, knowledge, time, epistemic, temporal, logic, bit, transmission, leadership, election, dining, cryptographer, SPIN

1. Introduction

Since the mid 1980s, *modal logics of knowledge* have been increasingly deployed in the formal specification of distributed systems, where they are used to make precise the concept of what a process knows [8, 20]. Temporal logics of knowledge — temporal logics enriched by modal knowledge operators — have also been widely used for reasoning about distributed systems [11, 27].

Model checking as an approach to the automatic verification of finite state systems has focussed predominantly on system specifications expressed in temporal logic — linear temporal logic in the case of SPIN [15, 16] and FORSPEC [26], branching temporal logic in the case of SMV [19] and its relatives. However, the model checking problem for *temporal logics of knowledge* has received comparatively little attention. While Halpern and Vardi proposed the use of model checking as an alternative to deduction for logics of knowledge as long ago as 1991, their proposal focussed on logics with no temporal component [12]. Ron van der Meyden studied the complexity of the model checking problem for a particular class of (essentially infinite state) systems of knowledge and time, and showed that the problem was complex (PSPACE-complete in the best case, undecidable in the worst) for this class [22].

In this paper, we address ourselves to the problem of model checking as an approach to showing that finite state systems satisfy specifications expressed in logics that combine temporal and knowledge components.

The remainder of this paper is organised as follows. In section 1.1, we shortly elaborate on modal logics of knowledge — readers familiar with the epistemic logic literature may wish to skip this section. In Section 2, we introduce a temporal logic of knowledge (Halpern and Vardi's logic CKL_n [11]). We then develop an approach to CKL_n model checking that combines ideas from the *interpreted systems* semantics for knowledge [8] with the *logic of local propositions* developed by Engelhardt et al [7]. In our approach, CKL_n model checking can be reduced to linear temporal logic model checking. After introducing and exploring the ideas underpinning the approach, we present a three case studies — the bit transmission problem, leadership election and the Dining Cryptographers — in which SPIN was used to establish temporal epistemic properties of a PROMELA system. The first two of these case studies (bit transmission and the leadership election scenario) are examples of *cooperative* systems, in the sense that we want to prove *positive* epistemic properties of the system (i.e.., a property of the form "eventually, agent i comes to know φ"). In such systems, we argue, our local propositions approach works well. However, in the third example, we wish to prove a *negative* epistemic property (i.e., a property of the form "no agent comes to know φ"). In the former case, we are proving an *existential* property, along the lines of "there exists a local proposition p such that…". However, in the latter case, we are proving a *universal* property — we need to quantify over local propositions in order to prove the absence of knowledge. As a consequence, we argue, while using the local propositions approach to establish epistemic properties of cooperative systems is usually straightforward, the approach is less well suited to adversarial systems. We conclude with some comments on issues for future research.

1.1. Background

Model checking techniques originated — and are most widely understood — as a technique for automatically verifying that finite state systems satisfy formal specifications [3]. These formal specifications are most commonly expressed either as formulae of the branching time temporal logic CTL (in the case of the SMV model checker and its relatives [19, 3]) or as formulae of Linear Temporal Logic (in the case of SPIN [15, 16] and FORSPEC [26]). Comparatively little attention has been given in the model checking community to *epistemic logic*: the modal logic of knowledge. Epistemic modal logics are widely recognised as having originated in the work of Jaakko Hintikka, a philosopher who in the early 1960s showed how certain modal logics could be used to formally capture some intuitions about the nature of knowledge [14]. In the 1980s, it was recognised that epistemic logics have an important role to play in the theory of distributed systems. In particular, it was demonstrated that epistemic logics can be used to formally express the desired behaviour of protocols. For example, when specifying a communication protocol, it is quite natural to wish to represent requirements such as "if process i knows that process j has received packet m, then i should send packet $m + 1$". Using epistemic logic, such requirements can be expressed both formally and naturally.

One of the key reasons why modal logics of knowledge have achieved such prominence was the discovery by Halpern and colleagues in the mid 1980s that S5 epistemic logics could be given a natural interpretation in terms of the states of processes — commonly called *agents* — in a distributed system. The model that has received the most widespread interest is known as the *interpreted systems* model [8].

In addition to interest in the use of epistemic logics in the specification of communicating systems, there has recently been interest in the use of knowledge logics for directly programming systems [8, 9]. A *knowledge-based program* has the general form:

```
case of
    if  $K_i\varphi_1$ do $a_1$
    ...
    if  $K_i\varphi_n$ do $a_n$
end case
```

The intuitive interpretation of such a program is that of a collection of rules; the left-hand side of each rule represents a condition, expressed in epistemic logic, of what an agent knows. If the condition is satisfied, then the corresponding action (program statement) is executed. Along with other researchers, (e.g., [7]), we take the view that such programs are best understood as specifications for systems — knowledge-based programs are not in a form that can be directly executed. There have been some studies on the computational complexity of automatically synthesising executable programs from knowledge-based programs [23, 24].

Despite the level of interest in using logics of knowledge for specifying communicating systems, there has been comparatively little work on *model checking* for such logics. In 1991 — somewhat prior to the current growth in interest in model checking — Halpern and Vardi proposed the use of model checking as an alternative to deduction for modal logics of knowledge [12]. They showed that the model checking problem for multi-agent S5 logics of knowledge was tractable, and speculated that the approach might have wider applications in this community; but to the best of our knowledge (no pun intended), no further work on this topic was reported. While the computational complexity of the satisfiability and validity problems for temporal logics of knowledge has been studied exhaustively by Halpern and Vardi [11], no such similar studies appear to have been carried out with respect to model checking. The closest work with which we are familiar is that of Vardi [25], who investigated the problem of when a concrete program could be said to implement a knowledge-based program. He showed that in the general case — where the knowledge-based program could contain knowledge tests with temporal modalities — the complexity of the problem coincided with that of the model checking problem for Linear Temporal Logic, i.e., it is PSPACE-complete (see, e.g., [6]).

Also closely related is the work of van der Meyden, who investigated the model checking problem for a small class of temporal knowledge logics: those in which agents are assumed to have *perfect recall* [22]. He established that the model checking problem for this class varies from PSPACE-complete in the "best" case to undecidable in the worst. However, van der Meyden did not investigate "practical" model checking for knowledge and time. Rao and Georgeff investigated the model checking problem for a range of logics combining temporal (CTL) and modal components, although their study was rather abstract — they did not implement any of the techniques they developed, and did not consider S5 logics of knowledge [21]. Finally, Benerecetti and Giunchiglia developed techniques for similar temporal modal logics, but these logics had an unusual (non-Kripke) semantics [1].

2. A Temporal Logic of Knowledge

We are concerned with modelling systems composed of multiple agents, each of which is an independently operating process. Let $Ag = \{1, \ldots, n\}$ denote the set of agents. We assume each agent $i \in Ag$ can be in any of a set L_i of local states. An agent's local state contains all the information required to completely characterise the state of the agent: the value of each of its local variables, together with the value of its program counter. In particular, the *information* available to an agent is determined by its local state. The state of a system at any moment can thus be characterised by a tuple $\langle l_1, \ldots, l_n \rangle$, where $l_i \in L_i$ is the local state of agent i at this moment. We let $G \subseteq L_1 \times \cdots \times L_n$ denote the *reachable global* states of the system (i.e., the set of states that a system may possibly enter during a legal computation sequence). (Notice that we have not explicitly introduced *environments*, although it is quite common to do so in the literature [8]: for simplicity, we assume that an environment can be modelled as an agent in the system.)

A *run* is a function

$$r : I\!N \to G$$

which associates with every natural number $u \in I\!N$ a global state $r(u)$. The idea is that a run represents one possible computation of a system: in general, a system may have a number of possible runs, and so we say a *system* is a set of runs; we use $\mathcal{R}$ to denote a system. A run together with a time point is a *point*: a point (r, u) defines a global state $r(u)$. We denote the i'th component of the tuple $r(u)$ by $r_i(u)$. Thus $r_i(u)$ is the local state of agent i in run r at "time" u.

Following conventional practice, we associate with every agent $i \in Ag$ an equivalence relation $\sim_i$ over the set of points [8, p.111]:

$$(r, u) \sim_i (r', v) \qquad \text{iff} \qquad r_i(u) = r'_i(v).$$

If $(r, u) \sim_i (r', v)$, then we say that (r, u) is indistinguishable from (r', v) from the point of view of i, or, alternatively, that i carries exactly the same information in (r, u) as in (r', v).

We use the relation $\sim_i$ to give a semantics to the knowledge modalities in CKL_n. To give a semantics to the "common knowledge" modality C_Γ, we introduce two further relations, $\sim^E_\Gamma$ and $\sim^C_\Gamma$. Given a set $\Gamma \subseteq Ag$ of agents, we define the relation $\sim^E_\Gamma$ as $\sim^E_\Gamma \mathrel{\hat{=}} \bigcup_{i \in \Gamma} \sim_i$ and we define the relation $\sim^C_\Gamma$ as the transitive closure of $\sim^E_\Gamma$.

A *model* or *interpreted system* for CKL_n is a pair $\mathcal{I} = \langle \mathcal{R}, \pi \rangle$, where $\mathcal{R}$ is a system and

$$\pi : \mathcal{R} \times I\!N \rightarrow 2^\Phi$$

is a valuation function, which gives the set of primitive propositions true at each point in $\mathcal{R}$ [8, pp.110–111].

Notice that we model a system as a set of infinite runs, which may at first sight appear to be at odds with the notion of the finite state systems that model checking is generally applied to. In fact, there is no contradiction. Given a Kripke structure $\langle G, R \subseteq G \times G, G_0 \subseteq G, \pi \rangle$ for a CTL-like logic (where R is a total "next time" relation and G_0 are initial states), we can obtain an interpreted system by "unwinding" the relation R starting from initial states G_0 to obtain a set of infinite runs.

Syntactically, CKL_n is propositional temporal logic augmented by an indexed set of modal operators K_i, one for each agent $i \in Ag$, and common knowledge operators C_Γ, where $\Gamma \subseteq Ag$. The formula $K_i\varphi$ is read as "agent i knows φ"; the formula $C_\Gamma\varphi$ means "it is common knowledge in Γ that φ".

Formulae are constructed from a set $\Phi = \{p, q, r, \ldots\}$ of *primitive propositions.* The language contains the standard propositional connectives $\neg$ (not), $\vee$ (or), $\wedge$ (and), $\rightarrow$ (implies) and $\leftrightarrow$ (if, and only if). For the temporal dimension we take the usual set of future-time connectives $\bigcirc$ (*next*),$\Diamond$ (*sometime* or *eventually*), $\Box$ (*always*), $\mathcal{U}$ (*until*) and $\mathcal{W}$ (*unless* or *weak until*).

The set *wff*(CKL_n) of well-formed formulae of CKL_n is defined by the following grammar:

$$\begin{array}{llcll}
\langle \mathcal{I},(r,u)\rangle & \models_{CKL_n} & \mathbf{true} & & \\
\langle \mathcal{I},(r,u)\rangle & \models_{CKL_n} & p & \text{iff} & p \in \pi(r,u) \qquad \text{(where } p \in \Phi) \\
\langle \mathcal{I},(r,u)\rangle & \models_{CKL_n} & \neg\varphi & \text{iff} & \langle \mathcal{I},(r,u)\rangle \not\models_{CKL_n} \varphi \\
\langle \mathcal{I},(r,u)\rangle & \models_{CKL_n} & \varphi \vee \psi & \text{iff} & \langle \mathcal{I},(r,u)\rangle \models_{CKL_n} \varphi \text{ or } \langle \mathcal{I},(r,u)\rangle \models_{CKL_n} \psi \\
\langle \mathcal{I},(r,u)\rangle & \models_{CKL_n} & K_i\varphi & \text{iff} & \text{for all } (r',v) \text{ in } \mathcal{I}, \text{ if } (r,u) \sim_i (r',v) \\
 & & & & \text{then } \langle \mathcal{I},(r',v)\rangle \models_{CKL_n} \varphi \\
\langle \mathcal{I},(r,u)\rangle & \models_{CKL_n} & C_\Gamma\varphi & \text{iff} & \text{for all } (r',v) \text{ in } \mathcal{I}, \text{ if } (r,u) \sim_\Gamma^C (r',v) \\
 & & & & \text{then } \langle \mathcal{I},(r',v)\rangle \models_{CKL_n} \varphi \\
\langle \mathcal{I},(r,u)\rangle & \models_{CKL_n} & \bigcirc\varphi & \text{iff} & \langle \mathcal{I},(r,u+1)\rangle \models_{CKL_n} \varphi \\
\langle \mathcal{I},(r,u)\rangle & \models_{CKL_n} & \varphi\,\mathcal{U}\,\psi & \text{iff} & \exists v \in I\!N \text{ s.t. } (u \leq v) \text{ and } \langle \mathcal{I},(r,v)\rangle \models_{CKL_n} \psi \\
 & & & & \text{and } \forall w \in \{u,\ldots,v-1\}, \text{ we have } \langle \mathcal{I},(r,w)\rangle \models_{CKL_n} \varphi
\end{array}$$

Figure 1. Semantics of CKL_n

$$\begin{array}{rcll}
\langle \mathit{wff}\rangle & ::= & \mathbf{true} & /\text{* logical constant for truth */} \\
 & | & \text{any element of } \Phi & /\text{* primitive propositions */} \\
 & | & \neg\langle \mathit{wff}\rangle & /\text{* negation */} \\
 & | & \langle \mathit{wff}\rangle \vee \langle \mathit{wff}\rangle & /\text{* disjunction */} \\
 & | & \bigcirc\langle \mathit{wff}\rangle & /\text{* next */} \\
 & | & \langle \mathit{wff}\rangle\,\mathcal{U}\,\langle \mathit{wff}\rangle & /\text{* until */} \\
 & | & K_i\langle \mathit{wff}\rangle & /\text{* } (i \in Ag) \text{ agent } i \text{ knows */} \\
 & | & C_\Gamma\langle \mathit{wff}\rangle & /\text{* } (\Gamma \subseteq Ag) \text{ it is common knowledge */} \\
 & & & /\text{* in } \Gamma \text{ that */}
\end{array}$$

The semantics of CKL_n are given via the satisfaction relation $\models_{CKL_n}$, which holds between pairs of the form $\langle \mathcal{I},(r,u)\rangle$, (where $\mathcal{I}$ is an interpreted system and (r,u) is a point in $\mathcal{I}$), and formulae of CKL_n. We read $\langle \mathcal{I},(r,u)\rangle \models_{CKL_n} \varphi$ as "φ is satisfied (equivalently, is true) at point (r,u) in $\mathcal{I}$". The rules defining $\models_{CKL_n}$ are given in Figure 1.

Semantic rules are only given for the temporal connectives $\bigcirc$ and $\mathcal{U}$: the remaining temporal connectives are introduced as abbreviations, as follows.

$$\begin{array}{rcl}
\Diamond\varphi & \hat{=} & \mathbf{true}\,\mathcal{U}\,\varphi \\
\Box\varphi & \hat{=} & \neg\Diamond\neg\varphi \\
\varphi\,\mathcal{W}\,\psi & \hat{=} & \varphi\,\mathcal{U}\,\psi \vee \Box\varphi
\end{array}$$

The remaining propositional connectives $(\wedge, \rightarrow, \leftrightarrow)$ are also assumed to be defined in terms of $\vee$ and $\neg$.

Notice that CKL_n is an expressive language. In particular, using the language it is possible to express the fact that a statement is true in all states of an interpreted system that can play a part in the interpretation of a formula. To see how this is done, we define a *universal modality* $\Box^*$, which is defined as the maximal solution to the following fixed point formula:

$$\Box^*\varphi \leftrightarrow \Box(\varphi \wedge C_{Ag}\,\Box^*\varphi).$$

To illustrate the properties of $\Box^*$, we define a *reachability relation.* First, we say point (r', v) is *directly* reachable from (r, u) (written $(r, u) \leadsto (r', v)$) iff:

- $r = r'$ and $v \geq u$; or
- $(r, u) \sim_i (r', v)$ for some agent $i \in Ag$.

We then define the reachability relation $\leadsto^*$ as the transitive closure of $\leadsto$. Now:

PROPOSITION 1 ([4]). *Let $\mathcal{I}$ be an interpreted system and (r, u) and (r', v) be points in $\mathcal{I}$ such that $\langle \mathcal{I}, (r, u)\rangle \models_{CKL_n} \Box^*\varphi$ and $(r, u) \leadsto^* (r', v)$. Then $\langle \mathcal{I}, (r', v)\rangle \models_{CKL_n} \varphi$.*

Proof: Assume that $\langle \mathcal{I}, (r, u)\rangle \models_{CKL_n} \Box^*\varphi$ and $(r, u) \leadsto^* (r', v)$. Then there exists a sequence of points $(r_1, u_1), (r_2, u_2), \ldots, (r_k, u_k)$ such that $(r, u) = (r_1, u_1)$, $(r', v) = (r_k, u_k)$, and for all $1 \leq l < k$, we have either:

1. $r_l = r_{l+1}$ and $u_{l+1} > u_l$; or
2. $(r_l, u_l) \sim_i (r_{l+1}, u_{l+1})$ for some agent $i \in Ag$.

The proof is then by induction on k; more precisely, we show that for all $1 \leq l < k$, if $\langle I, (r_l, u_l)\rangle \models_{CKL_n} \Box^*\varphi$ then $\langle I, (r_{l+1}, u_{l+1})\rangle \models_{CKL_n} \Box^*\varphi$. For the inductive step, there are two cases to consider, corresponding to the cases in the definition of $\leadsto$, above, and both are straightforward. □

Thus if $\langle \mathcal{I}, (r, u)\rangle \models \Box^*\varphi$ then φ is true *in all points that can play a part in the evaluation of formulae from* (r, u).

Linear Temporal Logic and Propositional Logic

Now consider the language and logic obtained from CKL_n by omitting knowledge and common knowledge modalities: we get *Linear Temporal Logic* (LTL) (see, e.g., [17, 18]). Formula of LTL are interpreted with respect to points in interpreted systems, as with CKL_n, but note that the interpretation of an LTL formula will depend *only* on the run within which it is interpreted. The truth or falsity of an LTL formula φ when interpreted on a point (r, u) in $\mathcal{I}$ will depend only on the run r, and will not be dependent on other runs in $\mathcal{I}$. This is not the case for knowledge modalities, as these can express properties of other runs. We write $\langle \mathcal{I}, (r, u) \rangle \models_{\text{LTL}} \varphi$ to indicate that LTL formula φ is satisfied at point (r, u) in $\mathcal{I}$: thus "$\models_{\text{LTL}}$" serves as the LTL satisfaction relation. We refer to the subset of LTL obtained by not permitting temporal logic connectives as propositional logic.

Local Propositions

We now introduce the notion of a *local* proposition [7]. Local propositions play an important role in our reduction of CKL_n model checking to LTL model checking. If i is an agent, then an i-local proposition is a formula of propositional logic whose interpretation is the same in each of the points in each equivalence class induced by the $\sim_i$ relation. Formally, a propositional logic formula φ is said to be i-local iff:

for all points $(r, u), (r', v)$ in $\mathcal{I}$,
if $(r, u) \sim_i (r', v)$, then $\langle \mathcal{I}, (r, u) \rangle \models_{CKL_n} \varphi$ iff $\langle \mathcal{I}, (r', v) \rangle \models_{CKL_n} \varphi$

To further understand the idea of an i-local proposition, assume — without loss of too much generality — that the local state of any agent i is a tuple of *local variables* $\langle b_{i_1}, \ldots, b_{i_k} \rangle$ each of which has the value 0 or 1 at any given time. (Of course, this is *exactly* what the state of any conventional computer process actually is.) The indexed set $\Phi_i = \{b_{i_1}, \ldots, b_{i_k}\} \subseteq \Phi$ of primitive propositions is assumed to form part of the vocabulary of the CKL_n language: $\Phi = \bigcup_{i \in Ag} \Phi_i$. Note that the Φ_i's need not be mutually disjoint — it is possible that variables are *shared* between agents, if the same variable appears in the state of more than one agent (although this requires an addition semantic constraint, described below).

We assume the obvious interpretation of local variables: $b_{i_j} \in \pi(r, u)$ iff the bit b_{i_j} has the value 1 in the state $r_i(u)$. If a variable b is shared between agents i and j, then we require that the variable has the same value inside both agents in any given system state: this requirement ensures that the valuation function π can give a unique, well-defined value to shared variables. It is straightforward to show the following:

PROPOSITION 2. *If φ is a formula of propositional logic containing only variables over Φ_i, then φ is i-local.*

Proof: Immediate from the fact that $(r, u) \sim_i (r', v)$ iff $r_i(u) = r'_i(v)$. □

The idea of such local propositions is that they relate the semantic definition of knowledge as given in Figure 1 with a syntactic one: let g be a global state verifying the i-local proposition φ_g. Then we have $\langle \mathcal{I}, (r, u) \rangle \models_{KL_n} K_i \psi$ iff for all (r', v) in $\mathcal{I}$, if $\langle \mathcal{I}, (r', v) \rangle \models_{KL_n} \varphi_g$ then $\langle \mathcal{I}, (r', v) \rangle \models_{KL_n} \psi$.

We can extend the notion of a local proposition to sets of agents. Given a set $\Gamma \subseteq Ag$ of agents and a propositional formula φ, we say that φ is Γ-*local* iff φ is i-local for all $i \in \Gamma$. We can prove an immediate analogue of Proposition 2:

PROPOSITION 3. *If φ is a formula of propositional logic containing only variables that are shared by the agents in Γ, then φ is Γ-local.*

Proof: We need to show that if φ depends only on Γ-shared variables, then if $(r, u) \sim_\Gamma^C (r', v)$ then $\langle \mathcal{I}, (r, u) \rangle \models_{CKL_n} \varphi$ iff $\langle \mathcal{I}, (r', v) \rangle \models_{CKL_n} \varphi$. Assume $(r, u) \sim_\Gamma^C (r', v)$. Then there is a sequence of points (r_1, u_1), (r_2, u_2), $\ldots, (r_k, u_k)$ such that $(r, u) = (r_1, u_1)$, $(r', v) = (r_k, u_k)$, and for all $1 \leq l < k$, we have $(r_l, u_l) \sim_i (r_{l+1}, u_{l+1})$ for some agent $i \in \Gamma$. Now if $(r_l, u_l) \sim_i (r_{l+1}, u_{l+1})$ then by definition the local state of i must the same in (r_l, u_l) and (r_{l+1}, u_{l+1}), and in particular, any Γ-shared variables must have the same values in (r_l, u_l) as (r_{l+1}, u_{l+1}). So any formula depending on these values will have the same interpretation in (r_l, u_l) and (r_{l+1}, u_{l+1}). Thus $\langle \mathcal{I}, (r, u) \rangle \models_{CKL_n} \varphi$ iff $\langle \mathcal{I}, (r', v) \rangle \models_{CKL_n} \varphi$. □

In addition, we can show:

PROPOSITION 4. *Let $\mathcal{I}$ be an interpreted system, let (r, u) and (r', v) be points in $\mathcal{I}$, and let φ be a Γ-local proposition. Then if $(r, u) \sim_\Gamma^C (r', v)$ then $\langle \mathcal{I}, (r, u) \rangle \models_{CKL_n} \varphi$ iff $\langle \mathcal{I}, (r', v) \rangle \models_{CKL_n} \varphi$.*

Proof: Assume $(r, u) \sim_\Gamma^C (r', v)$. Then as before, there exists a sequence of points $(r_1, u_1), (r_2, u_2), \ldots, (r_k, u_k)$ such that $(r, u) = (r_1, u_1)$, $(r', v) = (r_k, u_k)$, and for all $1 \leq l < k$, we have $(r_l, u_l) \sim_i (r_{l+1}, u_{l+1})$ for some agent $i \in \Gamma$. As φ is Γ-local, it is i-local for all $i \in \Gamma$. Hence by the definition of i-local, $\langle \mathcal{I}, (r_l, u_l) \rangle \models_{CKL_n} \varphi$ iff $\langle \mathcal{I}, (r_{l+1}, u_{l+1}) \rangle \models_{CKL_n} \varphi$. Hence $\langle \mathcal{I}, (r, u) \rangle \models_{CKL_n} \varphi$ iff $\langle \mathcal{I}, (r', v) \rangle \models_{CKL_n} \varphi$. □

3. *CKL$_n$* Model Checking through LTL Model Checking

The *model checking* problem for CKL_n is as follows. Given an interpreted system $\mathcal{I} = \langle \mathcal{R}, \pi \rangle$, together with a formula φ of CKL_n, return the set of points at which φ is satisfied in $\mathcal{I}$, i.e., the set

$$\{(r,u) \mid r \in \mathcal{R}, u \in I\!N, \text{ and } \langle \mathcal{I}, (r,u)\rangle \models_{CKL_n} \varphi\}.$$

This problem is too abstract for most practical model checking problems (invariant properties will be true in every state of the system — there is clearly no way a practical model checker would be able to enumerate this set!). For this reason, we are generally concerned with a slightly simpler version of this problem. Hereafter, when we refer to the model checking problem for CKL_n, we mean the problem of determining whether, given an interpreted system $\mathcal{I} = \langle \mathcal{R}, \pi \rangle$ and a formula φ, the formula φ is true in the initial state of every run in $\mathcal{R}$, i.e., whether or not

$$\forall r \in \mathcal{R} \text{ we have } \langle \mathcal{I}, (r,0)\rangle \models_{CKL_n} \varphi.$$

We say that $\mathcal{I}$ realises φ if it satisfies this property. Given an interpreted system $\mathcal{I}$ and CKL_n formula φ, we write $mc_{CKL_n}(\mathcal{I}, \varphi)$ to stand for the fact that $\mathcal{I}$ realises φ, i.e.,

$$mc_{CKL_n}(\langle \mathcal{R}, \pi\rangle, \varphi) \quad \text{iff} \quad \forall r \in \mathcal{R} \text{ we have } \langle\langle \mathcal{R}, \pi\rangle, (r,0)\rangle \models_{CKL_n} \varphi.$$

The Main Idea

At present, we do not have a model checker for CKL_n (although there is no reason in principle why one should not be implemented). What we *do* have available, however, is a model checker for Linear Temporal Logic (LTL), for example in the form of SPIN [15, 16]. SPIN takes as input a system, (described using the PROMELA language), and a formula of propositional LTL: it then checks whether or not this formula is satisfied in the first state of every run of the system. If it is not — if there is a run that fails to satisfy the formula — then it reports this run as a counter example. The model checking problem that SPIN solves is thus as follows. For any system $\mathcal{I} = \langle \mathcal{R}, \pi \rangle$ and formula φ of LTL, it determines whether or not:

$$\forall r \in \mathcal{R} \text{ we have } \langle \mathcal{I}, (r,0)\rangle \models_{\text{LTL}} \varphi.$$

If φ is an LTL formula and $\mathcal{I}$ is a system, then we write $mc_{\text{LTL}}(\mathcal{I}, \varphi)$ to indicate that $\mathcal{I}$ realises φ:

$$mc_{\text{LTL}}(\langle \mathcal{R}, \pi\rangle, \varphi) \quad \text{iff} \quad \forall r \in \mathcal{R} \text{ we have } \langle\langle \mathcal{R}, \pi\rangle, (r,0)\rangle \models_{\text{LTL}} \varphi.$$

We now turn to one of the main ideas underpinning this article: we show how CKL_n model checking can be reduced to LTL model checking. Our approach takes inspiration from work on the Logic of Local Propositions (LLP) [7]. LLP is a modal logic with a single universal modality, $\mathtt{Nec}$, and which allows quantification over propositions. A formula $\mathtt{Nec}\,\varphi$ of LLP means that φ is true in all states. LLP has a collection of quantifiers $\forall_i$, $\exists_i$, (where i is an agent), which allow quantification over propositions that are *local* to an agent. The intuition is that a proposition is local to an agent i if i is able to determine its truth using only locally available information — information available in its state. The key insight of [7] is that by using these quantifiers, one can define knowledge modalities. For example:

$$K_i\varphi \mathrel{\hat{=}} \exists_i q[q \wedge \mathtt{Nec}(q \rightarrow \varphi)] \qquad (1)$$

Thus an agent i knows φ iff there is a proposition q local to i such that q is true, and whenever q is true, φ is also true. In [7], it is proved that this definition of knowledge corresponds to the conventional one, given in terms of Kripke structures and accessible worlds [8].

We now show how we can make use of these ideas when considering the model checking problem for CKL_n. Suppose we want to determine whether or not the property $\Diamond K_i p$ is true of some system $\mathcal{I}$. That is, we want to determine whether or not

$$mc_{CKL_n}(\mathcal{I}, \Diamond K_i p)$$

Now (1) suggests the following approach to this problem. In order to show this, all we have to do is find some proposition ψ that is local to i (i.e., ψ is a predicate over i's state), such that

$$mc_{\mathrm{LTL}}(\mathcal{I}, \Diamond\psi \wedge \Box(\psi \rightarrow p))$$

Notice that the formula to be model checked has two components. The first ($\Diamond\psi$) corresponds in structure to the original input formula (with knowledge modalities replaced by propositions). The second component ($\Box(\psi \rightarrow p)$) represents a constraint (an invariant) that must hold.

Thus we have reduced a CKL_n model checking problem to an LTL model checking problem — and since we have LTL model checking tools available — SPIN — this suggests that we can — at least partially — automate the process of model checking CKL_n.

The Formal Details

We now present the formal details of our reduction approach. To begin with, we will consider just how model checking a statement of the form $K_i\varphi$ can be reduced to LTL model checking. We define a function lp_i, which takes as argument an interpreted system, and an LTL formula φ, and returns a local proposition that globally implies φ:

DEFINITION 1 (lp_i).

$$lp_i(\mathcal{I}, (r,u), \varphi) \hat{=} \begin{cases} \psi & \textit{ψ is an i-local proposition such that} \\ & \quad mc_{\mathrm{LTL}}(\mathcal{I}, \Box(\psi \to \varphi)) \\ & \quad \textit{and } \langle \mathcal{I}, (r,u) \rangle \models_{\mathrm{LTL}} \psi \\ \textbf{false} & \textit{if no such formula exists.} \end{cases}$$

If $lp_i(\mathcal{I}, (r,u), \varphi) = \psi$, then we say that ψ serves as an i-local proposition for φ in (r,u). (As an aside, note that the lp_i function is very similar in spirit to the "sound local predicate" function S_i of Engelhardt et al [7].) We can now show the following.

PROPOSITION 5. *Let $\mathcal{I}$ be an interpreted system, let (r,u) be a point in $\mathcal{I}$, and let φ be an* LTL *formula such that $lp_i(\mathcal{I}, (r,u), \varphi) = \psi$. Then:*

$$\langle \mathcal{I}, (r,u) \rangle \models_{CKL_n} K_i\varphi \qquad \textit{iff} \qquad \langle \mathcal{I}, (r,u) \rangle \models_{\mathrm{LTL}} \psi$$

Proof: (Left to right.) Immediate from the definition of lp_i. (Right to left.) We need to show that $\langle \mathcal{I}, (r,u) \rangle \models_{\mathrm{LTL}} \psi$ implies $\langle \mathcal{I}, (r,u) \rangle \models_{CKL_n} K_i\varphi$. From the definition of lp_i, we know that $\langle \mathcal{I}, (r,u) \rangle \models_{\mathrm{LTL}} \psi$ and in addition, that $mc_{\mathrm{LTL}}(\mathcal{I}, \Box(\psi \to \varphi))$. Since $mc_{\mathrm{LTL}}(\mathcal{I}, \Box(\psi \to \psi))$, then for all points (r', v) in $\mathcal{I}$, we have $\langle \mathcal{I}, (r', v) \rangle \models \psi \to \varphi$, and in particular, $\langle \mathcal{I}, (r'', w) \rangle \models_{\mathrm{LTL}} \psi \to \varphi$ for all (r'', w) such that $(r,u) \sim_i (r'', w)$. Since ψ is i-local, then if $\langle \mathcal{I}, (r,u) \rangle \models_{\mathrm{LTL}} \psi$ then $\langle \mathcal{I}, (r'', w) \rangle \models_{\mathrm{LTL}} \psi$ for all (r'', w) such that $(r,u) \sim_i (r'', w)$, and thus $\langle \mathcal{I}, (r'', w) \rangle \models_{\mathrm{LTL}} \varphi$ for all (r'', w) such that $(r,u) \sim_i (r'', w)$ and so $\langle \mathcal{I}, (r,u) \rangle \models_{CKL_n} K_i\varphi$. □

In the same way, we can extend the function lp_i to sets of agents. If $\Gamma \subseteq Ag$, then we can define lp_Γ as:

$$lp_\Gamma(\mathcal{I}, \varphi) \hat{=} \begin{cases} \psi & \text{ψ is an Γ-local proposition such that} \\ & \quad mc_{\mathrm{LTL}}(\mathcal{I}, \Box(\psi \to \varphi)) \\ & \quad \text{and } \langle \mathcal{I}, (r,u) \rangle \models_{\mathrm{LTL}} \psi \\ \textbf{false} & \text{if no such formula exists.} \end{cases}$$

The following result can now be proved.

PROPOSITION 6. *Let $\mathcal{I}$ be an interpreted system, let (r,u) be a point in $\mathcal{I}$, and let φ be an* LTL *formula such that $lp_\Gamma(\mathcal{I}, (r,u), \varphi) = \psi$. Then:*

$$\langle \mathcal{I}, (r,u) \rangle \models_{CKL_n} C_\Gamma\varphi \qquad \textit{iff} \qquad \langle \mathcal{I}, (r,u) \rangle \models_{\mathrm{LTL}} \psi$$

Proof: As Proposition 5, making use of Proposition 4. □

Finally, suppose we have some LTL formula φ for which we can establish $mc_{\text{LTL}}(\mathcal{I}, \ \Box\varphi)$. In this case φ is an invariant of system $\mathcal{I}$ — it is true in all the states of $\mathcal{I}$ that are reachable through some possible computation. From this we can immediately conclude the following.

PROPOSITION 7. *Let $\mathcal{I}$ be an interpreted system, and let φ be an* LTL *formula such that $mc_{\text{LTL}}(\mathcal{I}, \ \Box\varphi)$. Then for any point (r, u) in $\mathcal{I}$, we have $\langle\mathcal{I}, (r, u)\rangle \models_{CKL_n} \Box^*\varphi$.*

We now have a route to model checking (a subset of) CKL_n formulae by using only LTL model checking: When faced with the problem of determining whether $\langle\mathcal{I}, (r, u)\rangle \models_{CKL_n} K_i\varphi$, we can attempt to find a ψ such that $lp_i(\mathcal{I}, (r, u), \varphi) = \psi$, and check that $\langle\mathcal{I}, (r, u)\rangle \models_{\text{LTL}} \psi$. Notice that finding the i-local proposition ψ will itself require solving the LTL model checking problem $mc_{\text{LTL}}(\mathcal{I}, \ \Box(\psi \rightarrow \varphi))$.

Notice that the approach can deal with nested knowledge operators. We will see an example of this in the following section.

4. Case Studies

In this section, we will show how our CKL_n model checking technique was applied in two case studies.

4.1. The Bit Transmission Problem

Our first case study is the *bit transmission problem.* We adapt our discussion of this problem from [20, pp.39–44]. The bit transmission protocol was first studied in the context of epistemic logic by Halpern and Zuck [13]. The basic idea is that there are two agents, a sender and a receiver, who can communicate with one another through an unreliable communication medium. This medium may delete messages, but if a message does arrive at the recipient, then the message is correct. (It is also assumed that the environment satisfies a kind of fairness property, namely that if a message is sent infinitely often, then it eventually arrives.) The sender has a sequence of bits $x_0, x_1, \ldots, x_k$ that it desires to communicate to the receiver; when the receiver receives the bits, it prints them out. The goal is to derive a protocol that satisfies the safety requirement that the receiver never prints incorrect bits, and the liveness requirement that every bit will eventually be printed by the receiver.

The obvious solution to this problem involves sending acknowledgment messages, to indicate when a message was received. Halpern and Zuck's key insight was to recognise that an acknowledgment message in

```
$scount$ is a natural number variable
$rcount$ is a natural number variable

Sender
    $scount := 0$
    while $\mathbf{true}$ do
        read $x_{scount}$
        send $x_{scount}$ until $K_S K_R(x_{scount})$
        send "$K_S K_R(x_{scount})$" until $K_S K_R K_S K_R(x_{scount})$
        $scount := scount + 1$
    end-while
end-Sender

Receiver
    when $K_R(x_0)$ set $rcount := 0$
    while $\mathbf{true}$ do
        write $x_{rcount}$
        send "$K_R(x_{rcount})$" until $K_R K_S K_R(x_{rcount})$
        send "$K_R K_S K_R(x_{rcount})$" until $K_R(x_{rcount+1})$
        $rcount := rcount + 1$
    end-while
end-Receiver
```

Figure 2. The bit transmission protocol.

fact carries information about the knowledge state of the sender of the message. This motivated the development of the following knowledge-based protocol. After obtaining the first bit, the sender transmits it to the receiver. However, it cannot stop at this point, because for all it knows, the message may have been deleted by the environment. It thus continues to transmit the bit *until it knows the bit has been received.* At this point, the receiver knows the value of the bit that was transmitted, and the sender knows that the receiver knows the value of the bit — but the receiver does not know whether or not its acknowledgment was received. So the sender repeatedly sends a second acknowledgment, until it receives back a third acknowledgment from the receiver; when it receives this acknowledgment, it starts to transmit the next bit. When the receiver receives this bit, this indicates that its final (third) acknowledgment was received.

A pseudo-code version of the protocol is presented in Figure 2 (from [20, pp.39–44]). Note that we write x_i as a shorthand for "the value of bit x_i". Thus $K_R(x_i)$ means that the receiver (R) knows the value of bit x_i.

To demonstrate our ideas in a concrete setting, consider the PROMELA code given in Figure 3, where, for simplicity, we assume that message delivery is guaranteed. In a more complicated version, we may add deletion errors by having a process that can "steal" messages, but the knowledge properties at the specific points in the program — the labels, see below — would be the same.

Code is given for the sender and receiver agents and main variable declarations. The initialisation code is unremarkable, with one subtle exception. Suppose we were to initialise the `Send[]` array with the bits to be transmitted using a straightforward assignment statement. Then this array would remain fixed throughout every execution of the program — *and the values of the bits in this array would therefore be common knowledge to all agents in the system.* To get around this problem, we exploit SPIN's non-deterministic execution mechanism. We have a macro `INITIAL(V)`, where `V` is a variable name, which assigns a "random" (sic) bit to `V`.

```
#define INITIAL(V) \
  if               \
  :: 1 -> V = 0;   \
  :: 1 -> V = 1;   \
  fi
```

This macro is used to initialise both the `Send[]` and `Recv[]` arrays in the `init` process, ensuring that the values in these arrays may initially have different values in different computations. The goal of the protocol is that eventually, the values in the `Recv[]` array will have the same values that were initially in the `Send[]` array.

The general form of properties we prove is as follows:

$$at_i(\ell) \to K_i\varphi \tag{2}$$

where ℓ is a *program label*, and the unary predicate $at_i(\ell)$ means that the program counter of agent i is at the instruction labelled by ℓ. The use of the $at_i(\ldots)$ predicate in this way is common practice when reasoning about programs using temporal logic (see, e.g., [10, pp.70–71]). We use SPIN's remote reference mechanism (`P[X]@L`) to define $at_i(\ldots)$ predicates.

The first property we prove is that whenever the receiver is at state R_3, (and so is about to send the acknowledgment), that it knows the value of the bit it has received. Let $\mathcal{BTP}$ be the interpreted system for the bit transmission problem (where the assignment π is clear from the name for the atomic propositions that we use). We want to verify, that for all $r \in \mathcal{BTP}$:

```
#define ACK    10       /* K_R(x_i) */
#define ACK2   11       /* K_S K_R(x_i) */
#define ACK3   12       /* K_R K_S K_R(x_i) */
chan S = [10] of {int}; /* outward from sender */
chan R = [10] of {int}; /* outward from receiver */
local int Send[11];     /* message sent */
local int Recv[11];     /* message received */
int s_count = 0;        /* sender bit count */
int r_count = 0;        /* receiver bit count */
proctype SENDER() {
   S0:do
      :: (s_count < 10) ->
      S1: printf("sender sends bit %d", s_count);
      S2: S!Send[s_count];
      S3: R?ACK;
      S4: S!ACK2;
      S5: R?ACK3;
      S6: s_count++;
      :: (s_count == 10) ->
      S7: break
      od
}
proctype RECEIVER() {
   R0:do
      :: (r_count < 10) ->
      R1: S?Recv[r_count];
      R2: printf("receiver receives bit %d", r_count);
      R3: R!ACK;
      R4: S?ACK2;
      R5: R!ACK3;
      R6: r_count++;
      :: (r_count == 10) ->
      R7: break
      od
}
```

Figure 3. The bit transmission protocol in PROMELA (message delivery is guaranteed).

$$\langle \mathcal{BTP}, (r, 0)\rangle \models_{\mathrm{CKL}_n} \Box(at_R(R_3) \rightarrow K_R(Recv[r\_count] = Send[s\_count])) \tag{3}$$

This immediately translates to

$$\forall r \in \mathcal{BTP}, u \in I\!N : (5) \Rightarrow (6) \tag{4}$$

with

$$\langle \mathcal{BTP}, (r, u)\rangle \models_{\mathrm{CKL}_n} at_R(R_3) \tag{5}$$

and

$$\langle \mathcal{BTP}, (r,u)\rangle \models_{\mathrm{CKL}_n} K_R(\mathit{Recv}[r\_count] = \mathit{Send}[s\_count]) \tag{6}$$

To deal with (6), we must first find an R-local proposition for $\varphi = (\mathit{Recv}[r\_count] = \mathit{Send}[s\_count])$ to serve as the output of the function lp_R. We claim that $\psi = at_R(R_3)$ is a good candidate for that! Recall from Definition 1 that, in order to conclude that $lp_R(\langle \mathcal{BTP}, (r,u)\rangle, \varphi) = \psi$, we have to check three conditions, the first being that ψ is R-local, which is obvious, since $at_R(R_3)$ is about R's label. For the second condition, i.e. that $mc_{\mathrm{LTL}}(\mathcal{BTP},\ \Box(\psi \rightarrow \varphi)$, we generate a SPIN LTL claim for $\Box(at_R(R_3) \rightarrow (\mathit{Recv}[r\_count] = \mathit{Send}[s\_count)])$ as follows. We define propositions `p0` and `p1` to represent the CKL_n propositions $at_R(R_3)$ and φ, respectively.

```
#define p0 (RECEIVER[2]@R3)
#define p1 (Recv[r_count] == Send[s_count])
```

Finally, the property to be checked is written as the SPIN LTL formula:

```
!<>(p0 && !p1)
```

(We negate the claim to be verified, so that it can be used in a `never` claim, in the conventional manner for SPIN LTL claims.)

Now we come to the third condition, i.e., that $\langle \mathcal{BTP}, (r,u)\rangle \models_{\mathrm{LTL}} \psi$. This of course depends on the particular (r,u), but note that, by virtue of (5) and (6), we only are dealing with (r,u) that do satisfy ψ. Since we eliminated all the epistemic modalities, we have reduced problem (3) in fact to $\forall r \in \mathcal{BTP}, u \in I\!N : \langle \mathcal{BTP}, (r,u)\rangle \models_{\mathrm{LTL}} at_R(R_3) \rightarrow (at_R(R_3) \rightarrow (\mathit{Recv}[r\_count] = \mathit{Send}[s\_count]))$ which is equivalent to

$$\forall r \in \mathcal{BTP} \langle \mathcal{BTP}, (r,0)\rangle \models_{\mathrm{LTL}} \Box(at_R(R_3) \rightarrow (\mathit{Recv}[r\_count] = \mathit{Send}[s\_count])) \tag{7}$$

which, by our definition of mc_{LTL} gives

$$mc_{\mathrm{LTL}}(\mathcal{BTP},\ \Box(at_R(R_3) \rightarrow (\mathit{Recv}[r\_count] = \mathit{Send}[s\_count]))) \tag{8}$$

Summarising, we found a candidate for a local proposition $\psi_{(r,u)}$ for $\mathit{Recv}[r\_count] = \mathit{Send}[s\_count]$, by checking that (i) $\psi_{(r,u)}$ is R-local, and that (ii) $mc_{\mathrm{LTL}}(\mathcal{BTP},\ \Box(\psi_{(r,u)} \rightarrow (\mathit{Recv}[r\_count] = \mathit{Send}[s\_count])))$. We then observe that $\psi_{(r,u)}$ is indeed returned by lp_R, if moreover $\mathcal{BTP}, (r,u) \models_{\mathrm{LTL}} \psi_{(r,u)}$.

Next, we show a corresponding property for `SENDER`: when `SENDER` is at label `S4`, (i.e., about to send an `ACK2` message), then it knows that the receiver knows the value of the bit that was most recently sent.

$$\Box(at_S(S_4) \rightarrow K_S K_R(Recv[r\_count] = Send[s\_count])) \qquad (9)$$

Notice that this is a nested knowledge formula. To deal with it, we must first find an R-local proposition for $Recv[r\_count] = Send[s\_count]$. Note that, rather than taking $at_R(R_3)$ for this, we could also take $at_R(R_4)$ (R knows that $Recv[r\_count] = Send[s\_count]$ at position R_3, and he also know that this will not change when he just sends an acknowledgement). This can be checked in a similar way as was done above, for $at_R(R_4)$. This reduces the problem of checking (9) to that of checking:

$$\Box(at_S(S_4) \rightarrow K_S(at_R(R_4)))$$

But an S-local proposition for $at_R(R_4)$ is $at_S(S_4)$: when S is about to send `ACK2` then R is waiting for it. All in all, checking of (9) reduces to finding the local propositions $at_R(R_4)$ and $at_S(S_4)$ and then LTL checking

$$\Box(at_S(S_4) \rightarrow (at_S(S_4) \rightarrow at_R(R_4)))$$

Given the following macro

```
#define p2 (SENDER[1]@S4)
#define p3 (RECEIVER[2]@R4)
```

our property is easily represented and checked as the following SPIN LTL formula:

```
!<>(p2&&!p3)
```

In exactly the same way, we can check the following property:

$$\Box(at_R(R_5) \rightarrow K_R K_S K_R(Recv[r\_count] = Send[s\_count])) \ (10)$$

Finally, we give an example of proving the *absence* of knowledge. Let's suppose that agent R is at label R_0. Then in this case, the bits in the `Recv[]` array will have their initially assigned (essentially random) values. It may be that the bits are "correct", in the sense that they match those in `Send[]` but this is not necessarily the case:

$$\Box(at_R(R_0) \rightarrow \neg K_R(Recv[r\_count] = Send[s\_count]))$$

Now this formula will be *invalid* in a system if there is a single run of the system that satisfies the following property

$$\Diamond(at_R(R_0) \wedge Recv[r\_count] \neq Send[s\_count])$$

This property can be directly encoded and checked using SPIN.

4.2. The Leadership Election Protocol

In the leadership election protocol, a certain number of asynchronous communicating processes are arranged in a circular manner. Communication only occurs between neighbours. All processes have the same program, but differ by having a distinct *initial number* stored in their local memory. The objective for any process is to find the maximum of these numbers [5]. The process with the largest number is called the leader.

An unidirectional solution to this problem is an algorithm that sends messages in only one direction through the circle. A unidirectional solution requiring only $\mathcal{O}(n \log n)$ messages exists [5]. This algorithm has been implemented in PROMELA. This implementation is part of the example programs in the SPIN distribution. Two versions exist. The program for all processes is the same, but the initialisation procedure is different. In one version the initial numbers are assigned in always the same way. In the other version the initial numbers are assigned non-deterministically. For model checking knowledge another initialisation procedure is needed.

The SPIN distribution contains sample properties for this problem. The examples suggested are:

- `(<>[]oneLeader)`: In all runs eventually there will always be one leader
- `![] noLeader`: There is no run in which there is never a leader
- `<> elected`: In all runs eventually a leader is elected
- `[] (noLeader U oneLeader)`: For all runs there will be no leader unless there is one leader.

All these requirements are necessary requirements, but as a set they are not sufficient for characterizing a solution. An obvious requirement is for instance that the right leader is chosen. The right leader is the process with the largest initial number. One can consider the processes in the protocol to be communicating agents, with a common goal of gaining knowledge about leadership. This way of looking at the protocol is already suggested in the implementation, where a boolean variable knowswinner occurs. Apparently this variable indicates the presence of knowledge.

Using CKL_n we evaluate additional properties of this protocol. If the LTL example properties already described hold, exactly one leader is always chosen. We are interested in what knowledge the processes have when the protocol finishes. In particular we wish to prove that in every run, eventually every processor knows what the maximum number is.

We have modified the code provided slightly. Many interesting variables are local for a specific process. SPIN does not allow one to use these variables in requirements. We have replaced some local variables by global arrays of variables, one for each process. This change does not affect the implementation but makes it possible to refer to these variables in the requirements. The modified implementation is shown in Figure 4.

In order for an election protocol to be correct, the right leader must be chosen. The right leader in this case is the process with the highest initial value (variable Ini). Let, for any number x, the proposition m_x denote that x is the highest number. Since the numbers are assigned during the run, we should not test m_x at the beginning of the run. We use $\Diamond \Box m_x$, which expresses that from a certain moment on, m_x becomes and remains true. Let i be any process. The CKL_n property we wish to check is that $(\Diamond \Box m_x) \rightarrow \Diamond K_i m_x$. Since all processes behave the same, it is sufficient to check the property for $i = 0$.

In the initialisation procedures of both implementations provided, the number 5 is always the highest number. Everything that is always true is trivially known by all agents, so these initialisation procedures are of little use to us. We test the algorithm with four agents, using a new initialisation procedure that assigns, in a nondeterministic way, one of the following ranges:

```
4 5 6 7
5 6 7 8
6 7 8 9
7 8 9 10
```

The restriction to four agents was necessary to complete the model checking in reasonable time. The initialisation procedure is given in Figure 6.

Below, we list the formula to prove, the local proposition used and the two properties to check, respectively. From the ranges used by the initialisation one can see that 7 can either be a smallest or greatest number. Therefore, the fact that eventually agent 0 knows that 7 is the highest number is nontrivial.

```
byte nr_leaders = 0;

#define N 4 /* number of processes in the ring, N<7 */
#define L 8 /* 2xN */
byte I,J;
mtype = { one, two, winner };
chan q[N] = [L] of { mtype, byte};

byte Ini[6]; /* N<=6 randomize the process numbers */
byte nr[6];
```

Figure 4. Declarations for the leadership election protocol in PROMELA.

To prove the CKL_n property

$$(\Diamond \Box m7) \rightarrow \Diamond(K_0 m7)$$

using local proposition

```
p7 = (end&&nr[0]==7)
```

we check that

$$mc_{LTL}(\mathcal{I},\ \Box(p7 \rightarrow m7))$$

and that

$$mc_{LTL}(\mathcal{I}, (\Diamond \Box m7) \rightarrow (\Diamond p7)$$

The propositions used are defined as follows. Note that the initialisation procedure is the first process to start and thus gets process number 0. Agent 0 must therefore be referred to as `node[1]`. The proposition `end` expresses that process 0 is either at the label `LOST` or at label `LEADER`.

```
#define less6       (6>=Ini[0]&&6>=Ini[1]&&6>=Ini[2]&&6>=Ini[3])
#define less7       (7>=Ini[0]&&7>=Ini[1]&&7>=Ini[2]&&7>=Ini[3])
#define m7          (less7 && !less6)
#define end         (node[1]@LOST||node[1]@LEADER)
#define p7          (end&&nr[0]==7)
```

In summary, we have here verified knowledge properties that are stronger than the temporal properties provided with the example. Note that in order to do a meaningful verification, a new initialisation procedure was needed: in general one needs a non-deterministic initialisation procedure in order to verify knowledge properties, otherwise certain facts are trivially known to the agents.

```
proctype node (chan in, out; byte mynumber, procnum)
{  byte maximum = mynumber, neighbourR;
   bit Active=1,know_winner,is_leader;
   xr in;
   xs out;
   START:printf("MSC: %d", mynumber);
   out!one(mynumber);
   do
   :: in?one(nr[procnum]) ->
      if
      :: Active ->
         if
         :: nr[procnum] != maximum ->
         out!two(nr[procnum]);
         neighbourR = nr[procnum]
         :: else ->
            know_winner = 1;
            out!winner,nr[procnum];
         fi
         :: else ->
            out!one(nr[procnum])
         fi

   :: in?two(nr[procnum]) ->
      if
      :: Active ->
         if
         :: neighbourR > nr[procnum] && neighbourR > maximum ->
            maximum = neighbourR;
            out!one(neighbourR)
         :: else ->
            Active = 0
         fi
      :: else ->
         out!two(nr[procnum])
      fi
   :: in?winner,nr[procnum] ->
      if
      :: nr[procnum] != mynumber ->
         LOST:printf("MSC: LOST");
      :: else ->
         LEADER:printf("MSC: LEADER");
         is_leader=1;
         nr_leaders++;
         assert(nr_leaders == 1)
      fi;
      if
      :: know_winner
      :: else -> out!winner,nr[procnum]
      fi;
      break
   od
}
```

Figure 5. Agent definition for the leadership election protocol in PROMELA.

```
init {
byte proc;
atomic {
I = 1;/* pick a number to be assigned 1..N */
if
:: J=4;
:: J=5;
:: J=6;
:: J=7;
fi;
do
:: I <= N ->
if /* non-deterministic choice */
:: Ini[0] == 0 && N >= 1 -> Ini[0] = J
:: Ini[1] == 0 && N >= 2 -> Ini[1] = J
:: Ini[2] == 0 && N >= 3 -> Ini[2] = J
:: Ini[3] == 0 && N >= 4 -> Ini[3] = J
:: Ini[4] == 0 && N >= 5 -> Ini[4] = J
:: Ini[5] == 0 && N >= 6 -> Ini[5] = J /* works for up to N=6 */
fi;
I++;
J++;
:: I > N -> /* assigned all numbers 1..N */
break
od;

proc = 1;
do
:: proc <= N ->
run node (q[proc-1], q[proc%N], Ini[proc-1], proc-1);
proc++
:: proc > N ->
break
od
}
}
```

Figure 6. The `init` process for the leadership election protocol in PROMELA.

4.3. The Dining Cryptographers

The second case study is a good example of an important emerging class of applications for epistemic logic: proving knowledge properties of protocols.

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

mous payment, but they wonder if NSA is paying. They resolve their uncertainty fairly by carrying out the following protocol:

> Each cryptographer flips an unbiased coin behind his menu, between him and the cryptographer on his right, so that only the two of them can see the outcome. Each cryptographer then states aloud whether the two coins he can see — the one he flipped and the one his left-hand neighbor flipped — fell on the same side or on different sides. If one of the cryptographers is the payer, he states the opposite of what he sees. An odd number of differences uttered at the table indicates that a cryptographer is paying; an even number indicates that NSA is paying (assuming that the dinner was paid for only once). Yet if a cryptographer is paying, neither of the other two learns anything from the utterances about which cryptographer it is. [2]

An implementation of the Dining cryptographer's problem is given in Figure 8. It is perhaps not obvious from inspecting the code that the label `end` is always reached. If for instance the number of messages sent and expected does not match, a deadlock could occur. Before we move on to knowledge properties, we have verified this property: `<>end`. This gives us some confidence in the correctness of the implementation. We can now verify whether agents learn whether one of the cryptographers pays. We have to verify the next two formulas. We only verify for agent 1 since all agents play essentially the same role in this protocol.

$$\Box(\mathit{end} \rightarrow (\text{paysNsa} \rightarrow C\text{paysNSA}))$$
$$\Box(\mathit{end} \rightarrow (\neg\text{paysNsa} \rightarrow C\neg\text{paysNSA}))$$

In order to prove the first formula, we have to find a local proposition p. The description of the solution suggests that we use the parity of all announcements. This corresponds to the proposition `end && !sum` in the implementation. The following two formulas use the common proposition `end && !sum` and are together a proof of the first formula. Both formula's have been verified using SPIN.

```
[](end -> (paysNSA-> !sum ))
[](end -> ((end && !sum) -> paysNSA ))
```

The verification of the second property is very similar. The local proposition `end && sum` is used in the next LTL formulas.

```
[](end -> (!paysNSA-> sum ))
[](end -> ((end && sum) -> !paysNSA ))
```

It is harder to verify that agents do not learn who pays, in case one of the cryptographers pays. We must verify the following property.

$$\Box(\text{end} \rightarrow \neg K_0\text{pays1})$$

The problem here is that we must show that whatever the local state of agent 0 is, agent 0 is never certain that agent 1 is paying. We cannot supply a useful local proposition, but instead we must show that no local proposition exists. This cannot be done by verifying a single LTL property. Instead we must verify a property for every local state of the agent. Each agent has access to the following data: all announcements, whether it pays itself, and two coin flips. We have introduced eight propositions `an0,...,an7` for every possible combination of announcements, and `flip0,...,flip3` for every combination of coin flips for the two coins known to agent 0. We can proceed to falsify the next statement.

```
[] ((end && an0 && flip0) -> pays1)
```

The model checker will report that this formula does not hold. Therefore there must be a run for this local state of agent 0 in which agent 1 did not pay, which proves that in the given local state, the agent does not know that agent 1 paid. In order to draw the same conclusion for any local state of the agent, we must model check a formula for every local state of the agent. For this example we must verify 32 formulas. The remaining 31 formulas are these.

```
[] ((end && an0 && flip1) -> pays1)
[] ((end && an0 && flip2) -> pays1)
[] ((end && an0 && flip3) -> pays1)
[] ((end && an1 && flip0) -> pays1)
[] ((end && an1 && flip1) -> pays1)
[] ((end && an1 && flip2) -> pays1)
[] ((end && an1 && flip3) -> pays1)
...
[] ((end && an7 && flip0) -> pays1)
[] ((end && an7 && flip1) -> pays1)
[] ((end && an7 && flip2) -> pays1)
[] ((end && an7 && flip3) -> pays1)
```

This process can be automized by means of a scripting language. The script calculates, given the local variables, all possible combinations of

```
input system $\mathcal{I}$
input formula $f_1 = \Box(A \rightarrow \neg K_X B)$
let $(V_0, \ldots, V_n)$ be all local variables of $X$
let $L = \{(v_0, v_1, \ldots, v_n) | v_i \in dom(V_i)\}$
for $(v_0, v_1, \ldots, v_n) \in L$
    $f = \wedge_i V_i = v_i$
    if $mc_{LTL}(\mathcal{I}, \Box((A \wedge f) \rightarrow B))$
    if not $mc_{LTL}(\mathcal{I}, \Box\neg(A \wedge f))$
        return false
return true
```

Figure 7. A procedure for proving ignorance

variables and values. If one of these formulas cannot be verified, one possible reason could be that the local state is unreachable. Suppose for instance that only `[] ((end && an7 && flip1) -> pays1)` cannot be falsified. In that case one should check `[] !(end && an7 && flip1)`. If this second formula is true, then the given local state is unreachable and this formula should be ignored. If verification of the second formula fails, we have found a counterexample for the original knowledge formula. An algorithm in pseudocode for this procedure is given in Figure 7.

Notice that the Dining cryptographers is an example of an "adversarial" system, in the sense that the goal is to prove the *absence* of knowledge (as opposed to the presence of knowledge, as was the case in the bit transmission problem). The need to use the script to check all combinations of local propositions here suggests that the local propositions approach is less well-suited to such systems.

5. Concluding Remarks

Model checking as an approach to automatic verification has focussed almost exclusively on specifications expressed in temporal logic. Little attention has been given to temporal epistemic logics, although such logics have proven to be very useful and widely advocated in the specification of protocols [8]. In this paper, we have considered the model checking problem for such logics. We have introduced Halpern and Vardi's well-known temporal epistemic logic CKL_n, and demonstrated how, using ideas from the *interpreted systems* paradigm and the *logic of local propositions*, it is possible to reduce CKL_n model checking to LTL model checking. We then gave three case studies — the bit transmission problem, the leadership election protocol and the dining cryptographers problem — which were implemented in PROMELA, and showed

```
#define N 3 /*number of cryptographers*/
bool ancmt[N];
bool random[N];
bool paysG[N];
bool globalSum;
chan inG[N] = [0] of  bool;
chan outG[N] = [0] of  bool;

proctype cryptographer(chan in,out){
   bool pays,leftrandom,rightrandom;
   in?pays;
   in?leftrandom;
   in?rightrandom;
   if
      ::pays -> out!(leftrandom==rightrandom)
      ::!pays -> out!(!(leftrandom==rightrandom))
   fi
}

active proctype main() {
   byte i=0;
   do
   :: i < N ->
      if
         :: random[i]=false;
         :: random[i]=true;
      fi;
      paysG[i]=false;
      i++
   :: i == N ->
      break;
   od;
   if
      :: paysG[0]=true
      :: paysG[1]=true
      :: paysG[2]=true
      :: skip
   fi;
   i=0;
   do
   :: i < N ->
      run cryptographer(inG[i],outG[i]);
      inG[i]!paysG[i];
      inG[i]!random[i];
      inG[i]!random[(i+1)%N];
      outG[i]?ancmt[i];
      i++
   :: i == N ->
      break;
   od;
   globalSum=ancmt[0]^ ancmt[1]^ ancmt[2];
   END: skip
}
```

Figure 8. The Dining cryptographers in PROMELA.

```
#define an0 (ancmt[0]==0&&ancmt[1]==0&&ancmt[2]==0)
#define an1 (ancmt[0]==0&&ancmt[1]==0&&ancmt[2]==1)
#define an2 (ancmt[0]==0&&ancmt[1]==1&&ancmt[2]==0)
#define an3 (ancmt[0]==0&&ancmt[1]==1&&ancmt[2]==1)
#define an4 (ancmt[0]==1&&ancmt[1]==0&&ancmt[2]==0)
#define an5 (ancmt[0]==1&&ancmt[1]==0&&ancmt[2]==1)
#define an6 (ancmt[0]==1&&ancmt[1]==1&&ancmt[2]==0)
#define an7 (ancmt[0]==1&&ancmt[1]==1&&ancmt[2]==1)
#define flip0 (random[0]==0&&random[1]==0)
#define flip1 (random[0]==1&&random[1]==0)
#define flip2 (random[0]==0&&random[1]==1)
#define flip3 (random[0]==1&&random[1]==1)
#define pays0 (paysG[0]==1)
#define pays1 (paysG[1]==1)
#define pays2 (paysG[2]==1)
#define paysNSA (!pays0&&!pays1&&!pays2)
#define end (main[0]@END)
#define sum globalSum
```

Figure 9. Propositions used in the Dining Cryptographers.

how desirable temporal epistemic properties of this system could be proved using SPIN. Engelhardt et al suggested that local propositions might be used in a similar manner for implementing knowledge-based programs [7].

The main limitation of our approach is that, while it makes extensive use of model checking, the verification process still requires input from a human verifier (to obtain the local propositions used when reducing the CKL_n specification to LTL). A "direct" implementation of CKL_n model checking — perhaps as an extension or refinement to SPIN — would thus be desirable. However, there are some obstacles to building such a system: unlike pure LTL formulae, CKL_n formulae can express properties of *multiple runs*. For the moment, therefore, we believe our approach has something to offer which is theoretically well-founded and practically valuable to the verifier who desires to check epistemic temporal properties of systems. And, given the extent of interest in epistemic logic and its applications in the specification of communicating systems [8], we believe that our approach is potentially valuable indeed.

The first step for future work is to further investigate the theoretical foundations of our work, and in particular to discover the extent to which the approach is applicable. We plan to extend our ideas here to other knowledge programs, and also like to determine classes of programs for which the local propositions that are needed can be easily determined. Using model checking to verify that programs implement knowledge-based programs is another obvious application. Also, the role and use of these local propositions, especially in contexts different from distributed systems, is, to the best of our knowledge, still not explored.

Acknowledgements

This work was supported by the UK EPSRC (project GR/R27518 — "Verifiable Languages and Protocols for Multiagent Systems"), and by the European Commission's IST programme (project IST-1999-10948 — "Sustainable Lifecycles for Information Ecosystems"). We would like to thank Alessio Lomuscio and Ron van der Meyden, who both gave detailed helpful comments on a preliminary draft of this paper. A shorter, preliminary version of this paper was published at the 9th International SPIN Workshop on Model Checking of Software, April 11-13, 2002, Grenoble, France.

References

1. M. Benerecetti, F. Giunchiglia, and L. Serafini. A model checking algorithm for multiagent systems. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V (LNAI Volume 1555)*. Springer-Verlag: Berlin, Germany, 1999.
2. D Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1:65–75, 1988.
3. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press: Cambridge, MA, 2000.
4. C. Dixon, M. Fisher, and M. Wooldridge. Resolution for temporal logics of knowledge. *Journal of Logic and Computation*, 8(3):345–372, 1998.
5. D Dolev, Maria Klawe, and M Rodeh. An o(n log n) unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, 3:245–260, 1982.
6. E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science Volume B: Formal Models and Semantics*, pages 996–1072. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1990.
7. K. Engelhardt, R. van der Meyden, and Y. Moses. Knowledge and the logic of local propositions. In *Proceedings of the 1998 Conference on Theoretical Aspects of Reasoning about Knowledge (TARK98)*, pages 29–41, Evanston, IL, July 1998.
8. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. The MIT Press: Cambridge, MA, 1995.
9. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. Knowledge-based programs. *Distributed Computing*, 10(4):199–225, 1997.
10. R. Goldblatt. *Logics of Time and Computation (CSLI Lecture Notes Number 7)*. Center for the Study of Language and Information, Ventura Hall, Stanford, CA 94305, 1987. (Distributed by Chicago University Press).

11. J. Y. Halpern and M. Y. Vardi. The complexity of reasoning about knowledge and time. I. Lower bounds. *Journal of Computer and System Sciences*, 38:195–237, 1989.
12. J. Y. Halpern and M. Y. Vardi. Model checking versus theorem proving: A manifesto. In V. Lifschitz, editor, *AI and Mathematical Theory of Computation — Papers in Honor of John McCarthy*, pages 151–176. The Academic Press: London, England, 1991.
13. J. Y. Halpern and L. D. Zuck. A little knowledge goes a long way: knowledge-based derivations and correctness proofs for a family of protocols. *Journal of the ACM*, 39(3):449–478, 1992.
14. J. Hintikka. *Knowledge and Belief.* Cornell University Press: Ithaca, NY, 1962.
15. G. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall International: Hemel Hempstead, England, 1991.
16. G. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
17. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag: Berlin, Germany, 1992.
18. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems — Safety.* Springer-Verlag: Berlin, Germany, 1995.
19. K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers: Boston, MA, 1993.
20. J.-J. Ch. Meyer and W. van der Hoek. *Epistemic Logic for AI and Computer Science.* Cambridge University Press: Cambridge, England, 1995.
21. A. S. Rao and M. P. Georgeff. A model-theoretic approach to the verification of situated reasoning systems. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 318–324, Chambéry, France, 1993.
22. R. van der Meyden and N. Shilov. Model checking knowledge and time in systems with perfect recall. In *Proceedings of the Conference on Foundations of Software Technology and Theoretical Computer Science (LNCS Volume 1738)*, pages 432–445. Springer-Verlag: Berlin, Germany, 1999.
23. Ron van der Meyden. Finite state implementations of knowledge-based programs. In *Proceedings of the conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *LNCS*, pages 262–273, 1996.
24. Ron van der Meyden. Knowledge based programs: On the complexity of perfect recall in finite enviroments. In Y. Shoham, editor, *Proceedings of the Sixth conference on Theoretical Aspects of Rationality and Knowledge*, pages 31–50, Renesse, Holland, 1996. De Zeeuwse Stromen.
25. M. Y. Vardi. Implementing knowledge-based programs. In *Proceedings of the Sixth Conference on Theoretical Aspects of Rationality and Knowledge (TARK 1996)*, pages 15–30, De Zeeuwse Stromen, The Netherlands, 1996.
26. M. Y. Vardi. Branching vs. linear time: Final showdown. In T. Margaria and W. Yi, editors, *Proceedings of the 2001 Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2001 (LNCS Volume 2031)*, pages 1–22. Springer-Verlag: Berlin, Germany, April 2001.
27. M. Wooldridge, C. Dixon, and M. Fisher. A tableau-based proof method for temporal logics of knowledge and belief. *Journal of Applied Non-Classical Logics*, 8(3):225–258, 1998.