Model Checking Knowledge in Multi-Agent Systems

Mario R. F. Benevides[1], Carla Amor Divino M. Delgado[1], Carlos G. Lopez Pombo[2], Luis Roberto M. Lopes[1], and Ricardo F. Ribeiro[1]

[1] Universidade Federal do Rio de Janeiro, Brasil,
Email: {mario, delgado, luislops, ribeiro}@cos.ufrj.br
[2] Universidad de Buenos Aires, Argentina. Email: clpombo@dc.uba.ar. *

Abstract. The basic knowledge-based model for concurrent multi-agent systems (*MAS*) [Lam78] captures local and global views of a distributed computation. The usual knowledge modal language associated [FHM$^+$95] is capable of expressing things from one agent's local point of view, and about the whole system as well. This approach is very suggestive to talk about rational interactions at the knowledge level: each agent has it's part of local knowledge and uncertainty and, as interaction takes place, this knowledge can be changed by gaining new information and refining uncertainties, just like it happens in many real life situations. The growing relevance of social software field [MP02] drives attentions to formal modelling and verifying multi-agent rational interaction by means of computational methods. Since Model Checking [EC81] is a very successful technique for verifying finite state concurrent systems, we consider it an appropriate tool to be explored within this field. In this paper we propose a formal language and the corresponding Model Checking process to model and verify multi-agent systems at the knowledge level. The language we use is an extension of *CTL*, which we call *KCTL*, with an operator ($\mathcal{K}_k$) that provides the capability of observing the occurrence of an event from the point of view of one agent k. Algorithms for checking it's semantics in a branching time model (like the one for *CTL*) are also presented.

1 Introduction

The growing relevance of social software field [MP02] drives attentions to formal modelling and verifying multi-agent rational interaction by means of computational methods.

Model Checking is a very powerful and mature technique for verifying finite state concurrent systems [EC81], and many efforts are being made to contemplate social software aspects with model checking facilities.

The use of temporal epistemic models to formalize and check compositional systems in linear time was presented in [EJT99].

* The authors want to thank CAPES, CNPQ and FAPERJ.

In [BGL98], agents are modelled as concurrent reactive non-terminating finite state processes able to have BDI atitudes, i. e., beliefs, desires and intentions, reflecting in an specification with two ortogonal aspects: temporal and "mental attitudes". The mental attitudes are represented for each possible state of mind of each agent, in an hierarchical structure where all levels of knowledge are explicitly represented, what requires special computational treatment.

These initiatives, as many others, drive our attention to plausibility of automatic verification when epistemic aspects need to be considered.

This paper presents a formal language and the corresponding Model Checking process to model and verify Multi-Agent Systems (*MAS*) at the knowledge level.

We extend branching time *CTL* logic with knowledge operators $\mathcal{K}_k$ for each agent k. The resulting language, which we call *KCTL*, provides the capability of observing the occurrence of an event from the point of view of one agent k. A semantics based on equivalence "possibility relations" is given for *KCTL*.

Model Checking algorithms to verify *KCTL* formulae over state transition systems (interactive MASs representations) are defined and described in detail.

This paper is organized as follows. Section 2 reviews the branching-time temporal logic *CTL* [EC81] and the corresponding model checking process [CGP$^+$99]. Section 3 states the requirements to talk about knowledge in *MASs*. We present our approach to handle knowledge in a Model Checking process in section 4, and in section 5 we state the algorithms of interest. Section 6 presents an example model for the alternating bit protocol in this logic. Our final remarks go on section 7 .

2 *CTL* branching-time temporal logic and Model Checking

In [BAMP81], Ben-Ari et al. presented for the first time the logic of branching-time *CTL* with the aim of dealing with the set of every possible execution tree generated by a given program. This logic was specially designed to take care of the consequences of the non-determinism just like the one generated by programs that interact asynchronously.

It was in [EC81] where Emerson and Clarke gave the final shape to *CTL* providing a decision procedure, and that is the reason why the way we present the logic is close to that of the previously mentioned article.

2.1 *CTL* Language

Definition 21 *(Syntax of* CTL *formulae)*

Let $\mathcal{P}$ be a set of propositions. The language of CTL *formulae is defined as follows:*

$ForCTL(\mathcal{P})$ is the smallest set For of formulae such that:

- *$p \in For$ iff $p \in \mathcal{P}$,*
- *$\{\neg\phi_1, \phi_1 \vee \phi_2, \exists\mathsf{X}\phi_1, \exists\mathsf{G}\phi, \exists[\phi_1\mathsf{U}\phi_2]\} \subseteq For$ iff $\{\phi_1, \phi_2\} \subseteq For$.*

The rest of the propositional operators are defined in terms of negation ("$\neg$") and disjunction ("$\vee$") in the usual way. Let $\phi, \psi \in ForCTL(\mathcal{P})$, then the rest of the temporal operators are defined as follows: $\exists\mathsf{F}\phi = \exists[\mathtt{true}\mathsf{U}\phi]$, $\forall\mathsf{X}\phi = \neg\exists\mathsf{X}\neg\phi$, $\forall\mathsf{G}\phi = \neg\exists\mathsf{F}\neg\phi$, $\forall[\phi\mathsf{U}\psi] = \neg\exists[\neg\psi\mathsf{U}(\neg\phi \wedge \neg\psi)] \wedge \neg\exists\mathsf{G}\neg\psi$ and $\forall\mathsf{F}\phi = \neg\exists\mathsf{G}\neg\phi$.

The intended meaning of *CTL* formulae is given as usual in terms of Kripke models.

Definition 22 *(Kripke model)*
Let $\mathcal{P}$ be a set of propositions. Then $\mathfrak{M} = \langle S, S_0, R, \mathcal{P}, \mathcal{L}\rangle$ is said to be a Kripke model if it satisfies the following properties:

- *S is a non-empty set of states,*
- *$S_0 \subseteq S$ and $S_0 \neq \emptyset$,*
- *$R \subseteq S \times S$ and $Dom(R) = S$* [3]*,*
- *$\mathcal{L} : S \to 2^{\mathcal{P}}$.*

Definition 23 *(Set of runs of a Kripke model)*
Let $\mathfrak{M} = \langle S, S_0, R, \mathcal{P}, \mathcal{L}\rangle$ be a Kripke model. Then, the runs of $\mathfrak{M}$, denoted by $\mathcal{R}^{\infty}_{\mathfrak{M}}$ are characterized as follows:

$$\mathcal{R}^{\infty}_{\mathfrak{M}} = \{\sigma \in seq^{\infty}(S) \mid \pi_1(\sigma) \in S_0 \ \wedge (\forall i \in I\!N : \pi_i(\sigma)\ R\ \pi_{i+1}(\sigma))\}.$$ [4]

We will use $\mathcal{R}_{\mathfrak{M}}$ to denote the infinite set of finite prefixes of the sequences of $\mathcal{R}^{\infty}_{\mathfrak{M}}$.

Notation 21 *(Prefix of a run)*
Let $\mathfrak{M} = \langle S, S_0, R, \mathcal{P}, \mathcal{L}\rangle$ be a Kripke model, $T \subseteq \mathcal{R}^{\infty}_{\mathfrak{M}}$, $\sigma \in \mathcal{R}^{\infty}_{\mathfrak{M}}$ and $i \in I\!N$, ${}_i\sigma$ will denote the prefix of length i of σ, defined as ${}_i\sigma = \sigma' \in seq(S) \mid Length(\sigma') = i \wedge (\forall j \in I\!N : 1 \leq j \leq i \implies \pi_j(\sigma') = \pi_j(\sigma))$.

Definition 24 *(Satisfiability relation for* CTL *formulae)*
Let $\mathfrak{M} = \langle S, S_0, R, \mathcal{P}, \mathcal{L}\rangle$ be a Kripke model, the satisfiability relation is defined as follows:

[3] *Dom* is the set-theoretical domain function, and this restriction states that every state in S has at least one successor through the accessibility relation R.

[4] We use $seq^{\infty}(S)$ to denote the set of infinite sequences of elements taken from the set S, and π_i as the projection of the i^{th} element of a sequence.

$$\begin{array}{ll}
\mathfrak{M}, \langle \sigma, i \rangle \models p & \textit{iff}\, p \in \mathcal{L}(\pi_i(\sigma)) \\
\mathfrak{M}, \langle \sigma, i \rangle \models \neg\phi & \textit{iff}\, \mathfrak{M}, \langle \sigma, i \rangle \not\models \phi \\
\mathfrak{M}, \langle \sigma, i \rangle \models \phi \vee \psi & \textit{iff}\, \mathfrak{M}, \langle \sigma, i \rangle \models \phi \;\textit{or}\; \mathfrak{M}, \langle \sigma, i \rangle \models \psi \\
\mathfrak{M}, \langle \sigma, i \rangle \models \exists \mathsf{X}\phi & \textit{iff}\, \exists \sigma' \in \mathcal{R}^{\infty}_{\mathfrak{M}} : {}_i\sigma' = {}_i\sigma \wedge \mathfrak{M}, \langle \sigma', i+1 \rangle \models \phi \\
\mathfrak{M}, \langle \sigma, i \rangle \models \exists \mathsf{G}\phi & \textit{iff}\, \exists \sigma' \in \mathcal{R}^{\infty}_{\mathfrak{M}} : {}_i\sigma' = {}_i\sigma \wedge \forall j : i \leq j \implies \mathfrak{M}, \langle \sigma', j \rangle \models \phi \\
\mathfrak{M}, \langle \sigma, i \rangle \models \exists [\phi \mathsf{U} \psi] & \textit{iff}\, \exists \sigma' \in \mathcal{R}^{\infty}_{\mathfrak{M}} : {}_i\sigma' = {}_i\sigma \wedge \\
& \qquad (\exists j \in I\!\!N : i < j \wedge \mathfrak{M}, \langle \sigma', j \rangle \models \psi \wedge \\
& \qquad\quad (\forall k \in I\!\!N : i \leq k < j \implies \\
& \qquad\qquad \mathfrak{M}, \langle \sigma', k \rangle \models \phi))
\end{array}$$

2.2 *CTL* Model Checking

Given a Kripke Model $\mathfrak{M} = \langle S, S_0, R, \mathcal{P}, \mathcal{L} \rangle$ that represents a finite state concurrent system with it's properties of interest and a *CTL* formula f expressing some desired specification, the *model checking problem* is to find the set of states in S that satisfy f [CGP+99]: $\{s \in S | \mathfrak{M}, s \models f\}$. In other words, the state-transition system underlying a Kripke structure is checked to see whether it is a model of the specification written in *CTL*.

Normally some states are designated initial states, and we say that the system satisfies the specification provided that all of the initial states are in the set. Formally, $\mathfrak{M}, S_0 \models f$ means $\forall_{s_0 \in S_0} \mathfrak{M}, s_0 \models f$.

A *CTL* formula f can be identified with a set of states in a given model $\mathfrak{M}$, namely those states $Q_{\mathfrak{M}} \subseteq S$ that satisfy the formula: $Q_{\mathfrak{M}}(f) = \{s | \mathfrak{M}, s \models f\}$. Model checking a *CTL* formula therefore entails the manipulation of sets of states: $S_0 \subseteq Q_{\mathfrak{M}}(f)$. Algorithms for doing so are given in [CGP+99].

3 Knowledge in Concurrent Systems

A Distributed System is a concurrent system composed of a set of agents, each running it's corresponding program, that communicate by sending and receiving messages along previously defined communication channels.

The interesting fact about this model is that it permits to talk in separate about local and global computation. An agent is not concerned about the way other agents carries on their local computations. All interaction happens by sending and receiving messages, just like many computational systems operate.

This approach is very suggestive to talk about rational agents at the knowledge level. Each agent has it's part of local knowledge and uncertainty and, as interaction takes place, this knowledge can be changed by gaining new information and refining uncertainties.

3.1 Multi-Agent Kripke Models for Knowledge

The usual way to deal with knowledge and uncertainty in Kripke Models is by means of "indistinguishable states", presented in [FHM+95].

To accomplish the notion that each agent has it's own private information set, we label propositions with its corresponding agent identification, assigning propositions to one agent information set.

We also enhance the Kripke Model with possibility relations $\sim_k$. For each agent k, we define an equivalence relation $\sim_k$ over S. Two states s and t of S are related by $\sim_k$ if and only if agent k can not tell them apart. This means that the information agent k has in both s and t states is the same.

Definition 31 *(K-extended Kripke Model)*
Let $\{\mathcal{P}_k\}_{1\leq k\leq j}$ be a set of disjoint sets of propositions. Then $\mathfrak{M} = \langle S, S_0, R, \{\sim_k\}_{1\leq k\leq j}, \cup_{k=1}^{j}\mathcal{P}_k, \mathcal{L}\rangle$ is said to be a K-extended Kripke Model if it satisfies the following properties:

- *S, S_0 and R are as defined for a Kripke Model;*
- *$\mathcal{L} : S \rightarrow 2^{\cup_{k=1}^{j}\mathcal{P}_k}$.*
- *$\{\sim_k\}_{1\leq k\leq j}$ is a set of binary equivalence relations on S.*

3.2 A Language for Knowledge

To reason about knowledge in a *MAS* it is necessary to assume that agents are able to reason about the world and also about other agents' knowledge. A complete axiomatic characterization of the notion of knowledge and common knowledge, and an accurate analysis of the role played by time in *MAS*' evolution was given in [Leh84].

We adopt a propositional multi-modal language, with a knowledge modality $\mathcal{K}_k$ for each agent k. Knowledge modalities permits to talk about information from each agent's point of view. Intuitively, formula $\mathcal{K}_k\varphi$ indicates that "agent k knows φ".

4 Model Checking Knowledge in Multi-Agent Systems

We now present a formal language and the corresponding Model Check process to verify multi-agent systems at the knowledge level. The language proposed is an extension of *CTL*, which we call *KCTL*. After defining *KCTL*, we present Algorithms for checking it's semantics in a branching time model.

4.1 Multi-agent architecture

Computation in a Multi-agent System (*MAS*) is dictated by the local programs each agent runs. Global states of the system are compositions of local states of each agent. [Lam78] presents a classic event-based model for Asynchronous *MAS*. The model is basically composed by:

- a network with m agents, connected by communication channels;
- a set R of asynchronous runs (distributed computations or parallel runs of all agents involved);

– a set E of events, including internal actions and communication events;
– a set C of global states of the system; and
– a protocol P (or distributed algorithm) corresponding to a set of local programs that specifies the behavior of each agent.

For Model Check purposes, we represent the program for each agent in the MAS as an automaton. Each automaton represents the local states (nodes) and events from E (edges) for an agent. P is set of all automata.

Definition 41 *Automaton*
Let Σ be an alphabet, an automaton is a structure $\mathfrak{A} = \langle S, \Sigma^*, E, \{p_i\}_{i \in \mathcal{I}}, \mathcal{L}\rangle$ such that S is a set of states, Σ^* is the language generated by the alphabet Σ, $E \subseteq S \times \Sigma^* \times S$ a set of edges, $\{p_i\}_{i \in \mathcal{I}}$ a set of propositions, and $\mathcal{L} : S \rightarrow 2^{\{p_i\}_{i \in \mathcal{I}}}$ the function that assigns to each state a subset of the propositions.

When we put all the agents running together, we get to a global automaton that is the parallel asynchronous composition of the automata for each agent.

The idea is that the states of the composed automaton are n-tuples where its components corresponds to a local state of each of the agents, and the edges corresponds to all edges of local automata.

Definition 42 *Parallel Asynchronous Composition of Automata*
Let $\mathfrak{A}_i = \langle S_i, \Sigma^*, E_i, \{p_k\}_{k \in \mathcal{I}_i}, \mathcal{L}_i\rangle$ be an automaton for all $i \in [1, \ldots, n]$. Then the parallel asynchronous composition of the automatons, denoted as $||_{1 \le i \le n}\mathfrak{A}_i$, is the structure $\langle S, \Sigma^*, E, \mathcal{P}, \mathcal{L}\rangle$ defined as follows:

- $S = \Pi_{1 \le i \le n} S_i$,
- $((s_1, \ldots, s_n), l, (s_1{}', \ldots, s_n{}')) \in E$ *iff* $\bigvee_{1 \le i \le n} l \in \Pi_2(E_i)$ [5],
- $\mathcal{P} = \bigcup_{1 \le i \le n} \mathcal{P}_i$
- $\mathcal{L}((s_1, \ldots, s_n)) = \bigcup_{1 \le i \le n} \mathcal{L}_i(s_i)$

States in the composed automaton corresponds to the MAS's global states, and are the elements in C. The set of runs can be easily obtained from the global automaton: each run in R is a path on the global automaton's computation tree.

When making the Parallel Asynchronous Composition of a set of Automata where each automaton dictates the behavior of an agent, it is possible and reasonable to get a composed global automaton $G = \langle S, \Sigma^*, E, \mathcal{P}, \mathcal{L}\rangle$ where many states from S corresponds to the same local state component for a particular agent. When two such states s and t with the same local state component for agent i are connected by an edge $(s, l, t) \in E$, $l \in \Pi_2(E_j)$, $j \neq i$, this means that agent i is incapable of noticing that a global state change has occurred when the global state passes from s to t. This edges denote local actions made by other agents different from i, and for this reason are imperceptible for agent i.

An equivalence relation for each agent can be defined over the states with this "indistinguishable" property.

[5] Given a label $l \in \Sigma^*$, a set of states S and a set of edges $E \in S \times \Sigma^* \times S$, $l \in \Pi_2(E)$ if and only if there exists $s, s' \in S$ such that $(s, l, s') \in E$.

Definition 43 *Possibility Relation $\sim_i$ for agent i*
Let $||_{1\leq i\leq n}\mathfrak{A}_i = \langle S, \Sigma^*, E, \mathcal{P}, \mathcal{L}\rangle$ be the parallel asynchronous composition of automata $\mathfrak{A}_i = \langle S_i, \Sigma^*, E_i, \{p_k\}_{k\in\mathcal{I}_i}, \mathcal{L}_i\rangle$ for all $i \in [1,\ldots,n]$. The possibility relation $\sim_i \in S\times S$ for each agent i is the smaller equivalence relation containing all pairs (s,t) such that $s,t \in S$ and there is an edge $(s,l,t) \in E$, $l \in \Pi_2(E_j)$ and $j \neq i$.

Intuitively, two states are related by $\sim_i$ if agent i, being in one of them, considers it possible that the other one is the current state. In other words, agent i can't tell the current state from the other possible states.

4.2 Extended *CTL* Language: *KCTL*

We describe here a logic to reason about knowledge in state transition systems as the ones presented in the previous section. The language we will use is an extension of *CTL* with an operator ($\mathcal{K}_k$) that provides the capability of observing the occurrence of an event from the point of view of one of the automaton involved in the system. From now on, this language will be referred as *KCTL*.

As in *CTL*, *KCTL* formulae reason about (knowledge) properties of computation trees. The tree is formed just by unwinding the global automaton (or Kripke Structure [CGP$^+$99]) that represents the *MAS* from it's initial state. The computational tree illustrates all possible runs in R.

Definition 44 *(Syntax of* KCTL *formulae)*
Let $j \in I\!N$ and $\{\mathcal{P}_k\}_{1\leq k\leq j}$ be the set of disjoint sets of propositions. The language of KCTL *formulae is defined as follows:*
$ForCTL(j, \{\mathcal{P}_k\}_{1\leq k\leq j})$ is the smallest set For of formulae such that:

- *$p \in For$ iff there exists k such that $1 \leq k \leq j$ and $p \in \mathcal{P}_k$,*
- *$\mathcal{K}_i(\phi) \in For$ iff $1 \leq k \leq j$ and $\phi \in For$,*
- *any other compound formula is formed in the same way as in* CTL *(see definition 21).*

Just like in section 2, the semantics of the set of *KCTL* formulae is given in terms of Kripke models, but this time, considering as the set of propositions the union of all the components of the set of sets provided as argument for the construction of the set of *KCTL* formulae.

The intuition behind the next definition is that of a run where, given a binary relation over the set of states, successions of states which are related via that relation are compressed keeping only one of them, for instance, the first one. This definition somehow establishes a notion of indistinguishability of states in a run.

Definition 45 *($\sim$-quotient of a run)*
Let S be a non-empty set, and $\sim \subseteq S\times S$ a binary relation on S, we define the

$\sim$-quotient of a sequence, of elements of S, σ as $\sigma|^{\sim} = \sigma' \in seq^{\infty}(S) \mid \pi_1(\sigma) = \pi_1(\sigma') \wedge (\forall i \in I\!N : (\exists j, k \in I\!N : j < k \wedge \pi_i(\sigma') = \pi_j(\sigma) \wedge \pi_{i+1}(\sigma') = \pi_k(\sigma) \wedge (\forall r \in I\!N : j \leq r < k \implies \pi_r(\sigma) \sim \pi_j(\sigma))))$.

The satisfiability relation for a *KCTL* formula will be slightly modified to consider the previous definition to give meaning to the operator "K".

Definition 46 *(Satisfiability relation for* KCTL *formulae)*
Let $\{\mathcal{P}_k\}_{1\leq k\leq j}$ be a set of disjoint sets of propositions and $\mathfrak{M} = \langle S, S_0, R, \{\sim_k\}_{1\leq k\leq j}, \cup_{k=1}^{j}\mathcal{P}_k, \mathcal{L}\rangle$ be a K-extended Kripke Model, the satisfiability relation is defined in exactly the same way that it was done for CTL *formulae, except for the new operator that is interpreted as follows:*

$$\mathfrak{M}, \langle\sigma, i\rangle \models \mathcal{K}_k(\phi) \textit{ iff } \forall\sigma' \in \mathcal{R}^{\infty}_{\mathfrak{M}} : \forall j \in I\!N : {}_j\sigma'|^{\sim_k} = {}_i\sigma|^{\sim_k} \implies \mathfrak{M}, \langle\sigma', j\rangle \models \phi$$

4.3 A Model Checking Process for Knowledge

We now present algorithms for the model checking problem described in section 2.2. We use an explicit representation of K-extended Kripke Models $\mathfrak{M} = \langle S, S_0, R, \{\sim_k\}_{1\leq k\leq j}, \cup_{k=1}^{j}\mathcal{P}_k, \mathcal{L}\rangle$ as automata where each state is labeled with the propositions associated by $\mathcal{L}$.

The process is the usual model checking process presented in [CGP+99]: "To check whether a *KCTL* formula f is satisfied in some state(s) of S, the process consists on labeling each state $s \in S$ with the set *label(s)* of subformulae of f which are true in s. Initially, *label(s)* is $\mathcal{L}(s)$. Then the algorithm goes through a series of iterations, adding subformulae do *label(s)*. During i^{th} iteration, subformulae with $i-1$ nested *KCTL* operators are processed and added to the labels of states where it is satisfied. At the end, $\mathfrak{M}, s \models f$ if and only if $f \in$ *label(s)*".

For the intermediate stages of the algorithm, it is necessary to handle seven cases: atomic formulae, $\neg, \vee, \exists\mathsf{X}, \exists\mathsf{G}, \exists\mathsf{U}$ and $\mathcal{K}$. The six first cases are the same for *CTL*:

For formulae of the form:

- Atomic formulae, already handled;
- $\neg f_1$, label those states that are not labeled by f_1;
- $f_1 \vee f_2$, label those states that are labeled by either f_1, f_2 or both;
- $\exists\mathsf{X} f_1$, label those states that have a sucessor labeled by f_1;
- $\exists\mathsf{G} f_1$, first construct a restricted Kripke Model $\mathfrak{M}'$[6], then partition the graph (S', R') into strongly connected components, next find those states that belong to nontrivial components, and then work backwards using the converse of R' and find all of those states that can be reached by a path in which each state is labeled with f_1, finally label these states with $\exists\mathsf{G} f_1$;

[6] $\mathfrak{M}' = \langle S', S'_0, R', \{\sim_k\}'_{1\leq k\leq j}, \cup_{k=1}^{j}\mathcal{P}_k, \mathcal{L}'\rangle$ is obtained from $\mathfrak{M}$ by deleting from S all those states at which f_1 does not hold and restricting R and L accordingly. R' may not be total

- $\exists[f_1 \mathsf{U} f_2]$, first find all states labeled with f_2, then work backwards using the converse relation R and find all states that can be reached by a path in which each state is labeled by f_1, then label all this states with $\exists[f_1 \mathsf{U} f_2]$;

Detailed algorithms for this cases with time complexity of $O(|S| + |R|)$ are given in [CGP$^+$99].

We shall give special treatment to the seventh case, where the knowledge operator must be handled.

Following the intuitive meaning and the semantics defined for $\mathcal{K}_k$ operators in *KCTL*, to model check a formula of the form $\mathcal{K}_k f$ we must look to the indistinguishable states for agent k, related by $\sim_k$ equivalence possibility relation.

First, find the set of all states s labeled with f. Then, for each state found s, recursively check if all states t related to s (the current one) by $\sim_k$ (all t such that $s \sim_k t$) are labeled with f. If this is the case, label all them (the current state s and all states t, $s \sim_k t$) with $\mathcal{K}_k f$.

In spite of being a recursive process, this procedure is linear to the number of pairs in $\sim_k$. This is achieved because $\sim_k$ is an equivalence relation, what makes $s \sim_k t$ the same as $t \sim_k s$. The algorithm chooses a component and look for the possibilities for the second component. Each state is elected as first component just once, because we keep in track the states already elected in set L.

Once we have algorithms to the seven cases listed, to handle an arbitrary *KCTL* formula f just successively apply the state-labeling algorithm to the subformulae of f, starting with the shortest and most deeply nested one, and work outward until f is entirely checked. The complete process takes time $O(|f| \cdot (|S| + |R| + \Sigma_{k=1}^{j} | \sim_k |))$.

5 Algorithms

We now examine in detail the new algorithms to model check formulae of the form $\mathcal{K}_k f$.

Function `CheckK(`G, f, k`)`

```
Data   : $G$ [in parameter] is a (global) automaton (representing a $MAS$);
         $f$ [in parameter] is a formula to be evaluated;
         $k$ [in parameter] is an agent identifier;
         $L$ is a set of states, initially empty;
begin
    $L := \emptyset$ ; $S := \{s | f \in label(s)\}$
    foreach state $s$ in $S$ do
        if $RecursiveCheckK(G, f, k, L, s)$ then
            foreach state $t$ in $L$ do
                $label(t) := label(t) \cup \{\mathcal{K}_k f\}$
            end
    end
end
```

Function `RecursiveCheckK(`G, f, k, L, s`)`

```
Data   : $G$ [in parameter] is a (global) automaton (representing a $MAS$);
         $f$ [in parameter] is a formula to be evaluated;
         $k$ [in parameter] is an agent identifier;
         $L$ [in parameter] is a set of states;
         $s$ [in parameter] a state;
begin
    if $s \notin L$ then
        if $f \in$ label($s$) then
            $L := L \cup \{s\}$
            foreach state $t \in G$, so that $t \sim_k s$ do
                if RecursiveCheckK($G$, $f$, $k$, $L$, $t$) then
                    return True
                    else
                        return False
                    end
            end
    else
        return True
    end
end
```

6 Example

6.1 The Alternating Bit Protocol

As an example of how the logic presented in this text is proposed to work, we present a model for the *alternating bit protocol* [KR00] [Mil89].

The *alternating bit protocol* is a well-known basic communications protocol. It is often used as a test case, either for some algebraic formalism or for some tool for the analysis or verification of concurrent systems.

It consists of three components, or "agents": a **sender** agent, a **receiver** agent and a **communications channel**. The sender agent has a set of **messages** to send to the receiver over the communications channel. However, *the channel isn't reliable*, that is, it can lose any messages going through it. It is assumed that the channel can only transport *one* message at a time and that it is *bi-directional*, that is, it can transport messages from the sender to the receiver and it can also transport messages from the receiver to the sender.

The protocol starts with the sender selecting the first message to send. This message is extended with a **control bit** (initially 0) to form a **frame** and this frame is sent along the communications channel. Right as the sending of the frame starts, the sender also starts a **timer**. When this timer counts down to zero, the sender will assume the frame was lost and will retransmit it.

The communications channel then transmits the frames from the sender to the receiver. There are two situations that can occur. The frame is properly transmitted, or the frame is lost during transmission. If the fame is lost, there is nothing to be done but to wait until the timer of the sender to count down to zero.

If the frame wasn't lost, the receiver reads the frame from the channel. The receiver then checks the control bit in the frame. If this bit matches the internal control bit of the receiver, the message in the frame is *acknowledged*, that is, the receiver sends an **acknowledgement message** with the control bit to the sender over the communications channel. Receiver then flips his internal control bit and waits for another frame. If the bit of the received frame was wrong, the receiver sends a negative acknowledgement (with a flipped control bit), and waits for a retransmission of the frame.

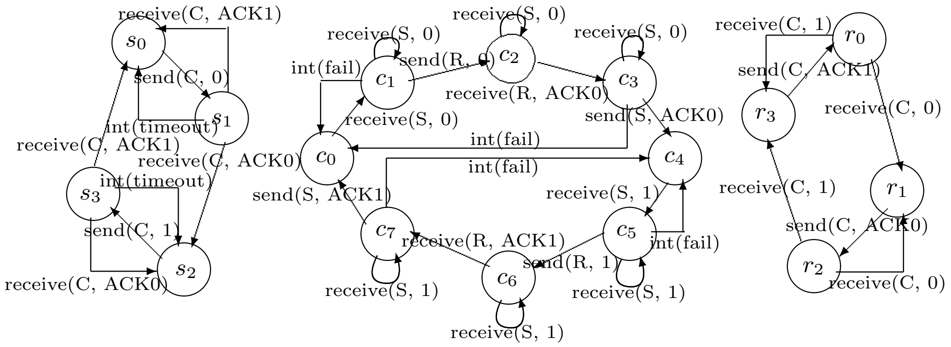
The communications channel then is used to transmit the acknowledgement from the receiver to the sender. As it is able to lose messages, the acknowledgement can be lost. If it happens, again there is nothing to do but wait until the timer runs down to zero. The sender will then retransmit the frame and, assuming the frame reaches the receiver, it will cause the receiver to transmit a new acknowledgement equal to the one which was lost.

The sending of the frame continues until the sender receives the acknowledgement of a successful transmission over the communications channel. Such acknowledgements are the ones with the control bit matching the internal control bit of the sender. If the bit doesn't match, the acknowledgement message is ignored. After a successful transmission, the sender flips the control bit, selects the next message to send and starts all over again.

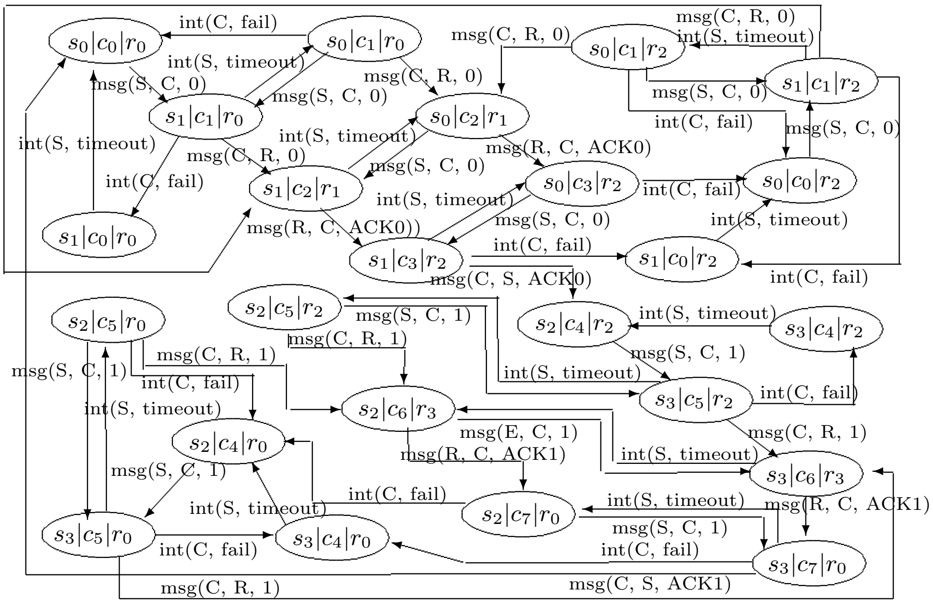
6.2 The agents automata

Diagrams of states

Next, the local automata for sender, channel and receiver, respectively.



Here, follows the global automaton.



The databases for sender and receiver

Now, it will be shown the databases for sender and receiver, with the propositions that holds on each state.

Sender

```
S0: sent_msg_bit_1
    received_Ack1
    p

S1: sending_msg_bit_0
    receiving_Ack0
    p

S2: sent_msg_bit_0
    received_Ack0
    p

S3: sending_msg_bit_1
    receiving_Ack1
    p
```

Receiver

```
R0: receiving_msg_bit_0
    sent_Ack1
    q

R1: received_msg_bit_0
    sending_Ack0
    q

R2: receiving_msg_bit_1
    sent_Ack0
    q

R3: received_msg_bit_1
    sending_Ack1
    q
```

6.3 Queries

High-level queries

Query #1 Is deadlock possible on this protocol?

$G,(s_0|c_0|r_0) \models \exists\mathsf{F}(\neg\exists\mathsf{X}(\mathcal{K}_s p \vee \mathcal{K}_r q))$

Result for $\mathcal{K}_s p$: All global states

Result for $\mathcal{K}_r q$: All global states

Result for $(\mathcal{K}_s p \vee \mathcal{K}_r q)$: {all global states} ∪ {all global states} = {all global states}

Result for $\exists\mathsf{X}(\mathcal{K}_s p \vee \mathcal{K}_r q)$: Set of states which have at least one successor in {all global states} = {all global states}

Result for $\neg\exists\mathsf{X}(\mathcal{K}_s p \vee \mathcal{K}_r q)$: Complementary set of {all global states} = $\emptyset$

Result for $\exists\mathsf{F}(\neg\exists\mathsf{X}(\mathcal{K}_s p \vee \mathcal{K}_r q))$: Is the initial state at the beginning of any path that contains any state in $\emptyset$? No. Then returns **false**.

Query #2 $G,(s_0|c_0|r_0) \models \exists\mathsf{F}(\mathcal{K}_r sending\_ACK0 \wedge \mathcal{K}_s receiving\_ACK0)$

Result for $\mathcal{K}_r sending\_ACK0$: $L_1 = \{(s_0|c_2|r_1),\ (s_1|c_2|r_1)\}$

Result for $\mathcal{K}_s receiving\_ACK0$: $L_2 = \{(s_1|c_2|r_1),\ (s_1|c_3|r_2),\ (s_1|c_0|r_2),\ (s_1|c_0|r_0),\ (s_1|c_1|r_0),\ (s_1|c_1|r_2)\}$

Result for $\mathcal{K}_r sending\_ACK0 \wedge \mathcal{K}_s receiving\_ACK0$: Intersection between L_1 and $L_2 = \{(s_1|c_2|r_1)\}$

Result for $\exists\mathsf{F}(\mathcal{K}_r sending\_ACK0 \wedge \mathcal{K}_s receiving\_ACK0)$: As exists the path $(s_0|c_0|r_0) \rightarrow (s_1|c_1|r_0) \rightarrow (s_1|c_2|r_1)$, this query returns **true**.

Query #3 $G,(s_1|c_0|r_0) \models \exists[\mathcal{K}_s sending\_msg\_bit\_0 \ \mathsf{U}\ \mathcal{K}_r receiving\_msg\_bit\_0]$

Result for $\mathcal{K}_r receiving\_msg\_bit\_0$: $\{(s_0|c_0|r_0),\ (s_1|c_1|r_0),\ (s_1|c_0|r_0),\ (s_0|c_1|r_0),\ (s_2|c_5|r_0),\ (s_3|c_5|r_0),\ (s_2|c_4|r_0),\ (s_3|c_4|r_0),\ (s_2|c_7|r_0),\ (s_3|c_7|r_0)\}$

Result for $\exists[\mathcal{K}_s sending\_msg\_bit\_0 \ \mathsf{U}\ \mathcal{K}_r receiving\_msg\_bit\_0]$: Since $\mathcal{K}_r receiving\_msg\_bit\_0$ is satisfied in $(s_1|c_0|r_0)$, this query returns **true**.

More detailed queries

Query #4 Is possible the receiver to remain in starvation?

$G,(s_0|c_0|r_0) \models \exists\mathsf{G}\mathcal{K}_r receiving\_msg\_bit\_0$

Result for $\mathcal{K}_r receiving\_msg\_bit\_0$:
CheckK(G, *receiving_msg_bit_0*, r)

% States that contains r_0
RecursiveCheckK(G, *receiving_msg_bit_0*, r, L, $(s_0|c_0|r_0)$)

L = $\{(s_0|c_0|r_0),\ (s_1|c_1|r_0),\ (s_1|c_0|r_0),\ (s_0|c_1|r_0),\ (s_2|c_5|r_0),\ (s_3|c_5|r_0),\ (s_2|c_4|r_0),\ (s_3|c_4|r_0),\ (s_2|c_7|r_0),\ (s_3|c_7|r_0)\}$

Result for $\exists\mathsf{G}\mathcal{K}_r receiving\_msg\_bit\_0$: As exists the path $(s_0|c_0|r_0) \rightarrow (s_1|c_1|r_0) \rightarrow (s_1|c_0|r_0) \rightarrow (s_0|c_0|r_0) \rightarrow$ ..., where all elements there contains '$\mathcal{K}_r receiving\_msg\_bit\_0$', this query returns **true**.

Query #5 $G, (s_0|c_0|r_0) \models \exists\mathsf{F}\mathcal{K}_r\mathcal{K}_s\neg \mathit{sending\_msg\_bit\_1}$

Result for $\mathcal{K}_s\neg \mathit{sending\_msg\_bit\_1}$:
CheckK(G, $\neg \mathit{sending\_msg\_bit\_1}$, s)
% States that contains s_0
RecursiveCheckK(G, $\neg \mathit{sending\_msg\_bit\_1}$, s, L, $(s_0|c_0|r_0)$)
L = $\{(s_0|c_0|r_0)$, $(s_0|c_1|r_0)$, $(s_0|c_2|r_1)$, $(s_0|c_3|r_2)$, $(s_0|c_0|r_2)$, $(s_0|c_1|r_2)\}$

% States that contains s_1
RecursiveCheckK(G, $\neg \mathit{sending\_msg\_bit\_1}$, s, L, $(s_1|c_1|r_0)$)
L = $\{(s_1|c_1|r_0)$, $(s_1|c_0|r_0)$, $(s_1|c_2|r_1)$, $(s_1|c_3|r_2)$, $(s_1|c_0|r_2)$, $(s_1|c_1|r_2)\}$

% States that contains s_2
RecursiveCheckK(G, $\neg \mathit{sending\_msg\_bit\_1}$, s, L, $(s_2|c_4|r_2)$)
L = $\{(s_2|c_4|r_2)$, $(s_2|c_5|r_2)$, $(s_2|c_6|r_3)$, $(s_2|c_5|r_0)$, $(s_2|c_4|r_0)$, $(s_2|c_7|r_0)\}$

Result for $\mathcal{K}_r\mathcal{K}_s\neg \mathit{sending\_msg\_bit\_1}$:
CheckK(G, $\mathcal{K}_s\neg \mathit{sending\_msg\_bit\_1}$, r)
% States that contains r_0
RecursiveCheckK(G, $\mathcal{K}_s\neg \mathit{sending\_msg\_bit\_1}$, r, L, $(s_0|c_0|r_0)$)
RecursiveCheckK(G, $Ks\neg \mathit{sending\_msg\_bit\_1}$, r, L, $(s_3|c_7|r_0)$) $\Rightarrow$ FAIL!

% States that contains r_1
RecursiveCheckK(G, $\mathcal{K}_s\neg \mathit{sending\_msg\_bit\_1}$, r, L, $(s_1|c_2|r_1)$)
L = $\{(s_1|c_2|r_1)$, $(s_0|c_2|r_1)\}$

% States that contains r_2
RecursiveCheckK(G, $\mathcal{K}_s\neg \mathit{sending\_msg\_bit\_1}$, r, L, $(s_1|c_3|r_2)$)
RecursiveCheckK(G, $\mathcal{K}_s\neg \mathit{sending\_msg\_bit\_1}$, r, L, $(s_3|c_5|r_2)$) $\Rightarrow$ FAIL!

% States that contains r_3
RecursiveCheckK(G, $\mathcal{K}_s\neg \mathit{sending\_msg\_bit\_1}$, r, L, $(s_2|c_6|r_3)$)
RecursiveCheckK(G, $\mathcal{K}_s\neg \mathit{sending\_msg\_bit\_1}$, r, L, $(s_3|c_6|r_3)$) $\Rightarrow$ FAIL!

Result for: $\exists\mathsf{F}\mathcal{K}_r\mathcal{K}_s\neg \mathit{sending\_msg\_bit\_1}$: Path $(s_0|c_0|r_0) \rightarrow (s_1|c_1|r_0) \rightarrow (s_1|c_2|r_1)$, where the last state contains '$\mathcal{K}_r\mathcal{K}_s\neg \mathit{sending\_msg\_bit\_1}$'. Then, the query returns **true**.

7 Conclusions

This paper focus on knowledge logics for *MAS*s and Model Checking. Our contribution is a formal language and the corresponding Model Checking process to model and verify multi-agent systems at the knowledge level.

First, a model for *MAS* adequate for Model Checking purposes is defined. Labeled composed automata, in correspondence to K-extended Kripke Models, are constructed from local automata given for each agent.

As a language for knowledge, we presented *KCTL*, an extension of *CTL* with knowledge operators $\mathcal{K}_k$ for each agent k. *KCTL* is capable to reason about knowledge in state transition systems that corresponds to interactive MASs representations. A semantics based on equivalence possibility relations is given for *KCTL*.

With adequate model and language in hands, a Model Checking process close to the usual process for *CTL* [CGP$^+$99] is also defined. The algorithms for checking $\mathcal{K}_k$ formulae, recursively working through the possibility relation over the set of states, is defined and examined in detail. It's efficiency is almost the same for *CTL*.

References

[BAMP81] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 164–176, Williamsburg, Virginia, 1981. ACM Press. ISBN: 0-89791-029-X.

[BGL98] M. Benerecetti, F. Giunchiglia, and L. Serafini. Model Checking Multiagent Systems. In *Journal of Logic and Computation 8(3)*, pages 401–423. 1998.

[CGP$^+$99] E.M. Clark, O. Grumberg, and D.A. Peled. *Model Checking.* MIT Press, 1999.

[EC81] E. A. Emerson and E. M. Clarke. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of the IBM Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 164–176. Springer-Verlag, 1981.

[EJT99] J. Engelfriet, C. M. Jonker and J. Treur. Compositional Verification of Multi-Agent Systems in Temporal Multi-Epistemic Logic. *Proceedings of the 5th International Workshop on Intelligent Agents V, Agent Theories, Architectures, and Languages*, 177–193, Springer-Verlag, 1999. ISBN: 3-540-65713-4.

[FHM$^+$95] R. Fagin, J.Y. Halpern, Y. Moses, et al. *Reasoning About Knowledge.* MIT Press, Cambridge, Massachussetts, 1995.

[KR00] J. F. Kurose and K. W. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet.* Addison-Wesley, Boston, Massachussetts, 2000.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[Leh84] D. Lehmann. Knowledge, common knowledge, and related puzzles. *Proc. 3rd Ann. ACM Conf. on Principles of Distributed Computing*, 62–67, 1984.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[MP02] M. Pauly. A Modal Logic for Coalitional Power in Games. *Journal of Logic and Computation* Vol.12(1), 149–166, 2002.