Automatically Verifying Expressive Epistemic Properties of Programs

F. Belardinelli[1] I. Boureanu[2] V. Malvone[3] S. F. Rajaona[2]

[1]Imperial College London,
[2]Surrey Centre for Cyber Security, University of Surrey,
[3]Télécom Paris,
francesco.belardinelli@imperial.ac.uk, {i.boureanu, s.rajaona}@surrey.ac.uk,
vadim.malvone@telecom-paris.fr

Abstract. We propose a new approach to the verification of epistemic properties of programs. First, we introduce the new "program-epistemic" logic $\mathcal{L}_{P\mathsf{K}}$, which is strictly richer and more general than similar formalisms appearing in the literature. To solve the verification problem in an efficient way, we introduce a translation from our language $\mathcal{L}_{P\mathsf{K}}$ into first-order logic. Then, we show and prove correct a reduction from the model checking problem for program-epistemic formulas to the satisfiability of their first-order translation. Both our logic and our translation can handle richer specification w.r.t. the state of the art, allowing us to express the knowledge of agents *about* facts pertaining to programs (i.e., agents' knowledge before and after a program is executed). Furthermore, we implement our translation in Haskell in a general way (i.e., independently of the programs in the logical statements), and we use existing SMT-solvers to check satisfaction of $\mathcal{L}_{\mathsf{PK}}$ formulas on a benchmark example in the AI/agency field.

1 Introduction

Explainability in learning, in security, etc [29] is at the forefront of AI. In this vein, we aim to verify knowledge beyond systems and onto code: e.g., to formally answer what programs know, and what an observer knows about a program modulo what is not private to them. Our goal is to no longer "just" model check knowledge of systems, but rather do knowledge verification of programs or reason about programs' knowledge. Moreover, if we could do so by leveraging the benefits of program-verification being often reduced to SMT-solving, this would be ideal.

But, whilst reductions of general verification to SMT-solving are often done and prove worthwhile for Hoare logics or program-intrinsic logics (such as separation logic [2]), it is not always clear if it is possible to give such reductions for high-level, non-classical logics defined on top of programs, such as linear dynamic logic [6]. Furthermore, this is even less so the case for logics of knowledge, or a sound mix of knowledge with code-driven logics. We come to fill this gap.

To this end, in 2017, [15] promisingly introduced a "*bespoke*" epistemic logic for programs, denoted $\mathcal{L}_{\Box\mathsf{K}}$. The programs/command C in [15] could be seen as formed of multiple threads/agents and each thread i would have observable and non-observable variables w.r.t. the program space. Then, a formula $\Box_C K_i \varphi$ in $\mathcal{L}_{\Box\mathsf{K}}$ denotes the thread/agent

i knowing the state of affairs φ at the end of executing program C. Under given conditions (e.g., set of program-instructions, variable domain, mathematical behaviour of program transformer), [15] proved that the model checking problem for $\mathcal{L}_{\Box\mathsf{K}}$ can be reduced to SMT-solving. Whilst interesting, there are several limitations to [15]:

a. The translation could handle reasoning about knowledge only at states at the end of the execution of a given program, i.e., formulas $\Box_C K_i\varphi$ were handled by the translation, whereas formulas of type $K_i\Box_C\varphi$ were not.
b. The logic $\mathcal{L}_{\Box\mathsf{K}}$ only allowed to expressing aspects like "at the end of executing program C, a formula holds", where the programs were not part of the logic, but rather ad-hoc instructions defined outside of the logic.
c. The logic was "bespoke" in that, predicate transformers were necessary to be defined w.r.t. the programs in the logic and the transition relation of the logic be linked to these predicate transforms. To this end, the reduction of model checking $\mathcal{L}_{\Box\mathsf{K}}$ to SMT was also dependent on these predicate transformers. However, no formal characterisation was given for these, and so we cannot ascertain if their program semantics is therefore "standard".

We overcome the limitations (a)-(c) above, and produce the first approach that allows verifying knowledge over program without totally foregoing the temporal aspects and by giving a logic and translation that is program-semantics independent and more general, in the sense of being "free" of dependencies on "bespoke" predicate transformers.

Contributions . Our contribution is threefold. First, we introduce a new logic to reason about agents' knowledge on program/command execution, in ways richer that the state-of-the-art. Concretely, we define a *program-epistemic logic* $\mathcal{L}_{\mathsf{PK}}$ that is strictly more expressive than the program-epistemic logic $\mathcal{L}_{\Box\mathsf{K}}$ in [15]: i.e., in $\mathcal{L}_{\mathsf{PK}}$, the epistemic and the knowledge operators can commute. Moreover, our program-epistemic logic $\mathcal{L}_{\mathsf{PK}}$ is more general than the logic in [15]: our relational semantics is not dependent on programs' predicate transformers, and our programs are fully mapped to logic operators. In that sense, our logic $\mathcal{L}_{\mathsf{PK}}$ can be seen as an extension of star-free linear dynamic logic (LDL) [7] with epistemic operators, or equivalently dynamic logic (DL) [18] extended with an epistemic operator. A nice consequence is that, like in DL [18] and unlike in $\mathcal{L}_{\Box\mathsf{K}}$ [15], in our program-epistemic logic $\mathcal{L}_{\mathsf{PK}}$, the non-atomic program operators for sequential composition and non-deterministic choice are respectively obtained by applying atomic program operators in sequence, and by using the OR logic connective over atomic program operators.

Second, for our logic, we show a *totally new* mechanism of translating its formulas to first-order logic, in such a way that we obtain that model checking $\mathcal{L}_{\mathsf{PK}}$ reduces to first-order satisfaction. Indeed, because our logic is more aligned to "standard" logics (such as linear dynamic logic – LDL [7] and dynamic logic – DL [18]), our translation is entirely recursive, without the need to leverage special cases separately and/or Hoare-style predicate transformers. In this translation and reduction result, we include formulas that [15] could not treat, i.e., $K_a[program]\varphi$ – expressing that agent a knows fact φ about the execution of "$program$".

Third, we mechanise our translation in Haskell and experiment with SMT-solving being called to answer w.r.t. satisfaction of $\mathcal{L}_{\text{PK}}$ formulas. We mechanise our translation in a general way (i.e., not for a given bespoke program as per [15], but for the whole logic). This is possible also because our logic itself "builds" the program operators within. We report on the experiments on one "home-made" relevant use-case, as well as the canonical example of the dining cryptographers, used in [15] and in most epistemic model checking benchmarks.

2 Program-Epistemic Logic

In this section we define an epistemic extension of dynamic logic [18], in particular strictly more expressive[1] than the language in [15].

Agents & Program Variables. We assume that *agents* (or *threads*) in set Ag have access to a countable set V of *variables*, that are modified concurrently by them and/or an outer *program*. Variables may belong the one of the following finite sets:

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

We use $\vec{o}$ and $\vec{n}$ for observable and non-observable variables in general, i.e., unindexed by a given agent. We use $\vec{x} = \langle x_1, \ldots, x_n \rangle$ to denote both the vector and the set of variables $x_1, \ldots, x_n$, according to some enumeration, as clear from the context. Finally, both $\vec{x}_i$ and $\vec{x}(i)$ denote the i-th element in vector $\vec{x}$.

2.1 Languages – Syntax

As in [15], we assume a *base language* $\mathcal{L}_{\text{QF}}$ to be a quantifier-free, first-order language including identity '$=$', whose variables are user- or domain-specific, i.e., integers, reals, etc. The particular choice of $\mathcal{L}_{\text{QF}}$ is left under-specified here, but we assume that $\mathcal{L}_{\text{QF}}$ is decidable.

We now define two languages based on $\mathcal{L}_{\text{QF}}$.

The first-order language $\mathcal{L}_{FO}$ is the extension of $\mathcal{L}_{\text{QF}}$ with quantification.

Definition 1 ($\mathcal{L}_{\text{FO}}$). *Formulas ϕ in $\mathcal{L}_{FO}$ are defined in BNF as follows, where $\pi \in \mathcal{L}_{QF}$ and $x \in V$:*

$$\phi ::= \pi \mid \neg\phi \mid \phi \wedge \phi \mid \forall x \phi$$

We extend quantification from variables to vectors as usual, writing $\forall \vec{x} \phi$ for $\forall x_1 \ldots \forall x_n \phi$. Further, we can introduce Boolean operators $\vee$, $\rightarrow$, $\leftrightarrow$, and the existential quantifier $\exists$ as standard. Moreover, the simultaneous substitution of variables $\vec{x}$ with expressions $\vec{e}$ in formula ϕ (provided that all $\vec{e}$s are free for $\vec{x}$s in ϕ and $|\vec{x}| = |\vec{e}| = m$) is denoted as $\phi[x_1/e_1, \ldots, x_m/e_m]$, or $\phi[\vec{x}/\vec{e}]$ in short.

[1] This is directly from the fact that the program operators and the knowledge operator can commute in our language, whereas in [15] they cannot. The programs are also more general herein, but this is secondary.

Our program-epistemic language $\mathcal{L}_{\mathsf{PK}}$ extends the base language $\mathcal{L}_{\mathrm{QF}}$ with the *epistemic modality* K_a, expressing the knowledge of agent a, as well as the *program operator* $[\rho]$ of dynamic logic [18], for some *program* ρ.

Definition 2 ($\mathcal{L}_{\mathsf{PK}}$)**.** *Formulas* α *and programs* ρ *in* $\mathcal{L}_{\mathsf{PK}}$ *are defined in BNF as follows:*

$$\alpha ::= \pi \mid \neg\alpha \mid \alpha \wedge \alpha \mid (\mathsf{K}_a\alpha)[\vec{x}/\vec{e}] \mid [\rho]\alpha$$
$$\rho ::= x := e \mid \phi?$$

where $\pi \in \mathcal{L}_{QF}$*,* a *is a fixed agent in* Ag *(i.e., we do not allow multi-agent nesting of epistemic modalities),* $\vec{e}$ *are domain-specific expressions over program variables* V *up to computable and non-recursive mathematical function,* $\phi \in \mathcal{L}_{FO}$ *is a first-order formula over* V*.*

Note that we indicate substitutions $[\vec{x}/\vec{e}]$ explicitly in epistemic formulas of type $(\mathsf{K}_a\alpha)[\vec{x}/\vec{e}]$. This is because – as we will see shortly – formulas $(\mathsf{K}_a\alpha)[\vec{x}/\vec{e}]$ and $\mathsf{K}_a(\alpha[\vec{x}/\vec{e}])$ are not equivalent in our semantics. That is, substitution does not commute with epistemic operators in general, differently from what happens with Boolean operators. For instance, even though Venus is the morning star, some agent a might not know that it is also the evening start. As a result, $(\mathsf{K}_a(Venus = morn\_st))[Venus/even\_st]$ can be true in our semantics, but $\mathsf{K}_a((Venus = morn\_st)[Venus/even\_st]) = \mathsf{K}_a(morn\_st = even\_st)$ might still be false [23].

Moreover, we write $\mathsf{K}_a\alpha$ as a shorthand for $(\mathsf{K}_a\alpha)[x/x]$, where $[x/x]$ is the identity substitution,

Derived dynamic operators. Given the syntax in Def. 2, we can introduce the arbitrary assignment $x := *$ of dynamic logic, as the following shorthand, provided that the domain D of interpretation is finite:

$$[x := *]\alpha ::= \bigwedge_{c \in D} [x := c]\alpha$$

Further, dynamic operators for sequential composition ';' and non-deterministic choice $\sqcup$ can be introduced as the following abbreviations, which are standard in dynamic logic [18]:

$$[\rho; \rho']\,\alpha ::= [\rho]\,[\rho']\,\alpha$$
$$[\rho \sqcup \rho']\,\alpha ::= [\rho]\,\alpha \vee [\rho']\,\alpha$$

So, hereafter we will make use of language $\mathcal{L}_{\mathsf{PK}}$ with all the derived operators described above, simply intended as the corresponding shorthands.

We write $FV(\Phi)$ for the set of free variables of a formula Φ. We normally use Greek letters $\phi, \psi, \ldots$, and $\alpha, \beta, \ldots$ to denote formulas in the first-order language $\mathcal{L}_{\mathrm{FO}}$ and program-epistemic formulas in $\mathcal{L}_{\mathsf{PK}}$, respectively. We refer to the fragment of $\mathcal{L}_{\mathsf{PK}}$ without program operator as the *epistemic language* $\mathcal{L}_{\mathsf{K}}$. Finally, for a tuple $\vec{x}$ of variables and a tuple $\vec{e}$ of terms such that $|\vec{x}| = |\vec{e}|$, we write $\bigwedge(\vec{x} = \vec{e})$ as a shorthand for $\bigwedge_{i \leq |\vec{x}|}(x_i = e_i)$.

Program Operators $[\rho]$ vs Programs ρ. In $\mathcal{L}_{\text{PK}}$ we consider program operators $[\rho]$ of dynamic logic. Each such operator $[\rho]$ corresponds to a *program* ρ over variables, as implied by the syntax of $\mathcal{L}_{\text{PK}}$. Hereafter we use "program-operator" and "program" interchangeably, when the context allows it.

Our program-operators $[\rho]$ applied at a state s can do: "$x := e$", i.e., assign the value of expression e to variable x; "$\phi?$", i.e., check the truth of first-order condition ϕ; "$x := *$", i.e., assign an arbitrary value to variable x; "$\rho; \rho'$", i.e., compose sequentially programs ρ and ρ'; "$\rho \sqcup \rho'$", i.e., choose non-deterministically to execute either program ρ or program ρ'. We remark that our program operators are interpreted similarly to operators in Linear Dynamic Logic (LDL) [14]; however, we do not consider the full expressivity of LDL, i.e., a program operator may not necessarily be a full regular expression, as we explicitly do not consider the Kleene star in our syntax.

2.2 Languages – Semantics

We now provide the semantics for the languages introduced in the previous section. First, let D be the *domain* of interpretation for variables and quantifiers. Then, a *valuation* is a total function $s : V \to D$, naturally lifted to tuples $\vec{x}$ and $\vec{e}$. Such a valuation is a *state*. We write $s[x \mapsto c]$ to denote the state s' that leaves s unchanged apart from assigning variable $x \in V$ to element $c \in D$, i.e., $s'(x) = c$ and $s'(y) = s(y)$ for all $y \in V$ different from x. Let U be the set of all such states.

Given the definition of state s as above, s is in particular an interpretation of $\mathcal{L}_{\text{QF}}$.

Definition 3 (Semantics of $\mathcal{L}_{\text{FO}}$). *Given a state $s \in U$ and formula $\phi \in \mathcal{L}_{FO}$, we define the* satisfaction relation $\models$ *for $\mathcal{L}_{FO}$ inductively as follows:*

$$
\begin{array}{lll}
s \models \pi & \textit{iff} & s \models_{QF} \pi \\
s \models \neg\phi & \textit{iff} & s \not\models \phi \\
s \models \phi \wedge \phi' & \textit{iff} & s \models \phi \textit{ and } s \models \phi' \\
s \models \forall x \phi & \textit{iff} & \textit{for all } c \in D,\ s[x \mapsto c] \models \phi
\end{array}
$$

where $\models_{QF}$ is the underlying satisfaction relation for $\mathcal{L}_{QF}$.

We remark that Def. 3 of satisfaction for $\mathcal{L}_{\text{FO}}$ is completely standard [25].

To introduce the semantics of $\mathcal{L}_{\text{PK}}$ we need a few more notions, starting with a relation of indistinguishability for the interpretation of epistemic operators.

Definition 4 (Indistinguishability). *Let $X \subseteq V$ be a set of variables. The* indistinguishability relation $\sim_X$ *is a binary relation over U, defined as $s \sim_X s'$ iff for all $x \in X$, $s(x) = s'(x)$. Clearly, $\sim_X$ is an equivalence relation over U, for any $X \subseteq V$.*

Further, to define the state updates entailed by program operators, we define a family of binary relations. Namely, for each program ρ, let $R_\rho \subseteq U \times U$ be a binary relation representing the *transition relation* induced by ρ, considered at a state s. Intuitively, $R_\rho(s, s')$ denotes that we can reach state s' from s via program ρ. We can then naturally lift R_ρ to a function from states to sets of states, as well as a function from sets of states to sets of states, as follows, where $s \in U$ and $S \subseteq U$:

$$
\begin{aligned}
R_\rho(s) &= \{s' \in U \mid R_\rho(s, s')\} \\
R_\rho(S) &= \textstyle\bigcup_{s \in S} R_\rho(s)
\end{aligned}
$$

Definition 5 (Semantics of $\mathcal{L}_{\mathsf{PK}}$). *Given a state $s \in W \subseteq U$, and formula $\alpha \in \mathcal{L}_{\mathsf{PK}}$, we define the* satisfaction relation $\models$ *for $\mathcal{L}_{\mathsf{PK}}$ inductively as follows.*

$(W, s) \models \pi$ *iff* $s \models_{QF} \pi$
$(W, s) \models \neg\alpha$ *iff* $(W, s) \not\models \alpha$
$(W, s) \models \alpha \wedge \alpha'$ *iff* $(W, s) \models \alpha$ *and* $(W, s) \models \alpha'$
$(W, s) \models (K_a\alpha)[\vec{x}/\vec{e}]$ *iff for all* $s' \in W$,
$s' \sim_{\vec{o}_a} s[\vec{x} \mapsto s(\vec{e})]$ *implies*
$(W, s') \models \alpha$
$(W, s) \models [\rho]\alpha$ *iff for all* $s' \in R_\rho(s)$, $(R_\rho(W), s') \models \alpha$

where the relation R_ρ is inductively defined as follows:

$R_{x:=e}(s) = \{s[x \mapsto s(e)]\}$;
$R_{\phi?}(s) = \{s\}$ *if* $s \models \phi$, *and* $\emptyset$ *otherwise.*

As a consequence of Def. 5, we have the following clauses for dynamic operators:

$(W, s) \models [x := e]\alpha$ iff $(R_{x:=e}(W), s[x \mapsto e]) \models \alpha$
$(W, s) \models [\phi?]\alpha$ iff $(R_{\phi?}(W), s) \models \phi \to \alpha$

Notice that we use the same symbol $\models$ for the satisfaction relations for of both language $\mathcal{L}_{\mathsf{FO}}$ and $\mathcal{L}_{\mathsf{PK}}$; the context will disambiguate. Moreover, in the derived truth clause for $[\phi?]\alpha$, the expression $\phi \to \alpha$ is not a formula in $\mathcal{L}_{\mathsf{PK}}$ strictly speaking, as ϕ is a generic first-order formula in $\mathcal{L}_{\mathsf{FO}}$. Nonetheless, we are able to interpret formulas of type $[\phi?]\alpha$ by using the corresponding clause in Def. 5. Hereafter we use $\vec{x} := \vec{e}$ to denote the simultaneous assignment of expressions $\vec{e}$ to variables $\vec{x}$.

In $\mathcal{L}_{\mathsf{PK}}$, we can write formulas such as $[x := e]\alpha$ to be evaluated at a state s. By Def. 5, this means that we first evaluate the expression e over the variables at state s, then assign the result to x, and finally check if α holds at the updated state $s[x \mapsto s(e)]$. In particular, if $\alpha = \mathsf{K}_a\alpha'$, we evaluate α' in all indistinguishable states s' that also assign value $s(e) \in D$ to variable x, whenever x is observable by agent a. As a result, formulas $[x := e](\mathsf{K}_a\alpha')$ and $(\mathsf{K}_a\alpha')[x/e]$ are not equivalent in our semantics, with the latter amounting to a *de re* interpretation of the epistemic modality [12], whereby formula $\mathsf{K}_a\alpha'$ is true for individuals $\vec{e}$ at state s. On the other hand, formulas $\mathsf{K}_a(\alpha'[x/e])$ and $(\mathsf{K}_a\alpha')[x/e]$ are not equivalent in general, as the former expresses a *de dicto* reading of the epistemic modality, whereby expressions $\vec{e}$ might denote different individuals in different indistinguishable states. This feature of the semantics motivates the explicit notation of substitutions in epistemic formulas.

We now state the model checking problem for $\mathcal{L}_{\mathsf{PK}}$.

Definition 6 (Model Checking). *Given a set $W \subseteq U$ of states, a state $s \in W$, and a formula $\alpha \in \mathcal{L}_{\mathsf{PK}}$, the model checking problem amounts to determining whether $(W, s) \models \alpha$.*

3 Translation into First-Order Logic

In this section, we show how program-epistemic formulas in $\mathcal{L}_{\mathsf{PK}}$ can be translated into first-order formulas in $\mathcal{L}_{\mathsf{FO}}$. The latter can then be fed into an SMT solver. The translation will be recursive on the structure of a formula $\alpha \in \mathcal{L}_{\mathsf{PK}}$.

Satisfaction Objects. To be able to define the translation, we first introduce some notation to denote states where a formula is satisfied. We generically call these sub-parts of the state-space *satisfaction objects* $[\![\beta]\!]$ of a formula β.

For formulas ϕ in $\mathcal{L}_{\text{FO}}$, the corresponding *satisfaction object* $[\![\phi]\!]_{\mathcal{L}_{FO}}$ is indeed the standard notion $[\![\phi]\!]$, that is, the set of states satisfying ϕ:

$$[\![\phi]\!]_{\mathcal{L}_{\text{FO}}} = \{s \in U \mid s \models \phi\}.$$

The *satisfaction object* $[\![\alpha]\!]_{\mathcal{L}_{\text{PK}}}$ for formulas α in $\mathcal{L}_{\text{PK}}$ is the set in $\wp(\wp(U))$ defined as follows:

$$[\![\alpha]\!]_{\mathcal{L}_{\text{PK}}} = \{W \in \wp(U) \mid \text{for all } s \in W, (W, s) \models \alpha\}$$

Programs. In $\mathcal{L}_{\text{PK}}$ we defined program operators $[\rho]$. Abstracting away the semantics for now, we consider the *program* ρ over variables as implied by the syntax in $\mathcal{L}_{\text{PK}}$. The *set of all epistemic programs* is denoted as $\{\rho\}$.

All programs update states, by changing an input into an output. Classically, a non-deterministic program C is modelled as a set-valued function $f_C : U \to \wp(U)$, stating that a state can be updated into one of a series of possible states. Our programs lift this representation uniformly from states to set of states. So, a program ρ can be represented as functions $f_\rho : \wp(U) \to \wp(\wp(U))$. This denotes that program ρ takes as input a set of states and this set can be transformed into one of a series of sets of states.

Now, we have all the elements to present our translation.

Definition 7 (Translation τ). *The translation $\tau : \mathcal{L}_{FO} \times \mathcal{L}_{\text{PK}} \to \mathcal{L}_{FO}$ is such that for $\phi \in \mathcal{L}_{FO}$ and $\alpha \in \mathcal{L}_{\text{PK}}$, $\tau(\phi, \alpha)$ is inductively defined as follows:*

$$\begin{aligned}
&\tau(\phi, \pi) && = \pi \\
&\tau(\phi, \neg\alpha) && = \neg\tau(\phi, \alpha) \\
&\tau(\phi, \alpha_1 \wedge \alpha_2) && = \tau(\phi, \alpha_1) \wedge \tau(\phi, \alpha_2) \\
&\tau(\phi, (\mathsf{K}_a\alpha)[\vec{x}/\vec{e}]) && = \forall\vec{k}(\bigwedge(\vec{k} = \vec{e}) \to \\
& && \quad \to \forall\mathbf{n}'_a(\phi[\vec{x}/\vec{k}] \to \tau(\phi[\vec{x}/\vec{k}], \alpha[\vec{x}/\vec{k}]))) \\
&\tau(\phi, [x := e]\alpha) && = \tau(\exists y((x = e[x/y]) \wedge \phi[x/y]), \alpha[x/e]) \\
&\tau(\phi, [\psi?]\alpha) && = \tau(\psi \wedge \phi, \psi \to \alpha)
\end{aligned}$$

where $\vec{k}$ is a tuple of new variables not appearing in α, ϕ, and for every $i \leq |\vec{n}'_a|$, $\vec{n}'_a(i) = \vec{k}(i)$ if $\vec{x}(i) = \vec{n}(i)$; otherwise $\vec{n}'_a(i) = \vec{n}_a(i)$.

The clause for translation τ for the base case $\alpha = \pi$ is immediate, as π itself is returned, as it is already a first-order formula by the syntax of $\mathcal{L}_{\text{FO}}$ in Def. 1. Then, translation τ commutes with Boolean operators. The case of epistemic operators is more complex. Intuitively, the values of expressions $\vec{e}$ are assigned to variables $\vec{k}$. Then, quantification on non-observable variables $\vec{n}'_a$ is applied to mimic the fact that operator K_a ranges over all states that are observationally indistinguishable for agent a from the current state. Moreover, variables $\vec{x}$ occurring in ϕ, α are replaced by $\vec{k}$, and therefore

if some of the $\vec{x}$ are equal to some of the $\vec{n}_a$, then the latter have to be replaced by the corresponding $\vec{k}$. This is basically the meaning of $\vec{n}'_a$. Finally, as regards the dynamic operators, atomic assignments $x := e$ are translated as substitutions, in such a way that if the expression e contains bindings over x, then these binding are evaluated first and they are consistently carried forward in the satisfaction object ϕ via substitutions; this is the purpose of $\exists y$ in $\exists y(\ldots \phi[x/y])$. This is intuitively in line with the interpretation of strongest-postconditions for assignment in programming languages. Tests $\psi?$ simply become implications. Again, we observe that, strictly speaking, $\psi \to \alpha$ is not a formula in $\mathcal{L}_{\mathsf{PK}}$, but our translation τ can take care of these formulas as well, by translating first-order formula ψ simply as itself.

Next, we prove a lemma, which will be used to show the main theorem. This lemma is simply saying what the relation translations for the assignment and test programs are, respectively, in our logic semantics. It is easy to see that these expressions are natural and we prove these rather for preciseness.

Lemma 1. *For every $\phi \in \mathcal{L}_{FO}$, we have that*

$$R_{\vec{x}:=\vec{e}}([\![\phi]\!]) = [\![\exists \vec{y}(\bigwedge(\vec{x} = \vec{e}[\vec{x}/\vec{y}]) \wedge \phi[\vec{x}/\vec{y}])]\!]$$
$$R_{\psi?}([\![\phi]\!]) = [\![\phi \wedge \psi]\!]$$

Proof. For $\rho = \vec{x} := \vec{e}$, we have that $s' \in R_{\vec{x}:=\vec{e}}([\![\phi]\!])$ iff for some $s \in [\![\phi]\!]$, $s' = s[\vec{x} \mapsto s(\vec{e})]$. Consider assignment $s'' = s'[\vec{y} \mapsto s(\vec{x})] = s[\vec{y} \mapsto s(\vec{x}), \vec{x} \mapsto s(\vec{e})]$. Since $s \models \phi$ and $s''(\vec{y}) = s(\vec{x})$, we obtain that $s'' \models \phi[\vec{x}/\vec{y}]$. Moreover, $s''(\vec{x}) = s(\vec{e}) = s''(\vec{e}[\vec{x}/\vec{y}])$, as s'' coincides with s on all variables different from $\vec{x}, \vec{y}$. Hence, $s'' \models \bigwedge(\vec{x} = \vec{e}[\vec{x}/\vec{y}])$. As a result, $s'' \models \bigwedge(\vec{x} = \vec{e}[\vec{x}/\vec{y}]) \wedge \phi[\vec{x}/\vec{y}]$, that is, $s' \in [\![\exists \vec{y}(\bigwedge(\vec{x} = \vec{e}[\vec{x}/\vec{y}]) \wedge \phi[\vec{x}/\vec{y}])]\!]$

For $\rho = \psi?$, we have that $s \in R_{\psi?}([\![\phi]\!])$ iff for $s \in [\![\phi]\!]$, $s \models \psi$. In particular, then, $s \models \phi \wedge \psi$, that is, $s \in [\![\phi \wedge \psi]\!]$.

In the next theorem, we prove that our translation is correct, that is, it preserves satisfaction between $\mathcal{L}_{\mathsf{PK}}$ and $\mathcal{L}_{\mathsf{FO}}$.

Theorem 1. *For every $\phi \in \mathcal{L}_{FO}$, state $s \in [\![\phi]\!]$, and $\alpha \in \mathcal{L}_{PK}$ such that $FV(\phi) \cup FV(\alpha) \subseteq \vec{p}$, we have that*

$$([\![\phi]\!], s) \models \alpha \textit{ iff } s \models \tau(\phi, \alpha)$$

Proof. We prove the theorem by structural induction on α.

Base case $\alpha = \pi$: We prove that $([\![\phi]\!], s) \models \pi$ iff $s \models \tau(\phi, \pi) = \pi$, which follows immediately by Def. 5 of satisfaction.

Inductive cases.

Case $\alpha = \neg\alpha'$: We have that $([\![\phi]\!], s) \models \alpha$ iff $([\![\phi]\!], s) \not\models \alpha'$, iff by induction hypothesis $s \not\models \tau(\phi, \alpha')$, iff $s \models \neg\tau(\phi, \alpha') = \tau(\phi, \alpha)$.

Case $\alpha = \alpha_1 \wedge \alpha_2$: We have that $([\![\phi]\!], s) \models \alpha$ iff $([\![\phi]\!], s) \models \alpha_1$ and $([\![\phi]\!], s) \models \alpha_2$, iff by induction hypothesis $s \models \tau(\phi, \alpha_1)$ and $s \models \tau(\phi, \alpha_2)$, iff $s \models \tau(\phi, \alpha_1) \wedge \tau(\phi, \alpha_2) = \tau(\phi, \alpha)$.

Case $\alpha = (\mathsf{K}_a\alpha')[\vec{x}/\vec{e}]$: We need to prove that $([\![\phi]\!], s) \models (\mathsf{K}_a\alpha')[\vec{x}/\vec{e}]$ iff $s \models \tau(\phi, \alpha) = \forall\vec{k}(\bigwedge(\vec{k} = \vec{e}) \rightarrow \forall\mathbf{n'}_a(\phi[\vec{x}/\vec{k}] \rightarrow \tau(\phi[\vec{x}/\vec{k}], \alpha'[\vec{x}/\vec{k}])))$, where $\vec{k}$ is a tuple of new variables not appearing in α, ϕ, and for every $i \leq |\vec{n}'_a|$, $\vec{n}'_a(i) = \vec{k}(i)$ if $\vec{x}(i) = \vec{n}(i)$; otherwise $\vec{n}'_a(i) = \vec{n}_a(i)$.

$\Rightarrow$ Suppose by contraposition that $s \not\models \forall\vec{k}(\bigwedge(\vec{k} = \vec{e}) \rightarrow \forall\mathbf{n'}_a(\phi[\vec{x}/\vec{k}] \rightarrow \tau(\phi[\vec{x}/\vec{k}], \alpha'[\vec{x}/\vec{k}])))$. In particular, we have that

$$s[\vec{k} \mapsto s(\vec{e})] \not\models \forall\mathbf{n'}_a(\phi[\vec{x}/\vec{k}] \rightarrow \tau(\phi[\vec{x}/\vec{k}], \alpha'[\vec{x}/\vec{k}]))$$

That is, for some $\vec{c} \in D^{|\vec{n}'_a|}$,

$$s[\vec{k} \mapsto s(\vec{e}), \vec{n}'_a \mapsto \vec{c}] \models \phi[\vec{x}/\vec{k}] \wedge \neg\tau(\phi[\vec{x}/\vec{k}], \alpha'[\vec{x}/\vec{k}])$$

Hence, there is a state $s' = s[\vec{n}'_a \mapsto \vec{c}]$ such that for all $y \in V \setminus \vec{o}_a$,$s'(y) = s[\vec{n}_a \mapsto \vec{c}](y)$, and $s'[\vec{k} \mapsto s'(\vec{e})] \models \phi[\vec{x}/\vec{k}] \wedge \neg\tau(\phi[\vec{x}/\vec{k}], \alpha'[\vec{x}/\vec{k}])$. By Def. 4 and Lemma **??**(1), we obtain that $s' \sim_{\vec{o}_a} s[\vec{k} \mapsto s(\vec{e})]$.

Since $s'[\vec{k} \mapsto s'(e)] \models \phi[\vec{x}/\vec{k}] \wedge \neg\tau(\phi[\vec{x}/\vec{k}], \alpha'[\vec{x}/\vec{k}])$, we derive that $s'[\vec{k} \mapsto s'(e)] \in [\![\phi[\vec{x}/\vec{k}]]\!]$ and $([\![\phi[\vec{x}/\vec{k}]]\!], s'[\vec{k} \mapsto s'(e)]) \not\models \alpha'[\vec{x}/\vec{k}]$ by induction hypothesis. Finally, for $s'' = s'[\vec{x} \mapsto s'(\vec{k})]$, we have $s'' \in [\![\phi]\!]$ and $s'' \sim_{\vec{o}_a} s[\vec{x} \mapsto s(\vec{e})]$ is such that $([\![\phi]\!], s'') \not\models \alpha'$. But then $([\![\phi]\!], s) \not\models (\mathsf{K}_a\alpha')[\vec{x}/\vec{e}]$.

$\Leftarrow$ Suppose by contraposition that $([\![\phi]\!], s) \not\models (\mathsf{K}_a\alpha')[\vec{x}/\vec{e}]$. Hence, there exists a state $s' \in [\![\phi]\!]$ such that $s' \sim_{\vec{o}_a} s[\vec{x} \mapsto s(\vec{e})]$ and $([\![\phi]\!], s') \not\models \alpha'$. Now consider state $s'' = s'[\vec{k} \mapsto s'(\vec{x})]$. Since $s''(\vec{k}) = s'(\vec{x})$ and $s''(\vec{e}) = s'(\vec{e})$, we have that $s'' \in [\![\phi[\vec{x}/\vec{k}]]\!]$, $s'' \sim_{\vec{o}_a} s[\vec{k} \mapsto s(\vec{e})]$, and $([\![\phi[\vec{x}/\vec{k}]]\!], s'') \not\models \alpha'[\vec{x}/\vec{k}]$.

By induction hypothesis, we obtain that $s'' \not\models \phi[\vec{x}/\vec{k}] \rightarrow \tau(\phi[\vec{x}/\vec{k}], \alpha'[\vec{x}/\vec{k}])$. In particular, for all $y \in V \setminus \vec{o}_a$, $s''(y) = s[\vec{k} \mapsto s(\vec{e})](y)$, and therefore $s[\vec{k} \mapsto s(\vec{e})] \not\models \forall\vec{n}'_a(\phi[\vec{x}/\vec{k}] \rightarrow \tau(\phi[\vec{x}/\vec{k}], \alpha'[\vec{x}/\vec{k}]))$.

Finally, we derive that $s \not\models \forall\vec{k}(\bigwedge(\vec{k} = \vec{e}) \rightarrow \forall\vec{n}'_a(\phi[\vec{x}/\vec{k}] \rightarrow \tau(\phi[\vec{x}/\vec{k}], \alpha'[\vec{x}/\vec{k}]))) = \tau(\phi, \alpha)$.

Case $\alpha = [x := e]\alpha'$: We have that $([\![\phi]\!], s) \models \alpha$ iff $(R_{x:=e}([\![\phi]\!]), s[x \mapsto s(e)]) \models \alpha'$ by Def. 5. By Lemma 1, this is equivalent to $([\![\exists y((x = e[x/y]) \wedge \phi[x/y])]\!], s) \models \alpha'[x/e]$. By induction hypothesis we obtain that $s \models \tau(\exists y((x = e[x/y]) \wedge \phi[x/y]), \alpha'[x/e]) = \tau(\phi, \alpha)$.

Case $\alpha = [\psi?]\alpha'$: We have that $([\![\phi]\!], s) \models \alpha$ iff $(R_{\psi?}([\![\phi]\!]), s) \models \psi \rightarrow \alpha'$ by Def. 5. By Lemma 1, this is equivalent to $([\![\phi \wedge \psi]\!], s) \models \psi \rightarrow \alpha'$. By induction hypothesis we obtain that $s \models \tau(\phi \wedge \psi, \psi \rightarrow \alpha') = \tau(\phi, \alpha)$.

By Theorem 1 we can reduce the model checking problem for $\mathcal{L}_{\mathsf{PK}}$ to satisfaction in first-order logic.

On Our Translation.

We compare the translation above with other similar methodologies in the literature.

Our translation is entirely recursive and does not need to treat different parts of the logic distinctly, as [15] did.

To see better how our translation stands out, let us detail on the intuition and the main thrust of the translation. For that, imagine a satisfaction query $M \models \varphi$, with $\varphi \in \mathcal{L}_{\mathsf{PK}}$ and M a Kripke structure. To compute the relation $\models$, one would recursively produce sets $[\![\texttt{subformula}(\varphi)]\!]$ of states in M that satisfy sub-formulas of φ. To this end, intuitively, our translation $\tau(\phi, \varphi)$ of a modal $\mathcal{L}_{\mathsf{PK}}$ formula into a quantified FO formula keeps the evolving sets $[\![\texttt{subformula}(\varphi)]\!]$ in the formula ϕ. In absolute terms, this ϕ encapsulates the set of states in the model M which satisfy a FO logic formula ϕ, i.e., $[\![\phi]\!]$, which in turn equates to the set of states that satisfy a subformula of φ. Then, $\tau(\phi, \varphi)$ is recursively applied until all subformulae of φ are consumed. In what follows, we will refer to ϕ as the *"satisfaction context"* (rather than " satisfaction object", as it was called in Section 3).

There are two points of main interest in the translation. One is w.r.t. program assignments and the other w.r.t. to the treatment of the K operator.

Regarding program assignments $x := e$ (where e is an expression on program variables and x is a logic/program variable), our translation $\tau(\phi, [x := e]\alpha)$ resorts to logic substitutions $[x/e]$ that are applied to both the recursive "satisfaction context" ϕ and the $\mathcal{L}_{\mathsf{PK}}$ formula. So, program assignments $x := e$ are treated in our translation like in dynamic epistemic logics (DEL) [27], creating an "update of the model" via substitutions.

As for the translation of epistemic formulas $\mathsf{K}_a\alpha[x/e]$, the aforesaid substitutions are also used but have to carefully treat the dichotomy of observable vs. non-observable variables. That is, a variable x non-observable by agent a can be assigned to a value produced by expression e where this expression e may contain values of variables observable to a. This needs to be handled in such a way that the non-observability of x is not affected. Thus, our translation for epistemic formulae $\mathsf{K}_a\alpha[x/e]$ contains an implicit double quantification over the non-observable variables of agent a and a renaming of variables in the "satisfaction context" ϕ to make sure that the non-observability is not lost. All of these allow us to evaluate variable at states coherently over different "satisfaction contexts" and epistemic contexts.

4 Mechanisation

In this section, we primarily present the implementation of the translation in Section 3.

Our code mechanising the translation and examples encoded/tested in/with it is available at http://people.itcarlson.com/ioana/epistemic-program-verifier/src.

4.1 A Generic Mechanisation of $\mathcal{L}_{\mathsf{PK}}$ Verification

We mechanise, in `Haskell`, the verification problem "$M \models \alpha$" for a model M and a formula $\alpha \in \mathcal{L}_{\mathsf{PK}}$, using our main result in Theorem 1. To this end, we implement our translation τ in Section 3 and check satisfiability/validity of the first-order formulae. To check the satisfiability/validity of the first-order formula resulting from the translation, we use the Haskell library `SBV`. The `SBV`—SMT Based Verification [11]— is an `Haskell` library that allows to call an SMT solver in Haskell. `SBV` allows to use several SMT solvers, with Z3 as its default solver.

Our implementation takes as input the description of system that constitutes a program (and produces the program in program-operators terms), and as second input the problem "$\phi \models \alpha$". It then sets $\phi \in \mathcal{L}_{\text{FO}}$ as a constraint on the initial states of the program, and proceeds with the implementation of the translation Section 3, to finally call Z3 on final quantified first-order formula.

The implementation is generic, of the translation in Section 3, and not specific/particular to a given input example, as is the case in [15].

4.2 A Modular Mechanisation of $\mathcal{L}_{\text{PK}}$ Verification

Our implementation is modular. Our verifier contains three main modules.

***Logics* module** defines $\mathcal{L}_{\text{FO}}$, $\mathcal{L}_{\text{K}}$, and $\mathcal{L}_{\text{PK}}$, with the necessary syntax for boolean and numerical expressions. This module also defines the programming commands, necessary for dynamic formulas $[\rho]\alpha$.

***Translation* module** implements the translation τ as defined in Definition 7.

***ToSBV* module** takes the problem inputs: the variables, the constraint ϕ, and the property α, and transform the problem $\phi \models \alpha$ into a predicate in `SBV`. To do so, the module interprets expressions and formulas into `SBV`'s symbolic types.

An additional module *ToString* provides function that gives the string representation of all the objects (expressions, programs, formulas). The modules *Logics*, *Translation*, and *ToString* can be used (independently from the module *ToSBV*) to translate a modal satisfiability $\phi \models \alpha$ into the corresponding first order satisfiability $\phi \models \tau(\phi, \alpha)$ (by Theorem 1). The latter can be solved by an SMT-solver API, other than `SBV`, or by a Theorem Prover such as Vampire [22] or iProver [21].

4.3 Experimentation

This work lends itself to checking several canonical examples in epistemic verification and a quite few others. In the benchmarks for these types of systems [19, 20, 24], one de-facto example is the dining cryptographers example.

Dining Cryptographers. This system is described by n cryptographers dining round a table [4]. The cryptographers may have paid for the dinner or their employer (the NSA) may have done. They execute a protocol to reveal whether one of the cryptographers paid, but without revealing which one. Each pair of cryptographers sitting next to each other have an un-biased coin, which can be observed only by that pair. The pair tosses the coins. Each cryptographer announces the result of XORing three booleans: the two coins they see and the fact of them having paid for the dinner. The XOR of all announcements is proven to be equal to the disjunction of whether any agent paid.

We follow the notation in [15], and model this problem as follows.

The domain of variables is $\mathbb{B} = \{T, F\}$. The program variables are $\vec{p} = \{x\} \cup \{p_i, c_i \mid 0 \leq i < n\}$; 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 $i \in Ag$ are

$\vec{o}_i = \{x, p_i, c_i, c_{i+1 \bmod n}\}$, and $\vec{n}_i = \vec{p} \setminus \vec{o}_i$. We model the protocol by an assignment ρ:

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

In the above, $\oplus$ denotes the XOR operator. The program ρ is therefore the result of the tossing all the coins the pairs of cryptographers see.

We check the following formulae:

$$\begin{aligned}
\alpha_1 &= \neg p_0 \Rightarrow \left(K_0 \left(\textstyle\bigwedge_{i=1}^{n-1} \neg p_i\right) \vee \bigwedge_{i=1}^{n-1} \neg K_0 p_i\right), \\
\alpha_2 &= [\rho] \left(K_0 \left(x \Leftrightarrow \textstyle\bigvee_{i=0}^{n-1} p_i\right)\right), \\
\alpha_2' &= K_0 \left([\rho] \left(x \Leftrightarrow \textstyle\bigvee_{i=0}^{n-1} p_i\right)\right), \\
\alpha_3 &= K_0 p_1.
\end{aligned}$$

We use the same formulae α_1, α_2 and α_3 as defined in [15] to make meaningful comparison of our approach with theirs. We added the formula α_2', which, as far as we know, can be expressed only in our new framework. In particular, formula α_1 states that if cryptographer 0 has not paid then she knows that no cryptographer paid, or (in case a cryptographer paid) she does not know which one. Formula α_2 states that cryptographer 0 knows that x is true iff one of the cryptographers paid. Formula α_2' states cryptographer 0 knows that, at the end of the program execution, x is true iff one of the cryptographers paid, where x is the result of the coin tossing. Finally, formula α_3 states that cryptographer 0 knows that cryptographer 1 has paid.

	SAT by our translation			SAT by the translation in (Gorogiannis et al., 2017)			Model Checking in MCMAS (Lomuscio et al., 2015)		
Formula	result	time		result	time		result	time	
		$n=5$	$n=10$		$n=5$	$n=10$		$n=5$	$n=10$
$\neg\alpha_1$	unsat	0.07sec	70sec	unsat	0.03sec	0.1sec	unsat	0.17sec	0.18sec
$\neg\alpha_2$	unsat	0.03sec	7sec	unsat	0.02sec	0.1sec	unsat	0.10sec	0.12sec
$\neg\alpha_2'$	unsat	0.15sec	17sec	N/A	-	0.1sec	unsat	0.20sec	0.25sec
$\neg\alpha_3$	sat	0.04sec	7sec	sat	0.01sec	0.1sec	sat	0.10sec	0.12sec

Table 1: Performances on Verifying the Dining-cryptographers Problem

Note: To see more, please see our code, file `ExampleDiningCryptographer.hs`.

In Table 1, all the answers to the formulas α_1, α_2, α_2' and α_3 checked are as expected. In contrast to [15], we see that our framework allows for checking the formula α_2' as well.

To see the performance, we report the time it took to get the satisfaction results, for $n = 5$ and $n = 10$ cryptographers. In terms of speed performance, Table 1 shows a reduction of scalability in our approach compared to [15]. To give an example, on the respective formulations of the same formula α_2, for 10 cryptographers, our translation takes approx seven seconds to reply, whereas [15] answers less than one second. In general, from this table, we see that for $n = 5$ cryptographers our reported times compare more closely with in [15], but for $n = 10$ cryptographers our efficiency drops by a factor of 10, compared to [15]. Using the data in [15], we also see that we are

generally twice faster than the model checker MCMAS in checking the problem for $n = 5$ cryptographers, but again much slower for $n = 10$.

However, the depreciation in scalability w.r.t. [15] is to be expected. On the one hand, to be able to treat the K operator before the program operator (e.g., checking α'_2) our translation uses more complex satisfaction objects $[\![\phi]\!]$ and their update inside our translation happens in various points (i.e., see the translation for K and the two satisfaction objects manipulated therein). More on this aspect can be found in in the paragraph "On Efficiency". On the other hand, our current tool does not make use of the full power of the SMT solver. Our translation yielding alternating quantifiers, yet these are not supported by the `SBV`. To this end, we transformed quantifiers into conjunction and disjunctions. This was possible for our examples working the finite domains, but foregoing quantifiers in this way "kills" the optimisations of such SMT-solving algorithms.

We note that the translation in [15] is specific to the examples that they study, and they do not provide a generic automation neither for the translation, nor for transforming the translated formulas into the right form for the SMT solver. Our formalism could be implemented in a performant and generic tool by feeding the translated first-order formula into a dedicated theorem prover such as Vampire [22] or iProver [21].

A Simple Example Focused on Non-observability. Now, we use an easy-to-follow program to show how the translation behaves around non-observable variables. Our simple program is "$x := y$" such that "reveals" the value of a non-observable variable y, by assigning it into an observable variable x. Formally, we consider the domain $\mathbb{B} = \{T, F\}$, an agent ag, and the variables $x \in \mathbf{o}_{ag}, y \in \mathbf{n}_{ag}$.

Note:To see more, please see our code, file `ExampleRevelation.hs`.

We checked the satisfiability of the following formulae:

$$\alpha = [x := y](K_{ag}(x \Leftrightarrow T))$$
$$\alpha' = K_{ag}([x := y](x \Leftrightarrow T))$$

Our tool returned two satisfying states for α, namely the states where $y = T$. Meanwhile, for formula α', it answered correctly saying it is unsatisfiable.

Note that these results returned by the tool are in line with our semantics for these formulae. That is, for α, in our semantics, at the states after the assignment of x to $y == T$, the agent will know that x is equal to T; and there are two such states, $(x = T, y = T)$ and $(x = F, y = T)$, right before the assignment where α is evaluated and holds. However, for formula α', at no state will the agent know in our semantics that after x is attributed y, then x is equal to T; this is so since from any state s that one would consider to evaluate α' there exists a state s' indistinguishable by ag from s (since $y \in \mathbf{n}_{ag}$) where $\neg[x := y](x \Leftrightarrow T)$ holds.

Once again, this example and others like it were used by us to empirically test the soundness of our implementation.

On Efficiency. In [15], because they did not treat the case of K operators being evaluated before program operators, then their "satisfaction contexts" and epistemic contexts (i.e., set of states) were sufficient to give their main result. In turn, our main result (Th. 1) need an extra quantification: it quantifies over every state s in such a "satisfaction contexts"/model $[\![\phi]\!]$. Also, our translation of knowledge operators already

contains one extra quantification compared to [15] and inner variable renaming, to cater for similar reason (i.e., programmes changing variables "ahead" of an epistemic operator be evaluated). Thus, all in all, for a given formula containing one K operator to be translated, we may iterate four times more over the state-space than [15] had to. So, in practice, our ability to translate a richer logic to FO comes at a depreciation in efficiency (on average) compared to [15], yet we can solve the dinning cryptographers problems for 5 cryptographers ten times faster than the MCMAS model checker [24] can.

5 Related Work

On SMT-Based Verification of Epistemic Properties of Programs. With the work of Gorogiannis et al. [15], we compared in the introduction already, so now we only discuss other related lines. [26] verify epistemic properties of programs not via dynamic logic, but by reasoning with an ignorance-preserving refinement. Like here, their notion of knowledge is based on observability of arbitrary domain program variables. Also, this work has no relation with first-order satisfaction nor translations of validity of programme-epistemic logics to that, nor their implementation.

On Dynamic Epistemic Logics (DEL). DEL [9]) is a family of logics that extend epistemic logic with dynamic operators.

On the one hand, DEL logics are mostly propositional, and their extensions with assignment only considered propositional assignment (e.g., [8]); contrarily, we support assignment on variables on arbitrary domains. Also, we have a denotational semantics of programmes (via predicate transformers), whereas DEL operates on more abstract semantics. On the other, action models in DEL can describe complex private communications that cannot be encoded with our current programming language. The line on *semi-public environments* in DEL also builds indistinguishability relations from the observability of propositional variables [31, 3, 17]. [16] explores the interaction between knowledge dynamics and non-deterministic choice/sequential composition.

Current DEL model checkers include `DEMO` [10] and `SMCDEL` [28]. We are not aware of the verification of DEL fragments being reduced to satisfiability problems. An online report [30] discusses –at some high level– the translation `SMCDEL` knowledge structures into QBF and the use of `YICES`.

Other Works. [15] discussed work related more tenuously, such as on general verification of temporal-epistemic properties of systems which are not programs in tools like `MCMAS` [24], `MCK` [13], `VERICS` [20], or one line of epistemic verification of models specifically of `JAVA` programs [1]. [15] also discussed some incomplete method of SMT-based epistemic model checking [5], or even bounded model checking techniques, e.g., [19]. All those are only loosely related to us, so no reiteration needed.

Why This Methodology

The value of our methodology is in the AI-based theory, a well-founded combination of dynamic and epistemic logic in a way that can be used to systematically verify knowledge over programmes in a manner that was not possible before. This can be used for,

e.g., private-information flow verification or explaining why a decision was taken under partial information [29]. Even if our implementation is not yet efficient, we stress that this is a proof-of-concept that can be further optimised by us leveraging in the future the full power of the SMT-solver, without us foregoing quantifications into disjunctions/conjunctions.

6 Conclusions

We defined a rich program-epistemic logic (mixing a Kleene-star-free fragment of LDL [7] with knowledge operators) and showed that its model checking problem can be reduced to SMT-solving. Indeed, our translation from our epistemic-program logic to FO logic treats a richer and more generic logic than ever before, w.r.t. knowledge of programs. We implemented this translation and tested it against a number of use-cases.

Acknowledgements

I. Boureanu and S. Rajaona were partly supported by the EPSRC project "AutoPaSS", EP/S024565/1.

References

1. Balliu, M., Dam, M., Le Guernic, G.: ENCoVer: Symbolic exploration for information flow security. In: Proc. of CSF-25. pp. 30–44 (2012)
2. Botincan, M., Parkinson, M., Schulte, W.: Separation Logic Verification of C Programs with an SMT Solver. Electr. Notes Theor. Comput. Sci. **254**, 5–23 (10 2009). https://doi.org/10.1016/j.entcs.2009.09.057
3. Charrier, T., Herzig, A., Lorini, E., Maffre, F., Schwarzentruber, F.: Building epistemic logic from observations and public announcements. In: Fifteenth International Conference on the Principles of Knowledge Representation and Reasoning (2016)
4. Chaum, D.: The dining cryptographers problem: Unconditional sender and recipient untraceability. Journal of Cryptology **1**(1), 65–75 (1988)
5. Cimatti, A., Gario, M., Tonetta, S.: A lazy approach to temporal epistemic logic model checking. In: Proc. of AAMAS-38. pp. 1218–1226. IFAAMAS (2016)
6. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence. p. 854–860. IJCAI '13, AAAI Press (2013)
7. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence. pp. 854–860. IJCAI '13, AAAI Press (2013)
8. van Ditmarsch, H.P., van der Hoek, W., Kooi, B.P.: Dynamic epistemic logic with assignment. Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems - AAMAS '05 p. 141 (2005)
9. Ditmarsch, H.v., Hoek, W.v.d., Kooi, B.: Dynamic Epistemic Logic. Synthese Library, Springer (2007)
10. van Eijck, J.: A demo of epistemic modelling. Interactive Logic p. 303 (2007)
11. Erkök, L.: Sbv: Smt based verification in haskell (2011), http://leventerkok.github.io/sbv/
12. Fitting, M., Mendelsohn, R.L.: First-Order Modal Logic. Kluwer Academic Publishers, USA (1999)
13. Gammie, P., van der Meyden, R.: MCK: Model checking the logic of knowledge. In: Proc. of CAV-16. pp. 479–483. Springer (2004)
14. Giacomo, G.D., Vardi, M.Y.: Synthesis for LTL and LDL on finite traces. In: Yang, Q., Wooldridge, M.J. (eds.) Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015. pp. 1558–1564. AAAI Press (2015), http://ijcai.org/Abstract/15/223
15. Gorogiannis, N., Raimondi, F., Boureanu, I.: A Novel Symbolic Approach to Verifying Epistemic Properties of Programs. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17. pp. 206–212 (2017). https://doi.org/10.24963/ijcai.2017/30, https://doi.org/10.24963/ijcai.2017/30
16. Grossi, D., Herzig, A., van der Hoek, W., Moyzes, C.: Non-determinism and the dynamics of knowledge. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (2017)
17. Grossi, D., van der Hoek, W., Moyzes, C., Wooldridge, M.: Program models and semi-public environments. Journal of Logic and Computation **29**(7), 1071–1097 (01 2016). https://doi.org/10.1093/logcom/exv086
18. Harel, D.: Dynamic Logic, pp. 497–604. Springer Netherlands, Dordrecht (1984). https://doi.org/10.1007/978-94-009-6259-0_10, https://doi.org/10.1007/978-94-009-6259-0_10
19. Kacprzak, M., Lomuscio, A., Niewiadomski, A., Penczek, W., Raimondi, F., Szreter, M.: Comparing BDD and SAT based techniques for model checking Chaum's dining cryptographers protocol. Fundamenta Informaticae **72**(1-3), 215–234 (2006)

20. Kacprzak, M., Nabiałek, W., Niewiadomski, A., Penczek, W., Półrola, A., Szreter, M., Woźna, B., Zbrzezny, A.: VerICS 2007 – a model checker for knowledge and real-time. Fundamenta Informaticae **85**(1-4), 313–328 (2008)
21. Korovin, K.: iprover–an instantiation-based theorem prover for first-order logic (system description). In: International Joint Conference on Automated Reasoning. pp. 292–298. Springer (2008)
22. Kovács, L., Voronkov, A.: First-order theorem proving and vampire. In: International Conference on Computer Aided Verification. pp. 1–35. Springer (2013)
23. Kripke, S.A.: Semantical considerations on modal logic. Acta Philosophica Fennica **16**, 83–94 (1963)
24. Lomuscio, A., Qu, H., Raimondi, F.: 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). https://doi.org/10.1007/s10009-015-0378-x, http://dx.doi.org/10.1007/s10009-015-0378-x
25. Mendelson, E.: Introduction to Mathematical Logic. Van Nostrand Reinhold, New York (1964)
26. Morgan, C.: The Shadow Knows: Refinement of ignorance in sequential programs. In: Mathematics of Program Construction, pp. 359–378. Lecture Notes in Computer Science, Springer (2006)
27. Plaza, J.: Logics of public communications. Synthese **158**(2), 165–179 (2007). https://doi.org/10.1007/s11229-007-9168-7, http://dx.doi.org/10.1007/s11229-007-9168-7
28. Van Benthem, J., Van Eijck, J., Gattinger, M., Su, K.: Symbolic model checking for dynamic epistemic logic. In: International Workshop on Logic, Rationality and Interaction. pp. 366–378. Springer (2015)
29. Viganò, L., Magazzeni, D.: Explainable security (2018)
30. Wang, S.: Dynamic epistemic model checking with Yices. https://airobert.github.io/FSA_report.pdf (2016), accessed 14/02/2017
31. Wooldridge, M., Lomuscio, A.: A computationally grounded logic of visibility, perception, and knowledge. Logic Journal of IGPL **9**(2), 257–272 (2001)