A Novel Symbolic Approach to Verifying Epistemic Properties of Programs*

Nikos Gorogiannis, Franco Raimondi
Department of Computer Science
Middlesex University, London, UK
{n.gkorogiannis,f.raimondi}@mdx.ac.uk

Ioana Boureanu
Department of Computer Science
University of Surrey, Guilford, UK
i.boureanu@surrey.ac.uk

Abstract

We introduce a framework for the symbolic verification of epistemic properties of programs expressed in a class of general-purpose programming languages. To this end, we reduce the verification problem to that of satisfiability of first-order formulae in appropriate theories. We prove the correctness of our reduction and we validate our proposal by applying it to two examples: the dining cryptographers problem and the ThreeBallot voting protocol. We put forward an implementation using existing solvers, and report experimental results showing that the approach can perform better than state-of-the-art symbolic model checkers for temporal-epistemic logic.

1 Introduction

Epistemic logics capture rich properties such as unlinkability, diagnosability and anonymity [Fagin *et al.*, 1995], and have been applied to complex systems. Several verifiers for temporal-epistemic properties have been developed [Lomuscio *et al.*, 2015; Gammie and van der Meyden, 2004; Kacprzak *et al.*, 2008], and used on a number of applications, ranging from security protocols [Boureanu *et al.*, 2009] and cache-coherence protocols [Baukus and van der Meyden, 2004] to under-water vehicles [Ezekiel *et al.*, 2011].

Typically, verification of epistemic properties in such systems is done by translating the specifications into the input language of a model checker (e.g., [Boureanu *et al.*, 2009]). Thus it is not possible, in general, to verify epistemic properties directly on program code. Also, it is hard to capture richer (e.g., first-order) state specifications, because the base logic of most temporal-epistemic verifiers is propositional.

This is in contrast to *program verification*. Therein, the full power of temporal logic is foregone (properties only constrain final states, all or some), but programs are expressed in concrete languages, and base logics are very expressive (dealing with integers, reals, arrays, strings etc). In this context, predicate transformers (e.g., strongest postconditions [Dijkstra, 1976]) are used to reduce verification to first-order queries fed to SMT solvers (e.g., Z3 [De Moura and Bjørner, 2008]).

Inspired by program verification, we propose a methodology for verifying epistemic properties of programs. We focus on S5-*like epistemic properties about program states* (e.g., "*I know that variable x is equal to $y + 5$*"). As in interpreted systems [Fagin *et al.*, 1995], agents can *observe* certain program variables. Also, *the program (its transition relation) is assumed to be known by all agents*, similarly to most existing model checkers. We summarise our contributions below.

A New Methodology. We propose a framework whereby a program-epistemic logic of agents can be defined, given user-chosen parameters. These parameters include (i) a base logic over program variables; (ii) the programming language; and, (iii) the set of observable program variables for each agent.

Reducing to First-Order. We show that if the *strongest postcondition* operator is computable for the chosen base logic/programming language, then validity of program-epistemic specifications is reducible to validity in first-order fragments (such as QBF and Presburger arithmetic).

Over-approximation. When the strongest postcondition can only be over-approximated (such as in programming languages with unbounded loops), we show that the validity of *positive* epistemic specifications still reduces to that of first-order fragments, in a sound but incomplete way.

Case Studies. We instantiate this framework on two case studies: the dining cryptographers problem [Chaum, 1988] and the ThreeBallot protocol [Rivest and Smith, 2007]. We then show how the behaviour of the two systems can be specified using epistemic properties employing propositional logic, and linear integer arithmetic, respectively.

Evaluation. We use off-the-shelf SMT solvers to verify epistemic properties via our reduction to first-order fragments. We experimentally assess the performance of the obtained verifiers, and compare with temporal-epistemic model checkers. We find that our method yields verifiers which can be implemented with minimal effort, whilst remaining competitive for properties in our language with state-of-the-art tools.

Paper Structure. Sec. 2 introduces the syntax and semantics of the proposed framework. Sec. 3 presents the reduction of epistemic to first-order validity. Our case studies and experimental results are detailed in Sec. 4. Finally, we discuss related work in Sec. 5 and conclude in Sec. 6.

*Raimondi was supported by EPSRC grant EP/K033921/1; Boureanu was funded by the Marie Skłodowska-Curie grant 661362.

2 An Epistemic Logic of Programs

We start from a user-specified *base language* and build on top of it, up to more expressive logics that specify epistemic properties of programs. We introduce these languages below.

Notation. We denote vectors $(x_1, \ldots, x_n)$ by $\mathbf{x}$ and their length by $|\mathbf{x}| = n$. We extend quantifiers over vectors of variables: $\forall \mathbf{x}.\, \phi$ means $\forall x_1.\, \forall x_2.\, \ldots \forall x_n.\, \phi$. We treat vectors as sets: $\mathbf{x} \subseteq \mathbf{y}$ means that every element of $\mathbf{x}$ appears in $\mathbf{y}$. We write $FV(P)$ for the set of free variables of a formula P.

2.1 Logical Languages: Syntax

Agents & Program Variables. Let $\mathcal{A}$ be a finite set of *agents*. Formulas are over a countable set of variables $\mathcal{V}$. To this end, we fix the following finite sets of variables:

- $\mathbf{p} \subseteq \mathcal{V}$ is a non-empty set of *program variables*;
- $\mathbf{o}_A \subseteq \mathbf{p}$ are the variables the agent $A \in \mathcal{A}$ can *observe*;
- $\mathbf{n}_A = \mathbf{p} \setminus \mathbf{o}_A$ are variables agent $A \in \mathcal{A}$ *cannot observe*.

Syntax. Let the *base language*, $\mathcal{L}_{QF}$, be a quantifier-free, first-order language. We leave $\mathcal{L}_{\text{QF}}$ under-specified to allow various instantiations; we denote formulas of $\mathcal{L}_{\text{QF}}$ with π.

We now define three languages on top of $\mathcal{L}_{\text{QF}}$.

The *first-order language*, $\mathcal{L}_{FO}$, is the extension of $\mathcal{L}_{\text{QF}}$ with quantifiers. Its definition is standard,

$$\phi ::= \pi \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \Rightarrow \phi_2 \mid \forall x.\, \phi \mid \exists x.\, \phi$$

where $x \in \mathcal{V}$ and $\pi \in \mathcal{L}_{\text{QF}}$. We use the Greek letters ϕ, ψ to denote formulas given in this first-order language.

The *epistemic language* language, $\mathcal{L}_{\mathsf{K}}$, is the extension of $\mathcal{L}_{\text{QF}}$ with epistemic modalities K_A. The modality K_A expresses the knowledge of agent A. Its syntax is as follows:

$$\alpha ::= \pi \mid \neg\alpha \mid \alpha_1 \wedge \alpha_2 \mid \alpha_1 \vee \alpha_2 \mid \alpha_1 \Rightarrow \alpha_2 \mid \mathsf{K}_A\alpha$$

where π is a base-language formula and $A \in \mathcal{A}$ is an agent. We use the Greek letter α to denote formulas in $\mathcal{L}_{\mathsf{K}}$.

We fix a (possibly infinite) set of *commands*[1], $\mathcal{C}$. The *program specification language*, $\mathcal{L}_{\Box\mathsf{K}}$, extends $\mathcal{L}_{\mathsf{K}}$ with every formula $\beta = \Box_C\alpha$, meaning "*at all final states of C, α holds*", where $\alpha \in \mathcal{L}_{\mathsf{K}}$ and $C \in \mathcal{C}$ is a command.

Finally, we fix $\mathcal{L}^+_{\mathsf{K}}$ as the $\neg, \Rightarrow$-free fragment of $\mathcal{L}_{\mathsf{K}}$ (note that $\neg$ and $\Rightarrow$ over base-language formulas π *are* allowed, while the dual operator of K, $\neg\mathsf{K}\neg$ is not allowed).

2.2 Logic Languages: Semantics

Let $\mathcal{D}$ be a set, used as the *domain* for interpreting variables and quantifiers. A *valuation* is a total function $s : \mathcal{V} \to \mathcal{D}$. We lift valuations to tuples of variables and terms (pointwise). Such a valuation is a *state*. Let $\mathcal{U}$ be the set of all states.

We use $s[x \mapsto c]$ to denote the state s' which agrees with s on all variables except for x, and which also has the property that $s'(x) = c$. We extend this notation to vectors of variables/values, i.e., $s[\mathbf{x} \mapsto \mathbf{c}] = s[x_1 \mapsto c_1] \cdots [x_n \mapsto c_n]$.

First-order Semantics. Let I be an interpretation of constants, functions and predicates in $\mathcal{L}_{\text{QF}}$ over the domain $\mathcal{D}$. Then, the standard first-order semantics is as follows.

$$\begin{array}{lcl}
s \models \pi & \iff & \text{in accordance to interpretation } I \\
s \models \phi_1 \circ \phi_2 & \iff & (s \models \phi_1) \circ (s \models \phi_2) \\
s \models \neg\phi & \iff & s \not\models \phi \\
s \models \exists x.\phi & \iff & \exists c \in \mathcal{D}.\, s[x \mapsto c] \models \phi \\
s \models \forall x.\phi & \iff & \forall c \in \mathcal{D}.\, s[x \mapsto c] \models \phi.
\end{array}$$

where $\circ$ is $\wedge$, $\vee$ or $\Rightarrow$.

The *interpretation* $[\![\phi]\!]$ of a first-order formula ϕ is the set of states satisfying it, i.e., $[\![\phi]\!] = \{s \in \mathcal{U} \mid s \models \phi\}$.

Epistemic and Program Semantics. For each command $C \in \mathcal{C}$, let $R_C \subseteq \mathcal{U} \times \mathcal{U}$ be a binary relation over $\mathcal{U}$, representing the *transition relation* of C. We overload R_C as a function from states to sets of states, as well as a function from sets of states to sets of states, as standard.

$$R_C(s) = \{s' \mid (s, s') \in R_C\} \qquad R_C(W) = \textstyle\bigcup_{s \in W} R_C(s)$$

Definition 2.1. The *strongest postcondition operator* is a partial function $SP(-,-) : \mathcal{L}_{\text{FO}} \times \mathcal{C} \rightharpoonup \mathcal{L}_{\text{FO}}$ defined as follows.

$$SP(\phi, C) = \psi \quad \text{iff} \quad [\![\psi]\!] = R_C([\![\phi]\!])$$

(here ψ is assumed to be in some canonical form).

Definition 2.2. A function $f : \mathcal{L}_{\text{FO}} \times \mathcal{C} \to \mathcal{L}_{\text{FO}}$ is *well-behaved* if it is total and computable, and, $FV(\phi) \subseteq \mathbf{p}$ implies $FV(f(\phi, C)) \subseteq \mathbf{p}$ for all $\phi \in \mathcal{L}_{\text{FO}}$ and $C \in \mathcal{C}$.

A function $f : \mathcal{L}_{\text{FO}} \times \mathcal{C} \to \mathcal{L}_{\text{FO}}$ *over-approximates the strongest postcondition* iff it is well-behaved, and, $[\![f(\phi, C)]\!] \supseteq R_C([\![\phi]\!])$ for all $\phi \in \mathcal{L}_{\text{FO}}$ and $C \in \mathcal{C}$.

Definition 2.3. Let $X \subseteq \mathcal{V}$. The *indistinguishability* relation $\sim_X$ is a binary relation over $\mathcal{U}$, defined as follows:

$$s \sim_X s' \iff \forall x \in X.\, (s(x) = s'(x))$$

Clearly $\sim_X$ is an equivalence relation over $\mathcal{U}$, for any X.

We now define the *interpretation of a program specification* β, by defining a ternary satisfaction relation $W, s \Vdash \beta$, where W is a set of states and s is a state in W, as follows:

$$\begin{array}{lcl}
W, s \Vdash \pi & \iff & s \models \pi \\
W, s \Vdash \neg\alpha & \iff & W, s \not\Vdash \alpha \\
W, s \Vdash \alpha_1 \circ \alpha_2 & \iff & (W, s \Vdash \alpha_1) \circ (W, s \Vdash \alpha_2) \\
W, s \Vdash \mathsf{K}_A\alpha & \iff & \forall s' \in W.\, (s \sim_{\mathbf{o}_A} s' \implies W, s' \Vdash \alpha) \\
W, s \Vdash \Box_C\alpha & \iff & \forall s' \in R_C(s).\, (R_C(W), s' \Vdash \alpha)
\end{array}$$

where $\circ$ is $\wedge$, $\vee$, or $\Rightarrow$, and $C \in \mathcal{C}$ is a command. Intuitively, in a judgement $W, s \Vdash \beta$, the set W designates the reachable, or *epistemically relevant* states, of which s must be a member.

Definition 2.4 (Validity of program specifications.)**.** Let $\phi \in \mathcal{L}_{\text{FO}}$ and β be a program specification. We write $\phi \Vdash \beta$ to mean that for all $s \in [\![\phi]\!]$, we have that $[\![\phi]\!], s \Vdash \beta$. We will also, sometimes, write $W \Vdash \beta$, where W is a set of states.

Defn. 2.4 sets out the format of our program-epistemic specifications. For instance, $\phi \Vdash \mathsf{K}_A\pi$ means that in all states satisfying ϕ, agent A knows π. Alternatively, $\phi \Vdash \Box_C\neg\mathsf{K}_A\pi$ means: if command C starts at a state satisfying ϕ, then in all states where the execution finishes, agent A does not know π.

Lemma 2.5. *For any first-order formula ϕ and epistemic formula α, $\phi \Vdash \Box_C\alpha$ iff $R_C([\![\phi]\!]) \Vdash \alpha$.*

Lemma 2.6. *Let $\phi \in \mathcal{L}_{FO}$, $s, s' \in \mathcal{U}$ and $X, Y \subseteq \mathcal{V}$. Then,*

1. *If $X \subseteq Y$ and $s \sim_Y s'$ then $s \sim_X s'$.*
2. *If $s \sim_X s'$ and $s \sim_Y s'$ then $s \sim_{X \cup Y} s'$.*
3. *If $FV(\phi) \subseteq X$ and $s \sim_X s'$, then $s \models \phi$ iff $s' \models \phi$.*

[1] NB: we use the terms *command* and *program* interchangeably, as a program can always be seen as a compound command.

3 Reducing to First-Order Validity

In this section, we show that the validity/satisfaction of a formula in $\mathcal{L}_{\Box\mathsf{K}}$ reduces to that of a first-order formula. More concretely, the main results of this section (Theorems 3.3 and 3.5) can be understood in the following way: the validity of certain first-order formulas is equivalent to, or implies, the validity of a program specification in the format given by Defn. 2.4. Underpinning this reduction is a translation of epistemic formulas into the first-order language.

Definition 3.1. We define the function $\tau : \mathcal{L}_{\mathsf{K}} \to \mathcal{L}_{\mathrm{FO}}$ by recursion on the structure of epistemic formulas as follows.

$$\begin{array}{ll} \tau(\phi,\pi) = \pi & \tau(\phi,\alpha_1\circ\alpha_2) = \tau(\phi,\alpha_1)\circ\tau(\phi,\alpha_2) \\ \tau(\phi,\neg\alpha) = \neg\tau(\phi,\alpha) & \tau(\phi,\mathsf{K}_A\alpha) = \forall\mathbf{n}_A.\,(\phi\Rightarrow\tau(\phi,\alpha)) \end{array}$$

The first parameter of τ is a first-order formula representing the set of reachable, or epistemically relevant, states. The translation maps epistemic formulas to (guarded) quantified first-order formulas, exploiting the definition of the indistinguishability relation in terms of state variables.

We first show that satisfaction of α at a state coincides with first-order satisfaction of $\tau(-,\alpha)$ at the same state.

Lemma 3.2. *For any $\phi \in \mathcal{L}_{FO}$, state $s \in [\![\phi]\!]$ and $\alpha \in \mathcal{L}_{\mathsf{K}}$ such that $FV(\phi)\cup FV(\alpha)\subseteq \mathbf{p}$, $[\![\phi]\!], s \Vdash \alpha$ iff $s \models \tau(\phi,\alpha)$.*

Proof. By structural induction on α. The inductive hypothesis is that for all subformulas of α, the result holds.

Cases for π, $\neg$, $\wedge$, $\vee$, $\Rightarrow$: Immediate by the definitions of τ and of the first-order satisfaction relation.

Case for $\mathsf{K}_A\alpha$: We need to show $[\![\phi]\!], s \Vdash \mathsf{K}_A\alpha$ iff $s \models \forall\mathbf{n}_A.\,(\phi\Rightarrow\tau(\phi,\alpha))$.

Direction $\Rightarrow$: Suppose $s \not\models \forall\mathbf{n}_A.\,(\phi\Rightarrow\tau(\phi,\alpha))$. Thus, there exists $\mathbf{c}\in\mathcal{D}^{|\mathbf{n}_A|}$ such that $s[\mathbf{n}_A\mapsto\mathbf{c}]\models\phi\wedge\neg\tau(\phi,\alpha)$. Expanding $s[-]$, there is a state s' such that for all $x\in\mathcal{V}\setminus\mathbf{n}_A$, $s(x)=s'(x)$, and $s'\models\phi\wedge\neg\tau(\phi,\alpha)$. By Defn. 2.3, we first obtain that $s\sim_{\mathcal{V}\setminus\mathbf{n}_A}s'$, which then implies $s\sim_{\mathbf{o}_A}s'$ by item (1) of Lem. 2.6. Also, $s'\models\phi$ and $s'\not\models\tau(\phi,\alpha)$, which means that $[\![\phi]\!], s' \not\Vdash \alpha$, by the inductive hypothesis. Therefore, there exists a state $s'\models\phi$ such that $s\sim_{\mathbf{o}_A}s'$ and $s'\not\models\tau(\phi,\alpha)$. Thus, $[\![\phi]\!], s\not\Vdash\mathsf{K}_A\alpha$, completing the proof.

Direction $\Leftarrow$: Assume $s\models\forall\mathbf{n}_A.\,(\phi\Rightarrow\tau(\phi,\alpha))$ but $[\![\phi]\!], s\not\Vdash\mathsf{K}_A\alpha$. The former implies that: for all value tuples $\mathbf{c}\in\mathcal{D}^{|\mathbf{n}_A|}$, we have $s[\mathbf{n}_A\mapsto\mathbf{c}]\models\phi\Rightarrow\tau(\phi,\alpha)$. Expanding $s[-]$, we get that if for all $x\in\mathcal{V}\setminus\mathbf{n}_A$, $s(x)=s'(x)$, then $s'\models\phi\Rightarrow\tau(\phi,\alpha)$. Using Defn. 2.3, we conclude that for all states s', if $s\sim_{\mathcal{V}\setminus\mathbf{n}_A}s'$ then $s'\models\phi\Rightarrow\tau(\phi,\alpha)$.

The second assumption implies that there exists a state $s_0\in[\![\phi]\!]$ such that $s\sim_{\mathbf{o}_A}s_0$ and $[\![\phi]\!], s_0\not\Vdash\alpha$. By the inductive hypothesis, this means $s_0\not\models\tau(\phi,\alpha)$. So, $s\sim_{\mathbf{o}_A}s_0$ and $s_0\models\phi\wedge\neg\tau(\phi,\alpha)$. Define a new state $s_1=s[\mathbf{p}\mapsto s_0(\mathbf{p})]$. But, if $FV(\phi)\cup FV(\alpha)\subseteq\mathbf{p}$, then $FV(\tau(\phi,\alpha))\subseteq\mathbf{p}$ (by simple induction). Thus, since $s_1\sim_{\mathbf{p}}s_0$ by construction, item (3) of Lem. 2.6 gives us that $s_1\models\phi\wedge\neg\tau(\phi,\alpha)$.

Also, since $\mathbf{o}_A\subseteq\mathbf{p}$ and $s_1\sim_{\mathbf{p}}s_0$, it follows from item (1) of Lem. 2.6 that $s_1\sim_{\mathbf{o}_A}s_0$. By the transitivity of $\sim_{\mathbf{o}_A}$ and the fact that $s\sim_{\mathbf{o}_A}s_0$ we have that $s\sim_{\mathbf{o}_A}s_1$. Also, by construction, $s\sim_{\mathcal{V}\setminus\mathbf{p}}s_1$, and given that $(\mathcal{V}\setminus\mathbf{p})\cup\mathbf{o}_A=\mathcal{V}\setminus\mathbf{n}_A$, we obtain $s\sim_{\mathcal{V}\setminus\mathbf{n}_A}s_1$ by item (2) of Lem. 2.6. This, along with the fact that $s_1\models\phi\wedge\neg\tau(\phi,\alpha)$, directly contradicts that for all states s', if $s\sim_{\mathcal{V}\setminus\mathbf{n}_A}s'$ then $s'\models\phi\Rightarrow\tau(\phi,\alpha)$. □

We can now state the first main result: when $SP(-,-)$ is well-behaved, validity of program specifications coincides with that of first-order formulas produced by the τ function.

Theorem 3.3. *Let $\phi\in\mathcal{L}_{FO}$ and $\alpha\in\mathcal{L}_{\mathsf{K}}$ be formulas such that $FV(\phi)\cup FV(\alpha)\subseteq\mathbf{p}$. If SP is well-behaved, then*

$$\phi \Vdash \Box_C\alpha \iff SP(\phi,C)\models\tau(SP(\phi,C),\alpha)\,.$$

Proof. Assume the left-hand side. By Lem. 2.5 this is equivalent to $R_C([\![\phi]\!])\Vdash\alpha$. By the semantics and Defn. 2.1, this is equivalent to claiming that for all s, if $s\models SP(\phi,C)$, then $SP(\phi,C), s\Vdash\alpha$. By Lem. 3.2, for all s, if $s\models SP(\phi,C)$, then $s\models\tau(SP(\phi,C),\alpha)$, completing the proof. □

The strongest postcondition operator is often not well-behaved, e.g., when the transition relation R is generated by programs containing unbounded loops over infinite domains. Our second main result states that when *positive* epistemic formulas are involved ($\mathcal{L}^+_{\mathsf{K}}$), and it is possible to compute an over-approximation of $SP(-,-)$, then validity of the translated formulas implies validity of the program specification. This mirrors the situation in program analysis, where undecidability often leads to sound but incomplete analyses.

We start with a technical lemma and lift it up to validity.

Lemma 3.4. *Let $\phi\in\mathcal{L}_{FO}$ and $\alpha\in\mathcal{L}^+_{\mathsf{K}}$ be formulas such that $FV(\phi)\cup FV(\alpha)\subseteq\mathbf{p}$. Then, for any set of states $W\subseteq[\![\phi]\!]$ and any state $s\in W$, if $[\![\phi]\!], s\Vdash\alpha$ then $W, s\Vdash\alpha$.*

Proof. By structural induction on α. The only interesting case is the epistemic one, that is, if $[\![\phi]\!], s\Vdash K_A\alpha$ then $W, s\Vdash\mathsf{K}_A\alpha$. Suppose the former. Then, for any state $s'\in[\![\phi]\!]$ such that $s\sim_{\mathbf{o}_A}s'$, it is the case that $[\![\phi]\!], s'\Vdash\alpha$. But, since $W\subseteq[\![\phi]\!]$, we can restrict attention to states in W, i.e., for any state $s'\in W$ such that $s\sim_{\mathbf{o}_A}s'$, it is the case that $[\![\phi]\!], s'\Vdash\alpha$. Now we can apply the inductive hypothesis on s' and obtain that for any state $s'\in W$ such that $s\sim_{\mathbf{o}_A}s'$, $W, s'\Vdash\alpha$ holds, which completes the proof. □

Theorem 3.5. *Suppose $f : \mathcal{L}_{FO}\times\mathcal{C}\to\mathcal{L}_{FO}$ over-approximates the strongest postcondition. If $\phi\in\mathcal{L}_{FO}$ and $\alpha\in\mathcal{L}^+_{\mathsf{K}}$ are formulas such that $FV(\phi)\cup FV(\alpha)\subseteq\mathbf{p}$, then*

$$f(\phi,C)\models\tau(f(\phi,C),\alpha) \implies \phi\Vdash\Box_C\alpha\,.$$

Proof. Assume $f(\phi,C)\models\tau(f(\phi,C),\alpha)$. By Lem. 3.2, this means that $f(\phi,C)\Vdash\alpha$, or (Defn. 2.4), that for all states $s\models f(\phi,C)$, it is the case that $[\![f(\phi,C)]\!], s\Vdash\alpha$. By assumption, $R_C([\![\phi]\!])\subseteq[\![f(\phi,C)]\!]$. Therefore we can soundly restrict the relevant set of states, obtaining that for all states $s\in R_C([\![\phi]\!])$, it is the case that $[\![f(\phi,C)]\!], s\Vdash\alpha$. By Lem. 3.4, for all s, if $s\in R_C([\![\phi]\!])$, then $R_C([\![\phi]\!]), s\Vdash\alpha$. This is equivalent to $\phi\Vdash\Box_C\alpha$, thus completing the proof. □

Remark 3.6. Without fixing the various framework parameters, it is hard to determine the decidability and complexity of program specifications. Some observations are possible:

First, the length of $\tau(\phi,\alpha)$ is linear in $\|\phi\|+\|\alpha\|$, where $\|\phi\|$ is the number of symbols in ϕ, defined as usual.

Second, the number of quantifier alternations in $\tau(\phi,\alpha)$ (in negation-normal form) is bounded by the quantifier alternations in ϕ plus the number of alternations of $\neg$ and K in α.

3.1 A Simple Programming Language

We informally present a simple, loop-free imperative programming language, and how it can be used to interpret execution modalities $\Box_C$ and $SP(-,-)$. In the (infinite) set of commands $\mathcal{C}$ we include assignments, conditionals and sequencing. We omit unbounded loops in order to guarantee that the strongest postcondition is well-behaved. We do not fix the types of variables, to allow different user choices for domains. In Sections 4.1 and 4.2 we show how we can use this language with Booleans and integers. Below we show the syntax of commands and the rules for computing $SP(-,-)$.

Command C	$SP(\phi, C)$
$x := *$	$\exists y.\, \phi[y/x]$
$x := e$	$\exists y.\, (x = e[y/x] \wedge \phi[y/x])$
$\texttt{if}(\pi)\ C_1\ \texttt{else}\ C_2$	$SP(\pi \wedge \phi, C_1) \vee SP(\neg\pi \wedge \phi, C_2)$
$C_1; C_2$	$SP(SP(\phi, C_1), C_2)$

Above, x is a program variable and y is a fresh logical variable. The transition relations R_C are defined as usual.

Remark 3.7. The $SP(-,-)$ operator for the above programming language is well-behaved (cf. Defn. 2.2) for any choice of other framework parameters (domain, program variables, base language etc). In addition, we can make the following observations that we use in Sections 4.1 and 4.2:

- $SP(-,-)$ may only introduce existential quantifiers.
- If $x \notin FV(\phi)$, then $SP(\phi, x := e) = (\phi \wedge x = e)$. That is, if x is unrestricted, no quantifiers are introduced.
- For a fixed C, the size of $SP(\phi, C)$ is polynomial in $\|\phi\|$.

4 Evaluation

We now present two case studies, where we instantiate the framework presented in Sec. 2, mechanise the verification of relevant epistemic properties by using appropriate tools such as SMT solvers, and assess verification performance.

All the experiments[2] have been performed on a 4-core 2.4 GHz Intel Core i7 MacBook Pro with 16 GB of RAM running OS X 10.11.6. The version of MCMAS is 1.2.2 and Z3 is 4.5.1; both tools have been compiled from source on the target machine.

4.1 Dining Cryptographers

Problem description. This is a scenario where n agents (cryptographers) dine at a round table [Chaum, 1988]. The dinner may have been paid by their employer (the NSA), or by one of the agents. They execute a protocol to reveal whether one of the agents paid, but without revealing which one. The protocol supplies each pair of adjacent agents with a random coin, which can be observed only by that pair. Each agent announces the result of XORing three Booleans: the two coins observable by her and the status of whether she paid for the dinner. The XOR of all announcements is proven to be equal to the disjunction of whether any agent paid.

This protocol has been used as an evaluation case study for several methods of verifying epistemic properties of programs [Kacprzak *et al.*, 2006; Kacprzak *et al.*, 2008; Lomuscio *et al.*, 2015].

[2]Our code is released open-source at `goo.gl/so8bQc`.

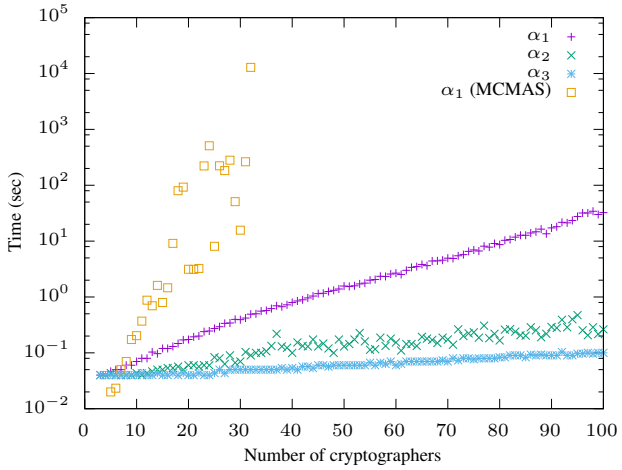


Figure 1: Dining cryptographers, performance vs MCMAS.

Framework Instantiation. The *domain* is $\mathcal{D} = \{\top, \bot\}$. The *base language* $\mathcal{L}_{\text{QF}}$ is propositional logic. The set of *agents* is $\mathcal{A} = \{0, \ldots, n-1\}$. The *program variables* are $\mathbf{p} = \{x\} \cup \{p_i, c_i \mid 0 \leq i < n\}$; x is the XOR of announcements; p_i encodes whether agent i has paid; and, c_i encodes the coin shared between agents $i-1$ and i. Observable variables for agent $i \in \mathcal{A}$ are $\mathbf{o}_i = \{x, p_i, c_i, c_{i+1 \bmod n}\}$, and $\mathbf{n}_i = \mathbf{p} \setminus \mathbf{o}_i$. We model the protocol by an assignment C:

$$x := \bigoplus_{i=0}^{n-1} p_i \oplus c_i \oplus c_{(i+1 \bmod n)} \qquad (C)$$

The language $\mathcal{L}_{\text{FO}}$ is that of QBFs. As seen in Sec. 3.1, $SP(\phi, C)$ is well-behaved. Validity of program specifications is in Σ_j^P or Π_j^P [Stockmeyer, 1977], where j is the sum of quantifier alternations in ϕ and alternations of $\neg$ and K in α.

The initial states, I, are those where at most one agent paid. Thus, we can compute the strongest postcondition:

$$I = \bigwedge_{i=0}^{n-1} \left(p_i \Rightarrow \bigwedge_{j=0, j\neq i}^{n-1} \neg p_j \right)$$
$$SP(I, C) = I \wedge \left(x \Leftrightarrow \bigoplus_{i=0}^{n-1} p_i \oplus c_i \oplus c_{(i+1 \bmod n)} \right)$$

Experiments. We list the specifications we verify. Formula $\alpha_1 = \neg p_0 \Rightarrow \left(\left(\mathsf{K}_0 \bigwedge_{i=0}^{n-1} \neg p_i \right) \vee \left(\bigwedge_{i=1}^{n-1} \neg \mathsf{K}_0 p_i \right) \right)$ states that if agent 0 has not paid then she knows that no agent paid, or (in case an agent paid) she does not know which one. Formula $\alpha_2 = \mathsf{K}_0 \left(x \Leftrightarrow \bigvee_{i=0}^{n-1} p_i \right)$ states that agent 0 knows that x is true iff one of the agents paid. Formula $\alpha_3 = \mathsf{K}_0 p_1$ states that agent 0 knows that agent 1 has paid; this is false, but we include this to assess performance on negative instances too.

To verify $I \Vdash \Box_C \alpha_1$, $I \Vdash \Box_C \alpha_2$ and $I \nVdash \Box_C \alpha_3$ we construct the QBF formula $SP(I, C) \wedge \neg\tau(SP(I, C), \alpha_i)$, feed it to Z3, and test for unsatisfiability, as per Theorem 3.3.

Experimental results for the verification of these formulae are reported in Fig. 1. As computing the reachable-states space is a dominant factor for MCMAS, its performance does not vary significantly across specifications α_1, α_2, α_3. Thus,

for clarity of exposition, we omit the MCMAS curves on specifications α_2, α_3. We observe that (i) MCMAS is faster, or equally fast, for $n \leq 7$, but slower for all $n > 7$; (ii) there are cases where our method is faster than MCMAS by a factor of ≥ 100 (e.g., when $n = 32$) when checking α_1 (which is the computationally most expensive in our case), whilst when verifying α_3 our speed-up is several orders of magnitudes higher.

4.2 ThreeBallot Voting Protocol

Problem description. In the *ThreeBallot voting protocol* [Rivest and Smith, 2007], a voter is given a *multi-ballot* formed of three *atomic* ballots. All atomic ballots are identical and show all candidates in the same fixed order. To vote for a candidate, a voter ticks the name of the candidate on exactly two atomic ballots. To vote against a candidate, a voter ticks the name of the candidate on exactly one atomic ballot. The voting system posts all the atomic ballots cast, randomly ordered, on a public bulletin board. The number of votes for the j-th candidate is the number of atomic ballots with the j-th position marked minus the total number of voters. Several voting requirements exist: (universal) verifiability, coercion-resistance, vote-privacy, etc. *Vote-privacy* broadly means that no observer in a voting protocol can know how some voter (other than themselves) voted. Here we only consider vote-privacy, and only in the absence of active adversaries. As such, the presentation above and our modelling are restricted to aspects of the ThreeBallot protocol relevant to this task, leaving aside several sub-parts of the system (e.g., scanning machines, identifiers on atomic bulletins, etc). We also leave out properties that are not yet expressible in our formalism, such as common-knowledge properties.

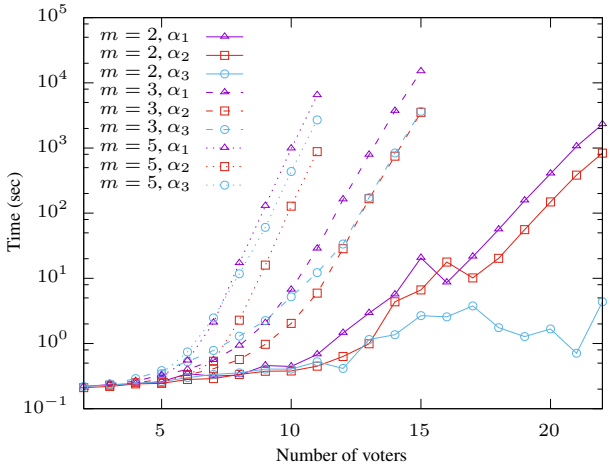
Framework Instantiation. We assume $m \geq 2$ candidates and $n \geq 2$ voters. The *domain* is $\mathcal{D} = \mathbb{N}$. The *base language* $\mathcal{L}_{\text{QF}}$ is linear integer arithmetic. The set of *agents* is $\mathcal{A} = \{1, \ldots, n, P\}$, where P is a 'public observer' agent (surveying all public aspects). The *program variables* are $\mathbf{p} = \bigcup_{j=1}^{m}\{c_j\} \cup \bigcup_{i=1}^{n}\bigcup_{j=1}^{m}\bigcup_{k=1}^{3}\{b_{ijk}\}$. Variable c_j stores the total number of atomic-ballot ticks for candidate j. Variable b_{ijk} (with values in $\{0, 1\}$) represents whether or not voter i ticked next to candidate j on the k-th atomic ballot.

For an agent $i \in \mathcal{A} \setminus \{P\}$, the observable variables are $\mathbf{o}_i = \bigcup_{j=1}^{m}\{c_j\} \cup \bigcup_{j=1}^{m}\bigcup_{k=1}^{3}\{b_{ijk}\}$, i.e., voter i can observe the totals for each candidate, as well as her own vote. For agent P, the observable variables are $\mathbf{o}_P = \bigcup_{j=1}^{m}\{c_j\}$. That is, P only observes the totals for each candidate. For every agent $i \in \mathcal{A}$, the non-observable variables are $\mathbf{n}_i = \mathbf{p} \setminus \mathbf{o}_i$.

To model vote-counting, we first introduce the *macro* $S_{i,j} \equiv \sum_{k=1}^{3} b_{ijk}$, denoting the number of ticks voter i has entered for candidate j. Then, the program C is:

$$c_1 := \sum_{i=1}^{n} S_{i,1} \; ; \; \ldots \; ; \; c_m := \sum_{i=1}^{n} S_{i,m} \qquad (C)$$

The language $\mathcal{L}_{\text{FO}}$ is Presburger arithmetic. For any $\phi \in \mathcal{L}_{\text{FO}}$, $SP(\phi, C)$ is well-behaved (Sec. 3.1). Program specification validity is in $\Sigma_\ell^{\mathsf{EXP}}$ or Π_ℓ^{EXP} (classes of the *weak exponential hierarchy* [Haase, 2014]), where $\ell + 1$ is the sum of quantifier alternations in ϕ and alternations of $\neg$ and K in α.



(a) Performance on the 3-ballot protocol.

$$\begin{aligned}
S_{i,j} &\equiv \textstyle\sum_{k=1}^{3} b_{ijk} \\
B &\equiv \textstyle\bigwedge_{i=1}^{n}\bigwedge_{j=1}^{m}\bigwedge_{k=1}^{3}(b_{ijk} = 0 \vee b_{ijk} = 1) \\
V_{i,j} &\equiv (S_{i,j} = 2) \\
\bar{V}_{i,j} &\equiv (S_{i,j} = 1) \\
CV_i^{\geq 0} &\equiv \textstyle\bigvee_{j=1}^{m} V_{i,j} \\
CV_i^{\leq 1} &\equiv \textstyle\bigwedge_{j=1}^{m}\left(V_{i,j} \Rightarrow \bigwedge_{j'=1,j'\neq j}^{m} \bar{V}_{i,j'}\right) \\
CV &\equiv \textstyle\bigwedge_{i=1}^{n}(CV_i^{\geq 0} \wedge CV_i^{\leq 1}) \\
NU &\equiv \textstyle\bigwedge_{j=1}^{m}\bigvee_{i=1}^{n} V_{i,j} \\
NU_{\text{mod } i} &\equiv \textstyle\bigwedge_{j=1}^{m}\bigvee_{i'=1,i'\neq i}^{n} V_{i',j} \\
I &\equiv B \wedge CV \wedge NU \\
I_{\text{mod } i} &\equiv B \wedge CV \wedge NU_{\text{mod } i}
\end{aligned}$$

(b) Macros used in the Three-Ballot system.

Figure 2: Experimental evaluation of case studies.

Fig. 2b depicts a number of macros which we use in composing the program specifications verified. Macro B bounds the possible values of the entries on atomic ballots to 0 or 1, i.e., no tick or a tick. Macro $V_{i,j}$ (resp. $\bar{V}_{i,j}$) states that voter i voted for (resp. against) candidate j, i.e., two ticks vs one tick. Macro $CV_i^{\geq 0}$ states that voter i voted for at least one candidate and $CV_i^{\leq 1}$ states that voter i voted for at most one candidate. Macro CV states that all voters voted correctly. Macro NU states that the vote is not unanimous, as every candidate received at least one vote; we will require non-unanimity to avoid trivial vote-privacy breaches. Macro $NU_{\text{mod } i}$ is a variation on NU, and states that every candidate received at least one vote from voters other than a designated voter i; we will require this non-unanimity condition for asserting voter-privacy relative to a voter agent. Accordingly, Fig. 2b shows the two possible initial conditions, I and $I_{\text{mod } i}$.

Thus, $SP(I, C) = I \wedge \big(\mathbf{c} = (\sum_{i=1}^{n} S_{i,1}, \ldots, \sum_{i=1}^{n} S_{i,m})\big)$ where $\mathbf{c}$ is the tuple $(c_1, \ldots, c_m)$, and similarly for $I_{\text{mod } i}$.

Experiments. The formula $\alpha_1 = \neg \mathsf{K}_P V_{1,1}$ states that the public observer does not know that voter 1 voted for candidate 1. The formula $\alpha_2 = \neg \mathsf{K}_1 V_{2,1}$ states that voter 1 does not know that voter 2 voted for candidate 1. We verify the following scenarios: (a) $I \Vdash \Box_C \alpha_1$; (b) $I_{\text{mod } 1} \Vdash \Box_C \alpha_2$; and, (c) $I \nVdash \Box_C \alpha_2$. We construct the relevant Presburger formulas obtained by Theorem 3.3, and issue the queries to Z3.

We make the following observations. With a timeout of roughly 1h, for $m = 2$ candidates, we can verify all formulas for up to $n = 22$ voters. For $m = 3$, we can check the specifications up to $n = 15$. For $m = 5$, we stop at $n = 11$. Runtime is evidently exponential in both n and m. Increasing m rather than n has a more pronounced impact on verification runtime; this is apparent on the graphs, as different formulas for the same m value cluster together. Finally, disproof (i.e., α_3) seems to be more tractable when $m = 2$.

5 Related Work

Several model checkers exist for *model verification* against properties expressed in temporal-epistemic logics [Lomuscio *et al.*, 2015; Gammie and van der Meyden, 2004; Kacprzak *et al.*, 2008]. Their underlying techniques include binary decision diagrams, automata, and bounded model checking using SAT solvers. Abstraction and parametrisation have also been introduced in some of these tools, to tackle infinite-state systems [Belardinelli *et al.*, 2016; Kouvaros and Lomuscio, 2016]. A fix-point technique for verifying bounds on resources in multi-agent systems using SMT solvers is described in [Zbrzezny *et al.*, 2016]. As per Section 4, we generally outperform significantly MCMAS [Lomuscio *et al.*, 2015], a state-of-the-art tool in this domain.

Verification of epistemic properties in (possibly) infinite state systems is also addressed in [Cimatti *et al.*, 2016]. The authors employ a counter-example guided abstraction refinement loop to translate a class of temporal-epistemic properties to LTL properties, and to generate a satisfiability problem for an infinite-state model checker. Beside the core technique, the key differences between [Cimatti *et al.*, 2016] and our approach are three-fold. (1) Our input is a program in a general-purpose language rather than the model of a system. (2) We can handle nested epistemic operators (but our temporal expressivity is limited to the final states of programs). (3) Our approach performs better on the epistemic properties of the dining cryptographers that can be expressed both by our framework and by [Cimatti *et al.*, 2016]: for the formula $p_0 \Rightarrow \mathsf{K}_0(\neg \bigvee_{i=1}^{n-1} p_i)$, the authors report running times of 287, 598 and 765 seconds for 280, 360 and 400 cryptographers, respectively; in all cases, this formula is checked in less than 1 second in our framework.

Verification of *programs* against properties expressed in (temporal-)epistemic logics is a less explored area of research compared to model-based verification. Going back some decades, the verification of epistemic properties for a LISP-like language called REX was investigated in [Rosenschein and Kaelbling, 1986]; the authors employed a variant of LTL+K and presented a calculus to prove logical consequences of a REX program. On a slightly different path, [Gelfond, 1994; Zhang, 2006] addressed the theoretical problem of extending (declarative) programming languages with epistemic operators. More recently, the approach described in [Balliu *et al.*, 2012] verifies whether Java programs respect non-interference properties (modulo declassification), given as epistemic formulae. The authors employ a version of the model checker JPF [Păsăreanu and Rungta, 2010] to generate the state space of a Java program and then either MCMAS or Z3 to verify a symbolic representation of the reachable state-space. The key differences between [Balliu *et al.*, 2012] and our approach stem from their focus on non-interference properties and the need of a concolic execution engine. Finally, dynamic epistemic logic [Plaza, 2007] lends itself to the expression of epistemic properties of programs; preliminary work on a verification tool in this space is described in [Wang, 2016].

6 Conclusions

In this paper, we proposed a new approach to verifying epistemic properties of programs. We showed how this method can be applied to arbitrary logics and programming languages. We use *program-epistemic* specifications, expressing the requirement that the given epistemic properties hold on all final states of the program. We showed how program-epistemic specifications can be reduced to appropriate queries to tools such as SMT solvers. We instantiated our approach in two case studies, the Dining Cryptographers problem and the ThreeBallot voting protocol, and experimentally evaluated verification performance.

For any given instantiation of our framework, the translation of program-epistemic properties into first-order sentences can be automated with great ease. In addition, we expect that advances in SMT technology will directly translate into performance gains for tools based on our methodology.

Our approach is not, of course, without limitations. We traded off temporal expressivity, to deal with arbitrary programming languages. Thus we cannot directly encode properties utilising, e.g., *until* operators or other complex temporal properties. This is a research direction we plan to pursue, drawing inspiration from attempts to lift infinite-state program verification to the verification of true temporal properties [Cook *et al.*, 2012].

Common-knowledge properties are outside the current reach of our approach. We plan to investigate an extension of our framework that can deal with common knowledge, possibly by viewing it as a fixpoint and using *cyclic proof* to discharge its instances [Brotherston *et al.*, 2012].

Another limitation is the restriction to positive epistemic properties ($\mathcal{L}_{\mathsf{K}}^{+}$) when the strongest postcondition is not well-behaved (cf. Theorem 3.5). To lift this limitation, we believe that *static analysis* methods such as *abstract interpretation* [Cousot and Cousot, 1977] can be fruitfully employed here, especially when over- and under-approximating analyses are combined. We plan to follow this thread in future work.

Another avenue of future work is a twofold extension of the work in [Armando *et al.*, 2014] which, one, focused on SAT solvers and, two, analysed only trace-based properties of systems. To this end, we will look into embedding different security semantics into our methodology and thus move

towards the verification of privacy and anonymity properties of programs with a security bearing (e.g., reference implementations of cryptographically rich e-voting protocols).

References

[Armando *et al.*, 2014] A. Armando, R. Carbone, and L. Compagna. SATMC: A SAT-based model checker for security-critical systems. In *Proc. of TACAS-20*, pages 31–45. Springer, 2014.

[Balliu *et al.*, 2012] M. Balliu, M. Dam, and G. Le Guernic. ENCoVer: Symbolic exploration for information flow security. In *Proc. of CSF-25*, pages 30–44, 2012.

[Baukus and van der Meyden, 2004] K. Baukus and R. van der Meyden. A knowledge based analysis of cache coherence. In *Proc. of ICFEM-6*, pages 99–114. Springer, 2004.

[Belardinelli *et al.*, 2016] F. Belardinelli, A. Lomuscio, and J. Michaliszyn. Agent-based refinement for predicate abstraction of multi-agent systems. In *Proc. of ECAI-22*, pages 286–294. IOS Press, 2016.

[Boureanu *et al.*, 2009] I. Boureanu, M. Cohen, and A. Lomuscio. Automatic verification of temporal-epistemic properties of cryptographic protocols. *Journal of Applied Non-Classical Logics*, 19(4):463–487, 2009.

[Brotherston *et al.*, 2012] J. Brotherston, N. Gorogiannis, and R.L. Petersen. A generic cyclic theorem prover. In *APLAS-10*, volume 7705, pages 350–367. Springer, 2012.

[Chaum, 1988] D. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1):65–75, 1988.

[Cimatti *et al.*, 2016] A. Cimatti, M. Gario, and S. Tonetta. A lazy approach to temporal epistemic logic model checking. In *Proc. of AAMAS-38*, pages 1218–1226. IFAAMAS, 2016.

[Cook *et al.*, 2012] B. Cook, E. Koskinen, and M. Vardi. Temporal property verification as a program analysis task. *Formal Methods in System Design*, 41(1):66–82, 2012.

[Cousot and Cousot, 1977] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL-4*, pages 238–252. ACM, 1977.

[De Moura and Bjørner, 2008] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of TACAS-14*, pages 337–340. Springer-Verlag, 2008.

[Dijkstra, 1976] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[Ezekiel *et al.*, 2011] J. Ezekiel, A. Lomuscio, L. Molnar, S. Veres, and M. Pebody. Verifying fault tolerance and self-diagnosability of an autonomous underwater vehicle. In *Proc. of IJCAI-22*, pages 1659–1664. AAAI Press, 2011.

[Fagin *et al.*, 1995] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about knowledge*. MIT press, 1995.

[Gammie and van der Meyden, 2004] P. Gammie and R. van der Meyden. MCK: Model checking the logic of knowledge. In *Proc. of CAV-16*, pages 479–483. Springer, 2004.

[Gelfond, 1994] M. Gelfond. Logic programming and reasoning with incomplete information. *Annals of Mathematics and Artificial Intelligence*, 12(1):89–116, 1994.

[Haase, 2014] C. Haase. Subclasses of presburger arithmetic and the weak EXP hierarchy. In *Proc. of LICS-23*, pages 47:1–47:10. ACM, 2014.

[Kacprzak *et al.*, 2006] M. Kacprzak, A. Lomuscio, A. Niewiadomski, W. Penczek, F. Raimondi, and M. Szreter. Comparing BDD and SAT based techniques for model checking Chaum's dining cryptographers protocol. *Fundamenta Informaticae*, 72(1-3):215–234, 2006.

[Kacprzak *et al.*, 2008] M. Kacprzak, W. Nabiałek, A. Niewiadomski, W. Penczek, A. Półrola, M. Szreter, B. Woźna, and A. Zbrzezny. VerICS 2007 – a model checker for knowledge and real-time. *Fundamenta Informaticae*, 85(1-4):313–328, 2008.

[Kouvaros and Lomuscio, 2016] P. Kouvaros and A. Lomuscio. Parameterised verification for multi-agent systems. *Artificial Intelligence*, 234:152–189, 2016.

[Lomuscio *et al.*, 2015] A. Lomuscio, H. Qu, and F. Raimondi. MCMAS: an open-source model checker for the verification of multi-agent systems. *International Journal on Software Tools for Technology Transfer*, 19(1):9–30, 2015.

[Plaza, 2007] J. Plaza. Logics of public communications. *Synthese*, 158(2):165–179, 2007.

[Păsăreanu and Rungta, 2010] Corina S. Păsăreanu and Neha Rungta. Symbolic PathFinder: Symbolic execution of java bytecode. In *Proc. of ASE'10*, pages 179–180. ACM, 2010.

[Rivest and Smith, 2007] R. L. Rivest and W. D. Smith. Three Voting Protocols: ThreeBallot, VAV, and Twin. In *Proc. of EVT'07*, pages 16–16. USENIX, 2007.

[Rosenschein and Kaelbling, 1986] S. J. Rosenschein and L. P. Kaelbling. The synthesis of digital machines with provable epistemic properties. In *Proc. of TARK'86*, pages 83–98. Morgan Kaufmann, 1986.

[Stockmeyer, 1977] L. J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3:1–22, 1977.

[Wang, 2016] S. Wang. Dynamic epistemic model checking with Yices. `https://airobert.github.io/FSA_report.pdf`, 2016. Accessed 14/02/2017.

[Zbrzezny *et al.*, 2016] A. M. Zbrzezny, A. Zbrzezny, and F. Raimondi. Efficient model checking timed and weighted interpreted systems using SMT and SAT solvers. In *Proc. of KES-AMSTA-10*, pages 45–55. Springer, 2016.

[Zhang, 2006] Y. Zhang. Computational properties of epistemic logic programs. In *Proc. of KR-10*, pages 308–317. AAAI Press, 2006.