Give Inconsistency a Chance: Semantics for Ontology-Mediated Verification

Clemens Dubslaff, Patrick Koopmann, and Anni-Yasmin Turhan

Technische Universität Dresden, Dresden, Germany
{clemens.dubslaff,patrick.koopmann,anni-yasmin.turhan}@tu-dresden.de

Abstract. Recently, we have introduced ontologized programs as a formalism to link description logic ontologies with programs specified in the guarded command language, the de-facto standard input formalism for probabilistic model checking tools such as Prism, to allow for an ontology-mediated verification of stochastic systems. Central to our approach is a complete separation of the two formalisms involved: guarded command language to describe the dynamics of the stochastic system and description logics are used to model background knowledge about the system in an ontology. In ontologized programs, these two languages are loosely coupled by an interface that mediates between these two worlds. Under the original semantics defined for ontologized programs, a program may enter a state that is inconsistent with the ontology, which limits the capabilities of the description logic component. We approach this issue in different ways by introducing consistency notions, and discuss two alternative semantics for ontologized programs. Furthermore, we present complexity results for checking whether a program is consistent under the different semantics.

Keywords: Description Logics, Inconsistency, Probabilistic Model Checking, Ontology-Mediated Verification

1 Introduction

Probabilistic model checking (PMC, see, e.g., [2,7] for surveys) is an automated technique for the quantitative analysis of dynamic systems. PMC has been successfully applied in many areas, e.g., to ensure that the system meets quality requirements such as low error probabilities or an energy consumption within a given bound. The de-facto standard specification language for dynamic systems in PMC is given by *stochastic programs*, a probabilistic variant of Dijkstra's guarded command language [5,9] used within many PMC tools such as Prism [10]. Usually, the behavior described by a stochastic program is part of a bigger system, or might be even used within a collection of systems that have an impact on the operational behavior as well. There are different ways how to

model this by using stochastic programs. First, one could integrate additional knowledge about the surrounding system directly into the stochastic program. While this approach provides accurate PMC results, it has the disadvantage that the guarded command language is not well-suited for describing complex static knowledge. Alternatively, one can use non-determinism to over-approximate the possible behaviors of the surrounding system. This approach has the disadvantage that a best- and worst-case PMC analysis might lead to many possible results that do not allow for a meaningful interpretation. As third option, one can use *ontology-mediated PMC*, an approach to integrate knowledge described by a description logic (DL) ontology into the PMC process, which we recently proposed in [6]. The core of this approach are *ontologized (stochastic) programs* which can be subject of PMC.

A central idea of ontologized programs is the complete separation of concerns and formalisms: the operational behavior is described using a *program component* expressed by guarded commands. Static knowledge about the system is described in an ontology using DL formalisms. An interface mediates between these two worlds by providing mappings from statements in guarded command language to DL and vice versa. The loose coupling achieved in this way allows to reuse and replace both components easily, so that the same behavior can be analyzed with different ontologies (to describe different system configurations), and the same ontology can be used with different programs (e.g., to describe different strategies of behavior to be analyzed within the same system). This approach distinguishes our work from other approaches that use DLs for the verification of dynamic systems, in which DL constructs are used directly within the program [1,4,8,14]. To be able to analyze ontologized programs practically, we developed a two-step procedure in [6]. First, the ontologized program and the property to be analyzed are *rewritten* into a plain stochastic program and a modified property. Then, these are evaluated using the PMC tool Prism. This way, essentially all model-checking tasks supported by Prism can be performed on ontologized programs, ranging from verification of standard properties (like the probability or the expected time for reaching a failure state) to more advanced quantitative analysis (cf. [7,3]). Our approach was implemented and evaluated in [6].

Under the original semantics of ontologized programs in [6], program components are allowed to enter states that are inconsistent with the ontology. This leaves the task of resolving the inconsistency to the program component by implementing a sort of exception handling. However, inconsistent states may also stand for situations that never can arise in practice. In this case, a semantics that avoids inconsistent states when possible is favorable, leading to a consistency notion of ontologized programs. In this paper, we answer the following open questions: 1) how to handle inconsistent states in a meaningful way and 2) what does it mean for an ontologized program to be inconsistent. We approach these questions by presenting three possible semantics of ontologized programs: consistency-independent, probability-normalizing, and probability-preserving semantics. Each semantics specifies the behavior of the program in a different way,

and additionally comes with its own notion of consistency. For each of these notions, we give tight complexity bounds for deciding consistency of programs.

2 Preliminaries

We assume familiarity of the reader with the basics of description logics. The results in this paper apply to different DLs, on which we make some basic assumptions: 1) they are fragments of $\mathcal{SROIQ}$, and 2) they use unary encoding for numbers. This was done for convenience in our proofs, and we conjecture that our results can be lifted to a more general class of DLs.

We briefly address relevant preliminaries on probabilistic operational models and their verification. A probability *distribution* over S is a function $\mu\colon S \to [0,1]\cap\mathbb{Q}$ with $\sum_{s\in S}\mu(s) = 1$, denoting the set of distributions over S by $\mathit{Distr}(S)$.

Markov decision processes (MDPs) are tuples $\mathbf{M} = \langle Q, \mathit{Act}, P, q_0, \Lambda, \lambda\rangle$ where Q and Act are countable sets of *states* and *actions*, respectively, P is a *partial probabilistic transition function* $P\colon Q \times \mathit{Act} \rightharpoonup \mathit{Distr}(Q)$ and $q_0 \in Q$ an *initial state*, Λ is a *set of labels* assigned to states via a *labeling function* $\lambda\colon Q \to \wp(\Lambda)$. For convenience, for $q_1, q_2 \in Q$, $\alpha \in \mathit{Act}$, we abbreviate $P(q_1,\alpha)(q_2)$ by $P(q_1,\alpha,q_2)$. An action $\alpha \in \mathit{Act}$ is *enabled* in a state $q \in Q$ if $P(q,\alpha)$ is defined. We assume that $\mathbf{M}$ does not have *final states*, i.e., in each state at least one action is enabled. A *finite path* in $\mathbf{M}$ is a sequence $\pi = q_0\,\alpha_0\,q_1\,\alpha_1\,\ldots\alpha_{k-1}q_k$ where $p_i = P(q_i,\alpha_i,q_{i+1}) > 0$ for all $i < k$. The probability of π is $\Pr(\pi) = p_0 \cdot p_1 \cdot \ldots \cdot p_{k-1}$. Intuitively, the behavior of $\mathbf{M}$ in state q is to select an enabled action α and move to a successor state q' with probability $P(q,\alpha,q')$. Such a selection is done via *(randomized) schedulers*, i.e., functions $\mathfrak{S}$ that map a finite paths to a distribution over actions. For schedulers, we require that for each finite path π in $\mathbf{M}$ with last state q, we have $P(q,\alpha)$ is defined for any $\alpha \in \mathit{Act}$ with $\mathfrak{S}(\pi)(\alpha) > 0$. Then, a probability measure $\Pr^{\mathfrak{S}}_{\mathbf{M}}$ over maximal $\mathfrak{S}$-paths, i.e., infinite paths that follow $\mathfrak{S}$, is defined in the standard way (cf. [13]).

Probabilistic model checking (PMC, cf. [2]) is used in this paper for a quantitative analysis of MDPs. Usually one has given a temporal logical property ϕ over the set of labels Λ that defines sets of maximal paths in the MDP that fulfill ϕ. For instance, *linear temporal logic (LTL, [12])* can be used to specify the set of paths reaching states labeled by an $\ell \in \Lambda$, formally $\phi_1 = \Diamond\ell$, or where always when such a state is reached, within two time steps a state not labeled by ℓ is reached again, formally $\phi_2 = \Box\big(\ell \to (\mathsf{X}\neg\ell \vee \mathsf{XX}\neg\ell)\big)$. For a scheduler $\mathfrak{S}$, any LTL property constitutes a measurable set of $\mathfrak{S}$-paths such that we can reason about the probability $\Pr^{\mathfrak{S}}_{\mathbf{M}}(\phi)$ of $\mathbf{M}$ satisfying an LTL property ϕ w.r.t. $\mathfrak{S}$. By ranging over all possible schedulers, this enables a best- and worst-case analysis. A *stochastic property* is an expression $\Phi = \mathsf{P}^{\mathrm{ex}}(\phi) \bowtie \tau$ where ϕ is an LTL formula, $\tau \in \mathbb{Q}$ a threshold, $\bowtie \in \{<,\leq,\geq,>\}$ a relation, and $\mathrm{ex} \in \{\min,\max\}$. Then, $\mathbf{M}$ *satisfies* Φ, denoted $\mathbf{M} \models \Phi$, iff $\mathrm{ex} = \min$ and $\inf_{\mathfrak{S}} \Pr^{\mathfrak{S}}_{\mathbf{M}}(\phi) \bowtie \tau$ or

ex = max and $\sup_{\mathfrak{S}} \Pr^{\mathfrak{S}}_{\mathbf{M}}(\phi) \bowtie \tau$. The *PMC problem* for Φ and $\mathbf{M}$ amounts to decide whether $\mathbf{M} \models \Phi$. For more details on MDPs and PMC, we refer to standard textbooks such as [13,2,7]. Note that in [6] we considered weighted MDP for ontology-mediated PMC over expected accumulated costs. For brevity, we consider only plain MDPs in this paper.

Arithmetic Constraints and Boolean Expressions are defined over a countable set $\mathcal{V}$ of *variables*. An *evaluation* over $\mathcal{V}$ is a function η that assigns to each $v \in \mathcal{V}$ a value in $\{-\delta(v), -\delta(v)+1, \ldots, \delta(v)\}$ where $\delta\colon \mathcal{V} \to \mathbb{N}$ defines bounds of each variable. The set of evaluations is denoted by $\mathit{Eval}(\mathcal{V})$. Let z range over $\mathbb{Z}$ and v range over $\mathcal{V}$. The set of *arithmetic expressions* $\mathbb{E}(\mathcal{V})$ is defined by the grammar $\alpha \ ::= \ z \mid v \mid (\alpha + \alpha) \mid (\alpha \cdot \alpha)$. Evaluations naturally extend to arithmetic expressions, e.g., $\eta(\alpha_1 + \alpha_2) = \eta(\alpha_1) + \eta(\alpha_2)$. $\mathbb{C}(\mathcal{V})$ denotes the set of *arithmetic constraints* over $\mathcal{V}$, which are terms of the form $(\alpha \bowtie z)$ with $\alpha \in \mathbb{E}(\mathcal{V})$, $\bowtie \in \{>, =, <\}$, and $z \in \mathbb{Z}$. For a given evaluation $\eta \in \mathit{Eval}(\mathcal{V})$ and constraint $\gamma \in \mathbb{C}(\mathcal{V})$, we write $\eta \models \gamma$ iff γ holds in η. We denote by $\mathbb{C}(\eta)$ the *constraints entailed by* η, i.e., $\mathbb{C}(\eta) = \{c \in \mathbb{C}(\mathcal{V}) \mid \eta \models c\}$. $\mathbb{B}(\mathcal{V})$ denotes the set of Boolean expressions over $\mathbb{C}(\mathcal{V})$, i.e., the set of propositional formulas where arithmetic constrains over $\mathcal{V}$ are used as atoms. Entailment of Boolean expressions $\mathbb{B}(\mathcal{V})$ from evaluations is defined in the usual way.

Stochastic programs provide in the area of PMC the de-facto standard to concisely describe MDPs. They are essentially a probabilistic variant of Dijkstra's *guarded command language* [5,9]. We call a function $u\colon \mathcal{V} \to \mathbb{E}(\mathcal{V})$ *update*, and a distribution $\sigma \in \mathit{Distr}(\mathit{Upd})$ over a given finite set *Upd* of updates *stochastic update*. The effect of an update $u\colon \mathcal{V} \to \mathbb{E}(\mathcal{V})$ on an evaluation $\eta \in \mathit{Eval}(\mathcal{V})$ is their composition $\eta \circ u \in \mathit{Eval}(\mathcal{V})$, i.e., $(\eta \circ u)(v) = \max\{-\delta(v), \min\{\eta(u(v)), \delta(v)\}\}$ for all $v \in \mathcal{V}$. This notion naturally extends to *stochastic updates* $\sigma \in \mathit{Distr}(\mathit{Upd})$ where for any $\eta, \eta' \in \mathit{Eval}(\mathcal{V})$ we have that $(\eta \circ \sigma)(\eta')$ is the sum over all $\sigma(u)$ for which $\eta \circ u = \eta'$. A *stochastic guarded command* over a finite set of updates *Upd*, briefly called *command*, is a pair $\langle g, \sigma \rangle$ where $g \in \mathbb{B}(\mathcal{V})$ is a *guard* and $\sigma \in \mathit{Distr}(\mathit{Upd})$ is a stochastic update, which we write in the following form:

$$(\texttt{server\_proc1} = 2 \wedge \texttt{server\_proc2} = 2) \mapsto \begin{array}{l} 1/2 : \texttt{server\_proc1} := 1 \\ 1/2 : \texttt{server\_proc1} := 3 \end{array} \qquad (1)$$

A *stochastic program* over $\mathcal{V}$ is a tuple $\mathbf{P} = \langle \mathcal{V}, C, \eta_0 \rangle$ where C is a finite set of commands and $\eta_0 \in \mathit{Eval}(\mathcal{V})$ is an initial variable evaluation. For brevity, we write $\mathit{Upd}(\mathbf{P})$ for the set of all updates in C. The *MDP induced by* $\mathbf{P}$ is defined as $\mathbf{M}[\mathbf{P}] = \langle S, \mathit{Act}, P, \eta_0, \Lambda, \lambda \rangle$ where

- $S = \mathit{Eval}(\mathcal{V})$,
- $\mathit{Act} = \mathit{Distr}(\mathit{Upd}(\mathbf{P}))$,
- $\Lambda = \mathbb{C}(\mathcal{V})$,
- $\lambda(\eta) = \mathbb{C}(\eta)$ for all $\eta \in S$, and
- $P(\eta, \sigma, \eta') = (\eta \circ \sigma)(\eta')$ for $\eta, \eta' \in S$ and $\langle g, \sigma \rangle \in C$ with $\lambda(\eta) \models g$.

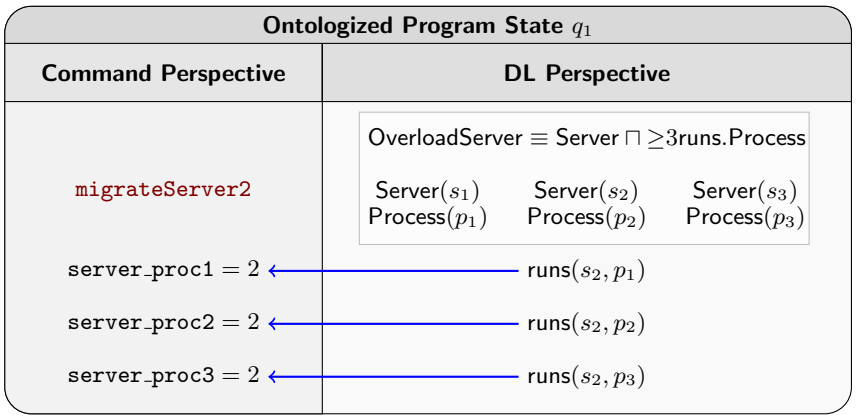

Fig. 1. Perspectives on the ontologized program state q_1: the command perspective (left) and DL perspective (right), linked by the interface (arrows).

3 Ontologized Stochastic Programs

In general, an ontologized program comprises the following three components [6]:

The Program is a specification of the operational behavior.
The Description Logic Ontology contains static background knowledge that may influence the behavior of the program.
The Interface links program and ontology by providing mappings between the languages used for the program and the ontology.

Ontologized programs combine the two formalisms guarded command language and description logics (DLs). To achieve maximum division of concerns, the behaviors and the static knowledge about the context of the program are specified separately and are only loosely coupled by an interface. This allows for specifying operational behavior in the usual way and to extend and reuse existing program specifications, and similarly, to easily link established or even standardized ontologies to a program.

Before we formally define ontologized programs, we explain how global states of ontologized programs are represented. Both formalisms, guarded command language and DL, may represent abstracted versions of global states in different ways. For the guarded command language a state is described as an evaluation of variables, while in DL a state is described using a set of axioms. Both formalisms require different degrees of abstraction and detail, depending on their dedicated purpose. Consequently, global states of ontologized programs are composed of two components, which we call the *command perspective* and the *DL perspective* on the global system state (see Figure 1). As can be seen in the figure, there are some elements in the two perspectives that correspond to each other, while others have no direct correspondence. The correspondence, depicted by arrows in

the figure, is defined in the interface component of the ontologized program via a function Dc (**D***L to* **c***ommand*), mapping DL axioms to command expressions. The part of the DL perspective inside the box is not mapped to anything on the command side – this is the *static part*, and contains the background knowledge about the system that is independent of the current state of the system. In contrast, the mapped axioms in the ontology perspective can change, which is why we call them *fluents*. Note that we also allow the command perspective to use variables that have no correspondence in the DL perspective.

So far, our description of global states does not depend on any DL reasoning, which is, however, required by the DL component to provide background knowledge for the operational component of the program. To invoke a reasoning task from the command side of ontologized programs, we have introduced the notion of hooks in [6]. A *hook* is a propositional variable in the program component that is linked by the interface to a Boolean *query* on the DL component of the ontologized program state. The result of the query determines the value of the propositional variable using the function cD (**c***ommand to* **D***L*). In our example, we would for each server $i \in \{1, 2, 3\}$ define one hook `migrateServer`i that stands for the knowledge that processes should be migrated from Server i:

$$\mathsf{cD}(\texttt{migrateServer}i) = \{\mathsf{OverloadedServer}(s_i)\}.$$

In the ontologized state shown in Figure 1, the hook `migrateServer2` is active because the ontology in DL perspective entails $\mathsf{OverloadedServer}(s_2)$.

To enable the actual use of hooks in the guarded command language, we have to slightly extend the definition of commands. Specifically, we define *abstract stochastic programs* as stochastic programs where guards are Boolean expressions over arithmetic expressions and hooks. For example a command in an abstract stochastic program looks like the following:

$$(\texttt{migrateServer2} \wedge \texttt{server\_proc1} = 2) \;\mapsto\; \begin{array}{l} 1/2 : \texttt{server\_proc1} := 1 \\ 1/2 : \texttt{server\_proc1} := 3 \end{array} \qquad (2)$$

While the intention of this command is similar to the command (1), this version omits the specification of the condition under which Server 2 needs to migrate. Specifying and testing this condition is now delegated to the DL component.

Now we are ready to formally define ontologized programs. Given an abstract stochastic program $\mathbf{P}$, we denote by $\mathcal{H}_{\mathbf{P}}$ the hooks used in $\mathbf{P}$.

Definition 1. *An* ontologized program *is a tuple* $\mathfrak{P} = \langle \mathbf{P}, \mathcal{K}, \mathfrak{I} \rangle$ *where*

- $\mathbf{P} = \langle \mathcal{V}_{\mathfrak{P}}, C_{\mathfrak{P}}, W_{\mathfrak{P}}, \eta_{\mathfrak{P},0} \rangle$ *is an abstract stochastic program,*
- $\mathcal{K}$ *is a DL ontology called the* static DL knowledge*,*
- $\mathfrak{I} = \langle \mathcal{V}_{\mathfrak{I}}, \mathcal{H}_{\mathfrak{I}}, \mathcal{F}, \mathsf{Dc}, \mathsf{cD} \rangle$ *is a tuple describing the* interface*, where* $\mathcal{V}_{\mathfrak{I}}$ *is a set of* public variables*,* $\mathcal{H}_{\mathfrak{I}}$ *is a set* interface hooks*,* $\mathcal{F}$ *is a set of* DL axioms *called* fluents*, and two mappings* $\mathsf{cD}\colon \mathcal{H}_{\mathfrak{I}} \to \wp(\mathbb{A})$ *and* $\mathsf{Dc}\colon \mathcal{F} \to \mathbb{B}(\mathcal{V}_{\mathfrak{I}})$*, and*

$\mathbf{P}$ *is* compliant *with* $\mathfrak{I}$*, i.e.,* $\mathcal{H}_{\mathbf{P}} \subseteq \mathcal{H}_{\mathfrak{I}}$ *and* $\mathcal{V}_{\mathfrak{I}} \subseteq \mathcal{V}_{\mathfrak{P}}$*. If for a DL* $\mathcal{L}$ *every axiom in* $\mathcal{K} \cup \mathcal{F}$ *is an* $\mathcal{L}$ *axiom, we call* $\mathfrak{P}$ *an* $\mathcal{L}$-ontologized program*.*

4 Three Semantics of Ontologized Programs

Stochastic programs are used as concise representations of MDPs that can be subject of a quantitative analysis. Analogously, ontologized programs are concise representations of *ontologized MDPs*, where a quantitative analysis on these MDPs can further rely on information provided by the ontology. Since there is an additional logical component in ontologized programs, there are different meaningful ways in which ontologized MDPs can be defined to represent the operational behavior of ontologized programs. We present three semantics for ontologized programs that differ in their treatment of inconsistencies. Such inconsistencies arise when variable evaluations in the stochastic program have an inconsistent meaning in the ontology. Choosing an appropriate semantics is a modeling decision to be made when constructing the ontologized program. Therefore we analyze the impact of this choice on the computational complexity of deciding consistency w.r.t. each of the different semantics. First we define formally that ontologized states comprise a command perspective and a DL perspective (see Figure 1).

Definition 2. *Let $\mathfrak{P} = \langle \mathbf{P}, \mathcal{K}, \mathfrak{I} \rangle$ be an ontologized program as in Definition 1. An* ontologized state in $\mathfrak{P}$ *is a tuple $q = \langle \eta_q, \mathcal{K}_q \rangle$, with* command perspective η_q *and* DL perspective $\mathcal{K}_q$*, where*

- $\mathcal{K} \subseteq \mathcal{K}_q$,
- $\mathcal{K}_q \subseteq \mathcal{K} \cup \mathcal{F}$,
- $\eta_q \in \mathit{Eval}(\mathcal{V}_\mathfrak{P})$, *and*
- $\alpha \in \mathcal{K}_q$ *iff* $\eta_q \models \mathsf{Dc}(\alpha)$ *for every* $\alpha \in \mathcal{F}$.

We call an ontologized state q inconsistent *if $\mathcal{K}_q$ is inconsistent.*

For every evaluation $\eta \in \mathit{Eval}(\mathcal{V}_\mathfrak{P})$, there is always a unique KB $\mathcal{K}_\eta$ s.t. $\langle \eta, \mathcal{K}_\eta \rangle$ is an ontologized state in $\mathfrak{P}$. We denote this state by $s(\mathfrak{P}, \eta)$. To define updates on ontologized states, we exploit the fact that 1) updates can only directly modify the command perspective of ontologized states, and 2) the DL perspective of ontologized states is fully determined by the command perspective. Thus, we can easily lift the effect of updates on evaluations to updates on ontologized states. For an update $u \in \mathit{Upd}(\mathcal{V}_\mathfrak{P})$ and an ontologized state q, we define $u(q) = s(\mathfrak{P}, u(\eta_q))$. Correspondingly, we extend stochastic updates $\sigma\colon \mathit{Upd}(\mathcal{V}_\mathfrak{P}) \to [0,1]$ to ontologized states q by setting $\sigma(q, u(q)) = \sigma(u)$ for all $u \in \mathit{Upd}(\mathcal{V}_\mathfrak{P})$. To illustrate how commands are executed on ontologized states, we consider the ontologized state q_1 shown in Figure 1 and the abstract command in (2). First, we note that the guard in the command is active, since the hook `migrateServer2` is active in q_1. The command can thus be executed on the state, and executes each of its updates with a 50% chance. For the first update, `server_proc1` := 1, we obtain the ontologized state q_2 in which `server_proc1` = 2 is replaced by `server_proc1` = 1, and $\mathsf{runs}(s_2, p_2)$ is replaced by $\mathsf{runs}(s_1, p_2)$. Consequently, the hook `migrateServer2` becomes inactive in q_2, since $\mathsf{cD}(\texttt{migrateServer2}) = \{\mathsf{OverloadedServer}(s_2)\}$ and $\mathsf{OverloadedServer}(s_2)$ is not entailed by the DL perspective on q_2.

4.1 Consistency-Independent Semantics

We first present the semantics for ontologized programs originally introduced in [6]. This semantics does not give inconsistent ontologized states a priori a special meaning, therefore named *consistency-independent semantics*. The definition of the MDP induced by an ontologized program under consistency-independent semantics closely follows the one for MDPs induced by plain stochastic programs.

Definition 3. *Let $\mathfrak{P} = \langle \mathbf{P}, \mathcal{K}, \mathfrak{I} \rangle$ be an ontologized program as in Definition 1. The* MDP induced by $\mathfrak{P}$ *under consistency-independent semantics is given by $\mathbf{M}_{ci}[\mathfrak{P}] = \langle Q, Act, P, q_0, \Lambda, \lambda \rangle$, where*

- $Q = \{s(\mathfrak{P}, \eta) \mid \eta \in \mathit{Eval}(\mathcal{V}_\mathfrak{P})\}$,
- $\mathit{Act} = \mathit{Distr}(\mathit{Upd}(\mathbf{P}))$,
- $q_0 = s(\mathfrak{P}, \eta_0)$,
- $\Lambda = \mathcal{H}_\mathbf{P} \cup \mathbb{C}(\mathcal{V}_\mathfrak{P})$,
- $\lambda(q) = \mathbb{C}(\eta_q) \cup \{\ell \in \mathcal{H}_\mathbf{P} \mid \mathcal{K}_q \models \mathsf{cD}(\ell)\}$ *for every $q \in Q$, and*
- $P(q, \sigma) = \sigma(q)$ *for all $\langle g, \sigma \rangle \in C$ with $\lambda(q) \models g$.*

$\mathfrak{P}$ *is* inconsistent *if there exists an inconsistent $q \in Q$ reachable from q_0.*

We can test consistency of ontologized programs using a reachability analysis on the state space. For this, we generate states one after the other, using DL reasoner calls for each state to determine their active hooks and their consistency. As each state requires polynomial space, this can be done in polynomial space with a k-oracle, where k is the complexity of entailment in the DL. A corresponding lower bound can be obtained by reduction of the word problem for polynomially space-bounded Turing machines with k-oracle. The construction used here shows a close relationship between Turing machines with k-oracle and $\mathcal{L}$-ontologized programs, provided that $\mathcal{L}$ is k-hard. This relationship is also used for later complexity results presented in this paper.

Theorem 1. *Let $\mathcal{L}$ be a DL such that deciding entailment in $\mathcal{L}$ is k-complete. Then, deciding consistency of $\mathcal{L}$-ontologized programs is* PSPACE^k*-complete, even for non-stochastic programs.*

4.2 Probability-Normalizing Semantics

If a stochastic program is inconsistent, consistency-independent semantics relies on the program specification to deal with inconsistent states. On the one hand, this can offer flexibility. On the other hand, it can be desirable to let the semantics handle the situation by definition. Since inconsistent states have no correspondence to states of the modeled system, they could stand for undesired states that should not occur in the MDP at all. The idea of probability-normalizing semantics is to remove all states of paths that can lead to an inconsistent state and locally normalize probabilistic choices accordingly. Under this semantics, the ontology serves an additional purpose: not only does it assign meaning to the hooks, it also specifies which program states are possible, and thus restricts the

transitions from a given state. This allows for a further separation of concerns, and thus for reusability of the behavioral and the DL component of the program: we may use the same behavioral component with different ontologies that put different constraints on the state space. In the running server example, we may want to analyze a system in which the servers have different software and/or hardware configurations, such that some processes cannot run on all servers. For instance, we might use a different ontology that specifies that Process 1 cannot run on Server 3. This restricts the possible outcomes of applying the command in (2) on the state in Figure 1 (with the modified ontology). It can now only move Process 1 to Server 1, so that this migration is performed with 100% probability.

To achieve this behavior under consistent-independent semantics, we would have to extend the commands of the program with a case distinction over all possible successors to inconsistent states, which is clearly undesired in the above example. Instead, we capture the restriction to possible states of the MDP by defining *probability-normalizing semantics* for ontologized programs. Intuitively, under probability-normalizing semantics, the induced MDP can be obtained from the corresponding consistency-independent semantics by removing inconsistent states and then normalizing the resulting probabilities. It is possible that for some state q and command $\langle g, \sigma \rangle$ with $\lambda(q) \models g$ such a normalization is not possible, since all successor states that should have a positive probability lead to an inconsistent state. In this case, the command cannot be applied on q. If no command can be applied anymore on a state, this state is removed from the MDP as well.

Definition 4. *Let* $\mathbf{M}_{ci}[\mathfrak{P}] = \langle Q, Act, P, q_0, \Lambda, \lambda \rangle$ *for an ontologized program* $\mathfrak{P}$*. An MDP* $\mathbf{M}$ *is* probability-normalizing w.r.t. $\mathfrak{P}$ *if* q_0 *is consistent and* $\mathbf{M} = \langle Q', Act, P', q_0, \Lambda, \lambda' \rangle$*, where*

1. $Q' \subseteq Q$ *contains only consistent states,*
2. λ' *is obtained from* λ *by restricting its domain to* Q'*,*
3. *for all* $q \in Q$ *and* $\sigma \in Act$ *s.t.* $P(q, \sigma)$ *is defined, we have for all* $q' \in Q'$*:*

$$P(q, \sigma, q') \;=\; P'(q, \sigma, q') \cdot \sum\nolimits_{\hat{q} \in Q'} P(q, \sigma, \hat{q}).$$

$\mathfrak{P}$ *is called* probability-normalizable *if there exists a probability-normalizing MDP w.r.t. by* $\mathfrak{P}$*. In case* Q' *and* P' *are subset-maximal,* $\mathbf{M}$ *is the called* the MDP induced by $\mathfrak{P}$ under probability-normalizing semantics*, denoted* $\mathbf{M}_{pn}[\mathfrak{P}]$*.*

Recall that according to our definition, MDPs cannot have final states. Therefore, to obtain the probability probability-normalizing MDP, it does not suffice to remove inconsistent state, but also states that would only lead to inconsistent states would have to be removed. As this operation might lead to a removal of the initial state, not every ontologized program is probability-normalizable. While for consistency-independent semantics consistency of a program can be decided by determining reachability of inconsistent states, we have to solve a complementary task for probability-normalizability. Specifically, we have to decide whether one can find an unbounded path that never enters an inconsistent

state, for which again polynomial space and an oracle for the reasoner is sufficient. As a result, deciding probability-normalizability has the same complexity as deciding consistency under consistency-independent semantics.

Theorem 2. *Let $\mathcal{L}$ be a DL for which deciding entailment has complexity k. Then, deciding whether an $\mathcal{L}$-ontologized program is probability-normalizable is* PSPACE^k*-complete, even for non-stochastic programs.*

4.3 Probability-Preserving Semantics

If we use probability-normalizing semantics, it is possible that the probabilities specified in the commands change. This is the right choice if the program uses probabilities to model randomized behavior as in our running example: the program migrates a process as soon as this becomes necessary, and it does so by choosing each possible server with equal probability. If the ontology restricts the number of possible choices, then consequently the probabilities need to change as well. However, there are other scenarios where a change of probabilities is not desired: rather than using stochastic commands to model randomized behavior, the program component may use stochastic commands to describe probabilistic outcomes of the simulated system that are based on measured probabilities. For instance, we might know that at any point in time the server has a probability of 1% of becoming disfunct and needs a restart. This probability should not be affected by the specifications: there might be a program state in which a disfunct server would necessarily lead to an inconsistency in the future. This should not mean that the server cannot become disfunct and that the current program state should be possible by definition. To capture such phenomena in a semantics, we introduce *probability-preserving semantics*.

Definition 5. *Let $\mathbf{M}_{ci}[\mathfrak{P}] = \langle Q, Act, P, q_0, \Lambda, \lambda\rangle$ for an ontologized program $\mathfrak{P}$. An MDP $\mathbf{M}$ is* probability-preserving w.r.t. $\mathfrak{P}$ *if q_0 is consistent and $\mathbf{M} = \langle Q', Act, P', q_0, \Lambda, \lambda'\rangle$ where*

1. *$Q' \subseteq Q$ contains only consistent states,*
2. *λ' is obtained from λ by restricting its domain to Q', and*
3. *for all $P'(q, \sigma, q') > 0$ we have that $P'(q, \sigma, q') = P(q, \sigma, q')$.*

$\mathfrak{P}$ is called probability-preservable *if there exists a probability-preserving MDP w.r.t. $\mathfrak{P}$. In case Q' and P' are subset-maximal, $\mathbf{M}$ is the called* the MDP induced by $\mathfrak{P}$ under probability-preserving *semantics, denoted $\mathbf{M}_{pp}[\mathfrak{P}]$.*

Note that Definition 5 requires $\mathbf{M}$ to be an MDP and hence, P' is a probability transition function. Hence, Condition 3 implies that any transition that reaches an inconsistent state with positive probability w.r.t. P has no corresponding transition in P'.

To verify whether a program is probability-preservable, we have to take the non-deterministic and the probabilistic choices of the program differently into account. This is different to the other notions of consistency discussed here, because

only a single path, going over non-deterministic and probabilistic choices indifferently, is sufficient to witness inconsistency or probability-normalizability. We can characterize probability-preservability as follows: a program is probability-preservable iff there *exists* a resolution of the non-deterministic choices in the program such that for *all* probabilistic choices an inconsistent state is never reached. Thus, testing this property requires both existential and universal explorations of the state space, which is why we can use alternating Turing machines with polynomial space to solve the corresponding problem.

Theorem 3. *Let $\mathcal{L}$ be a DL for which deciding entailment has complexity k. Then, deciding whether an $\mathcal{L}$-ontologized program is probability-preservable is* APSPACE^k*-complete. For non-stochastic programs, it is* PSPACE^k*-complete.*

Recall that APSPACE = EXPTIME (see for example Corollary 2 to Theorem 16.5 and Corollary 3 to Theorem 2.20 in [11]). However, as an exponential Turing machine may also write exponentially long queries to the oracle, we may not always have APSPACE^k = EXPTIME^k.

4.4 Ontology-mediated PMC

In [6] we presented a technique to apply PMC techniques to ontologized programs to perform quantitative analysis tasks on knowledge-intensive systems. For this, we treated inconsistent states as consistent ones, following the consistency-independent semantics presented in Section 4.1. Inconsistencies in ontologized programs is not as bad as usually in other logic-based formalisms, as stochastic properties can also be evaluated on MDPs with inconsistent states. In fact, they can even have a designated meaning, e.g., modeling exceptions the program has to face. An advantage of ontologized programs is that commands could be used to detect when a program is in an inconsistent state. To this end, a guard employing a hook $\texttt{inc}$ that is satisfied in exactly those states that are inconsistent, e.g., with $\mathsf{cD}(\texttt{inc}) = (\top \sqsubseteq \bot)$, can invoke the actions to undertake when an exception has occurred. Note that while all inconsistent states agree on the hooks that are active (it is always the complete set of hooks), the command perspective of states can still be different. Here, private variables may encode additional information about the current state that can be used to decide how to react to inconsistencies. An example for ontology-mediated PMC on consistency-independent semantics, is to decide whether

$$\mathsf{P}^{\min}\left(\Box(\texttt{inc} \rightarrow (\mathsf{X}\neg\texttt{inc} \vee \mathsf{XX}\neg\texttt{inc}))\right) \;>\; 95\%$$

Intuitively, this states that an exception is always successfully handled with high probability of at least 95% in the sense that whenever an inconsistent state is reached, a consistent state is reached again within at most two steps. Such a stochastic property can be checked using our method in [6] and the PMC tool PRISM [10].

Ontology-mediated PMC for probability-normalizing and -preserving semantics is not as straight forward as under consistency-independent semantics. While

we could achieve a stochastic program that disallows to reach inconsistent states also under consistency-independent semantics, this requires an explicit handling of inconsistencies and modifications in the program that "provisions" the possible variable evaluations. More specifically, we then need a the possibility to decide for each update in the command whether it would lead to an inconsistent state or not. However, while this would work out well in our running example, we cannot expect such an encoding for any ontologized program. Fortunately, Definitions 4 and 5 are defined in a constructive way, i.e., these consistency-dependent semantics are defined based on the consistency-independent one by manipulating the state space and probabilistic transition function. Hence, we could restrict the state space to consistent states and adjust further states and probabilities on-the-fly during the construction process of the MDP.

5 Outlook

We are currently implementing a scenario to evaluate our implementation for ontology-mediated PMC under the different semantics we presented in this paper. Here, we can benefit from the constructive definition of our semantics that evaluates inconsistent states beforehand and could hence directly included into the model building process, e.g., in the probabilistic model checker Prism [10]. There are some theoretical results that are still left open but should be easy to obtain: the complexity of the actual model-checking tasks, or the size of the rewritings that our practical method produces. Furthermore, there are some extensions and modifications worth investigating: our semantics offer two possible ways of treating probabilities occurring in the program to "restore" consistency. The choice of which one to select depends on the kind of stochastic phenomena actually modeled. It is easy to think of scenarios where both kinds of stochastics occur. This would require a more flexible type of ontologized programs. In such a program, some distributions may be marked as "fixed", while others could be subject to normalization. In our framework, modifications of program states happen on the level of DL axioms and thus affect all models similarly to [4,8]. Another approach are DL action-based programs [1,14], where modifications of states affect the level of interpretations. So, a variant of ontologized programs where states are associated not with ontologies, but with interpretations, similar to [1,14] is certainly an interesting direction for future research.

Acknowledgements

The authors are supported by the DFG through the Collaborative Research Center TRR 248 (see `https://perspicuous-computing.science`, project ID 389792660), the Cluster of Excellence EXC 2050/1 (CeTI, project ID 390696704, as part of Germany's Excellence Strategy), and the Research Training Groups QuantLA (GRK 1763) and RoSI (GRK 1907).

References

1. Baader, F., Zarrieß, B.: Verification of Golog programs over description logic actions. In: Proceedings of FroCos 2013. LNCS, vol. 8152, pp. 181–196. Springer (2013)
2. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
3. Baier, C., Dubslaff, C., Klüppelholz, S.: Trade-off analysis meets probabilistic model checking. In: Proc. of the 23rd Conference on Computer Science Logic and the 29th Symposium on Logic In Computer Science (CSL-LICS). pp. 1:1–1:10. ACM (2014)
4. Calvanese, D., De Giacomo, G., Lenzerini, M., Rosati, R.: Actions and programs over description logic knowledge bases: A functional approach. In: Knowing, Reasoning, and Acting: Essays in Honour of H. J. Levesque. College Publications (2011)
5. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)
6. Dubslaff, C., Koopmann, P., Turhan, A.: Ontology-mediated probabilistic model checking. In: Ahrendt, W., Tarifa, S.L.T. (eds.) Integrated Formal Methods - 15th International Conference, IFM 2019. Lecture Notes in Computer Science, vol. 11918, pp. 194–211. Springer (2019). https://doi.org/10.1007/978-3-0u30-34968-4_11
7. Forejt, V., Kwiatkowska, M.Z., Norman, G., Parker, D.: Automated verification techniques for probabilistic systems. In: Proc. of the School on Formal Methods for the Design of Computer, Communication and Software Systems, Formal Methods for Eternal Networked Software Systems (SFM'11). LNCS, vol. 6659, pp. 53–113. Springer (2011)
8. Hariri, B.B., Calvanese, D., Montali, M., De Giacomo, G., De Masellis, R., Felli, P.: Description logic knowledge and action bases. J. Artif. Intell. Res. **46**, 651–686 (2013)
9. Jifeng, H., Seidel, K., McIver, A.: Probabilistic models for the guarded command language. Science of Computer Programming **28**(2), 171 – 192 (1997)
10. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Proc. of the 23rd Intern. Conf. on Computer Aided Verification (CAV'11). LNCS, vol. 6806, pp. 585–591 (2011)
11. Papadimitriou, C.H.: Computational complexity. Academic Internet Publ. (2007)
12. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977. pp. 46–57 (1977). https://doi.org/10.1109/SFCS.1977.32
13. Puterman, M.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley & Sons, Inc., New York, NY (1994)
14. Zarrieß, B., Claßen, J.: Verification of knowledge-based programs over description logic actions. In: IJCAI. pp. 3278–3284. AAAI Press (2015)

A Appendix

We provide here the proofs of our results that have been omitted due to space constraints in the main paper.

A.1 Auxiliary Lemmas for the Hardness Results

We define the size of a concept/axiom/ontology in the usual way as its string length, that is, the number of symbols needed to write it down, where concept, role and individual names count as one, as do logical operators, and number restrictions are encoded using unary encoding.

The following auxiliary lemma will be helpful for the hardness results, as it shows how to represent ontologies based on a polynomially bounded enumeration of DL axioms. We first need a more liberal definition of conservative extensions.

Definition 6. *A* renaming *is a bijective function* $\sigma : \mathsf{N_C} \cup \mathsf{N_R} \cup \mathsf{N_I} \to \mathsf{N_C} \cup \mathsf{N_R} \cup \mathsf{N_I}$ *s.t.* $\sigma(X) \in \mathsf{N_C}$ *iff* $X \in \mathsf{N_C}$, $\sigma(X) \in \mathsf{N_R}$ *iff* $X \in \mathsf{N_R}$ *and* $\sigma(X) \in \mathsf{N_I}$ *iff* $X \in \mathsf{N_I}$. *Given an ontology* $\mathcal{O}$ *and a renaming* σ, *we denote by* $\mathcal{O}\sigma$ *the result of replacing every name* X *in* $\mathcal{O}$ *by* $\sigma(X)$.

Let $\mathcal{O}_1$ *and* $\mathcal{O}_2$ *be two ontologies. Then,* $\mathcal{O}_1$ *is a* conservative extension of $\mathcal{O}_2$ modulo renaming *iff there exists some renaming* σ *s.t. for every axiom* α *using only names in* $\mathcal{O}_2$, $\mathcal{O}_1\sigma \models \alpha$ *iff* $\mathcal{O}_2 \models \alpha$.

Lemma 1. *There exists a polynomial function* p *s.t. for every* $n \in \mathbb{N}$, *there exists a polynomially computable function* π *from indices* $\{0, \dots, p(n)\}$ *to* $\mathcal{SROIQ}$ *axioms s.t. for every ontology* $\mathcal{O}$ *of size at most* n, *we can compute in time polynomial in* $\mathcal{O}$ *a set* $I \subseteq \{0, \dots, p(n)\}$ *s.t.* $\{\pi(i) \mid i \in I\}$ *is a conservative extension of* $\mathcal{O}$ *modulo renaming.*

Proof. Let $n \in \mathbb{N}$ be our bound on ontology sizes. Using standard structural transformations, we can compute for every $\mathcal{SROIQ}$ ontology $\mathcal{O}$ in polynomial time a *normalized ontology* that is a conservative of $\mathcal{O}$ and in which every axiom is of one of the following forms:

1. $A_1 \sqsubseteq \{a_1\}$,
2. $A_1 \sqcap A_2 \sqsubseteq A_3$,
3. $A_1 \sqsubseteq A_2 \sqcup A_3$,
4. $A_1 \sqsubseteq \leq i r_1.A_2$,
5. $\leq i r_1.A_1 \sqsubseteq A_2$,
6. $A_1 \sqsubseteq \exists r.\mathsf{Self}$,
7. $\exists r.\mathsf{Self}. \sqsubseteq A_1$,
8. $A_1(a_1)$,
9. $r_1(a_1, a_2)$,
10. $r_1 \sqsubseteq r_2$,
11. $r_1 \circ r_2 \sqsubseteq r_3$,
12. $\mathsf{Disj}(r_1, r_2)$,

where $A_1, A_2, A_3 \in \mathsf{N_C} \cup \{\top, \bot\}$, $r_1, r_2, r_3 \in \mathsf{N_R}$, $i \in \mathbb{N}$ and $a_1, a_2 \in \mathsf{N_I}$. Let m be the maximal size of any ontology that is the result of normalizing an ontology of size n. Note that m is polynomial in n. The number of concept names, role names and individuals occurring in any normalized ontology $\mathcal{O}$ of size m is bounded by m, and every number occurring in $\mathcal{O}$ is bounded by m as well. This means, we can pick a set

$$\mathbf{X} = \{X_1, \dots, X_m\} \subseteq \mathsf{N_C} \cup \{\top, \bot\} \cup \mathsf{N_R} \cup \mathsf{N_I}$$

s.t. for every any normalized ontology $\mathcal{O}$ of size m there exists a renaming σ s.t. $\mathcal{O}\sigma$ uses only names from $\mathbf{X}$. As the resulting ontology is still normalized, every such ontology uses only axioms from the set

$$\mathfrak{A} = \{\alpha \mid \alpha \text{ is normalized and uses only names from } \mathbf{X} \text{ and numbers } \leq m\}$$

As there are 12 axiom types, and each uses at most 3 names from $\mathbf{X}$ and one number, the number of axioms in $\mathfrak{A}$ is bounded by $p(n) = 12 \cdot |\mathbf{X}|^3 \cdot m = 12m^4$, which is polynomial in n because m is polynomial in n.

The function π is required can then for example defined as follows: for $i \in \{0, \dots, p(n)\}$, we define $\pi(i) = \alpha$, where

1. α is of the form ($i \mod 12$) as above,
2. for $a \in \{1, 3\}$, X_j is the ath name occurring in α, where $j = \lfloor \frac{i}{12m^{a-1}} \rfloor \mod m$, and
3. if α is of the form (4) or (5), i.e. it contains a number, then this number is $j = \lfloor \frac{i}{12m^3} \rfloor \mod m$

For every ontology $\mathcal{O}$, we obtain the index set I as required by first normalizing $\mathcal{O}$, renaming it accordingly, and then assigning the indices. □

Using an index function π as in Lemma 1, we can define a special type of oracle Turing machines called *(alternating) Turing machines with DL oracles*. Such a Turing machine uses an oracle tape with tape alphabet $\{0, 1, \not\emptyset\}$ ($\not\emptyset$ being the blank symbol), and is specified using an enumeration π of DL axioms as above, and some axiom α. Based on the current oracle tape content, we obtain an index set I that contains the number i iff the oracle tape contains a 1 at position i. The oracle then answers **yes** if $\{\pi(i) \mid i \in I\} \models \alpha$, and **no** otherwise. For a given DL $\mathcal{L}$, we may additionally require that the oracle only accepts inputs for which the corresponding ontology is in $\mathcal{L}$, in which case we call the Turing machine a *Turing machine with $\mathcal{L}$-oracle for π, α*.

Lemma 2. *Let k be a complexity class and $\mathcal{L}$ a DL for which deciding entailment is k-hard. Then, for every Turing machine T with k-oracle and bound n on the size of the oracle tape, we can construct in time polynomial in n and T a Turing machine T' with $\mathcal{L}$-oracle for some axiom enumeration π and CI α that accepts the same set of words with at most polynomial overhead.*

Proof. We first argue that it suffices to focus on entailment of axioms of the form $\alpha = A(a)$, since entailment of CIs can be reduced to it. Specifically, for any ontology $\mathcal{O}$ and CI $C \sqsubseteq D$, we have $\mathcal{O} \models C \sqsubseteq D$ iff $\mathcal{O} \cup \{C(a), D \sqsubseteq A\} \models A(a)$, where a and A do not occur in $\mathcal{O}$. Because entailment in $\mathcal{L}$ is k-hard, we can construct for every query to the k-oracle in polynomial time an ontology $\mathcal{O}$ that entails α iff the query should return **true**. Because the oracle tape is bounded, we can use Lemma 1 to translate that ontology into a corresponding index set to serve as input for the $\mathcal{L}$-oracle. T' thus proceeds as T, but before sending a query to the oracle, it reduces it to the entailment of the axiom α from an ontology $\mathcal{O}$, which it then translates to a query to the $\mathcal{L}$-oracle. □

A.2 Consistency-Independent Semantics

Theorem 1. *Let $\mathcal{L}$ be a DL such that deciding entailment in $\mathcal{L}$ is k-complete. Then, deciding consistency of $\mathcal{L}$-ontologized programs is* PSPACE^k*-complete, even for non-stochastic programs.*

Proof. Let $\mathbf{M}[\mathfrak{P}]$ be the MDP induced by some ontologized program $\mathfrak{P}$ as in Definition 3 where $\mathcal{K}$ is expressed in $\mathcal{L}$. For the upper bound, we specify a non-deterministic PSPACE algorithm that, starting from the initial state, explores all reachable states in $\mathbf{M}[\mathfrak{P}]$ and decides in k whether the current state is inconsistent. Specifically, for each current variable evaluation in $\mathfrak{P}$, we non-deterministically select a command in $\mathbf{P}_{\mathfrak{P}}$ whose guard is satisfied. Then, we non-deterministically select an update that has a positive probability in the distribution of the command and apply this update on the current evaluation. In each step, we use a k-oracle to determine whether the current state is consistent, and which hooks are active. As each state can be stored in polynomial space, the algorithm runs in PSPACE^k.

For the lower bound, we provide a polynomial reduction of the acceptance problem for polynomially space bounded, deterministic Turing machines T with k-oracle. Let

$$T = \langle Q, \Gamma, \gamma_0, \Gamma_i, \Gamma_o, q_0, F, \delta, q_?, q_+, q_- \rangle$$

be such a Turing machine, where

- Q are the *states*,
- $\Gamma = \{\gamma_0, \dots \gamma_m\}$ is the *tape alphabet*,
- γ_0 is the *blank symbol*,
- $\Gamma_i \subseteq \Gamma$ is the *input alphabet*,
- Γ_o is the *oracle tape alphabet*,
- q_0 is the *initial state*,
- $F \subseteq Q$ is the set of accepting states,
- $\delta\colon \big((Q \setminus (F \cup \{q_?\})) \times \Gamma \times \Gamma_o\big) \to \big(Q \times \Gamma \times \{-1, 0, +1\} \times \Gamma_o \times \{-1, 0, +1\}\big)$ is the transition relation, (which works on two tapes, the standard tape and the oracle tape),
- $q_? \in Q \setminus F$ is the *query state* to query the oracle, and
- $q_+, q_- \in Q \setminus F$ are the *query answer states*.

The Turing machine uses two polynomially bounded tapes: one standard tape and one oracle tape. In every state that is non-final and not the query state, δ defines the successor state of T. Whenever the current state is $q_?$, the oracle is invoked, leading to T entering q_+ in the next step if the oracle accepts the content of the oracle tape, and entering q_- otherwise. Based on T and the input word $w = \sigma_0 \dots \sigma_l$, we build an ontologized program $\mathfrak{P}_{T,w}$ s.t. $\mathfrak{P}_{T,w}$ is inconsistent iff T accepts w.

Since entailment in $\mathcal{L}$ is k-hard, we can use Lemma 2 to construct a polynomially space-bounded Turing machine T' with $\mathcal{L}$-oracle for $\pi, A(a)$ that accepts

w iff so does T. Specifically, we obtain a set $\mathcal{F}_q$ of DL axioms, polynomially bounded in $|w|$, a mapping $\pi\colon \{0,\dots,r(|w|)\} \mapsto \mathcal{F}_q$, where r is a polynomial,

$$T' = \langle Q', \Gamma, \gamma_0, \Gamma_i, \{0,1\}, q_0, F', \delta', q_?, q_+, q_- \rangle,$$

where $Q' = \{q_0,\dots,q_n\}$, and T' accepts the same language as T, is still polynomially space-bounded, but uses a different oracle. If T' is in state $q_?$, and the oracle tape contains the word $\sigma_0 \dots \sigma_{r(|w|)}$, the Turing machine moves into the state q_+ iff $\{\pi(i) \mid 0 \le i \le r(|w|), \sigma_i = 1\} \models A(a)$, and otherwise moves to q_-. Based on this Turing machine, we construct the ontologized program $\mathfrak{P}_{T,w}$.

Let p be a polynomial s.t. T' uses at most $p(|w|)$ tape cells on input w. The program uses the following variables:

- $\texttt{state} \in \{0,\dots,n\}$ stores the state of the Turing machine,
- for $0 \le i \le p(|w|)$, $\texttt{dtape\_i} \in \{0,\dots,m\}$ stores the letter on the default tape position i,
- for $0 \le i \le r(|w|)$, $\texttt{otape\_i} \in \{0,1\}$ stores the letter on the oracle tape at position i,
- $\texttt{dpos} \in \{0,\dots,p(|w|)\}$ stores the current position on the default tape,
- $\texttt{opos} \in \{0,\dots,r(|w|)\}$ stores the current position on the oracle tape.

We use a single hook $\mathcal{H} = \{\texttt{oracle\_yes}\}$ standing for the positive oracle answer. For every

$$\langle \langle q_{i_1}, \gamma_{i_2}, \gamma_{i_3} \rangle, \langle q_{i_4}, \gamma_{i_5}, \mathrm{Dir}_d, \gamma_{i_6}, \mathrm{Dir}_o \rangle \rangle \in \delta',$$

every $i \in \{0,\dots,p(|w|)\}$ and every $j \in \{0,\dots,r(|w|)\}$, we use the following non-stochastic command:

$$\begin{pmatrix} \texttt{state}{=}i_1 \\ \wedge \texttt{dpos}{=}i \ \wedge \texttt{dtape\_i}{=}i_2 \\ \wedge \texttt{opos}{=}j \ \wedge \texttt{otape\_j}{=}i_3 \end{pmatrix} \mapsto \begin{pmatrix} \texttt{state} := i_4 \\ \texttt{dtape\_i} := i_5, \texttt{dpos} := \texttt{dpos} + \mathrm{Dir}_d, \\ \texttt{otape\_j} := i_6, \texttt{opos} := \texttt{opos} + \mathrm{Dir}_o \end{pmatrix} \tag{3}$$

To handle the behavior of the oracle, we further add the following commands, where $q_? = q_{i_1}$, $q_+ = q_{i_2}$ and $q_- = q_{i_3}$

$$\texttt{state} = i_1 \wedge \texttt{oracle\_yes} \quad \mapsto \quad \texttt{state} := i_2 \tag{4}$$

$$\texttt{state} = i_1 \wedge \neg\texttt{oracle\_yes} \quad \mapsto \quad \texttt{state} := i_3 \tag{5}$$

The initial state $\eta_{\mathfrak{P},0}$ of the program is the evaluation $\texttt{state} = 0$, $\texttt{dtape\_i} = w_i$ for $i \in \{0,\dots,l\}$, $\texttt{dtape\_i} = 0$ for $i \in \{l,\dots,p(|w|)\}$, $\texttt{otape\_i} = 0$ for $i \in \{0,\dots,r(|w|)\}$, $\texttt{dpos} = \texttt{otape} = 0$. This completes the definition of the program component $\mathbf{P}$ in $\mathfrak{P}_{T,w}$.

As DL axioms, we set for the static DL knowledge $\mathcal{K} = \emptyset$ and for the fluents $\mathcal{F} = \mathcal{F}_q \cup \{\top \sqsubseteq \bot\}$. It remains to specify the interface functions Dc and cD. For Dc, we set for every $i \in \{0,\dots,r(|w|)\}$,

$$\mathsf{Dc}(\pi(i)) = \big(\texttt{otape\_i} = 1\big)$$

to translate the oracle calls, and

$$\mathsf{Dc}(\top \sqsubseteq \bot) = \left(\bigvee_{q_i \in F} \mathtt{state} = i \right),$$

to encode that the ontologized states that correspond to accepting states in the Turing machine are inconsistent. Finally, we specify cD by setting

$$\mathsf{cD}(\mathtt{oracle\_yes}) = \{A(a)\}.$$

It is standard to verify that the Turing machine T' accepts w iff the ontologized program $\mathfrak{P}_{T,w}$ is inconsistent, i.e., can reach an inconsistent state. Since T accepts w iff so does T', this completes the reduction. □

A.3 Probability-Normalizing Semantics

Theorem 2. *Let $\mathcal{L}$ be a DL for which deciding entailment has complexity k. Then, deciding whether an $\mathcal{L}$-ontologized program is probability-normalizable is* PSPACE^k*-complete, even for non-stochastic programs.*

Proof. For the upper bound, note that the only way for $\mathfrak{P}$ to be inconsistent under probability-normalizing semantics is if all probabilistic choices would lead to an inconsistent state in the $\mathbf{M}[\mathfrak{P}]$. We can thus use a non-deterministic PSPACE^k-algorithm similar to the one described in the proof for Theorem 1 to determine whether a program *is* probability-normalizable: instead of testing for inconsistency of the current state, we here test for consistency. Note that it is sufficient to find a single execution path that never reaches an inconsistent state for an ontologized program to be probability-normalizable. This execution path might be infinite, however, because the state space is bounded, any infinite execution path would need to visit a state twice. We use a counter to put an exponential bound on the states to be visited, and store a non-deterministically selected state for later comparison. If this state repeats, we found an unbounded execution path that never visits an inconsistent state, and accept. If we reach the bound, such a loop was not found and we reject.

For the lower bound, we note that the reduction used in the proof for Theorem 1 can also be used for probability-normalizability. In that reduction, we reduce the word-problem of deterministic polynomially-space bounded Turing machines with k-oracle to consistency of ontologized programs under consistency-independent semantics. Inspection of the construction reveals that the constructed program $\mathfrak{P}_{T,w}$ is not only non-stochastic, but even *deterministic*, by which we mean that in every ontologized state, there is at most one command that can be executed. Specifically, there is exactly one Command (3) in the proof of Theorem 1 for each assignment to the variables `state`, `dpos`, `opos`, `dtape_i` and `otape_j`, which means that in each ontologized state, at most one of these commands is executable. Moreover, for the assignment $\mathtt{state} = i_1$, where $q_{i_1} = q_?$ denotes the oracle query state, there is no such command, as there

is no transition in the Turing machine for those states, and exactly one of the commands (4) and (5) is executable if $\texttt{state} = i_1$, unless if the state is inconsistent. We obtain that, if some path in $\mathbf{M}[\mathfrak{P}_{T,w}]$ leads from the initial state to an inconsistent state, then all paths lead to an inconsistent state, in which case the program is not probability-normalizable. Consequently, $\mathfrak{P}_{T,w}$ is probability-normalizable iff $\mathfrak{P}_{T,w}$ is consistent, which is the case iff T does not accept w. Since non-acceptance of words in polynomially space-bounded Turing-machines with k-oracle is also PSPACE^k-complete, we obtain that probability-normalizability is PSPACE^k-hard. □

A.4 Probability-Preserving Semantics

Theorem 3. *Let $\mathcal{L}$ be a DL for which deciding entailment has complexity k. Then, deciding whether an $\mathcal{L}$-ontologized program is probability-preservable is* APSPACE^k*-complete. For non-stochastic programs, it is* PSPACE^k*-complete.*

Proof. Both bounds are via reductions from and to alternating Turing machines. In an alternating Turing machine, we switch between ∃-non-determinism and ∀-non-determinism, meaning that the Turing machine can switch between ∃-non-deterministic choices and ∀-non-deterministic transitions. Acceptance is then defined inductively according to the choices involved: a configuration is accepting if 1) it is in an accepting state, 2) there exists an ∃-non-deterministic transition to a configuration that is accepting, or 3) all ∀-non-deterministic transitions lead to an accepting configuration.

For the upper bounds, we use a modification of the algorithm used in the proof for Theorem 1 that runs in alternating polynomial space and decides probability-preservability of ontologized programs. Let $\mathfrak{P}$ be an ontologized program as in Definition 1, and $\mathbf{M}_{ci}[\mathfrak{P}] = \langle Q, Act, P, \iota, \Lambda, \lambda, wgt\rangle$ the MDP induced by $\mathfrak{P}$ under consistency-independent semantics. The algorithm guesses the states $q \in Q$ to be included in the probability-preserving MDP one after the other, starting from ι, where we use alternation to switch between the non-deterministic and the probabilistic choices of the program. Specifically, we have to guess a set of states Q' s.t.: 1) no state in Q' is inconsistent, 2) for every Q' there is always some guarded command that can be executed on the current state, and 3) all successor-states into which this command would bring us with a positive probability are included in Q'. Our algorithm thus proceeds as follows based on the currently guessed state. If the current state is inconsistent, we reject. Otherwise, we non-deterministically select a guarded command $\langle g, \sigma\rangle \in C_{\mathfrak{P}}$ s.t. $q \models g$ (∃-non-determinism) and continue on all states $q' \in Q$ such that $P(q, \sigma, q') > 0$, $q' \in Q'$ (∀-non-determinism). Each state takes polynomial space, and we use a k-oracle to determine which hooks are active and whether the state is consistent. Since this procedure can be implemented by an alternating Turing machine with k-oracle that uses polynomial space, probability-preservability can be decided in APSPACE^k. If the program is non-stochastic, the ∀-non-determinism is not needed, and the algorithm runs in PSPACE^k.

For the lower bound, we adapt the reduction used in the proof for Theorem 1. However, this time, we reduce the word acceptance problem for polynomially space bounded *alternating* Turing machines with k-oracle. Specifically, let

$$T = \langle Q, \Gamma, \gamma_0, \Gamma_i, \Gamma_o, q_0, F, \delta, q_?, q_+, q_-, g \rangle,$$

be such a Turing machine, where $Q = \{q_0, \dots, q_n\}$, $\Gamma = \{\gamma_0, \dots \gamma_n\}$ is the standard tape alphabet, $\gamma_0 \in \Gamma_i$ is the blank symbol, $\Gamma_i \subseteq \Gamma$ is the input alphabet, Γ_o is the oracle alphabet, q_0 is the initial state, $F \subseteq Q_\wedge \cup Q_\vee$ is the set of accepting states,

$$\delta \colon (Q \setminus (F \cup \{q_?\})) \times \Gamma \times \Gamma_o \to \wp(Q \times \Gamma \times \{-1, +1\} \times \Gamma_o \cup \{-1, +1\})$$

is the transition function, the states $q_?$, q_+ $q_- \in Q$ manage the querying of the oracle, and $g \colon Q \to \{\forall, \exists, \texttt{accept}, \texttt{reject}\}$ partitions the states Q into four categories of universal and existential quantified states, and accepting and rejecting states, respectively. The definition of acceptance is standard, specifically, the oracle works as in the proof for Theorem 1, and a configuration with state q is *accepting* 1) if $g(q) = \texttt{accept}$, 2) $g(q) = \exists$ and one of the successor configurations is accepting, or 3) $g(q) = \forall$ and all successor configurations are accepting.

Again, we use Lemma 2 to construct, based on T and the input word w, a polynomially bounded set $\mathcal{F}_q$ of DL axioms, a mapping $\pi \colon \mathcal{F}_q \to \{0, \dots, r(|w|)\}$, where r is a polynomial, and an alternating Turing machine

$$T' = \langle Q', \Gamma, \gamma_0, \Gamma_i, \{0, 1, \not{b}\}, q_0, F', \delta', q_?, q_+, q_-, g' \rangle$$

with an $\mathcal{L}$-oracle for $\pi, A \sqsubseteq B$, that is polynomially space-bounded, and accepts w iff so does T'. If T' is in state $q_?$, and the oracle tape contains the word $\sigma_0 \dots \sigma_{r(|w|)}$, the Turing machines moves into the state q_+ iff

$$\left\{ \pi(i) \mid 0 \leq i \leq r(|w|), \sigma_i = 1 \right\} \models A(a),$$

and otherwise moves to q_-. Based on this Turing machine, we construct the ontologized program $\mathfrak{P}_{T,w}$ in the same way as in the proof for Theorem 1, but with a different set $C_\mathfrak{P}$ of commands.

We first model the $\forall$-transitions. For every

$$t = \langle q_{i_1}, \gamma_{i_2}, \gamma_{i_3} \rangle \in (Q \setminus (F \cup \{q_?, q_+, q_-\})) \times \Gamma \times \Gamma_o,$$

where $g'(q_{i_1}) = \forall$, every $\langle q_{i_4}, \gamma_{i_5}, \mathrm{Dir}_d, \gamma_{i_6}, \mathrm{Dir}_o \rangle \in \delta'(t)$, every $i \in \{0, \dots, p(|w|)\}$ and every $j \in \{0, \dots, r(|w|)\}$, we use the command

$$\begin{pmatrix} \texttt{state} = i_1 & & \\ \wedge\, \texttt{dpos} = i & \wedge\, \texttt{dtape\_i} = i_2 & \\ \wedge\, \texttt{opos} = j & \wedge\, \texttt{otape\_j} = i_3 & \end{pmatrix} \mapsto \begin{pmatrix} \texttt{state} := i_4 \\ \texttt{dtape\_i} := i_5, \texttt{dpos} := \texttt{dpos} + \mathrm{Dir}_d, \\ \texttt{otape\_j} := i_6, \texttt{opos} := \texttt{opos} + \mathrm{Dir}_o \end{pmatrix}$$

Note that for the program to be *not* probability preservable, every command that we can execute in an ontologized state leads to an inconsistency, which is

why we model ∀-transitions in this way. For the ∃-transitions, we make use of the stochastic component. Let

$$t_1 = \langle q_{i_1}, \gamma_{i_2}, \gamma_{i_3} \rangle \in (Q \setminus (F \cup \{q_?, q_+, q_-\})) \times \Gamma \times \Gamma_o,$$

where $g'(q_{i_1}) = \exists$. For every $i \in \{0, \dots, p(|w|)\}$ and $j \in \{0, \dots, r(|w|)\}$, we add the stochastic command

$$\begin{pmatrix} \mathtt{state} = i_1 & & \\ \wedge\, \mathtt{dpos} = i & \wedge\, \mathtt{dtape\_i} = i_2 \\ \wedge\, \mathtt{opos} = j & \wedge\, \mathtt{otape\_j} = i_3 \end{pmatrix} \;\mapsto\; \sigma$$

where σ is the stochastic update such that for every

$$t_2 = \langle q_{i_4}, \gamma_{i_5}, \mathrm{Dir}_d, \gamma_{i_6}, \mathrm{Dir}_o \rangle \in \delta'(t_1),$$

we have

$$\sigma \begin{pmatrix} \mathtt{state} := i_4 \\ \mathtt{dtape\_i} := i_5, \mathtt{dpos} := \mathtt{dpos} + \mathrm{Dir}_d, \\ \mathtt{otape\_j} := i_6,\ \mathtt{opos} := \mathtt{opos} + \mathrm{Dir}_o \end{pmatrix} = \frac{1}{|\delta'(t_1)|}.$$

The corresponding transitions are excluded from the probability-preserving MDP if there is a positive probability of getting into an inconsistent state. To this end, we can model ∃-transitions in this way.

The stochastic commands that handle the use of the oracle are as in the proof for Theorem 1. It is standard to show that the resulting program is probability-preserving iff w is accepted by the Turing machine, so that we obtain that probability-preservability is APSPACE^k-hard.

Now assume that for no state $q \in Q$, $g'(q) = \exists$. T then corresponds to a coNPSPACE^k-machine, and the ontologized program $\mathfrak{P}_{T,w}$ is non-stochastic. As a consequence, deciding whether non-stochastic programs are probability-preserving is coNPSPACE^k = PSPACE^k-hard. □