Under consideration for publication in Formal Aspects of Computing

Enhancing Probabilistic Model Checking with Ontologies

Clemens Dubslaff, Patrick Koopmann, and Anni-Yasmin Turhan
Technische Universität Dresden, Dresden, Germany

Abstract. Probabilistic model checking (PMC) is a well-established method for the quantitative analysis of state-based operational models such as Markov decision processes. Description logics (DLs) provide a well-suited formalism to describe and reason about knowledge and are used as basis for the web ontology language (OWL). We investigate how such knowledge described by DLs can be integrated into the PMC process, introducing *ontology-mediated PMC*. Specifically, we propose *ontologized programs* as a formalism that links ontologies to behaviors specified by probabilistic guarded commands, the de-facto standard input formalism for PMC tools such as Prism. Through DL reasoning, inconsistent states in the modeled system can be detected. We present three ways to resolve these inconsistencies, leading to different Markov decision process semantics. We analyze the computational complexity of checking whether an ontologized program is consistent under these semantics. Further, we present and implement a technique for the quantitative analysis of ontologized programs relying on standard DL reasoning and PMC tools. This way, we enable the application of PMC techniques to analyze knowledge-intensive systems. We evaluate our approach and implementation on a multi-server system case study, where different DL ontologies are used to provide specifications of different server platforms and situations the system is executed in.

Keywords: Probabilistic Model Checking, Ontologies, Description Logics, Ontology-Mediated Verification, Context-dependent Systems Analysis

1. Introduction

Probabilistic model checking (PMC) is an automated technique for the quantitative analysis of systems (see for example [BK08, FKNP11] for surveys). PMC has been successfully applied in many areas to verify non-functional quality requirements such as energy efficiency or low probabilities of failure. *Probabilistic guarded command (PGC) programs* rely on a probabilistic variant of Dijkstra's guarded command language [Dij76,

Correspondence and offprint requests to: Clemens Dubslaff `clemens.dubslaff@tu-dresden.de`, Patrick Koopmann `patrick.koopmann@tu-dresden.de`, or Anni-Yasmin Turhan `anni-yasmin.turhan@tu-dresden.de`. 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).

JSM97] and provide the de-facto standard specification formalism for PMC used in many PMC tools such as Prism [KNP11] and Storm [DJKV17].

Usually, the behavior described by a PGC program is part of a larger system, or might even be used within a collection of systems that have an impact on the operational behavior. There are different ways how to model such distributed systems with PGC programs. First, we could integrate additional knowledge about the surrounding system directly into the program. While this approach benefits from the full expressiveness of PGC programs, it has the disadvantage that guarded command languages are not well-suited for describing complex knowledge about the contexts. Alternatively, we could use the concept of nondeterminism to over-approximate the possible behaviors of the surrounding system. This approach, however, leads to a coarse model that does not fully specify the impact of the context onto the system described by PGC programs. When it comes to verifying the overall system, a best- and worst-case PMC analysis might then not allow for meaningful interpretation.

In this paper, we propose a third option, an approach we call *ontology-mediated PMC* where we separate the specification of the operational behavior of a system from the specification of the additional knowledge that influences the behavior. This allows us to use different, specialized formalisms for describing the operational behavior and knowledge about the surrounding system. As knowledge description formalism, we use the well-established family of *description logics (DLs)*. DLs are fragments of first-order logic balancing expressivity and decidability [BCM$^+$03, BHLS17]. While the worst-case complexity for DL reasoning can be very high, modern optimized systems often allow to reason about very large knowledge bases in short time [PMG$^+$17]. Logical theories formulated in a DL are called *ontologies*, hence the naming of ontology-mediated PMC: knowledge expressed in ontologies mediate the PMC process.

The core of our approach relies on *ontologized PGC programs*, which formalize the idea of completely separating concerns to describe operational behavior by PGC programs and knowledge by ontologies. To specify the interplay between both formalisms, we use the concept of an *interface*. The interface mediates between these two worlds by providing mappings from PGC statements to DL statements and vice versa. Specifically, ontologized programs are PGC programs that use *hooks* to refer to the ontology within the behavior description. These hooks are linked to DL expressions via the interface. The loose coupling achieved in this way allows us to reuse and replace both components easily. PGC programs then can be analyzed with different ontologies (to describe different system configurations and contexts), and the same ontology can be used with different programs (for example, to describe different implementations of the behavior to be analyzed while keeping the same context). This design distinguishes our approach from other approaches that use DLs for the verification of systems, in which DL constructs are used directly within the program [BZ13, CDGLR11, HCM$^+$13, ZC15].

Let us now briefly summarize our contributions before we explain our approach of ontology-mediated verification more in detail. Besides introducing ontologized programs and methods for their analysis through PMC, this paper makes the following contributions:

(1) Three semantics of ontologized programs that differ in the treatment of inconsistent states,
(2) complexity results for checking consistency of ontologized programs under these semantics,
(3) a practical method to generate standard PGC programs from ontologized programs using DL reasoning,
(4) an implementation of the transformation for the quantitative analysis with PMC, and
(5) a case study and evaluation of the implementation on a multi-server system model.

Here, we rely on the semantical framework introduced in [DKT20] especially for the contributions (1) and (2). For (3) and (4), we extend the approach of [DKT19] by providing full proofs, a slightly more general formalisms for ontologized programs with weights, and introduce a formal notion of *mediators* that enable more elegant and clean proofs. Our evaluation case study (5) is extended towards more analyses than discussed in [DKT19].

Semantics of ontologized programs. The presented semantics of ontologized programs are all formalized in terms of *Markov decision processes (MDPs, cf. [Put94])*, while other state-based operational semantics of PGC programs are also imaginable. They follow a product construction of the MDP-semantics of the PGC program and those DL ontologies that are compatible with states of the program. In theory, this allows states to be associated with an ontology that is logically inconsistent, so that we have to treat *inconsistent states*. Consider the following example: A program compiled for a specific processor architecture might not run properly on a different architecture. But future architectures might not be known during program specification such that after a new architecture release, an execution of the program might run into inconsistent states with respect to the architecture properties specified in the ontology. According to our first semantics, called *consistency-independent semantics*, there is no special treatment of such inconsistencies and the resulting MDP may contain inconsistent states. We used this semantics initially as the basis for our definition of ontology-mediated PMC given in [DKT19]. However, consequences drawn by classical DL reasoning on inconsistent ontologies are rarely useful: The only valuable information we can infer here is the inconsistency itself. Thus, the task of resolving the inconsistency is left to the program component under consistency-independent semantics, e.g., by implementing an explicit notion of exception handling in the program.

While in some instances, handling inconsistencies by the program might be sufficient, in other situations it is more convenient to let the DL component directly control the state space of the program by using a semantics that excludes inconsistent states altogether. For example, inconsistencies that would appear under the consistency-independent semantics could also be a result of underspecification of the program component, where the model of the program relies on further knowledge. In our architecture example, there could be an external component that only allows for executing programs on a compatible architecture. Or there might be actions only partly enabled for the user, but modeled through nondeterminism in the program. Then, a semantics that rules out states with an inconsistent ontology is favorable. For this, we introduce *consistency notions* for ontologized programs and present in addition to the consistency-independent semantics two further semantics. These are the *probability-normalizing* and *probability-preserving semantics* [DKT20] that differ in the treatment of inconsistent states. Probability-normalizing semantics assumes that stochastic behaviors in the program component arise within the program itself, e.g., to break symmetries as done within randomized algorithms. Here, it is natural to redistribute the probability mass that leads to inconsistent states to consistent ones. We achieve this by normalizing the distributions of probabilistic choices in the program. Probability-preserving semantics assumes that probabilistic choices have their origin in the surrounding system, e.g., to stochastically model environmental influences. Here, we strictly disallow any execution that may reach inconsistent states by pruning nondeterministic choices. For all of the three semantics we present complexity results for deciding consistency of programs.

Practical analysis of ontologized programs. To be able to analyze ontologized programs practically, we develop a two-step procedure. First, the ontologized program and the properties to be analyzed are rewritten into a plain PGC program and modified versions of the properties. This way, essentially all model-checking tasks supported by state-of-the-art PMC tools such as Prism [KNP11] can be applied for analyzing ontologized programs. These tasks range from verifying standard statements about probabilities and expectations to the more advanced quantitative analysis such as presented in [BDK+14, KBC+18]. We first approach our rewrite method formally by defining what properties a rewrite function should satisfy to allow for a reduction. Then, we present two rewrite functions: one that is simple and easy to understand but leads to inefficient rewritings, and one that is efficient. Technically, our rewriting replaces expressions in the ontologized program by expressions that are derived using a DL reasoner. To allow for concise rewritings that can be computed efficiently, it is important to minimize the amount of DL reasoning. To this end, we use a technique that concentrates on the relevant axioms called *axiom pinpointing.*

Implementation and evaluation. Then, we turn to our practical implementation of this procedure in which the operational behavior is specified in the input language of Prism, and where the ontology is given in *web ontology language (OWL)* [MCH+09]. Since our approach is independent of any particular DL, the implementation supports any OWL fragment for which a DL reasoner exists. Technically, we transform the hooks from an ontologized program into expressions derived from reasoning on its ontology.

We evaluate our implementation based on a heterogeneous multi-server scenario and show that our approach facilitates the analysis of knowledge-intensive systems when varying behavior and ontology. In the case study, we consider multiple servers with architecture compatibility constraints running various job-executing processes. The job-scheduling protocol that decides which process is executed next is modeled through the program component. Here, we model two variants: one randomized algorithm and one that selects the next process in a round-robin fashion. Specific characteristics of the processes and servers are modeled in the ontology component, where we consider different contexts the program runs in. Processes may have different priorities depending on the importance of the job to be executed. An inconsistent system state occurs when a process is running on a server with different architecture compatibility. We resolve these inconsistencies explicitly in the program component, following ideas for resolving inconsistencies in the probability-normalizing and probability-preserving semantics. We analyze several instances of the multi-server system, arising from the two job-scheduling protocols, different server configurations, and four different contexts modeled by ontologies. Our focus of the analysis is to compute probabilities of running into certain critical situations specified by the ontologies, as well as an energy-utility analysis [BDD$^+$14] to investigate the trade-off between the number of successfully processed jobs and the consumed energy.

2. Preliminaries

We recall well-known notions and formalisms from probabilistic model checking and description logics required to ensure a self-contained presentation throughout the paper. By $\mathbb{Q}$, $\mathbb{Z}$, and $\mathbb{N}$ we denote the set of rationals, integers, and nonnegative integers, respectively. Let X be a countable set. We denote by $\wp(X)$ the powerset of X. A (rational) probability *distribution* over X is a function $\mu\colon X \to [0,1] \cap \mathbb{Q}$ with $\sum_{x\in X} \mu(x) = 1$. The set of distributions over X is denoted by $\mathit{Distr}(X)$.

2.1. Markov Decision Processes

The operational model used in this paper is given in terms of *Markov decision processes (MDPs)* (see, e.g., [Put94]). MDPs are tuples $\mathbf{M} = \langle Q, \mathit{Act}, P, \iota, \Lambda, \lambda\rangle$ where Q and Act are countable sets of *states* and *actions*, respectively, $P\colon Q \times \mathit{Act} \rightharpoonup \mathit{Distr}(Q)$ is a partial probabilistic transition function, $\iota \in Q$ an initial state, and Λ a set of labels assigned to states via the labeling function $\lambda\colon Q \to \wp(\Lambda)$. An action $\alpha \in \mathit{Act}$ is *enabled* in a state $q \in Q$ if $P(q,\alpha)$ is defined. The set of all enabled actions in $q \in Q$ of $\mathbf{M}$ is denoted by $\mathit{En}_{\mathbf{M}}(q)$. As usual, we assume that $\mathbf{M}$ does not have final states, i.e., $\mathit{En}_{\mathbf{M}}(q) \neq \varnothing$ for all $q \in Q$. 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 the product of its transition probabilities, i.e., $\Pr(\pi) = p_0 \cdot p_1 \cdot \ldots \cdot p_{k-1}$. Infinite paths are defined accordingly, i.e., as sequences $\pi = q_0\,\alpha_0\,q_1\,\alpha_1\,\ldots$ where $P(q_i,\alpha_i,q_{i+1}) > 0$ for all $i \in \mathbb{N}$. A state $q' \in Q$ is *reachable from* $q \in Q$ if there exists a path π in $\mathbf{M}$ as above for which $q_0 = q$ and there is an index j with $q_j = q'$.

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 $P(q,\alpha,q')$. A probability measure over sets of infinite paths is defined relying on the concept of *(randomized) schedulers*. Schedulers are functions $\mathfrak{S}$ that map any finite path to a distribution over actions. We require that for each finite path π in $\mathbf{M}$ ending in a state q that $P(q,\alpha)$ is defined for any $\alpha \in \mathit{Act}$ with $\mathfrak{S}(\pi)(\alpha) > 0$. A path $\pi = q_0\alpha_0q_1\ldots$ is an $\mathfrak{S}$*-path* if for all $i \in \mathbb{N}$ for which α_i is defined in π we have that $\mathfrak{S}(q_0\alpha_0\ldots q_i)(\alpha_i) > 0$. By the use of schedulers $\mathfrak{S}$, a probability measure $\Pr^{\mathfrak{S}}_{\mathbf{M},q}$ over infinite $\mathfrak{S}$-paths that start in q is defined in the standard way (cf. [Put94]).

Amending $\mathbf{M}$ with a *weight function* $\mathit{wgt}\colon Q \to \mathbb{N}$ turns $\mathbf{M}$ into a *weighted MDP* $\langle \mathbf{M}, \mathit{wgt}\rangle$. The weight of a finite path $\pi = q_0\,\alpha_0\,q_1\ldots q_k$ is defined as $\mathit{wgt}(\pi) = \sum_{i\leq k} \mathit{wgt}(q_i)$. For a scheduler $\mathfrak{S}$ we define by $\mathrm{Exp}[\mathit{wgt}]^{\mathfrak{S}}_{\mathbf{M}}$ the function that maps a measurable set of infinite $\mathfrak{S}$-paths to the expected value of the weight of these paths.

2.2. Probabilistic Model Checking

MDPs are suitable for a quantitative analysis using *probabilistic model checking (PMC, cf. [BK08, FKNP11])*. A property to be analyzed is usually defined using temporal logics over the set of labels, constituting a set of maximal paths for which the property is fulfilled after the resolution of nondeterministic choices. By ranging over all possible resolutions of nondeterminism, this enables a best- and worst-case analysis on the property. Standard analysis tasks ask, e.g., for the minimal and maximal probability of a given property, or the expected weight reaching a given set of states.

Quantitative properties. Let ϕ be a property expressed in a temporal logic over the set of labels Λ, defining a set of maximal paths in the MDP that fulfill ϕ. For the purpose of this paper, it suffices to consider path properties specified in *linear temporal logic (LTL, [Pnu77])*, which we extend with a dedicated operator to express reachability with constraints on the accumulated weight. Specifically, an LTL formula over a set of labels Λ is an expression of the grammar

$$\varphi ::= \mathtt{tt} \mid \ell \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathsf{X}\varphi \mid \varphi \mathsf{U} \varphi \mid \varphi \mathsf{U}^{wgt \bowtie \tau} \varphi$$

where ℓ ranges over Λ, $\bowtie \in \{<, =, >, \geq, \neq, \leq\}$, and $\tau \in \mathbb{N}$. Here, $\mathtt{tt}$ stands for the formula that is always satisfied, X for the *next operator*, and U for the *until operator.* We may also use LTL formulas with the common syntactic abbreviations, e.g., $\varphi \vee \psi \equiv \neg(\neg\varphi \wedge \neg\psi)$, the *eventually operator* $\Diamond\varphi \equiv \mathtt{tt}\mathsf{U}\varphi$, or the *globally operator* $\Box\varphi \equiv \neg\Diamond\neg\varphi$. The set of paths reaching states labeled by a particular $\ell \in \Lambda$ could be, e.g., formalized in LTL by $\varphi_1 = \Diamond\ell$. Similarly, the set of paths where always when such a state is reached, within two time steps a state not labeled by ℓ is reached again, could be formalized by $\varphi_2 = \Box\big(\ell \rightarrow (\mathsf{X}\neg\ell \vee \mathsf{X}\mathsf{X}\neg\ell)\big)$.

The semantics $[\![\varphi]\!]$ of an LTL formula φ in a weighted MDP $\mathbf{M} = \langle Q, \mathit{Act}, P, \iota, \Lambda, \lambda, \mathit{wgt}\rangle$ is recursively defined as the set of infinite paths $\pi = q_0\alpha_0q_1\ldots$ where

$$\begin{array}{rll}
\pi \in [\![\mathtt{tt}]\!] & & \\
\pi \in [\![\ell]\!] & \text{iff} & \ell \in \lambda(q_0) \\
\pi \in [\![\neg\varphi]\!] & \text{iff} & \pi \notin [\![\varphi]\!] \\
\pi \in [\![\varphi \wedge \psi]\!] & \text{iff} & \pi \in [\![\varphi]\!] \cap [\![\psi]\!] \\
\pi \in [\![\mathsf{X}\varphi]\!] & \text{iff} & q_1\alpha_1\ldots \in [\![\varphi]\!] \\
\pi \in [\![\varphi\mathsf{U}\psi]\!] & \text{iff} & \text{there is } j \in \mathbb{N} \text{ with } q_j\alpha_j\ldots \in [\![\psi]\!] \text{ and } q_i\alpha_i\ldots \in [\![\varphi]\!] \text{ for all } i < j \\
\pi \in [\![\varphi\mathsf{U}^{wgt\bowtie\tau}\psi]\!] & \text{iff} & \text{there is } j \in \mathbb{N} \text{ with } q_j\alpha_j\ldots \in [\![\psi]\!], q_i\alpha_i\ldots \in [\![\varphi]\!] \text{ for all } i < j, \text{ and } \mathit{wgt}(q_0\alpha_0\ldots q_j) \bowtie \tau
\end{array}$$

For a scheduler $\mathfrak{S}$, any LTL property constitutes a measurable set of $\mathfrak{S}$-paths [Var85] that start in the initial state ι. To this end, we can reason about the probability $\Pr^{\mathfrak{S}}_{\mathbf{M}}(\varphi)$ 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 consideration on the probabilities. A *probability property* Φ is an expression $\mathsf{P}^{\mathrm{ex}}(\varphi) \bowtie \theta$ where φ is an LTL formula, $\theta \in \mathbb{Q} \cap [0,1]$ a threshold, $\bowtie \in \{<, =, >, \geq, \neq, \leq\}$ a relation, and $\mathrm{ex} \in \{\min, \max\}$. Then, $\mathbf{M}$ *satisfies* Φ, denoted $\mathbf{M} \models \Phi$, iff $\mathrm{ex} = \min$ and $\inf_{\mathfrak{S}} \Pr^{\mathfrak{S}}_{\mathbf{M}}(\varphi) \bowtie \theta$ or $\mathrm{ex} = \max$ and $\sup_{\mathfrak{S}} \Pr^{\mathfrak{S}}_{\mathbf{M}}(\varphi) \bowtie \theta$. Likewise, Φ is an *expectation property* if Φ is of the form $\mathsf{E}[\mathit{wgt}]^{\mathrm{ex}}(\varphi) \bowtie \theta$ where φ is an LTL formula and θ, $\bowtie$, and ex are as within probability properties. For expectation properties, $\mathbf{M}$ *satisfies* Φ, denoted $\mathbf{M} \models \Phi$, iff $\mathrm{ex} = \min$ and $\inf_{\mathfrak{S}} \mathrm{Exp}[\mathit{wgt}]^{\mathfrak{S}}_{\mathbf{M}}\big(\varphi\big) \bowtie \theta$ or $\mathrm{ex} = \max$ and $\sup_{\mathfrak{S}} \mathrm{Exp}[\mathit{wgt}]^{\mathfrak{S}}_{\mathbf{M}}\big(\varphi\big) \bowtie \theta$. The *PMC problem* for Φ and $\mathbf{M}$ amounts to decide whether $\mathbf{M} \models \Phi$.

To reason about cost-utility trade-offs in weighted MDPs, we rely on *energy-utility quantiles* [BDD⁺14]. Formalized in our theoretical framework, we have given a weighted MDP where its weight function is non-negative and integer-valued and stands for costs invested, e.g., energy consumed. Then, for a probability threshold $\theta \in [0,1] \cap \mathbb{Q}$, and LTL formulas φ and ψ, the *min-cost quantile* with respect to θ, φ, and ψ is defined as

$$\min\big\{\, c \in \mathbb{N} \mid \mathbf{M} \models \mathsf{P}^{\max}(\varphi\mathsf{U}^{wgt\leq c}\psi) > \theta \,\big\} \quad . \tag{1}$$

The above quantile specifies the minimal cost budget required such that there is a scheduler under which the probability to reach some state in which ψ holds through states where φ holds is greater than θ, not exceeding the cost budget. Min-cost quantiles can be used to reason about cost-utility trade-offs in weighted MDPs by additionally encoding a utility value into the state space of the MDP. This additional value then formalizes the utility gained (e.g., number of tasks completed) during an execution of the MDP. LTL formulas that include arithmetic constraints on the utility value as labelings then can put utility constraints on the paths over which the costs are minimized in the min-cost quantile. As an example, a constraint $\mathit{utility} \geq 20$ would label all states with utility value at least 20.

Quantitative queries. While classical PMC tasks for MDPs are to decide whether a probabilistic property or expectation property is satisfied in the given (weighted) MDP, the computational task to determine the value of probabilities and expectations to fulfill an LTL property is also of interest. To this end, we define a *probability query* to be an expression of the form $\mathsf{P}^{\mathrm{ex}}(\varphi)$=? and an *expectation query* to be an expression of the form $\mathsf{E}[\mathit{wgt}]^{\mathrm{ex}}(\varphi)$=?, where φ is an LTL formula over the same label set as for the given (weighted) MDP and $\mathrm{ex} \in \{\min, \max\}$. Queries of the above form can be answered by using standard methods like value iteration or policy iteration. For more details on MDPs and PMC, we refer to standard textbooks and surveys such as [Put94, BK08, FKNP11].

To specify computational tasks of min-cost quantiles, we define *quantile queries* to be expressions of the form $\mathsf{Qu}_{>\theta}(\varphi \mathsf{U}^{\leq wgt}\psi)$=? where $\theta \in \mathbb{Q} \cap [0,1]$ and φ and ψ are LTL formulas. Answering a quantile query amounts to compute the value of the corresponding quantile, e.g., for the above query computing (1).

2.3. Probabilistic Guarded Command Programs

As a concise and natural representation of MDPs we use *probabilistic guarded command (PGC) programs*, which are essentially expressed in a probabilistic variant of Dijkstra's guarded command language [Dij76, JSM97]. Our PGC programs are compatible to the input language of the probabilistic model checker system PRISM [KNP11]. In such a program, initial integer values for variables are explicitly given, while the changes of values for variables are provided through commands of the form

$$\texttt{[action] guard} \rightarrow \texttt{p}_1\texttt{:update}_1 \texttt{ + } \texttt{p}_2\texttt{:update}_2 \texttt{ + } \ldots \texttt{ + } \texttt{p}_n\texttt{:update}_n \ .$$

Here, `action` describes the action the program performs when executing this command. Furthermore, `guard` is a Boolean expression over arithmetic constraints on variable evaluations, e.g., $(\texttt{x} = \texttt{1}) \wedge (\texttt{y} \leq \texttt{5})$ for some variables $\texttt{x}, \texttt{y}$ is fulfilled if `x` has value `1` and `y` has a value smaller or equal than `5`. In case the guard evaluates to `true` in a state, the command is enabled and the values of the program's variables could be updated. An update describes how variables change depending on the current variable evaluation, e.g., `x:=1+y` changes the value of `x` to the increment of the value of `y`. Each update $\texttt{update}_i$ is chosen with probability $\texttt{p}_i$ for $i \in \{1,\ldots,n\}$. That is, in every state fulfilling `guard` the evaluations of the expressions $\texttt{p}_1$, $\texttt{p}_2$, …, $\texttt{p}_n$ constitute a probability distribution, i.e., must sum up to 1.

Any execution of the program starts in the initial variable evaluation and in each arising evaluation, it nondeterministically picks a guarded command whose guard is satisfied, and then modifies the current evaluation according to the update.

Example 2.1 (Multi-server platform commands). Let us introduce an excerpt of a PGC program that models a multi-server platform where processes are scheduled for execution on servers. The scheduling strategy is modeled by a program over variables $\texttt{server\_proc}i$ that evaluate to the number of the server the process i runs on. To increase the efficiency of the server platform, processes can be migrated from one server to another. For instance, the command

$$\texttt{[migrate]} \ (\texttt{server\_proc1} = 2 \wedge \texttt{server\_proc2} = 2 \wedge \texttt{server\_proc3} = 2) \mapsto \begin{array}{ll} 1/2: & \texttt{server\_proc1} := 1 \\ 1/2: & \texttt{server\_proc1} := 3 \end{array}$$

describes that if Process 1 and Process 2 are on Server 2 and Process 3 is on Server 2, then Process 1 is migrated to either Server 1 or Server 3 with probability of 50% each. ◇

To formally define PGC programs, we first introduce states as evaluations of variables, conditions on states, and updates. Let $\mathcal{V}$ be a finite set of *variables* where a range function $\rho\colon \mathcal{V} \to \mathbb{N}$ defines the domain of each variable as non-negative integers of $\Delta(v) = \{-\rho(v), -\rho(v)+1, \ldots, \rho(v)\}$. An *evaluation* over $\mathcal{V}$ is a function η that assigns to each $v \in \mathcal{V}$ a value in $\Delta(v)$. We denote the set of evaluations over $\mathcal{V}$ by $\mathit{Eval}(\mathcal{V})$. The set $\mathbb{E}(\mathcal{V})$ of arithmetic expressions is defined by the following grammar, where $z \in \mathbb{Z}$ and $v \in \mathcal{V}$:

$$\chi \ ::= \ z \mid v \mid (\chi + \chi) \mid (\chi \cdot \chi) \ .$$

Variable evaluations are extended to arithmetic expressions in the natural way, i.e., $\eta(z) = z$, $\eta(\chi_1 + \chi_2) = \eta(\chi_1) + \eta(\chi_2)$, and $\eta(\chi_1 \cdot \chi_2) = \eta(\chi_1) \cdot \eta(\chi_2)$. We denote by $\mathbb{C}(\mathcal{V})$ the set of *arithmetic constraints* over $\mathcal{V}$, which are terms of the form $(\chi \bowtie z)$ with $\chi \in \mathbb{E}(\mathcal{V})$, $\bowtie \in \{<,=,>,\geq,\neq,\leq\}$, and $z \in \mathbb{Z}$. For a given evaluation $\eta \in \mathit{Eval}(\mathcal{V})$ and constraint $(\chi \bowtie z) \in \mathbb{C}(\mathcal{V})$, we write $\eta \models (\chi \bowtie z)$ iff $\eta(\chi) \bowtie z$ and say that $(\chi \bowtie z)$ is *satisfied by* η. We denote by $\mathbb{C}(\eta)$ the constraints satisfied by η, i.e., $\mathbb{C}(\eta) = \{\gamma \in \mathbb{C}(\mathcal{V}) : \eta \models \gamma\}$. For a countable set X and x ranging over X, we define *Boolean expressions* $\mathbb{B}(X)$ over X by the grammar

$$\phi \ ::= \ x \mid \neg\phi \mid \phi \wedge \phi \ .$$

We define a *satisfaction relation* $\models \ \subseteq \ \wp(X) \times \mathbb{B}(X)$ for Boolean expressions as usual: for $Y \subseteq X$, we have

$$\begin{aligned} Y \models x & \ \text{ iff } \ x \in Y \\ Y \models \neg\psi & \ \text{ iff } \ Y \not\models \psi \\ Y \models \phi \wedge \psi & \ \text{ iff } \ Y \models \phi \text{ and } Y \models \psi \end{aligned}$$

For an evaluation $\eta \in \mathit{Eval}(\mathcal{V})$ and $\phi \in \mathbb{B}(\mathbb{C}(\mathcal{V}))$, we write $\eta \models \phi$ iff $\mathbb{C}(\eta) \models \phi$. The connectives disjunction $\vee$ and implication $\rightarrow$ can be expressed using $\wedge$ and $\neg$ in the usual way. The length of arithmetic expressions, constraints, and Boolean expressions is the size of the string encoding following the defined grammars.

We call a function $u\colon \mathcal{V} \rightarrow \mathbb{E}(\mathcal{V})$ an *update*. Recall that $\mathbb{E}(\mathcal{V})$ stands for the set of arithmetic expressions over variables $\mathcal{V}$. A *probabilistic update* is a distribution $\sigma \in \mathit{Distr}(\mathit{Upd})$ over a given finite set Upd of updates. The effect of an update $u\colon \mathcal{V} \rightarrow \mathbb{E}(\mathcal{V})$ on an evaluation $\eta \in \mathit{Eval}(\mathcal{V})$ is their composition $\eta \circ u \in \mathit{Eval}(\mathcal{V})$, which we define for all $v \in \mathcal{V}$ to be $(\eta \circ u)(v) = \max\{-\rho(v), \min\{\eta(u(v)), \rho(v)\}\}$. Note that we extended the definition of evaluations towards arithmetic expressions through their natural meaning of addition and multiplication operations. This notion naturally extends to *probabilistic updates* $\sigma \in \mathit{Distr}(\mathit{Upd})$ by $\eta \circ \sigma \in \mathit{Distr}(\mathit{Eval}(\mathcal{V}))$, where for any $\eta' \in \mathit{Eval}(\mathcal{V})$ we have

$$(\eta \circ \sigma)(\eta') = \sum_{u \in \mathit{Upd}, \eta \circ u = \eta'} \sigma(u) \ .$$

To provide an intuition why we have to sum over multiple updates when evaluating probabilistic updates, let us consider a variable space $\mathcal{V} = \{v\}$ and variable evaluations $\eta, \eta' \in \mathit{Eval}(\mathcal{V})$ where $\eta(v) = 2$ and $\eta'(v) = 4$. Furthermore, let us consider three updates $\mathit{Upd} = \{u_1, u_2, u_3\}$ where $u_1(v) = v + v$, $u_2(v) = 4$, and $u_3(v) = 2 \cdot v$. We define a probabilistic update $\sigma\colon \{u_1, u_2, u_3\} \rightarrow \mathit{Eval}(\{v\})$ by $\sigma(u_1) = 1/2$, $\sigma(u_2) = 1/3$, and $\sigma(u_3) = 1/6$. Then, $\eta' = \eta \circ u_1 = \eta \circ u_2 = \eta \circ u_3$ and thus, $(\eta \circ \sigma)(\eta') = 1$, the sum of all probabilities of the above updates, as $2 + 2 = 4 = 2 \cdot 2$. Differently, when $\eta(v) = 1$, we would have $(\eta \circ \sigma)(\eta') = 1/3$, since then only the update u_2 is effective.

A *probabilistic guarded command (PGC)* over a finite set of updates Upd, briefly called *command*, is a pair $\langle g, \sigma \rangle$ where $g \in \mathbb{B}(\mathbb{C}(\mathcal{V}))$ is a *guard* and $\sigma \in \mathit{Distr}(\mathit{Upd})$ is a probabilistic update. Similarly, a *weight assignment* is a pair $\langle g, w \rangle$ where $g \in \mathbb{B}(\mathbb{C}(\mathcal{V}))$ is a guard and $w \in \mathbb{N}$ a *weight*. A *PGC program* over $\mathcal{V}$, also briefly called *program*, is a tuple $\mathbf{P} = \langle \mathcal{V}, \mathbf{C}, \mathbf{W}, \eta_0 \rangle$ where $\mathbf{C}$ is a finite set of commands, $\mathbf{W}$ a finite set of weight assignments, and $\eta_0 \in \mathit{Eval}(\mathcal{V})$ is an *initial variable evaluation*. We define $\mathit{Upd}(\mathbf{P})$ as the union of all supports of probabilistic updates σ for which there is a guard $g \in \mathbb{B}(\mathbb{C}(\mathcal{V}))$ with $\langle g, \sigma \rangle \in \mathbf{C}$. Note that for brevity we did not formalize the common notion of actions in commands here, which however does not restrict the expressiveness of the formalism. The *size* of program $\mathbf{P}$ is defined as the sum of the lengths of its commands, weight annotations, the initial state, and the range specification of each variable used, where we count variables and operators as one, and encode all numbers in binary.

Semantics. The semantics of a program $\mathbf{P} = \langle \mathcal{V}, \mathbf{C}, \mathbf{W}, \eta_0 \rangle$ is specified via the weighted MDP

$$\mathbf{M}[\mathbf{P}] = \langle S, \mathit{Act}, P, \eta_0, \Lambda, \lambda, \mathit{wgt} \rangle$$

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 $\eta \in S$,
- $P(\eta, \sigma) = \eta \circ \sigma$ for any $\eta \in S$ and $\langle g, \sigma \rangle \in \mathbf{C}$ with $\lambda(\eta) \models g$, and
- $\mathit{wgt}(\eta) = \sum_{\langle g, w \rangle \in \mathbf{W}, \lambda(\eta) \models g} w$ for any $\eta \in S$.

Note that $\mathbf{M}[\mathbf{P}]$ is indeed a weighted MDP and that $P(\eta, \sigma)$ is a probability distribution with finite support for all $\eta \in \mathit{Eval}(\mathcal{V})$ and $\sigma \in \mathit{Distr}(\mathit{Upd}(\mathbf{P}))$ due to the finite domains assigned to each variable. Further, if we use binary encoding of numbers, each state in the MDP can be represented in space polynomial in the size of the program, and there can be at most exponentially many states.

Non-probabilistic programs. While our focus is on analyzing PGC programs through probabilistic model checking, there are applications in which probabilities are not relevant, and a program without probability distributions is sufficient. Clearly, such programs can be captured by our formalism. However, as some of our complexity results improve if we discard probabilities, we explicitly define a non-probabilistic variant of programs. In the following, a program is called *non-probabilistic* if for every command $\langle g, \sigma \rangle \in \mathbf{C}$ there is an update $u \in \mathit{Upd}(\mathbf{P})$ such that $\sigma(u) = 1$. We then simply abbreviate such a command by $g \mapsto u$. Note that non-probabilistic programs might be nondeterministic, as there can be more than one command that can be executed in a state.

Concept constructor	Syntax	Semantics
Top-concept	$\top$	$\Delta^\mathcal{I}$
Bottom-concept	$\bot$	$\emptyset$
Negation	$\neg C$	$\Delta^\mathcal{I} \setminus C^\mathcal{I}$
Conjunction	$C \sqcap D$	$C^\mathcal{I} \cap D^\mathcal{I}$
Disjunction	$C \sqcup D$	$C^\mathcal{I} \cup D^\mathcal{I}$
Existential restriction	$\exists r.C$	$\{d \in \Delta^\mathcal{I} \mid \exists e.(d,e) \in r^\mathcal{I} \wedge e \in C^\mathcal{I}\}$
Value restriction	$\forall r.C$	$\{d \in \Delta^\mathcal{I} \mid \forall e.(d,e) \in r^\mathcal{I} \rightarrow e \in C^\mathcal{I}\}$
Qualified at-most restriction	$(\leqslant n\ r.C)$	$\big\{d \in \Delta^\mathcal{I} \mid \#\{e \in \Delta^\mathcal{I} \mid (d,e) \in r^\mathcal{I} \wedge e \in C^\mathcal{I}\} \leq n\big\}$
Qualified at-least restriction	$(\geqslant n\ r.C)$	$\big\{d \in \Delta^\mathcal{I} \mid \#\{e \in \Delta^\mathcal{I} \mid (d,e) \in r^\mathcal{I} \wedge e \in C^\mathcal{I}\} \geq n\big\}$

Table 1. Syntax and semantics of the concept constructors in $\mathcal{ALCQ}$.

2.4. Description Logics

Description Logics (DLs) are a family of logics that vary in expressivity and that are often fragments of first-order logic for which reasoning is decidable. Theories formulated in a DL are usually called *ontologies.* The results in this paper apply to the expressive DL $\mathcal{SROIQ}$ underlying the web ontology language OWL 2 [MCH$^+$09], but for ease of presentation, we introduce its fragment $\mathcal{ALCQ}$ in detail. $\mathcal{ALCQ}$ is sufficiently expressive for modeling and is the DL that is used in our running example and in our evaluation.

The description logic $\mathcal{ALCQ}$. An ontology is a finite set of DL *axioms*, which put *concepts*, *roles*, and *individuals* in relation to each other. Intuitively these describe sets, binary relations, and individual objects, respectively. The syntax of $\mathcal{ALCQ}$ is defined based on a set $\mathsf{N_C}$ of *concept names*, a set $\mathsf{N_R}$ of *role names*, and a set $\mathsf{N_I}$ of *individual names*. We first describe the syntax and semantics of concepts, and then consider axioms and ontologies. Concepts are built by the use of a set of concept constructors available to the respective DL. In the DL $\mathcal{ALCQ}$, *concepts* are defined inductively according to the following grammar, where $A \in \mathsf{N_C}$, $r \in \mathsf{N_R}$, and $n \in \mathbb{N}$:

$$C := \bot \mid \top \mid A \mid \neg C \mid C \sqcap C \mid C \sqcup C \mid \exists r.C \mid \forall r.C \mid (\leqslant n\ r.C) \mid (\geqslant n\ r.C)\ .$$

The semantics of concepts is given in terms of first-order logic interpretations. An *interpretation* $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$ consists of a non-empty set $\Delta^\mathcal{I}$ (the *domain*) and a mapping function $\cdot^\mathcal{I}$ that assigns to each individual name $a \in \mathsf{N_I}$ a domain element $a^\mathcal{I} \in \Delta^\mathcal{I}$, to each concept name $A \in \mathsf{N_C}$ a subset $A^\mathcal{I} \subseteq \Delta^\mathcal{I}$, and to each role name $r \in \mathsf{N_R}$ a binary relation $r^\mathcal{I} \subseteq \Delta^\mathcal{I} \times \Delta^\mathcal{I}$. In other words, individual names correspond to constants, concept names to unary predicates, and role names correspond to binary predicates in first-order logic. The mapping function $\cdot^\mathcal{I}$ extends to complex concepts as shown in Table 1. Here, given a set X, we use $\#X$ to denote the *cardinality* of X. For example, the qualified at-most restrictions $(\leqslant n\ r.C)$ describe sets of objects that are in the r-relation to *at most* n many objects that satisfy the concept C.

Let a and b be individual names, C and D be concepts, and r be a role. An *axiom* is a statement of one of the following forms:

- $C \sqsubseteq D$ is a *general concept inclusion (GCI)*,
- $C \equiv D$ is a *concept equivalence*,
- $C(a)$ is a *concept assertion*, and
- $r(a,b)$ is a *role assertion.*

GCIs are terminological axioms and state sub-concept relationships, concept assertions state concept membership of an individual, and role assertions state that two individuals are related via the mentioned role. Formally, an interpretation $\mathcal{I}$ *satisfies* a

- GCI $C \sqsubseteq D$, iff $C^\mathcal{I} \subseteq D^\mathcal{I}$,
- a concept equivalence $C \equiv D$ iff $C \sqsubseteq D$ and $D \sqsubseteq C$,
- concept assertion $C(a)$, iff $a^\mathcal{I} \in C^\mathcal{I}$, and
- a role assertion $r(a,b)$, iff $(a^\mathcal{I}, b^\mathcal{I}) \in r^\mathcal{I}$.

An *ontology* $\mathcal{O}$ is then a finite set of axioms. An interpretation $\mathcal{I}$ is a *model of an ontology* $\mathcal{O}$ iff all axioms in $\mathcal{O}$ are satisfied by $\mathcal{I}$. There are many reasoning services defined for DLs. The reasoning services relevant in our context are *axiom entailment* and *consistency*. An axiom α is entailed by an ontology, in symbols $\mathcal{O} \models \alpha$, if $\mathcal{I} \models \alpha$ for all models $\mathcal{I}$ of $\mathcal{O}$. Two important forms of such entailments are: *subsumption* w.r.t. $\mathcal{O}$ ($C \sqsubseteq D$) and *instance of* w.r.t. $\mathcal{O}$ (denoted $C(a)$). An ontology is *consistent* if it has any model, and otherwise it is *inconsistent*. Note that an inconsistent ontology entails every axiom, because it does not have any models.

Example 2.2 (Multi-server platform ontology). Let us continue the description of our running example started in Example 2.1, by describing an ontology for our multi-server platform. Here, qualified at-most restrictions are used to describe the concept of "servers that run up to two processes that have high priority":

$$\mathsf{Server} \sqcap (\leqslant 2\ \mathsf{runsProcess}.\exists \mathsf{hasPriority}.\mathsf{High})\ .$$

We can describe the state of a particular multi-server platform, in which different servers run processes with different priorities, using the following assertions:

$$\mathsf{hasServer}(\mathsf{platform}, \mathsf{server1}) \qquad \mathsf{hasServer}(\mathsf{platform}, \mathsf{server2}) \tag{2}$$
$$\mathsf{runsProcess}(\mathsf{server2}, \mathsf{process1}) \qquad \mathsf{runsProcess}(\mathsf{server2}, \mathsf{process2}) \tag{3}$$
$$\mathsf{hasPriority}(\mathsf{process1}, \mathsf{highP}) \qquad \mathsf{hasPriority}(\mathsf{process2}, \mathsf{highP}) \qquad \mathsf{High}(\mathsf{highP}) \tag{4}$$

General domain knowledge on the server application domain can be specified using the following axioms:

$$\exists \mathsf{runsProcess}.\top \sqsubseteq \mathsf{Server} \tag{5}$$
$$(\geqslant 4\ \mathsf{runsProcess}.\top) \sqsubseteq \mathsf{Overloaded} \tag{6}$$
$$(\geqslant 2\ \mathsf{runsProcess}.(\exists \mathsf{hasPriority}.\mathsf{High})) \sqsubseteq \mathsf{Overloaded} \tag{7}$$
$$\mathsf{PlatformWithOverload} \equiv \exists \mathsf{hasServer}.\mathsf{Overloaded} \tag{8}$$

These GCIs express that if something runs a process (of some kind) then it is a server (5), and something that runs more than four processes is overloaded (6), that something is already overloaded when it runs two processes with a high priority (7), and that PlatformWithOverload is a platform that has an overloaded server (8).

As an example for reasoning in DLs, the GCIs (5) and (7) imply that the subsumption relationship

$$(\geqslant 3\ \mathsf{runsProcess}.(\exists \mathsf{hasPriority}.\mathsf{High})) \sqsubseteq \mathsf{Server} \sqcap \mathsf{Overloaded}$$

holds. Furthermore, it can be derived from the assertions (3) and (4) together with the GCIs (5) and (7) that $\mathsf{Overloaded} \sqcap \mathsf{Server}(\mathsf{server2})$ holds. This together with assertion (2) entails the instance relationship $\mathsf{PlatformWithOverload}(\mathsf{platform})$ w.r.t. the example ontology. $\diamond$

Other description logics. As DLs can differ in expressivity, so can the complexity for reasoning in them. The fragment of $\mathcal{ALCQ}$ without qualified number restrictions is the DL $\mathcal{ALC}$, which is the smallest propositionally complete DL. The fragment of $\mathcal{ALC}$ that offers only conjunction, existential restrictions, and the top-concept is called $\mathcal{EL}$. The formal ontology languages standardized by the W3C are captured in the OWL 2 standard [MCH$^+$09] and most of these languages are based on DLs. The light-weight DL $\mathcal{EL}$ is the core language of the OWL-EL profile, because of its good computational properties. Reasoning in $\mathcal{EL}$ [Bra04] and in its moderate extensions [BBL05] can be done in polynomial time.

The most expressive ontology language in that standard is OWL-DL which corresponds to the DL $\mathcal{SROIQ}$ [HKS06]. This DL extends $\mathcal{ALCQ}$ by additional constructors for roles, concepts and axioms, shown in Table 2. For example, $\mathcal{SROIQ}$ provides *nominals*, which are individuals that can be used in complex concepts, and it admits the use of inverse roles. By the use of the nominal $\{\mathsf{ACME}\}$ and the inverse of the role produces, we can state that ACME servers are servers produced by ACME:

$$\mathsf{ACME\text{-}Server} \sqsubseteq \mathsf{Server} \sqcap \exists \mathsf{produces}^{-}.\{\mathsf{ACME}\}\ .$$

In addition, the DL $\mathcal{SROIQ}$ can express complex inclusions between (composition of) roles, for example $\mathsf{runsOS} \circ \mathsf{runs} \sqsubseteq \mathsf{runs}$. Such inclusions can model transitivity and more complex relationships. We omit the details of the full syntax of $\mathcal{SROIQ}$, which comes with additional restrictions on the use of role names to ensure decidability, and refer the interested reader to [HKS06]. Reasoning in $\mathcal{SROIQ}$ is N2ExpTime-complete [Kaz08], while reasoning in $\mathcal{ALCQ}$ [Tob01] (and already in $\mathcal{ALC}$ [Sch91]) is ExpTime-complete.

Construct	Syntax	Semantics
Nominal	$\{a\}$	$a^\mathcal{I}$
Local reflexivity	$\exists r.\mathsf{Self}$	$\{d \in \Delta^\mathcal{I} \mid (d,d) \in r^\mathcal{I}\}$
Inverse role	r^-	$\{(y,x) \mid (x,y) \in r^\mathcal{I}\}$
Universal role	U	$\Delta^\mathcal{I} \times \Delta^\mathcal{I}$
Complex role inclusion	$R_1 \circ \ldots \circ R_n \sqsubseteq S$	$(R_1^\mathcal{I} \circ \ldots \circ R_n^\mathcal{I}) \subseteq S^\mathcal{I}$
Role disjointness	$\mathsf{Disj}(R_1, R_2)$	$R_1^\mathcal{I} \cap R_2^\mathcal{I} = \emptyset$

Table 2. Syntax and semantics of additional constructors in $\mathcal{SROIQ}$.

There are several highly optimized implementations of OWL reasoners that can decide consistency and axiom entailment (see [KKS14, GHM$^+$14, SLG14, SPG$^+$07]). We use the DL $\mathcal{ALCQ}$ in the evaluation part of this paper. However, our approach for ontology-mediated PMC can employ any DL for which reasoning is decidable.

3. Ontologized Programs

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

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 of probabilistic guarded command (PGC) language and DLs. To achieve a division of concerns, the operational behavior and the static background knowledge are specified separately and are only loosely coupled by an interface. On the one hand, this allows for specifying operational behavior in the usual way and to extend and reuse existing program specifications. On the other hand this allows us to easily link well-established or standardized ontologies to a program.

In this section, we describe the syntax of ontologized programs, and specify a foundational MDP semantics for ontologized programs called *consistency-independent semantics*, which we originally introduced in [DKT19]. In Section 5 we define two other semantics that refine consistency-independent semantics in taking care of inconsistencies arising from the DL component.

3.1. Syntax of Ontologized Programs

In order to prepare for the formal definition of the syntax of ontologized programs, we first explain how states of ontologized programs are represented. Both formalisms, PGC language and DL, represent abstracted versions of states in different ways. For the PGC language, a state is described as an evaluation of variables, while for the 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, states of ontologized programs are composed of two components, which we call the *command perspective* and the *DL perspective* on the 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 counter-part. This 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*), that maps DL axioms to command expressions. The part of the DL perspective depicted inside the box is not mapped to anything on the command side. This is the so-called *static part*, which contains the background knowledge about the system independent of the current state of the system. In contrast, the set of mapped axioms in the ontology perspective may change, depending on changes in the program perspective. Ontologized programs also allow the command perspective to use variables that have

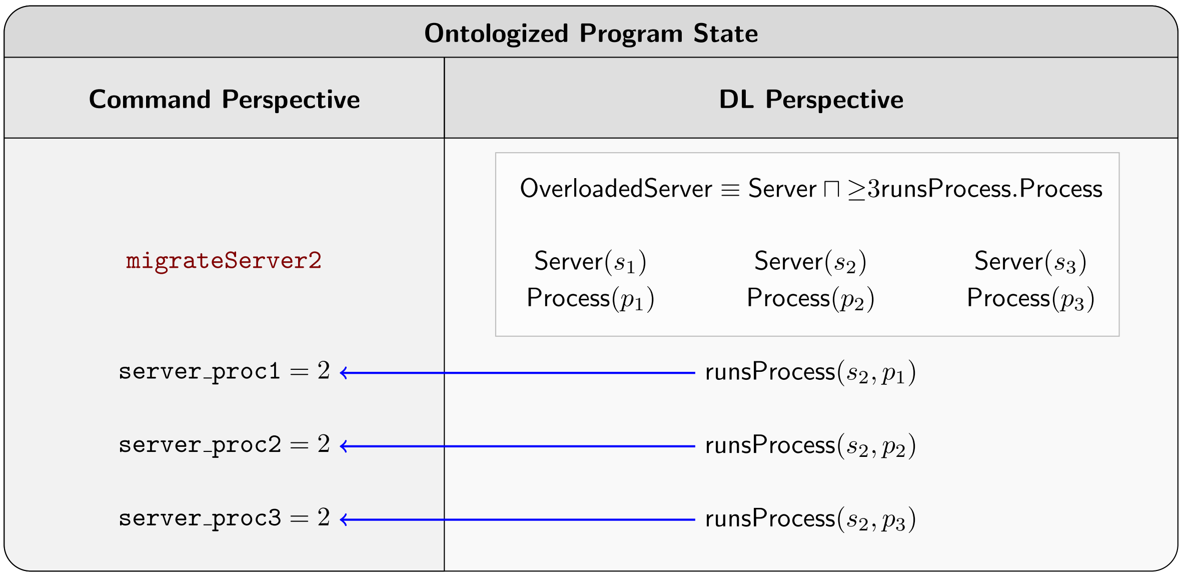

Fig. 1. Perspectives on the ontologized program state q_1: the command perspective (left) and DL perspective (right), linked by the interface (arrows).

no correspondence in the DL perspective, which we call *private variables*. (These are not depicted in the figure for brevity.)

In order for the DL component to provide background knowledge for the program component, the description of global states needs to depend on DL reasoning. To invoke a reasoning task from the command side of ontologized programs, we introduce the notion of hooks. A *hook* is a propositional variable in the program component that is linked by the interface to a *Boolean query* over the DL perspective of the ontologized program state. Technically, such a query consists of a set of DL axioms, to which the answer is true or false depending on whether or not all the axioms are entailed by the DL perspective. The result of the query determines the value of the hook using the function cD (**c***ommand to* **D***L*). To enable the actual use of hooks in the PGC language, we have to extend the definition of commands slightly. Specifically, we define *abstract programs* as programs where guards are Boolean expressions over arithmetic expressions as well as hooks. Given an abstract program $\mathbf{P}$, we denote by $\mathcal{H}_\mathbf{P}$ the set of hooks used in $\mathbf{P}$.

Example 3.1 (Hooks). In our server platform example from Example 2.1 and Example 2.2, we would define a hook migrateServer$i$ for each server $i \in \{1, 2, 3\}$ that represents the information 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 axioms in the DL perspective entail OverloadedServer(s_2).

To specify a scheduling strategy similarly as in Example 2.1 but with including the concept of hooks, we can specify a command of an *abstract* program in the following form:

$$[\texttt{migrate}] \quad (\texttt{migrateServer2} \wedge \texttt{server\_proc1} = 2) \quad \mapsto \quad \begin{array}{ll} 1/2: & \texttt{server\_proc1} := 1 \\ 1/2: & \texttt{server\_proc1} := 3 \end{array} \tag{9}$$

While the intention of this command is similar to the command in Example 2.1, this version omits the specification of the condition under which Server 2 needs to migrate. The idea is to specify and test this migration condition in the DL component rather than in the program itself, enabling a scheduling strategy that depends on background knowledge specified in the DL. ◇

We are now ready to formally define ontologized programs.

Definition 3.1. An *ontologized program* is a tuple $\mathfrak{P} = \langle \mathbf{P}, \mathcal{O}, \mathfrak{I} \rangle$ where

1. $\mathbf{P} = \langle \mathcal{V}_{\mathfrak{P}}, \mathbf{C}_{\mathfrak{P}}, \mathbf{W}_{\mathfrak{P}}, \eta_{\mathfrak{P},0} \rangle$ is an abstract program called the *program component*,
2. $\mathcal{O}$ is a DL ontology called the *static DL knowledge*,
3. $\mathfrak{I} = \langle \mathcal{V}_{\mathfrak{I}}, \mathcal{H}_{\mathfrak{I}}, \mathcal{F}, \mathsf{Dc}, \mathsf{cD} \rangle$ is a tuple describing the *interface*, which is composed of a set $\mathcal{V}_{\mathfrak{I}}$ of *public variables*, a set $\mathcal{H}_{\mathfrak{I}}$ of *interface hooks*, a set $\mathcal{F}$ of *DL axioms* called *fluents*, and two mappings $\mathsf{cD}\colon \mathcal{H}_{\mathfrak{I}} \to \wp(\mathbb{A})$ and $\mathsf{Dc}\colon \mathcal{F} \to \mathbb{B}(\mathbb{C}(\mathcal{V}_{\mathfrak{I}}))$, where $\mathbb{A}$ denotes the set of DL axioms, and

$\mathbf{P}$ and $\mathcal{O}$ are *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 $\mathcal{O} \cup \mathcal{F}$ is an $\mathcal{L}$ axiom, we call $\mathfrak{P}$ an *$\mathcal{L}$-ontologized program.*

To indicate the different components of ontologized programs, we sometimes use a subscript, that is, we adopt the notation $\mathfrak{P} = \langle \mathbf{P}_{\mathfrak{P}}, \mathcal{O}_{\mathfrak{P}}, \mathfrak{I}_{\mathfrak{P}} \rangle$.

3.2. Consistency-Independent Semantics for Ontologized Programs

PGC 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 the behavior specified can further rely on information provided by the ontology. There are different ways how the background knowledge formulated in the ontology can influence the operational behavior of an ontologized program, leading to different MDP semantics. Choosing an appropriate semantics is a modeling decision to be made when constructing the ontologized program. We first present the most basic semantics, called *consistency-independent semantics*, which also underlies our model-checking approach. Under this semantics, states with inconsistent corresponding DL query evaluations do not lead to a special treatment when specifying the operational behavior, i.e., the behavior is independent from the consistency status.

Before we go into detail, we specify how states in ontologized MDPs look like. For this, we formalize the intuitive description of states from Section 3.1 by defining ontologized states comprised of a command perspective and a DL perspective (as illustrated in Figure 1).

Definition 3.2. Let $\mathfrak{P} = \langle \mathbf{P}, \mathcal{O}, \mathfrak{I} \rangle$ be an ontologized program. An *ontologized state in* $\mathfrak{P}$ is a tuple $q = \langle \eta_q, \mathcal{O}_q \rangle$, with *command perspective* η_q and *DL perspective* $\mathcal{O}_q$, where

S1 η_q is an evaluation over $\mathcal{V}_{\mathfrak{P}}$,
S2 $\mathcal{O}_q \subseteq \mathcal{O} \cup \mathcal{F}$,
S3 $\mathcal{O} \subseteq \mathcal{O}_q$, and
S4 for every $\alpha \in \mathcal{F}$ holds $\alpha \in \mathcal{O}_q$ iff $\eta_q \models \mathsf{Dc}(\alpha)$.

The set of *active hooks in* q is $\mathcal{H}_q = \{\mathtt{h} \in \mathcal{H}_{\mathbf{P}} \mid \mathcal{O} \models \mathsf{cD}(\mathtt{h})\}$.

For every evaluation $\eta \in \mathit{Eval}(\mathcal{V}_{\mathfrak{P}})$, there is always a unique ontology $\mathcal{O}_\eta$ s.t. $\langle \eta, \mathcal{O}_\eta \rangle$ is an ontologized state in $\mathfrak{P}$. We denote the ontologized state induced by an evaluation η as $e(\mathfrak{P}, \eta)$. To define updates on ontologized states, we exploit the facts that

- updates can only modify the command perspective of ontologized states directly, and
- 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 $u \in \mathit{Upd}(\mathbf{P}_{\mathfrak{P}})$ and an ontologized state q, we define $u(q) = e(\mathfrak{P}, u(\eta_q))$. Correspondingly, we extend probabilistic updates $\sigma \in \mathit{Distr}(\mathit{Upd}(\mathbf{P}_{\mathfrak{P}}))$ to ontologized states q by setting $\sigma(q, u(q)) = \sigma(u)$ for all $u \in \mathit{Upd}(\mathbf{P}_{\mathfrak{P}})$.

Example 3.2. To illustrate how commands are executed on ontologized states, we consider the ontologized state q_1 shown in Figure 1 and the abstract command (9) of Example 3.1. First, we note that the guard in the command is active, since the hook `migrateServer2` is active and `server_proc1` $= 2$ holds in q_1. The command can thus be executed on this 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 shown in Figure 2 in which `server_proc1` $= 2$ is replaced by `server_proc1` $= 1$, and $\mathsf{runs}(s_2, p_2)$ is replaced by $\mathsf{runs}(s_1, p_2)$. As a result of this update, the hook `migrateServer2` becomes inactive in q_2. More precisely, the interface in the example

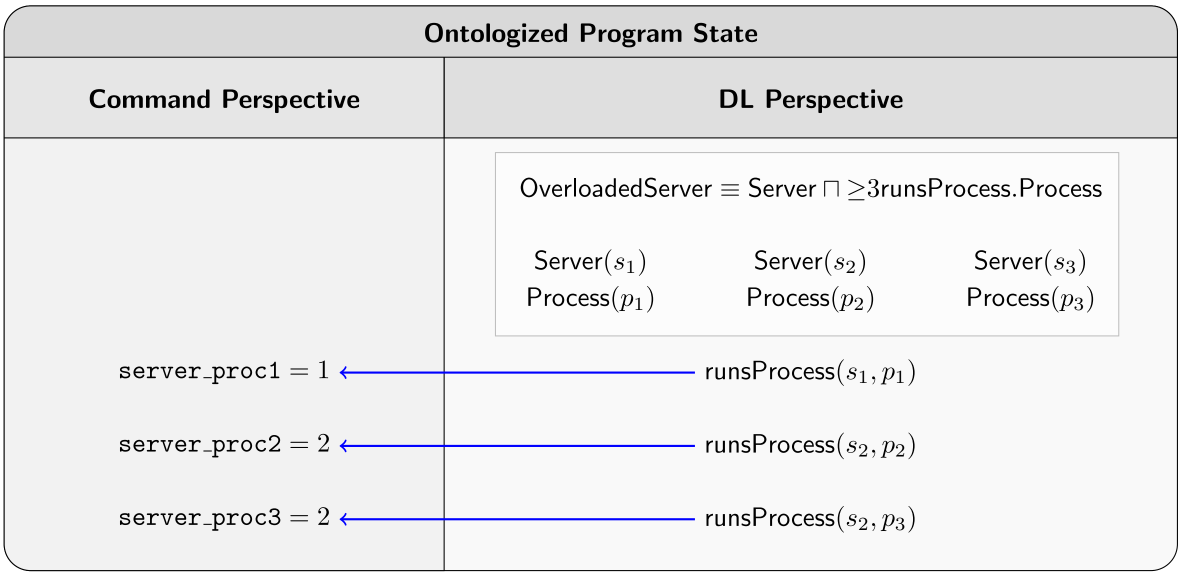
Ontologized Program State
Command Perspective
DL Perspective
OverloadedServer ≡ Server ⊓ ≥3runsProcess.Process
Server(s_1) Process(p_1)
Server(s_2) Process(p_2)
Server(s_3) Process(p_3)
server_proc1 = 1
server_proc2 = 2
server_proc3 = 2
runsProcess(s_1, p_1)
runsProcess(s_2, p_2)
runsProcess(s_2, p_3)

Fig. 2. State q_2 after the update `server_proc1` $:= 1$ on state q_1.

specifies $\mathsf{cD}(\texttt{migrateServer2}) = \{\mathsf{OverloadedServer}(s_2)\}$, and $\mathsf{OverloadedServer}(s_2)$ is not entailed by $\mathcal{O}_{q_2}$, i.e., by the DL perspective of q_2. ◇

The definition of the MDP induced by an ontologized program under consistency-independent semantics now closely follows the semantics for MDPs induced by plain programs.

Definition 3.3. Let $\mathfrak{P} = \langle \mathbf{P}, \mathcal{O}, \mathfrak{I} \rangle$ be an ontologized program. The *MDP induced by* $\mathfrak{P}$ is defined as $\mathbf{M}[\mathfrak{P}] = \langle Q, Act, P, \iota, \Lambda, \lambda, wgt \rangle$, where

1. $Q = \{e(\mathfrak{P}, \eta) \mid \eta \in Eval(\mathcal{V}_{\mathfrak{P}})\}$,
2. $Act = Distr(Upd(\mathbf{P}))$,
3. $\iota = e(\mathfrak{P}, \eta_0)$,
4. $\Lambda = \mathcal{H}_{\mathfrak{I}} \cup \mathbb{C}(\mathcal{V}_{\mathfrak{P}})$,
5. $\lambda(q) = \mathbb{C}(\eta_q) \cup \mathcal{H}_q$ for every $q \in Q$,
6. $P(q, \sigma, q') = e(\mathfrak{P}, (\eta_q \circ \sigma)(\eta_{q'}))$ for all $q, q' \in Q$, $\langle g, \sigma \rangle \in \mathbf{C}_{\mathfrak{P}}$ with $\lambda(q) \models g$, and
7. $wgt(q) = \sum_{\langle g, w \rangle \in \mathbf{W}_{\mathfrak{P}}, \lambda(q) \models g} w$

In an induced MDP, the set of states Q contains the ontologized states that can be constructed based on the evaluations of the variables $\mathcal{V}_{\mathfrak{P}}$ occurring in the program component $\mathbf{P}$. Correspondingly, the initial state q_0, is the ontologized state $e(\mathfrak{P}, \eta_0)$ obtained from the initial evaluation η_0 of the program component. The label set Λ determines which information on the nodes the probabilistic properties can access. As for MDPs induced by plain programs, this contains the set $\mathbb{C}(\mathcal{V}_{\mathfrak{P}})$ of arithmetic constraints on the program variables, but in addition it also contains the set $\mathcal{H}_{\mathfrak{I}}$ of hooks defined by the interface. The labeling function λ associates ontologized states with those labels that are active in that state. Finally, probabilistic transitions and weights are assigned to ontologized states based on which guards are entailed by the label of the respective node. This is done in the same way as for plain programs. Similarly as for the MDP induced by (plain) programs, the set of actions corresponds to the set of distributions over the updates that occur in the program.

4. Model Checking under Consistency-Independent Semantics

Following the definition of the inconsistency-independent semantics (see Definition 3.3), it is straightforward to establish an approach towards a quantitative analysis of ontologized programs under consistency-independent semantics. Recall that, because each program variable has a bounded range, the MDPs induced by our programs always have a finite number of states. Therefore, we could simply apply standard PMC techniques directly on the MDP induced by the ontologized program. However, in practice an explicit construction of the induced MDP is not feasible: even though the range of each program variable is bounded, the number of states is exponential in the size of the input. Moreover, associating labels to ontologized states is much more expensive than for plain programs, since we have to use DL reasoning to determine which hooks are active. Depending on the DL used, testing entailment of axioms can range up to N2ExpTime (see Section 2.4). Consequently, it is desirable to keep the number of entailment tests small. There exist advanced techniques to represent the MDP states concisely to mitigate the exponential blow-up of the states space, e.g., through symbolic representations via multi-terminal binary decision diagrams (MTBDDs) [MP04]. However, it is not clear how to extend those approaches to ontologized states, and we would still need a way to evaluate the hooks.

Instead of devising a new procedure for constructing the MDP from scratch, our approach is to reduce ontology-mediated PMC to standard PMC. This is inspired by *ontology-based data access* (OBDA) [PLC+08, BO15], where ontologies are used in connection with database systems to improve query answering. Specifically, the data in the database is considered incomplete, and the ontology is used to infer some of the missing information. Queries can then be used to query explicit as well as implicit information from the database. For some DLs, it is possible to use a technique called *query rewriting* to reduce query answering in an OBDA setting to querying without ontology. For this, the query is *rewritten* into a new query that yields the same answers as the original one, but can be evaluated without the ontology so that standard database systems can be used [CDL+07].

We propose a similar technique to perform PMC on ontologized programs, where we rewrite the ontologized program, together with the property to be checked, into a plain program and a translated property. These can then be directly evaluated using standard PMC methods. Our technique preserves the shape of the MDP induced by ontologized program based on a notion of isomorphism, which we introduce in Section 4.1. To show correctness of our approach, we also define an abstract specification of rewriting in terms of so-called mediators and rewriters.

If an MDP is isomorphic to another MDP, a *mediator* guarantees that state labels from the first MDP can be translated to those of the second MDP. This allows us to reduce model checking on the first MDP to model checking on the second. A *rewriter* is a function that rewrites an ontologized program into a plain program. If there exists a mediator from the MDP induced by the ontologized program to the MDP induced by the rewritten program, then model-checking the ontologized program can be reduced to model-checking of the rewritten program, and we call the rewriter *sound.*

We specify two instances of sound rewriters in Section 4.2. The first one, which we call *naive rewriter*, has a simpler definition and we use it mainly to illustrate the basic idea. Because the naive rewriter is not feasible in practice, we present an optimized variant, the *justification-based* rewriter. This rewriter reduces the amount of DL reasoning required by focusing on the relevant axioms. Our reduction technique works for analyzing programs under consistency-independent semantics.

4.1. Isomorphic MDPs and Mediators

We formalize a rewriting procedure that takes as input an ontologized program $\mathfrak{P}$ as in Definition 3.1 and generates a plain program $\widetilde{\mathfrak{P}}$. Our rewriting procedure has to ensure that the MDP $\mathbf{M}[\widetilde{\mathfrak{P}}]$ induced by $\widetilde{\mathfrak{P}}$ corresponds "in some way" to the MDP $\mathbf{M}[\mathfrak{P}]$ induced by $\mathfrak{P}$. To make sure that a variety of model-checking tasks on $\mathfrak{P}$ can be easily translated to model-checking tasks on $\widetilde{\mathfrak{P}}$, we require a strong correspondence between $\mathbf{M}[\widetilde{\mathfrak{P}}]$ and $\mathbf{M}[\mathfrak{P}]$, together with a means of translating properties on $\mathbf{M}[\mathfrak{P}]$ (which may use hooks) to properties on $\mathbf{M}[\widetilde{\mathfrak{P}}]$. This is captured by the following definition.

Definition 4.1. Let $\mathbf{M}_1$ and $\mathbf{M}_2$ be two MDPs, where $\mathbf{M}_1 = \langle Q_1, Act_1, P_1, \iota_1, \Lambda_1, \lambda_1, wgt_1\rangle$ and $\mathbf{M}_2 =$

$\langle Q_2, Act_2, P_2, \iota_2, \Lambda_2, \lambda_2, wgt_2\rangle$. Then, $\mathbf{M}_1$ and $\mathbf{M}_2$ are *isomorphic modulo labelings* iff there exist bijections

$$b\colon Q_1 \to Q_2 \qquad \text{and} \qquad a\colon Act_1 \to Act_2$$

that satisfy the following conditions:

I1 $b(\iota_1) = \iota_2$,
I2 for every $q \in Q_1$, $wgt_2(b(q)) = wgt_1(q)$,
I3 for every $q, q' \in Q_1$ and $\sigma \in Act_1$, $P_2(b(q), a(\sigma), b(q'))$ is defined iff $P_1(q, \sigma, q')$ is defined, in which case $P_2(b(q), a(\sigma), b(q')) = P_1(q, \sigma, q')$.

If for such MDPs, there exists a function $m\colon \Lambda_1 \to \mathbb{B}(\Lambda_2)$ s.t.

I4 for every $q \in Q_1$ and $\ell \in \Lambda_1$, $\ell \in \lambda_1(q)$ iff $\lambda_2(b(q)) \models m(\ell)$,

then m is called a *mediator from* $\mathbf{M}_1$ *to* $\mathbf{M}_2$.

Since probabilistic properties refer to states via their label sets, we need a way of translating the label sets. If $\mathbf{M}_1$ and $\mathbf{M}_2$ are not only isomorphic modulo labelings, but there also exists a mediator m from $\mathbf{M}_1$ to $\mathbf{M}_2$, we can use m to translate properties over label sets in $\mathbf{M}_1$ to properties over label sets in $\mathbf{M}_2$. In this case it is straightforward to reduce model-checking tasks for $\mathbf{M}_1$ to corresponding model-checking tasks in $\mathbf{M}_2$. The following lemma is an easy consequence of Definition 4.1.

Lemma 4.1. Let $\mathbf{M}_1$ be an MDP with labels Λ_1 and $\mathbf{M}_2$ an MDP with labels Λ_2 such that $\mathbf{M}_1$ and $\mathbf{M}_2$ are isomorphic modulo labelings and there exists a mediator $m\colon \Lambda_1 \to \mathbb{B}(\Lambda_2)$ from $\mathbf{M}_1$ to $\mathbf{M}_2$. Let Φ be a probabilistic property or an expectation property over the label set Λ_1, and Φ^m is the result of replacing every $\ell \in \Lambda_1$ by $m(\ell)$. Then, $\mathbf{M}_1 \models \Phi$ iff $\mathbf{M}_2 \models \Phi^m$.

Note that as a consequence of this lemma, we can also reduce translate probabilistic, expectation, and quantile queries using a mediator.

Fix an ontologized program $\mathfrak{P}$ as in Definition 3.1. Our method rewrites $\mathfrak{P}$ into a program $\widetilde{\mathfrak{P}}$ such that $\mathbf{M}[\mathfrak{P}]$ and $\mathbf{M}[\widetilde{\mathfrak{P}}]$ are isomorphic modulo labelings and have a mediator m. Since each evaluation $\eta \in Eval(\mathcal{V}_{\mathfrak{P}})$ can be uniquely extended to an ontologized state $e(\eta) = \langle \eta, \mathcal{O}_\eta\rangle$, there is a one-to-one correspondence between ontologized states and their command perspective. Our rewriting is such that the MDP induced by $\widetilde{\mathfrak{P}}$ is obtained from $\mathbf{M}[\mathfrak{P}]$ by replacing each ontologized state q by its command perspective η_q, and reducing the label set to $\mathbb{C}(\mathcal{V}_{\mathfrak{P}})$. The states in $\mathbf{M}[\mathfrak{P}]$ are labeled with arithmetic constraints and hooks, and the states in $\mathbf{M}[\widetilde{\mathfrak{P}}]$ are labeled with the same arithmetic constraints as the corresponding states in $\mathbf{M}[\mathfrak{P}]$. Hence, the mediator m only needs to take care of the hooks, and for every arithmetic constraint $\gamma \in \mathbb{C}(\mathcal{V}_{\mathfrak{P}})$, we have $m(\gamma) = \gamma$. This mediator does not only establish the correspondence between the MDP induced by the ontologized program and the MDP of the rewritten program, it is also used to rewrite the program in the first place. How a rewriting takes place is specified in the following definition.

Definition 4.2 (Rewriting). Let $\mathfrak{P}$ be an ontologized program as in Definition 3.1, Φ a probabilistic or expectation property over the labelling in $\mathbf{M}[\mathfrak{P}]$. A *rewriter for* $\mathfrak{P}$ is a function $r\colon \mathcal{H}_{\mathfrak{I}} \to \mathbb{B}(\mathbb{C}(\mathcal{V}_{\mathfrak{P}}))$. The *rewriting of* $\mathfrak{P}$ *using* r is the program $\mathfrak{P}^r$ obtained from $\mathbf{P}_{\mathfrak{P}}$ by replacing every hook $\mathtt{h} \in \mathcal{H}_{\mathfrak{I}}$ by $r(\mathtt{h})$. Correspondingly, the *rewriting of* Φ is the property Φ^r obtained by replacing every hook $\mathtt{h} \in \mathcal{H}_{\mathfrak{I}}$ in Φ by $r(\mathtt{h})$. The function r is called *sound rewriter for* $\mathfrak{P}$ iff its extension $m\colon \mathbb{C}(\mathcal{V}_{\mathfrak{P}}) \cup \mathcal{H}_{\mathfrak{I}} \to \mathbb{B}(\mathbb{C}(\mathcal{V}_{\mathfrak{P}}))$ given by

$$m(\gamma) = \begin{cases} \gamma & \text{if } \gamma \in \mathbb{C}(\mathcal{V}_{\mathfrak{P}}) \\ r(\gamma) & \text{if } \gamma \in \mathcal{H}_{\mathfrak{I}} \end{cases}$$

is a mediator from $\mathbf{M}[\mathfrak{P}]$ to $\mathbf{M}[\mathfrak{P}^r]$.

Example 4.1 (Rewriting). In our running example, the states of the MDP induced by the rewritten program just correspond to their command perspective (see Figure 1 and 2). The way we defined the ontologized program, the hook `migrateServer2` becomes active if there are three processes on the second server. As we have exactly three processes in our example, a possible mediator m could capture this by

$$m(\mathtt{migrateServer2}) = \big(\mathtt{server\_proc1} = 2 \land \mathtt{server\_proc2} = 2 \land \mathtt{server\_proc3} = 2\big)\ .$$

By restricting the domain of the mediator to $\mathcal{H}_{\mathfrak{I}}$, we obtain a sound rewriter r. Using this rewriter, our command (9) of Example 3.1

$$\texttt{[migrate]}\quad (\texttt{migrateServer2} \wedge \texttt{server\_proc1} = 2) \;\mapsto\; \begin{array}{ll} 1/2: & \texttt{server\_proc1} := 1 \\ 1/2: & \texttt{server\_proc1} := 3 \end{array}$$

would be replaced by the following command without hooks in the rewritten program:

$$\texttt{[migrate]}\quad \Big((\texttt{server\_proc1} = 2 \wedge \texttt{server\_proc2} = 2 \wedge \texttt{server\_proc3} = 2) \\ \wedge \texttt{server\_proc1} = 2 \Big) \;\mapsto\; \begin{array}{ll} 1/2: & \texttt{server\_proc1} := 1 \\ 1/2: & \texttt{server\_proc1} := 3 \end{array}$$

Note that there could be many rewriters in this example that would produce the same induced MDP. For instance, omitting $\texttt{server\_proc1} = 2$ in the rewritten command would yield the same operational behavior without directly including constraints on all process-server placements. $\diamond$

The following is an easy consequence of Lemma 4.1 and shows that sound rewriters are sufficient to reduce model checking in ontologized programs to model checking in plain programs.

Lemma 4.2. Let $\mathfrak{P}$ be an ontologized program with interface hooks $\mathcal{H}_{\mathfrak{I}}$, r a sound rewriter for $\mathfrak{P}$, and Φ a probabilistic or expectation property over the labeling in $\mathbf{M}[\mathfrak{P}]$. Then, $\mathbf{M}[\mathfrak{P}] \models \Phi$ iff $\mathbf{M}[\mathfrak{P}^r] \models \Phi^r$.

Thus, by computing a sound rewriter, we can reduce probabilistic model checking of ontologized programs to probabilistic model checking of programs. For those, standard tools are available [KNP11, DJKV17]. The main task is to obtain a sound rewriter that can be effectively and efficiently computed.

4.2. Constructing Sound Rewriters

To obtain a sound rewriter for a given program $\mathfrak{P}$, we have to define for each hook $\texttt{h} \in \mathcal{H}_{\mathfrak{I}}$ a Boolean expression over $Eval(\mathcal{V}_{\mathfrak{P}})$ that identifies exactly the ontologized states $q \in Q_{\mathfrak{P}}$ in which $\texttt{h}$ is active. In other words, we need to identify conditions on the program perspective of q that determine when $\texttt{h}$ is active.

Before looking at the program perspective, we recall the mechanism of hook activation starting from the DL perspective. Activation of $\texttt{h}$ is determined via the interface function cD: $\texttt{h}$ is active in q iff $\mathcal{O}_q \models \mathsf{cD}(\texttt{h})$. By Condition **S2** in Definition 3.2, every axiom in $\mathcal{O}_q$ is either taken from $\mathcal{O}_{\mathfrak{P}}$, the background ontology of the ontologized program, or from $\mathcal{F}$, the set of fluents. Since by Condition **S3**, $\mathcal{O}_{\mathfrak{P}}$ is contained in the DL perspective of every ontologized state for $\mathfrak{P}$, only the axioms in $\mathcal{F}$ are relevant. We can thus identify the ontologized states in which $\texttt{h}$ is active solely based on the axioms in $\mathcal{F}$. For a hook $\texttt{h} \in \mathcal{H}_{\mathfrak{I}}$, let

$$\mathfrak{F}_{\texttt{h}} = \{\mathbf{F} \subseteq \mathcal{F} \mid \mathcal{O}_{\mathfrak{P}} \cup \mathbf{F} \models \mathsf{cD}(\texttt{h})\}$$

be the set containing the sets of fluents responsible for (the activation of) $\texttt{h}$. Hence, a hook $\texttt{h}$ is active in an ontologized state q iff there exists some $\mathbf{F} \in \mathfrak{F}_{\texttt{h}}$ s.t. $\mathbf{F} \subseteq \mathcal{O}_q$. On the command perspective, this in turn can be determined using the interface function Dc. Namely, we have $\mathbf{F} \subseteq \mathcal{O}_q$ iff $\eta_q \models \bigwedge_{\alpha \in \mathbf{F}} \mathsf{Dc}(\alpha)$. We can thus identify the states in which $\texttt{h}$ is active using the rewriter $r_n : \mathcal{H}_{\mathfrak{I}} \to \mathbb{B}(\mathbb{C}(\mathcal{V}_{\mathfrak{P}}))$ defined by

$$r_n(\texttt{h}) = \bigvee_{\mathbf{F} \in \mathfrak{F}_{\texttt{h}}} \bigwedge_{\alpha \in \mathbf{F}} \mathsf{Dc}(\alpha).$$

We call r_n the *naive rewriter for* $\mathfrak{P}$.

Lemma 4.3. Let $\mathfrak{P}$ be an ontologized program and r_n be the naive rewriter for $\mathfrak{P}$. Then, r_n is a sound rewriter for $\mathfrak{P}$.

Proof. Let $\mathbf{M}_1 = \mathbf{M}[\mathfrak{P}] = \langle Q_1, Act_1, P_1, \iota_1, \Lambda_1, \lambda_1, wgt_1 \rangle$, be the MDP induced by $\mathfrak{P}$. Our proof proceeds in four steps: **(1)** We first construct an MDP $\mathbf{M}_2$ based on $\mathbf{M}_1$ by restricting the label set. **(2)** We show that $\mathbf{M}_1$ and $\mathbf{M}_2$ are isomorphic modulo labelings. **(3)** We then show that we obtain a mediator m_n from $\mathbf{M}_1$ to $\mathbf{M}_2$ if we extend r_n to $\mathbb{C}(\mathcal{V}_{\mathfrak{P}}) \cup \mathcal{H}_{\mathfrak{I}}$ by setting $m_n(\gamma) = \gamma$ for all $\gamma \in \mathbb{C}(\mathcal{V}_{\mathfrak{P}})$. **(4)** Finally, we show that $\mathbf{M}_2$ is the MDP induced by the program $\mathfrak{P}^{r_n}$ that is the result of rewriting $\mathfrak{P}$ using r_n.

(1) $\mathbf{M}_2 = \langle Q_2, Act_2, P_2, \iota_2, \Lambda_2, \lambda_2, wgt_2\rangle$ is defined by

1. $Q_2 = \{\eta_q \mid q \in Q_1\}$,
2. $Act_2 = Act_1$,
3. for all $q, q' \in Q_1$ and $\sigma \in Act_1$, $P_2(\eta_q, \sigma, \eta_{q'}) = P_1(q, \sigma, q')$ if $P_1(q, \sigma, q')$ is defined, and otherwise $P_2(\eta_q, \sigma, \eta_{q'})$ is undefined,
4. $\iota_2 = \eta_{\iota_1}$,
5. $\Lambda_2 = \mathbb{C}(\mathcal{V}_{\mathfrak{P}})$,
6. for all $q \in Q_2$, $\lambda_2(q) = \mathbb{C}(q) = \{\gamma \in \mathbb{C}(\mathcal{V}_{\mathfrak{P}}) : q \models \gamma\}$, and
7. for all $q \in Q_1$, $wgt_2(\eta_q) = wgt_1(q)$.

(2) Since every evaluation $\eta \in Eval(\mathcal{V}_{\mathfrak{P}})$ can be uniquely extended to an ontologized state $e(\eta, \mathfrak{P}) \in Q_1$, the function $b\colon Q_1 \to Q_2$ given by $b(q) = \eta_q$ is invertible by $b^-(q) = e(q, \mathfrak{P})$, and thus a bijection. Since $Act_1 = Act_2$, we can define another bijection $a\colon Act_1 \to Act_2$ by setting $a(\sigma) = \sigma$ for all $\sigma \in Act_1$. $\mathbf{M}_1$ and $\mathbf{M}_2$ are isomorphic modulo labelings via the bijections b and a.

(3) To show that m_n as defined above is a mediator from $\mathbf{M}_1$ to $\mathbf{M}_2$, we have to consider the labels in $\Lambda_1 = \mathbb{C}(\mathcal{V}_{\mathfrak{P}}) \cup \mathcal{H}_{\mathfrak{I}}$. By construction, for every $\phi \in \mathbb{C}(\mathcal{V}_{\mathfrak{P}})$ and $q \in Q_1$, we have $m_n(\phi) = \phi$ and thus $\phi \in \lambda_1(q)$ iff $\lambda_2(b(q)) = \lambda_2(\eta_q) \models \phi$. Consider $q \in Q_1$ and $\mathtt{h} \in \mathcal{H}_{\mathfrak{I}}$. We show that $\mathtt{h} \in \lambda_1(q)$ iff $\lambda_2(b(q)) \models m_n(\mathtt{h})$: If $\mathtt{h} \in \lambda_1(q)$, then $\mathcal{O}_q \models \mathsf{cD}(\mathtt{h})$. By definition of $\mathfrak{F}_{\mathtt{h}}$, we obtain that $\mathbf{F} = (\mathcal{O}_q \cap \mathcal{F}) \in \mathfrak{F}_{\mathtt{h}}$. For every $\alpha \in \mathbf{F}$, by Definition 3.2, $\eta_q \models \mathsf{Dc}(\alpha)$. Consequently, $\eta_q \models \bigwedge_{\alpha \in \mathbf{F}} \mathsf{Dc}(\alpha)$. By definition of r_n, we obtain $\eta_q \models r_n(\mathtt{h}) = m_n(\mathtt{h})$, and since $\lambda_2(b(q)) = \lambda_2(\eta_q)$, we have that $\lambda_2(b(q)) \models m_n(\mathtt{h})$. Assume $\mathtt{h} \notin \lambda_1(q)$. Then, $\mathcal{O}_q \not\models \mathsf{cD}(\mathtt{h})$, and $\mathbf{F} = (\mathcal{O}_q \cap \mathcal{F}) \notin \mathfrak{F}_{\mathtt{h}}$. By monotonicity of the entailment relation, there also cannot be any subset $\mathbf{F}' \subsetneq \mathbf{F}$ s.t. $\mathbf{F}' \in \mathfrak{F}_{\mathtt{h}}$. It follows that $\lambda_2(\eta_q) \not\models r_n(\mathtt{h})$ and thus that $\lambda_2(b(q)) \not\models m_n(\mathtt{h})$.

We have shown that for every label $\ell \in \Lambda_1$ and every state $q \in Q_1$, $\ell \in \lambda_1(q)$ iff $\lambda_2(b(q)) \models m_n(\ell)$. Hence, m_n is a mediator from $\mathbf{M}_1$ to $\mathbf{M}_2$.

(4) It remains to show that $\mathbf{M}_2$ is the MDP

$$\mathbf{M}_3 = \mathbf{M}[\mathfrak{P}^{r_n}] = \langle Q_3, Act_3, P_3, \iota_3, \Lambda_3, \lambda_3, wgt_3\rangle$$

induced by $\mathfrak{P}^{r_n}$. We show that $\mathbf{M}_2 = \mathbf{M}_3$ by checking the components of $\mathbf{M}_3$ one after the other.

1. $Q_3 = Eval(\mathcal{V}_{\mathfrak{P}}) = Q_2$.
2. $Act_3 = Distr(Upd(\mathfrak{P}^{r_n})) = Distr(Upd(\mathbf{P})) = Act_1 = Act_2$, where $\mathbf{P}$ is the program component of $\mathfrak{P}$, because the updates in $\mathbf{P}$ operate only on $\mathcal{V}_{\mathfrak{P}}$, as do the updates in $\mathfrak{P}^{r_n}$.
3. $\Lambda_3 = \mathbb{C}(\mathcal{V}_{\mathfrak{P}}) = \Lambda_2$.
4. For all $\eta \in Q_3 = Q_2$, $\lambda_3(\eta) = \mathbb{C}(\eta) = \lambda_2(\eta)$.
5. For $q \in Q_3$ and $\sigma \in Act_3$, $P_3(q, \sigma)$ is defined if there exists $(g, \sigma) \in \mathbf{C}^{r_n}$, where $\mathbf{C}^{r_n}$ are the commands in $\mathfrak{P}^{r_n}$, and $q \models g$. In this case, $P_3(q, \sigma, q') = (q \circ \sigma)(q')$ for all $q' \in Q_3$.
   Since $(g, \sigma) \in \mathbf{C}^{r_n}$, there must exist a command $(g', \sigma) \in \mathbf{C}_{\mathfrak{P}}$, where $\mathbf{C}_{\mathfrak{P}}$ is the set of commands in $\mathfrak{P}$, and g is obtained from g' by replacing each hook $\mathtt{h} \in \mathcal{H}_{\mathfrak{I}}$ with $r_n(\mathtt{h})$. Since m_n is a mediator from $\mathbf{M}_1$ to $\mathbf{M}_2$ and also $q \in Q_2$, $q \models g$ iff $b^-(q) = e(q, \mathfrak{P}) \models g'$. It follows that P_3 is defined iff P_1 is defined, which is the case iff P_2 is defined. Assume P_3 and P_2 are defined. We then have for all $q' \in Q_2 = Q_3$, that
   $$P_2(q, \sigma, q') \;=\; P_1(e(q, \mathfrak{P}), \sigma, e(q', \mathfrak{P})) = (e(q, \mathfrak{P}) \circ \sigma)(e(q', \mathfrak{P})) = (q \circ \sigma)(q') \;=\; P_3(q, \sigma, q').$$
   It follows that $P_3 = P_2$.
6. Let $\mathbf{W}_{\mathfrak{P}}$ be the weight assignments in $\mathfrak{P}$, and $\mathbf{W}^{r_n}$ the weight assignments in $\mathfrak{P}^{r_n}$. For every $q \in Q_3$, we have
   $$wgt_3(q) = \sum_{\langle g, w\rangle \in \mathbf{W}^{r_n}, \lambda_3(q) \models g} w.$$
   For every weight assignment $\langle g, w\rangle \in \mathbf{W}^{r_n}$, there exists a weight assignment $\langle g', w\rangle \in \mathbf{W}_{\mathfrak{P}}$, where g is obtained from g' by replacing every hook $\mathtt{h}$ by $r_n(\mathtt{h})$. Since r_n extends to a mediator, we have for every such weight $\langle g, w\rangle$ and state $q \in Q_2 = Q_3$ that $q \models g$ iff $b^-(q) = e(q, \mathfrak{P}) \models g'$. It follows that $wgt_3(q) = wgt_1(e(q, \mathfrak{P})) = wgt_2(q)$. Consequently, $wgt_3 = wgt_2$.

We have shown that $\mathbf{M}_3 = \mathbf{M}_2$, and thus that $\mathbf{M}_2$ is the MDP $\mathbf{M}[\mathfrak{P}^{r_n}]$ induced by $\mathfrak{P}^{r_n}$. Therefore, m_n mediates from $\mathbf{M}_1[\mathfrak{P}]$ to $\mathbf{M}_2[\mathfrak{P}^{r_n}]$, and by Definition 4.2, r_n is a sound rewriter for $\mathfrak{P}$. $\square$

While the rewriter r_n is a sound rewriter, it may not be efficiently computable in practice: For each hook $\mathtt{h} \in \mathcal{H}_{\mathfrak{I}}$, the set $\mathfrak{F}_{\mathtt{h}}$ contains up to exponentially many elements. Since entailment testing in DLs can be an expensive operation, a brute-force method for computing r_n would not perform well in practice. Furthermore, the rewritten program $\mathfrak{P}_{r_n}$ would be very large. As an optimization, we can discard from $\mathfrak{F}_{\mathtt{h}}$ all sets $\mathbf{F} \in \mathfrak{F}_{\mathtt{h}}$ for which there exists some $\mathbf{F}' \in \mathfrak{F}_{\mathtt{h}}$ s.t. $\mathbf{F}' \subset \mathbf{F}$, that is, we only need the subset-minimal sets in $\mathfrak{F}_{\mathtt{h}}$. Those are the subset minimal sets $\mathbf{F}' \subseteq \mathcal{F}$ s.t. $\mathcal{O}_{\mathfrak{P}} \cup F' \models \mathsf{cD}(\mathtt{h})$. Subset-minimal subsets of an ontology that entail a consequence are usually called *justifications* [Hor11].

Definition 4.3. Given an ontology $\mathcal{O}$ and an axiom (set) α s.t. $\mathcal{O} \models \alpha$, a subset $\mathcal{J} \subseteq \mathcal{O}$ is a *justification of* $\mathcal{O} \models \alpha$ iff $\mathcal{J} \models \alpha$ and for every $\mathcal{J}' \subsetneq \mathcal{J}$ we have $\mathcal{J}' \not\models \alpha$. We denote by $\mathsf{J}(\mathcal{O}, \alpha)$ the set of all justifications of $\mathcal{O} \models \alpha$. Let $\mathcal{O}' \subseteq \mathcal{O}$. The set of justifications for $\mathcal{O} \models \alpha$ *relative to* $\mathcal{O}'$ is the set

$$\mathsf{J}(\mathcal{O}, \alpha, \mathcal{O}') = \{\mathcal{J} \setminus \mathcal{O}' \mid \mathcal{J} \in \mathsf{J}(\mathcal{O}, \alpha)\}.$$

The computation of justifications is called *axiom pinpointing* for which various algorithms have been developed for different DLs [SC03, BPS07, Hor11]. Some of these algorithms can be adapted to compute also relative justifications (see also Section 6). To determine in which ontologized states a hook $\mathtt{h}$ is active, we only need to consider the justifications of $\mathcal{O}_{\mathfrak{P}} \cup \mathcal{F} \models \mathsf{cD}(\mathtt{h})$ relative to the static background knowledge $\mathcal{O}_{\mathfrak{P}}$, which are exactly the subset-minimal sets in $\mathfrak{F}_{\mathtt{h}}$ we are looking for. Thus we can replace the sets $\mathfrak{F}_{\mathtt{h}}$ used in the naive rewriter by the sets $\mathsf{J}(\mathtt{h}) = \mathsf{J}(\mathcal{O}_{\mathfrak{P}} \cup \mathcal{F}, \mathsf{cD}(\mathtt{h}), \mathcal{O}_{\mathfrak{P}})$. For $\mathtt{h} \in \mathcal{H}_{\mathfrak{I}}$ the rewriter r_J is defined as follows:

$$r_J(\mathtt{h}) = \bigvee_{\mathcal{J} \in \mathsf{J}(\mathtt{h})} \bigwedge_{\alpha \in \mathcal{J}} \mathsf{Dc}(\alpha). \tag{10}$$

We call r_J the *justification-based rewriter* for $\mathfrak{P}$.

Lemma 4.4. Let $\mathfrak{P}$ be an ontologized program and r_J the justification-based rewriter for $\mathfrak{P}$. Then, r_J is a sound rewriter for $\mathfrak{P}$.

Proof. The only element in the proof of soundness of the naive rewriter in Lemma 4.3 that we have to adapt is Step **(3)**, which establishes that the extension of the rewriter r_n to the label set of $\mathbf{M}_1 = \mathbf{M}[\mathfrak{P}]$ is a mediator from $\mathbf{M}_1$ to $\mathbf{M}_2$. Let m_J be defined as $m_J(\gamma) = \gamma$ for $\gamma \in \mathbb{C}(\mathcal{V}_{\mathfrak{P}})$ and $m_J(\gamma) = r_J(\gamma)$ otherwise. We show that m_J is a mediator from $\mathbf{M}_1$ to $\mathbf{M}_2$, after which the rest of the proof works as before. For this, we need to show that for every hook $\mathtt{h} \in \mathcal{H}_{\mathfrak{I}}$ and every ontologized state $q \in Q_1$, $\mathtt{h} \in \lambda_1(q)$ iff $\lambda_2(b(q)) = \lambda_2(\eta_q) \models m_J(\mathtt{h})$.

Assume $\mathtt{h} \in \lambda_1(q)$. Then, by Definition 3.3, $\mathtt{h}$ is active in q, which by Definition 3.2 means that $\mathcal{O}_q \models \mathsf{cD}(\mathtt{h})$. Consequently, there exists a justification $\mathcal{J}$ for $\mathcal{O}_q \models \mathsf{cD}(\mathtt{h})$ relative to $\mathcal{O}_{\mathfrak{P}}$. Since $\mathcal{O}_q \subseteq \mathcal{O}_{\mathfrak{P}} \cup \mathcal{F}$ by Condition **S2** in Definition 3.2, $\mathcal{J}$ also a justification for $\mathcal{O}_{\mathfrak{P}} \cup \mathcal{F} \models \mathsf{cD}(\mathtt{h})$ relative to $\mathcal{O}_{\mathfrak{P}}$, which means $\mathcal{J} \in \mathsf{J}(\mathtt{h})$. Moreover, by Condition **S4**, $\lambda_2(\eta_q) = \lambda_2(b(q)) \models \mathsf{Dc}(\alpha)$ for every $\alpha \in \mathcal{J} \subseteq \mathcal{O}_q$, which means that $\lambda_2(b(q)) \models \bigwedge_{\alpha \in \mathcal{J}} \mathsf{Dc}(\alpha)$ and $\lambda_2(b(q)) \models m_J(\mathtt{h})$. Assume $\mathtt{h} \notin \lambda_1(q)$. Then, $\mathcal{O}_q \not\models \mathsf{cD}(\mathtt{h})$, which also means that no subset of $\mathcal{O}_q$ entails $\mathsf{cD}(\mathtt{h})$. It follows that for no $\mathcal{J} \in \mathsf{J}(\mathtt{h})$, $\mathcal{J} \subseteq \mathcal{O}_q$, and thus that $\lambda_2(\eta_q) = \lambda_2(b(q)) \not\models m_J(\mathtt{h})$. $\square$

As direct consequence of Lemmata 4.2, 4.3, and 4.4 we obtain the main theorem of this section:

Theorem 4.1. Let $\mathfrak{P}$ be an ontologized program with interface hooks $\mathcal{H}_{\mathfrak{I}}$, r the naive or the justification-based rewriter for $\mathfrak{P}$, and Φ be any probabilistic property or expectation property over the label set of $\mathbf{M}[\mathfrak{P}]$. Then, $\mathbf{M}[\mathfrak{P}] \models \Phi$ iff $\mathbf{M}[\mathfrak{P}^m] \models \Phi^m$.

5. Inconsistency Handling and Further Semantics

Under consistency-independent semantics, it is well possible that an ontologized program enters a state whose DL perspective is inconsistent. This indicates that the state describes a situation that should be impossible according to the background knowledge.

Example 5.1. In our running example, it could be the case that some processes run only on servers with a specific architecture. Suppose our background ontology contains additionally the following axioms:

$$\mathsf{ArchitectureA} \sqcap \mathsf{ArchitectureB} \sqsubseteq \bot \qquad \mathsf{ArchitectureA} \sqsubseteq \forall \mathsf{runsProcess}.\mathsf{ArchitectureA}$$
$$\mathsf{ArchitectureA}(s_1) \qquad \mathsf{ArchitectureB}(p_1)$$

These axioms state that the two concepts $\mathsf{ArchitectureA}$ and $\mathsf{ArchitectureB}$ are disjoint, meaning that the two architectures A and B are incompatible. Furthermore, any server with architecture A can only run processes properly that were compiled for the same architecture. The Server 1 is of architecture A through the axiom $\mathsf{ArchitectureA}(s_1)$ and Process 1 is of architecture B through the axiom $\mathsf{ArchitectureB}(p_1)$. With these additional axioms, the ontologized program state shown in Figure 2 would be inconsistent, as Process 1 is now running on a server that does not support the architecture the process has been compiled for. ◇

Inconsistent ontologies entail any axiom and thus any hook. Furthermore, hooks are associated to states based on the entailments in the DL perspective and thus, all hooks are active in an inconsistent state. This usually indicates undesired behaviors of the ontologized program and asks for a special treatment of inconsistent states. There are different ways in how one can deal with such situations:

1. Make sure the program component deals with such a situation using a sort of exception handling,
2. make sure the ontologized program never enters an inconsistent state, or
3. use an alternative semantics which makes inconsistent states impossible.

In this section, we discuss these options in detail and specify alternative semantics to treat inconsistencies. We first have a closer look at inconsistencies under consistency-independent semantics, how hard it is to detect them, and how they might be used in a verification process using probabilistic model checking. Then, we define two alternative semantics which deal with inconsistent states in a different way, the *probability-normalized semantics* and the *probability-preserving semantics*. Which semantics is appropriate ultimately depends on the nature of the probabilities involved. The right semantics thus depends on the use case. While these semantics avoid inconsistent states altogether, it is still possible for an ontologized program to be *inconsistent* under these semantics, which is the case if no induced MDP for the respective semantics can be found. Therefore, a central reasoning task is to determine, whether an ontologized program is consistent under the respective semantics. We keep practical approaches for deciding this as future work, and analyze the computational complexity of determining inconsistencies under the three semantics from a theoretical perspective. This analysis also reveals a close relation between ontologized programs and space-bounded Turing machines with oracles.

5.1. Auxiliary Lemmas for the Hardness Results

Before we introduce and discuss the semantics, we give some technical results in preparation for the proofs of our complexity results. We define the size of a concept, an axiom, or an ontology in the usual way as its string length, that is, the number of symbols needed to write it down. Here concept, role, and individual names count as one, as do logical operators, and number restrictions are encoded using unary encoding. We first show how to represent ontologies using on a polynomially bounded enumeration of DL axioms by renaming axioms. In order to show this, we need a more liberal definition of conservative extensions where, in contrast to the usual conservative extensions, we may also rename concept or role names.

Definition 5.1. A *renaming* is a bijective function $\varsigma\colon \mathsf{N_C} \cup \mathsf{N_R} \cup \mathsf{N_I} \to \mathsf{N_C} \cup \mathsf{N_R} \cup \mathsf{N_I}$ s.t. $\varsigma(X) \in \mathsf{N_C}$ iff $X \in \mathsf{N_C}$, $\varsigma(X) \in \mathsf{N_R}$ iff $X \in \mathsf{N_R}$ and $\varsigma(X) \in \mathsf{N_I}$ iff $X \in \mathsf{N_I}$. Given an ontology $\mathcal{O}$ and a renaming ς, we denote by $\mathcal{O}\varsigma$ the result of replacing every name X in $\mathcal{O}$ by $\varsigma(X)$.

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 $\mathcal{O}_2$, $\mathcal{O}_1\varsigma \models \alpha$ iff $\mathcal{O}_2 \models \alpha$.

Lemma 5.1. There exists a polynomial function p s.t. for every $n \in \mathbb{N}$, there exists a polynomially computable *axiom enumeration* π, that is, a function mapping indices $\{0, \ldots, p(n)\}$ to $\mathcal{SROIQ}$ axioms, s.t. for every ontology $\mathcal{O}$ of size at most n, we can compute in time polynomial in n a set $I \subseteq \{0, \ldots, p(n)\}$ s.t. $\{\pi(i) \mid i \in I\}$ is a conservative extension of $\mathcal{O}$ modulo renaming.

Proof. Let $n \in \mathbb{N}$ be the bound on ontology sizes. Using standard structural transformations (see for example [RKGH16]), we can compute for every $\mathcal{SROIQ}$ ontology $\mathcal{O}$ in polynomial time a *normalized ontology* that is a conservative extension of $\mathcal{O}$ and in which every axiom is in one of the following twelve 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. $A_1 \sqsubseteq (\geqslant i\ r_1.A_2)$,
5. $(\geqslant i\ r_1.A_1) \sqsubseteq A_2$,
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. The number m is polynomial in n. The number of concept, role and individual names 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

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

s.t. for every any normalized ontology $\mathcal{O}$ of size m there exists a renaming ς s.t. $\mathcal{O}\varsigma$ uses only names from $\mathbf{X}$. Since 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 every normalized axiom is of one of the twelve forms above, it uses at most three 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 axiom enumeration π required by the lemma can then be defined as follows: for $i \in \{0, \ldots, p(n)\}$, we set $\pi(i) = \alpha$, where α is determined as follows:

1. α is of the form $(i \mod 12){+}1$ as above,
2. for $a \in \{1, 2, 3\}$, the ath (concept or role) name occurring in α is X_j, 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 5.1, 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 $\{0, 1, \square\}$ ($\square$ 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 $\langle \pi, \alpha \rangle$*.

Lemma 5.2. Let k be a complexity class and $\mathcal{L}$ a DL for which deciding entailment is k-hard, and let T be a Turing machine with k-oracle and bound n on the size of the oracle tape. Then, 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 5.1 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. □

5.2. Consistency-Independent Semantics

Before we can discuss inconsistencies under the consistency-independent semantics, we need a formal definition of what it means for an ontologized state of an ontologized program to be inconsistent. Based on this definition, we say an ontologized program is inconsistent under consistency-independent semantics if it can reach an inconsistent state.

Definition 5.2. An ontologized state q is *inconsistent* if $\mathcal{O}_q$ is inconsistent. Let $\mathfrak{P}$ be an ontologized program, and let $\mathbf{M}[\mathfrak{P}]$ be the weighted MDP induced by $\mathfrak{P}$ as in Definition 3.3. Then, $\mathfrak{P}$ is *inconsistent* if there exists a state $q \in Q$ that is inconsistent and reachable from the initial state ι.

Even though inconsistencies are allowed, we may still want to avoid inconsistent states under consistency-independent semantics, and directly construct the ontologized program in such a way that inconsistent states are never reached. Inconsistent states may then indicate a bug in the ontologized program that we might want to detect using appropriate tool support. We provide a theoretical analysis about the arising complexity in the next subsection. The proof shows a close relation between $\mathcal{L}$-ontologized programs and polynomially-space bounded Turing machines depending on the complexity k of deciding entailment in $\mathcal{L}$. Thus it also gives an idea of the expressivity of ontologized programs. The underlying reduction to prove hardness via a reduction to oracle Turing machines will be reused for later results.

Theorem 5.1. Let k be a complexity class and $\mathcal{L}$ a DL for which deciding entailment is k-complete. Then, deciding consistency of $\mathcal{L}$-ontologized programs is $\text{PS}\textsc{pace}^k$-complete, even for non-probabilistic programs.

Proof. Let $\mathbf{M}[\mathfrak{P}]$ be the MDP induced by some ontologized program $\mathfrak{P}$ as in Definition 3.3 where $\mathcal{O}$ 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 nondeterministically select a command in $\mathbf{P}_{\mathfrak{P}}$ whose guard is satisfied. Then, we nondeterministically 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 $\text{PS}\textsc{pace}^k$.

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

$$T = \langle Q, \Gamma, \gamma_0, \Gamma_i, \Gamma_o, q_0, F, \delta, q_?, q_{\text{yes}}, q_{\text{no}} \rangle$$

be such a Turing machine, where

- Q are the *states*,
- $\Gamma = \{\gamma_0, \dots \gamma_m\}$ is the *tape alphabet*,
- $\gamma_0 = \square$ 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
- $q_{\text{yes}}, q_{\text{no}} \in Q \setminus F$ 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_{yes} in the next step if the oracle accepts the content of the oracle tape, and entering q_{no} otherwise. Based on T and the input word $w = w_0 \dots w_l$, 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 5.2 to construct a polynomially space-bounded Turing machine T' with $\mathcal{L}$-oracle for $\langle \pi, A(a)\rangle$ that accepts w iff so does T. Specifically, we obtain a set $\mathfrak{A}$ of DL axioms, polynomially bounded in $|w|$, an axiom enumeration $\pi\colon \{0,\ldots,r(|w|)\} \mapsto \mathfrak{A}$, where r is a polynomial,

$$T' = \langle Q', \Gamma, \gamma_0, \Gamma_i, \{\Box, 0, 1\}, q_0, F', \delta', q_?, q_{\text{yes}}, q_{\text{no}}\rangle,$$

where $Q' = \{q_0, \ldots, q_n\}$, and T' accepts the same language as T, is still polynomially space-bounded, but uses an $\mathcal{L}$-oracle. If T' is in state $q_?$, and the oracle tape contains the word $w_0 \ldots w_{r(|w|)}$, the Turing machine moves into the state q_{yes} iff $\{\pi(i) \mid 0 \leq i \leq r(|w|), w_i = 1\} \models A(a)$, and otherwise moves to q_{no}. We can construct T' so that queries to the $\mathcal{L}$-oracle are always posed using consistent ontologies, and in fact even that the oracle tape never describes an inconsistent ontology. 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. We start by introducing the program component of the ontologized program. Recall that for each variable v in the program, we need to specify also its *range* via the function ρ. Specifically, variable v can only take values in $\{-\rho(v), \ldots, \rho(v)\}$.

The program uses the following variables:

- $\texttt{state}$ with $\rho(\texttt{state}) = |Q'|$ stores the state of the Turing machine,
- for $0 \leq i \leq p(|w|)$, $\texttt{wtape\_i}$ with $\rho(\texttt{wtape\_i}) = |\Gamma|$ stores the letter on the work tape position i,
- for $0 \leq i \leq r(|w|)$, $\texttt{otape\_i}$ with $\rho(\texttt{otape\_i}) = 1$ stores the letter on the oracle tape at position i,
- $\texttt{wpos}$ with $\rho(\texttt{wpos}) = p(|w|)$ stores the current position on the default tape,
- $\texttt{opos}$ with $\rho(\texttt{opos}) = r(|w|)$, 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

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

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

$$\begin{pmatrix} & \texttt{state}{=}i_1 & & \\ \wedge & \texttt{wtape\_i}{=}i_2 & \wedge & \texttt{wpos}{=}i \\ \wedge & \texttt{otape\_j}{=}i_3 & \wedge & \texttt{opos}{=}j \end{pmatrix} \mapsto \begin{pmatrix} \texttt{state} := i_4 & \\ \texttt{wtape\_i} := i_5, & \texttt{wpos} := \texttt{wpos} + \mathsf{Move}_d, \\ \texttt{otape\_j} := i_6, & \texttt{opos} := \texttt{opos} + \mathsf{Move}_o \end{pmatrix} \tag{11}$$

Note that we omitted the action label of the command, as it is irrelevant for our purposes. To handle the behavior of the oracle, we further add the following commands, where $q_? = q_{i_1}$, $q_{\text{yes}} = q_{i_2}$ and $q_{\text{no}} = q_{i_3}$

$$\texttt{state} = i_1 \wedge \texttt{oracle\_yes} \quad \mapsto \quad \texttt{state} := i_2 \tag{12}$$

$$\texttt{state} = i_1 \wedge \neg\texttt{oracle\_yes} \quad \mapsto \quad \texttt{state} := i_3 \tag{13}$$

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

As DL axioms, we set as background ontology $\mathcal{O} = \emptyset$ and for the fluents $\mathcal{F} = \mathfrak{A} \cup \{\top \sqsubseteq \bot\}$. It remains to specify the interface functions Dc and cD. The interface is used to translate the oracle calls, as well as to establish the correspondence between acceptance of the Turing machine to inconsistency of the ontologized program.

For Dc, we set for every $i \in \{0, \ldots, r(|w|)\}$,

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

to translate the oracle calls, and

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

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

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

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. $\Box$

Ontology-mediated PMC under consistency-independent semantics. Inconsistency in ontologized programs is not such a severe handicap as in other logic-based formalisms, as probabilistic properties can still be evaluated on MDPs with inconsistent states. In fact, they can even have a designated meaning, e.g., for modeling exceptions the program has to face. Note that commands can be used to detect if a program is in an inconsistent state. To this end, we can employ a designated hook `inc` that is satisfied in exactly those states that are inconsistent, i.e., a hook for which $\mathsf{cD}(\texttt{inc}) = (\top \sqsubseteq \bot)$. Using this hook in guards of the program, the program 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 full set of hooks), the command perspective of states can still differ. Specifically, private variables can encode additional information about the current state that can be used to decide how to react to inconsistencies.

The dedicated hook `inc` can then also be used to analyze whether the exception handling in the program satisfies the intended requirements. An example for an ontology-mediated PMC analysis under consistency-independent semantics would be to decide whether

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

Intuitively, this probabilistic property states that an exception is always successfully handled with a high probability of at least 95%. Here, successful handling means that, whenever an inconsistent state is reached, a consistent state is reached within at most two steps. Such a probabilistic property can be checked using the rewriting technique presented in Section 4 and hence, has full tool support by the PMC tool Prism [KNP11].

5.3. Probability-Normalizing Semantics

If a 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 how inconsistent states are handled. On the other hand, it is more convenient 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 paths that can lead to an inconsistent state, and then normalize the resulting probabilities 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 provides further reusability of the behavioral and the DL component of the program towards putting constraints on the state space. In the running example of the multi-server platform, our extended background ontology disallows execution of Process 1 on Server 1. Consequently, the outcome of the command (9) in Example 3.1 can only be the update in which Process 1 is moved to Server 3, which means that this update is now executed with a probability of 1. Note that we could achieve this behavior in this particular example also under consistency-independent semantics. However, this requires an explicit handling of inconsistencies and modifications in the program that "provisions" the possible variable evaluations. More specifically, we then need a possibility to decide for each update in the command whether it would lead to an inconsistent state or not. While this works out well in our running example (see Section 6.2), we cannot expect such an encoding for any ontologized program.

To capture the probability-normalizing semantics, we define a different MDP that fulfills the above conditions. Intuitively, under probability-normalizing semantics, we consider the MDP obtained by removing all inconsistent states in the induced MDP and normalizing its 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.

Definition 5.3. Let $\mathfrak{P}$ be an ontologized program, and $\mathbf{M}[\mathfrak{P}] = \langle Q, Act, P, \iota, \Lambda, \lambda, wgt\rangle$ the MDP induced by $\mathfrak{P}$. An MDP $\mathbf{M}$ is *probability-normalizing w.r.t.* $\mathfrak{P}$ if $\mathbf{M} = \langle Q', Act, P', \iota, \Lambda, \lambda', wgt'\rangle$, where

PN1 $Q' \subseteq Q$ contains only consistent states,
PN2 λ' and wgt' are obtained from λ and wgt by restricting their domain to Q',
PN3 for all $q \in Q$ and $\sigma \in Act$ s.t. $P(q, \sigma)$ is defined, we have for all $q' \in Q'$:

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

$\mathfrak{P}$ is called *probability-normalizable* if there exists a probability-normalizing MDP w.r.t. by $\mathfrak{P}$. In this case, we denote by $\mathbf{M}_{pn}[\mathfrak{P}]$ the *MDP induced by* $\mathfrak{P}$ *under probability-normalizing semantics* as the unique maximal probability-normalizing MDP w.r.t. $\mathfrak{P}$. Here, maximality is understood w.r.t. subset-maximality of Q' and where the P' is defined for the largest set of tuples $(q, \sigma) \in Q' \times Act$.

Recall that according to our definition, MDPs cannot have final states. Therefore, to obtain the probability-normalizing MDP, it does not suffice to remove inconsistent states, but also states that would otherwise become final, since all of their every successor states are removed. As this operation might lead to a removal of the initial state, not every ontologized program is probability-normalizable. If the program is probability-normalizable, we are not interested in any arbitrary probability-normalizing MDP, but in the one that maximizes both the set Q' of states and the set P' of transitions. The induced MDP under probability-normalizing semantics is always unique: Given two probability-normalizing MDPs $\mathbf{M}_1$ and $\mathbf{M}_2$ w.r.t. the same ontologized program $\mathfrak{P}$, we can construct their component-wise union, which is again a probability-preserving MDP. By doing this for all probability-normalizing MDPs for $\mathfrak{P}$, we obtain $\mathbf{M}_{pn}[\mathfrak{P}]$, the maximal probability-normalizing MDP.

Determining consistency under probability-normalizing semantics. Probability-normalizability is a stronger form of consistency in the following sense: if an ontologized program is not probability-normalizable, then not only can it reach an inconsistent state in $\mathbf{M}[\mathfrak{P}]$, but in fact every path in $\mathbf{M}[\mathfrak{P}]$ starting from the initial state eventually leads to an inconsistent state. We may thus also speak of *inconsistency under probability-normalizing semantics.* Even though this notion of inconsistency is much weaker than the one defined in Definition 5.2, the complexity of determining probability-normalizability is in fact the same. Intuitively, this is because we need to solve dual problems for both consistency notions: for inconsistency under consistency-independent semantics, we need to determine whether *some* path leads from the initial state to an inconsistent state, while for inconsistency under probability-preserving semantics, we need to determine whether *all* paths from the initial state lead to an inconsistent state.

Theorem 5.2. Let k be a complexity class and $\mathcal{L}$ a DL for which deciding entailment is k-complete. Then, deciding whether an $\mathcal{L}$-ontologized program is probability-normalizable is PSPACE^k-complete, even for non-probabilistic programs.

Proof. For the upper bound, we 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 5.1 to determine whether a program *is* probability-normalizable: instead of testing for inconsistency of the current state, we test for consistency. Note that for an ontologized program to be probability-normalizable, it suffices to have a single execution path that never reaches an inconsistent state. Because $\mathbf{M}[\mathfrak{P}]$ has no final states, this execution path has infinite length. However, because the state space is bounded, any infinite execution path would need to visit some state twice. We use a counter to put an exponential bound on the visited states. After we visited exponentially many states, we know that some state must have been repeated, and we found an unbounded execution path that never visits an inconsistent state, and accept.

For the lower bound, we note that the reduction used in the proof for Theorem 5.1 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-probabilistic, but even *deterministic*, in the sense that in every ontologized state, there is at most one command that can be executed. Specifically, there is exactly one command (11) for each assignment to the variables `state`, `wpos`, `opos`, `wtape_i` and `otape_j`, which means that in each ontologized state, at most one of these commands is executable. Moreover, for the assignment `state` $= i_1$, where $q_{i_1} = q_?$ denotes the oracle query state, there is no such command, because there is no transition in the Turing machine for the oracle state, and exactly one of the commands (12) and (13) is executable if `state` $= i_1$, unless 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. □

5.4. Probability-Preserving Semantics

The probability-normalizing semantics changes the probabilities specified in the commands to avoid inconsistent states. This is the right choice if the program uses probabilities to model randomized behavior. For instance, 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, e.g., disallowing migration to incompatible server architectures, then consequently the probabilities need to change as well. There are other scenarios where a change of probabilities is not desired: rather than using probabilistic commands to model randomized behavior, the program component may describe probabilistic outcomes of the modeled 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: the semantics should not allow this probability to become 0% because having a disfunct server would be inconsistent with the background ontology. Rather, states from which we cannot ensure that the server becomes disfunct with a probability of 1% should be impossible by the semantics as well. To capture this behavior, we introduce the *probability-preserving semantics.*

Definition 5.4. Let $\mathfrak{P}$ be an ontologized program, and $\mathbf{M}[\mathfrak{P}] = \langle Q, Act, P, \iota, \Lambda, \lambda, wgt\rangle$ the MDP induced by $\mathfrak{P}$. An MDP $\mathbf{M}$ is *probability-preserving w.r.t.* $\mathfrak{P}$ if $\mathbf{M} = \langle Q', Act, P', \iota, \Lambda, \lambda', wgt'\rangle$, where

PP1 $Q' \subseteq Q$ contains only consistent states,

PP2 λ' and wgt' are obtained from λ and wgt by restricting their domain to Q',

PP3 for all $q \in Q'$ and $\sigma \in Act$ for which $P'(q, \sigma)$ is defined, $P'(q, \sigma) = P(q, \sigma)$.

$\mathfrak{P}$ is called *probability-preservable* if there exists a probability-preserving MDP w.r.t. $\mathfrak{P}$. In this case, we denote by $\mathbf{M}_{pp}[\mathfrak{P}]$ the *MDP induced by* $\mathfrak{P}$ *under probability-preserving semantics*, which is the unique maximal *probability-preserving MDP w.r.t.* $\mathfrak{P}$, where maximality is understood as in Definition 5.3.

The difference to Definition 5.3 is Condition **PP3**, which does not normalize the probabilities in P, but keeps them if possible. Thus $\mathbf{M}_{pp}[\mathfrak{P}]$ can be obtained from $\mathbf{M}[\mathfrak{P}]$ by removing all transitions that lead with a positive probability to an inconsistent state, and then removing all inconsistent states and those states for which all outgoing transitions have to be removed.

Determining consistency under probability-preserving semantics. The consistency notion that is used for the probability-preserving semantics is that of probability-preservability. We can see that this consistency notion is stricter than the one of probability-normalizability: it does not only remove parts of a probabilistic transition that lead to an inconsistent state, but also removes the whole transition if it can lead to an inconsistency. It turns out that deciding probability-preservability is more challenging than for the other consistency notions. Intuitively, the main reason for this is that we now need to take full care of the alternations of non-deterministic and probabilistic choices that underlie the semantics of ontologized programs: in each state, we *non-deterministically* select the next command to execute, after which there is a *probabilistic* selection of the outcome of that command. If one of the probabilistic choices would lead to an inconsistency, then we have to restrict the set of non-deterministic choices. It is thus not sufficient anymore to find a single execution path that leads to an inconsistency, as in the case of inconsistency in consistency-independent semantics, or a single execution path that leads to a consistency, as is the case for probability-normalizing semantics. Instead, execution trees have to be analysed.

Theorem 5.3. Let k be a complexity class and $\mathcal{L}$ be a DL for which deciding entailment is k-complete. Then, deciding whether an $\mathcal{L}$-ontologized program is probability-preservable is APSPACE^k-complete. For non-probabilistic 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 $\exists$-non-determinism and $\forall$-non-determinism, meaning that the Turing machine can switch between $\exists$-non-deterministic choices and $\forall$-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 $\exists$-non-deterministic transition to a configuration that is accepting, or 3) all $\forall$-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 5.1 that runs in alternating polynomial space and decides probability-preservability of ontologized programs. Let $\mathfrak{P}$ be an ontologized program as in Definition 3.1, 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 command that can be executed on the current state, and 3) all successor-states into which this command brings us with a positive probability are included in Q'. Our algorithm thus proceeds as follows based on the currently guessed state q. If q is inconsistent, we reject. Otherwise, we non-deterministically select a command $\langle g, \sigma\rangle \in \mathbf{C}_{\mathfrak{P}}$ s.t. $q \models g$ ($\exists$-non-determinism) and continue on all states $q' \in Q$ such that $P(q, \sigma, q') > 0$ ($\forall$-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 $\mathrm{APS{\small PACE}}^k$. If the program is non-probabilistic, the $\forall$-non-determinism is not needed, and the algorithm runs in $\mathrm{PS{\small PACE}}^k$.

For the lower bound, we adapt the reduction used in the proof for Theorem 5.1. However, this time, we reduce the word problem for polynomially space bounded *alternating* Turing machines with k-oracle. Specifically, let

$$T = \langle Q, \Gamma, \gamma_0, \Gamma_i, \Gamma_o, q_0, \delta, q_?, q_{\text{yes}}, q_{\text{no}}, g\rangle,$$

be such a Turing machine, where $Q = \{q_0, \ldots, q_n\}$, $\Gamma = \{\gamma_0, \ldots \gamma_m\}$ is the standard tape alphabet, $\Gamma_i \subseteq \Gamma$ is the input alphabet containing the blank symbol $\gamma_0 = \Box \in \Gamma$, Γ_o is the oracle alphabet, q_0 is the initial state,

$$\delta\colon Q \times \Gamma \times \Gamma_o \to \wp(Q \times \Gamma \times \{-1, 0, +1\} \times \Gamma_o \cup \{-1, 0, +1\})$$

is the transition function, the states $q_?$, q_{yes}, $q_{\text{no}} \in Q$ 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 oracle works as in the proof for Theorem 5.1, and a configuration with state q is *accepting* if

- $g(q) = \texttt{accept}$,
- $g(q) = \exists$ and one of the successor configurations is accepting, or
- $g(q) = \forall$ and all successor configurations are accepting.

Again, we use Lemma 5.2 to construct, based on T and the input word w, a polynomially bounded set $\mathfrak{A}$ of DL axioms, an axiom enumeration $\pi\colon \mathfrak{A} \to \{0, \ldots, r(|w|)\}$, where r is a polynomial, and an alternating Turing machine

$$T' = \langle Q', \Gamma, \gamma_0, \Gamma_i, \{0, 1, \Box\}, q_0, \delta', q_?, q_{\text{yes}}, q_{\text{no}}, g'\rangle$$

with an $\mathcal{L}$-oracle for $\langle \pi, A \sqsubseteq B\rangle$, 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 $\sigma_0 \ldots \sigma_{r(|w|)}$, the Turing machines moves into the state q_{yes} iff

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

and otherwise moves to q_{no}. Based on this Turing machine, we construct the ontologized program $\mathfrak{P}_{T,w}$ in a similar way as in the proof for Theorem 5.1 by making the accepting states of the Turing machine correspond to inconsistent program states, except that this time we use a different set $\mathbf{C}_{\mathfrak{P}}$ of commands.

Let p be the polynomial that bounds the number of tape cells. We first model the $\forall$-transitions. For every

$$t = \langle q_{i_1}, \gamma_{i_2}, \gamma_{i_3}\rangle \in Q \times \Gamma \times \Gamma_o,$$

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 $i \in \{0, \ldots, p(|w|)\}$ and every $j \in \{0, \ldots, r(|w|)\}$, we use the (non-probabilistic) command

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

That is, we model $\forall$-non-determinism using the non-deterministic choices of the ontologized program. The reason for this is that for the program to be *not* probability preservable, *every* command that we can execute in an ontologized state would lead to an inconsistency. This corresponds to the situation in which every successor state in the Turing machine is accepting.

While $\forall$-transitions are modeled via non-determinism in the program, $\exists$-transitions are modeled via probabilities. Let

$$t_1 = \langle q_{i_1}, \gamma_{i_2}, \gamma_{i_3} \rangle \in Q \times \Gamma \times \Gamma_o,$$

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

$$\begin{pmatrix} & \texttt{state} = i_1 & & \\ \wedge & \texttt{wpos} = i & \wedge & \texttt{wtape\_i} = i_2 \\ \wedge & \texttt{opos} = j & \wedge & \texttt{otape\_j} = i_3 \end{pmatrix} \mapsto \sigma$$

where σ is the probabilistic 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

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

The corresponding transition has to be excluded from the probability-preserving MDP if *at least one* of the updates leads with a positive probability to an inconsistent state, which corresponds to the acceptance condition of $\exists$-states: *at least one* successor configuration has to be accepting.

The commands that handle the use of the oracle are as in the proof for Theorem 5.1. It is standard to show that the resulting program is probability-preserving iff w is accepted by the Turing machine. As a consequence, we obtain that probability-preservability is $\mathrm{APS{\small PACE}}^k$-hard.

Now assume that for no state $q \in Q$ we have $g(q) = \exists$. Then, T corresponds to a Turing machine in which *every* execution path should lead to an accepting state, for which the word problem is $\mathrm{coNPS{\small PACE}}^k = \mathrm{PS{\small PACE}}^k$-complete. The ontologized program $\mathfrak{P}_{T,w}$ is then non-probabilistic. As a consequence, deciding whether non-probabilistic programs are probability-preserving is $\mathrm{PS{\small PACE}}^k$-hard. $\square$

6. Implementation and Evaluation

We demonstrate the feasibility of ontology-mediated PMC by an implementation of the methods described in Section 4 according to consistency-independent semantics. We apply our implementation on the running example of the multi-server system presented in the last sections, modeled more in detail than described throughout the paper.

6.1. Implementation Details

Our implementation consists of two components: an ontology reasoning component, which computes the rewritings for each hook, and the program generation component, which then integrates those into the program. In the end, we obtain a program in the standard input language of the PMC tool Prism [KNP11].

Ontology reasoning. Our implementation relies on the justification-based rewriter as defined in Section 4.2. The ontology-component of the ontologized program is provided in the standard web ontology language OWL-DL [HKS06]. To compute the hook rewritings, we implemented a tool in Scala that uses the OWL-API [HB11] to parse and access the ontology, and the OWL reasoning system Openllet based on Pellet [SPG$^+$07] for computing the justifications. The OWL-API [KPHS07] comes with an implementation of the *black-box approach* for axiom pinpointing, which can be used with any reasoning system compatible with the OWL-API. However, this approach turned out to be too slow in practice for our purposes. To this end, we decided to rely on Openllet, the only state-of-the art reasoner that supports most of the OWL-DL profile and comes with an integrated *white-box* implementation for computing justifications [SPG$^+$07]. This method for computing justifications worked out to be much faster on our examples. We adapted this implementation to reduce computation times further. Neither the implementation of the black-box approach in the OWL-API nor the one of the white-box approach in Openllet directly support *relative* justifications. Note that, in order to compute the hook rewritings, we only need to vary the fluent axioms in $\mathcal{F}$, and not the axioms in the static DL knowledge $\mathcal{O}$. Since the justification algorithm implemented in Openllet varies both, and thus potentially explores the exponentially many subsets of $\mathcal{O}$, it was not able to create any hook rewriting without running into memory problems.

We adapted the justification algorithm in Openllet to support relative justifications w.r.t. $\mathcal{O}$ and to only return the axioms from $\mathcal{F}$. Especially for large numbers of axioms in $\mathcal{O}$, this reduces the search space for computing justifications drastically.

Program and query specification. For the program component of the ontologized program, we use the input language of Prism and the preprocessor templating functionalities provided by Prism. Note that programs as formally defined in Section 2 form a subclass of the Prism language. However, our implementation supports the full class of the Prism language and even Prism language extensions such as ProFeat [CDKB18]. We further allow the use of hooks in guards that are not specified in the program itself, approaching abstract programs as issued in Section 3.1. Thus, the specified program cannot be analyzed without a specification of hook rewriters, obtained through the interface and reasoning over the given ontology. In the current implementation, the interface directly provides the combination of hook names and axioms for their evaluation during the reasoning process on the given ontology. After the reasoning process, the hook rewritings are generated as standard `formula` expressions in the input language of Prism that allow for a macro-like replacement of hooks by the Boolean expressions returned from ontology reasoning. Technically, the concatenation of the program specification and the hook `formula` expressions then leads to a standard Prism program, which we use as input for a quantitative analysis by Prism. Note that due to the global evaluation of `formula` expressions, hook rewriters apply also for hooks used in probabilistic, expectation, and quantile queries and hence allow for expressing knowledge-dependent property specifications.

6.2. Case Study Setup

Our evaluation scenario is based on the multi-server platform example used throughout this paper, modeled in more detail. The test suite itself consists of the ontology, the program, and the quantitative queries for the analysis. For the automated generation of different system setups, we use scripts that take the number of servers and processes as input and generate ontologies and programs.

Ontology: multi-server knowledge base. The ontology conducted for this study uses the DL $\mathcal{ALCQ}$. It models background knowledge on different capacity constraints of the servers and different priority settings on the processes. To model compatibility of processes with server architectures (see Example 5.1), we distinguish between two types of architectures: $\lozenge$ and $\blacklozenge$. Servers could be of and processes could be compiled for either architecture. Then, $\lozenge$-processes can only be executed on servers having architecture $\lozenge$ and likewise, $\blacklozenge$-processes can only be executed on servers with architecture $\blacklozenge$. We used three different types of hooks to link the abstract program with the ontology:

critical system state hook "`critical`" is active in states the system should avoid and which we call *critical states* in the following,

migrate hook "`migrateServer`s" is active in those situations where the system should schedule the migration of some process on server s to another server, and

inconsistency hook "`inc_`p`_`s" is active when a process p assigned to a particular server s would lead to an inconsistent state of the system, taking into account both capacity and compatibility limitations.

We defined four ontologies in total, which we denote by $\mathcal{O}_1, \ldots, \mathcal{O}_4$ in the following. These ontologies differ in their capacity and compatibility constraints, as well as properties of the server and processes. Table 3 gives an overview of the four ontologies in terms of those concepts that enable the critical system state hook (C), the migrate hook (M), and the inconsistency hooks (I). For instance, for $\mathcal{O}_2$, the system is in a critical system state when a prioritized process runs on an overloaded server; the migrate hook becomes active when either a prioritized process runs on an almost overloaded server or some server is overloaded; and a system state is inconsistent when the maximal number of processes on a server is surpassed or a process runs on a server those architecture is not compatible with the process. For ontology $\mathcal{O}_1$, the situation is similar as for $\mathcal{O}_2$, but where processes compiled for incompatible architectures do not lead to an inconsistent

ontology	prioritized runs on overloaded	prioritized runs on almost overloaded	two prioritized on same server	server overloaded	max. number of processes on server	incompatible architecture
$\mathcal{O}_1$	C	M		M	I	
$\mathcal{O}_2$	C	M		M	I	I
$\mathcal{O}_3$	C	M			I	I
$\mathcal{O}_4$	C		C	M	I	I

Table 3. Varying different situations for different ontologies.

system state. Intuitively, the servers according to $\mathcal{O}_1$ could differ to the servers of other ontologies by running virtualization software that enables processing any kind of processes independent from the architecture the process is compiled for.

Program: job processing protocol. To evaluate the flexibility regarding the operational behavior in ontologized programs, we provide two programs components, differing in their policy on how jobs are processed on each server. The first program, denoted $\mathbf{P}_{\text{rand}}$, uses a randomized policy, i.e., the process to be scheduled next is selected by tossing a fair coin. The second, denoted $\mathbf{P}_{\text{rr}}$ is using a deterministic policy selecting the next process in a round-robin fashion. For each program we assume a fixed rate how likely it is in a step to spawn a new process. Processes will then be assigned to a server with compatible system architecture according to a uniform distribution.

As apparent in the specification of the ontologies, an inconsistent system state would occur when a process is running on a server with different architecture compatibility. This undesired situation is resolved explicitly in the program component, following ideas for resolving inconsistencies in the probability-normalizing and probability-preserving semantics. The probability mass of the randomized selection of processes to be executed next in the program $\mathbf{P}_{\text{rand}}$ is redistributed in case of leading to an inconsistent state. After the redistribution, the full probability mass leads to states where all processes run on servers with the same architecture as the process, implementing a similar approach as for the probability-normalizing semantics. Note that when spawning new processes, these processes are also assigned to servers with the same architecture as the process is compiled for, also avoiding inconsistent system states through redistribution. Differently, for the case of migrating processes, the non-deterministic choices responsible for the migration of processes to a server running incompatible architecture are disallowed in a similar fashion as done within probability-preserving semantics. Due to the mixture of these two kinds of inconsistency handling, we perform ontology-mediated PMC under consistency-independent semantics, but with explicit handling of inconsistencies.

Towards a quantitative analysis of both programs $\mathbf{P}_{\text{rand}}$ and $\mathbf{P}_{\text{rr}}$, we consider three different weight assignments. The weight *time* describes the time required for processing by a set of weight assignments $\mathbf{W}_{time} = \{\langle \mathtt{tt}, 1\rangle\}$ where to each step one unit of time is assigned to. Weight *critical* describes the number of critical states entered by weight assignments $\mathbf{W}_{critical} = \{\langle \mathtt{critical}, 1\rangle\}$ assigning weight one to each critical state. Note that thus, reasoning on the ontology is required to determine the weight structure *critical* for the MDP semantics of the program. Finally, *energy* describes energy consumption by weight annotations

$$\mathbf{W}_{energy} \;=\; \mathbf{W}^{\mathtt{migrate}}_{energy} \cup \big\{\langle \bigvee_{1\leq i\leq \#P} \mathtt{server\_proc}i = j, 1\rangle \mid 1 \leq j \leq \#S\big\}$$

where $\#S$ stands for the total number of servers, $\#P$ for the total number of processes in the multi-server system, and $\mathbf{W}^{\mathtt{migrate}}_{energy}$ is the set of weight assignments that assign one unit of energy to each process that has been migrated in the last step[1]. Intuitively, a server to consume one unit of energy when one or more processes are assigned to the server. In case no process is assigned to the server, this server is idling and does not consume any significant amount of energy, which we model with zero energy costs. Furthermore, a migration task that moves a process from one server to another requires one additional unit of energy.

[1] In our formal framework, we only considered *state weights*, i.e., weights that do not depend on the action of the executed command. However, here we employ weight assignments that only occur on a `migrate` action. This is without loss of generality as technically, we could encode the last performed action into the state space of the program and express such *action weights* by our formalism of state weights. However, as Prism also supports action weights, we directly used action weights in our implementation.

In each program, we include a dedicated counter variable *utility* that at any moment evaluates to the total number of jobs completed so far. Here, prioritized processes achieve double the utility of low prioritized processes. With the weight annotations above and the value of the utility counter, trade-offs between costs and utility can be considered where time, number of critical states, or energy consumption are interpreted as costs. For instance, it might be favorable to migrate a prioritized process to a different server. On the one hand this requires one additional unit of energy, but on the other hand it could, depending on the ontology, reduce the chance of entering a critical system state. Furthermore, migrating a process can increase the chance of completing this process and thus achieving more utility.

Quantitative queries. Towards a quantitative analysis we specified the following standard probability and expectation queries:

(i) What is the probability of reaching a critical system state within 15 time steps?

$$\mathsf{P}^{\mathrm{ex}}(\Diamond^{\mathit{time}\leq 15}\mathtt{critical})=?$$

(ii) What is the expected energy consumption until achieving a utility value of at least 20?

$$\mathsf{E}[\mathit{energy}]^{\mathrm{ex}}\big(\Diamond(\mathit{utility}\geq 20)\big)=?$$

(iii) What is the expected number of critical states entered before achieving a utility value of at least 20?

$$\mathsf{E}[\mathit{critical}]^{\mathrm{ex}}\big(\Diamond(\mathit{utility}\geq 20)\big)=?$$

For each of these queries, we consider their minimizing and maximizing variant when ranging over all resolutions of non-deterministic choices in the MDP, i.e., ex $\in \{\min, \max\}$ in the above formalization of queries. Note that query (i) directly contains the hook `critical` in the LTL formula, hence requiring knowledge of the ontology to be evaluated.

To show-case our approach for more advanced model-checking tasks, we investigate the cost-utility trade-off between energy or critical situations and utility in terms of *energy-utility quantiles* [BDD+14]. For this, we consider the following quantile queries:

(iv) What is the minimal energy required to achieve a utility value of at least 20 with probability at least 95%?

$$\mathsf{Qu}_{>0.95}\big(\Diamond^{\leq \mathit{energy}}(\mathit{utility}\geq 20)\big)=?$$

(v) What is the minimal number of critical states entered to achieve a utility value of at least 20 with probability at least 95%?

$$\mathsf{Qu}_{>0.95}\big(\Diamond^{\leq \mathit{critical}}(\mathit{utility}\geq 20)\big)=?$$

Note that the amount of utility achieved is encoded into the state space, i.e., there is a variable *utility* that evaluates to the utility achieved so far. To this end, $\mathit{utility}\geq 20$ is an arithmetic constraint that can be used as label in any LTL formula and evaluated as atomic proposition in the MDP semantics of the program (cf. Section 2.3).

6.3. Evaluation

For an evaluation of our implementation, we performed a quantitative analysis of the aforementioned queries on ontologized programs formed by the two programs $\mathbf{P}_{\mathrm{rand}}$ and $\mathbf{P}_{rr}$ and the four ontologies $\mathcal{O}_1, \ldots, \mathcal{O}_4$, each on systems with two and three servers, respectively. While the system with two servers comprises one server of each architecture, the three-server system has two servers of architecture $\Diamond$ and one of architecture $\blacklozenge$. Within all of these combinations, we obtained $4\cdot 2\cdot 2 = 16$ ontologized programs in total, which we translated into programs expressed in the input language of Prism.

Quantitative analysis results. The numerical results of the probability query (i), the expectation queries (ii) and (iii), and the quantile queries (iv) and (v) are shown in Table 4. In Figure 3, these results are visualized, ordered by probability queries (upper left), expectation and quantile queries for the *energy* weight annotations (upper right), expectation and quantile queries for the *critical* weight annotations (lower left),

server/			prob. critical (i)		exp. energy (ii)		exp. critical (iii)		cost-utility quantile	
processes	ontology	program	min	max	min	max	min	max	energy (iv)	critical (v)
$\Diamond\blacklozenge$/8	$\mathcal{O}_1$	$\mathbf{P}_{\mathrm{rand}}$	0.9988	0.9999	32.16	45.80	18.26	30.53	35	23
		$\mathbf{P}_{\mathrm{rr}}$	0.9985	0.9999	29.41	48.11	15.56	32.67	32	20
	$\mathcal{O}_2$	$\mathbf{P}_{\mathrm{rand}}$	0.9991	0.9991	34.06	34.06	27.87	27.87	37	34
		$\mathbf{P}_{\mathrm{rr}}$	0.9998	0.9998	32.57	32.57	25.66	25.66	35	30
	$\mathcal{O}_3$	$\mathbf{P}_{\mathrm{rand}}$	0.9991	0.9991	34.06	34.06	27.87	27.87	37	34
		$\mathbf{P}_{\mathrm{rr}}$	0.9998	0.9997	32.76	32.76	25.66	25.66	35	30
	$\mathcal{O}_4$	$\mathbf{P}_{\mathrm{rand}}$	0.9819	0.9819	28.22	28.22	20.05	20.05	31	28
		$\mathbf{P}_{\mathrm{rr}}$	0.9845	0.9845	26.77	26.77	16.57	16.57	29	24
$\Diamond\Diamond\blacklozenge$/6	$\mathcal{O}_1$	$\mathbf{P}_{\mathrm{rand}}$	0.6604	0.9983	29.09	58.57	1.73	28.38	31	4
		$\mathbf{P}_{\mathrm{rr}}$	0.6392	0.9983	27.92	62.25	1.53	28.72	30	4
	$\mathcal{O}_2$	$\mathbf{P}_{\mathrm{rand}}$	0.4558	0.9866	29.17	47.63	0.91	14.05	32	3
		$\mathbf{P}_{\mathrm{rr}}$	0.4256	0.9854	27.96	48.17	0.78	13.62	30	2
	$\mathcal{O}_3$	$\mathbf{P}_{\mathrm{rand}}$	0.4558	0.9866	29.17	47.63	0.91	14.05	32	3
		$\mathbf{P}_{\mathrm{rr}}$	0.4256	0.9855	28.73	49.09	0.79	13.66	31	2
	$\mathcal{O}_4$	$\mathbf{P}_{\mathrm{rand}}$	0.7776	0.7776	29.26	32.04	3.37	7.97	32	8
		$\mathbf{P}_{\mathrm{rr}}$	0.7612	0.7612	28.99	31.61	3.06	6.50	32	7

Table 4. Analysis results for the multi-server case study.

and the overall running time to evaluate these queries (lower right, logarithmic scale). The ontologies are denoted by "$\mathcal{O}_i(a)$" where i is the index of the ontology number and "a" indicates the architectures of the servers of the system setup. Left (blue) bars show results for randomized scheduling, right (red) bars show results for round-robin scheduling. Bars span the results for minimizing and maximizing queries, while dots indicate the quantile results.

In the case of the setup with two servers, only under ontology $\mathcal{O}_1$ there is some freedom in performing migrations while keeping a consistent system state as all processes of any architecture can be processed on both servers. The other ontologies $\mathcal{O}_2$, $\mathcal{O}_3$, and $\mathcal{O}_4$ disallow processes to be placed on server with a different architecture than the process, since this would lead to inconsistent system states (see Table 3). Hence, only in the setup $\mathcal{O}_1(\Diamond\blacklozenge)$ (ontology $\mathcal{O}_1$ with two servers, one of each architecture) the minimal and maximal expected number of critical situations and expected energy consumption differ. The probability of reaching a critical situation where a prioritized process runs on an overloaded server is very high in the two-server setup, as there are comparably many processes to be scheduled on two servers. In the three-server setup, the additional server provides enough freedom for migration strategies and less processes than in the two-server setup have to be scheduled. Hence, as we can see in the results of the minimizing probability to reach a critical situation, a clever scheduling can reduce the probability of reaching critical situations in settings $\mathcal{O}_1(\Diamond\Diamond\blacklozenge)$, $\mathcal{O}_2(\Diamond\Diamond\blacklozenge)$, and $\mathcal{O}_3(\Diamond\Diamond\blacklozenge)$. In the setting $\mathcal{O}_4(\Diamond\Diamond\blacklozenge)$, more critical situations can occur according to ontology $\mathcal{O}_4$ (see Table 3) and thus, there is less flexibility to avoid critical situations even with clever strategies.

While cost-utility trade-offs lead to higher costs than corresponding expectations in most of the two-server setups, they are close to the minimal expected cost values in most of the three-server setups. Again, the reason can be seen in the greater migration flexibility the three-server setups provide. This statement is underpinned by the three-server setup $\mathcal{O}_4(\Diamond\Diamond\blacklozenge)$, where ontology $\mathcal{O}_4$ is more restrictive than the other ontologies, and the cost-utility trade-offs are slightly greater than corresponding worst-case expectations. Likewise, in the two-server setup $\mathcal{O}_1(\Diamond\blacklozenge)$, the ontology $\mathcal{O}_1$ is less restrictive ontology than the ontologies within the other two-server setups, leading to cost-utility trade-offs within the expectation ranges.

In all scenarios we see that a round-robin job-selection policy performed slightly better than the randomized one when the objective is to minimize energy consumption, reaching critical system states, and minimizing cost-utility trade-off. This can be explained by the fact that when analyzing optimal schedulers, they can provision the next job selection and thus, can preemptively prevent critical situations by migrating processes or suppress migration to save energy in case there is no urgent need for such a migration.

From our experiments, we can draw the following conclusions: First of all, our approach of separating concerns by loosely coupled program and ontology components enables us to independently specify and

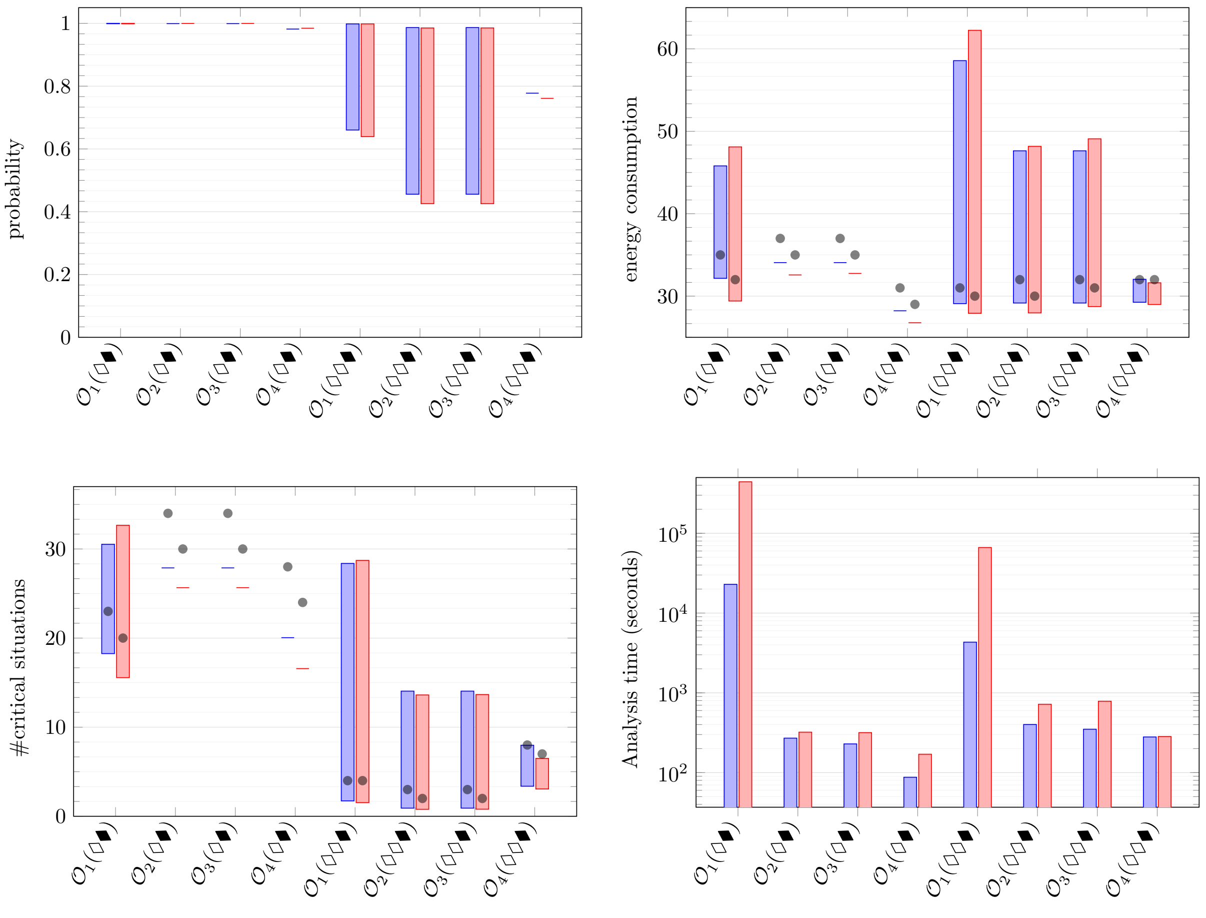

Fig. 3. Results of the ontology-mediated PMC analysis in the multi-server setting and runtime statistics. Left (blue) bars show results for $\mathbf{P}_{\text{rand}}$, right (red) bars show results for $\mathbf{P}_{\text{rr}}$. Bars span the results for minimizing and maximizing queries, dots indicate the quantile query results.

analyze all kinds of combinations of programs and ontologies, leading to different but explainable results. Varying program and ontology component thus has direct impact on the global system characteristics (see Table 4). Second, programs executed in more restrictive contexts modeled by ontologies show more restricted results with respect to the possibilities to resolve nondeterminism. Inconsistent states have great impact with this respect, as the greater the number of states inconsistent in the ontology, the less flexibility is given to a scheduler to resolve nondeterminism. In Figure 3, the effect of more restrictive ontologies is visualized by showing smaller bars. Third, our approach to translate ontologized programs to standard PRISM models enables the quantitative analysis of all state-of-the-art analyses supported by PRISM, involving the analysis of cost-utility trade-offs.

Performance statistics. All the experiments were carried out on an Intel Xeon E5-2680@2.70GHz server platform (Turbo Boost and HT enabled) with 128 GB of RAM running Debian GNU/Linux 9.1. The rewriting of all 16 ontologized programs into PRISM programs took 130 seconds in total, including the computation of hook formulas using justifications by our adjusted version of the DL reasoner OPENLLET. We used the symbolic MTBDD engine of PRISM in the version presented in [KBC+18], which supports the computation of energy-utility quantiles [BDD+14]. Furthermore, we applied variable-reordering techniques to reduce the size of the MTBDD representation of the model and speed-up the computations. In Table 5, the model characteristics in terms of system states of the resulting MDP, the symbolic representation of the model

server/					analysis time [s]	
processes	ontology	program	states	nodes	(i)-(iii)	(iv) + (v)
◊♦/8	$\mathcal{O}_1$	$\mathbf{P}_{\text{rand}}$	90 027 882	134 648	4 662.57	18 268.61
		$\mathbf{P}_{\text{rr}}$	66 116 970	2 933 937	56 935.82	384 886.94
	$\mathcal{O}_2$	$\mathbf{P}_{\text{rand}}$	934 122	6 518	79.03	191.39
		$\mathbf{P}_{\text{rr}}$	158 368	7 507	49.81	271.52
	$\mathcal{O}_3$	$\mathbf{P}_{\text{rand}}$	934 122	6 518	69.14	159.78
		$\mathbf{P}_{\text{rr}}$	158 432	7 372	46.52	325.56
	$\mathcal{O}_4$	$\mathbf{P}_{\text{rand}}$	934 122	6 830	35.68	51.79
		$\mathbf{P}_{\text{rr}}$	157 472	7 455	45.28	124.75
◊◊♦/6	$\mathcal{O}_1$	$\mathbf{P}_{\text{rand}}$	23 072 910	173 841	2 011.52	2 308.40
		$\mathbf{P}_{\text{rr}}$	37 231 023	3 718 247	18 759.57	47 598.00
	$\mathcal{O}_2$	$\mathbf{P}_{\text{rand}}$	800 814	16 782	164.18	236.93
		$\mathbf{P}_{\text{rr}}$	967 250	106 897	314.88	404.51
	$\mathcal{O}_3$	$\mathbf{P}_{\text{rand}}$	800 814	15 666	141.45	208.63
		$\mathbf{P}_{\text{rr}}$	975 526	96 030	325.56	412.91
	$\mathcal{O}_4$	$\mathbf{P}_{\text{rand}}$	773 598	15 769	117.88	161.77
		$\mathbf{P}_{\text{rr}}$	425 306	44 724	126.69	156.91

Table 5. Statistics on the resulting consistency-independent MDP and query-evaluation times.

using MTBDD nodes, and the analysis times for evaluating the queries (i)-(iii) and cost-utility quantile queries (iv) and (v) are shown.

Note that the different ontologies, as well as program variants have a great impact on the model and analysis characteristics. We see a big difference in both model size and analysis times between the experiments that involve ontology $\mathcal{O}_1$ and experiments with other ontologies. This can be explained by the fact that ontology $\mathcal{O}_1$ imposes no constraints on process and server architectures such that only servers exceeding the number of maximal processes are in an inconsistent state (see Table 3). To this end, those states where processes run on a server with a different architecture the process is compiled for are also reachable in a system execution. This is not the case in the other system setups with ontologies different than $\mathcal{O}_1$, where such states are pruned and hence, the state space is restricted there. The sizes of the models have direct impact on their analysis times. As for the analysis results, we visualized the run-time characteristics of evaluating all queries in Figure 3 on the lower right. This also explains the huge analysis times for the setups with ontology $\mathcal{O}_1$. Our experiments hence show that model-checking speed and results benefit from sufficiently precise and restrictively modeled knowledge given by ontologies.

7. Related Work

Model checking context-dependent systems. The idea of using different formalisms for behaviors and contexts to facilitate model checking goes back to [DBRL12], where a scenario-based *context description language (CDL)* based on message sequence charts is used to describe environmental behaviors. Their aim is to mitigate the state-space explosion problem by resolving nondeterminism in the system to model the environment by parallel composition with CDL descriptions of the context. However, the approach by [DBRL12] also uses an operational description of the context. Our approach focuses on separating the description of operational behavior and contexts by using dedicated specification formalisms. *Role-based conceptual modeling* describe formalisms to specify contexts and their dynamics w.r.t. role-playing entities (see, e.g., [KBGA15]). Here, components may play different roles in specific contexts (modeled through elements called *compartments*). Modeling and model checking role-based systems with exogenous coordination has been detailed in [CDB$^+$16, BCD$^+$18, CBDK20]. While their specification of the relationships between components in role-based systems follow a graphical specification language [KBGA15, CDB$^+$16], the semantics is provided following an automata-theoretic approach. Feature-oriented systems describe systems comprising features that can be active or inactive (see, e.g., [DBK15]). We used similar principles within our framework to combine ontological elements as show-cased in our evaluation in Section 6: in Table 3, the columns indicate different feature formalizations of the context while the rows indicate for which ontology description which of these features are considered. A reconfiguration framework for context-aware feature-oriented systems has been considered in [MNSY16]. All the above formalisms use an operational description of contexts, while we intentionally focused on a declarative representation through ontologies that allows for reasoning about complex information and enables the reuse of established knowledge bases.

Description logics in Golog programs. There is a relation between our work and work on integrating DLs and ConGolog programs [BZ13, ZC15]. The focus there is on verifying properties formulated in computation tree logic for ConGolog programs, where also DL axioms specify tests within the program and within the properties to be checked. Differently, we provide a generic approach that allows to employ various PMC tasks using existing tools, and allow for probabilistic guarded command programs. Furthermore, while in ConGolog programs the operational and DL part is integrated in one formalism, our approach separates both parts so that they can easily be varied within the constraints of the interface. However, the main difference is that in the semantics of [BZ13, ZC15], states are identified with interpretations rather than ontologies, which are directly modified by the program. This makes reasoning much more challenging, and easily leads to undecidability if syntactic restrictions are not carefully imposed. Closer to our semantics are the DL-based programs presented in [HCM$^+$13, CDGLR11], where actions consist of additions and removals of assertions in the ontology. Again, there is no separation of concerns in terms of program and ontology, and they only support a Golog-like program language that cannot describe probabilistic behavior.

Ontology-oriented programming. Within the concept of ontology-oriented programming [HCN08], ontologies are described and referred to in the program itself, without separating concerns by using different suitable formalisms for operational behaviors and ontology for knowledge representation. However, our approach currently supports only static ontologies, while the expression of ontologies in the same formalism as the behaviors allows for dynamic changes of the ontology.

Ontology-mediated query answering. There is a resemblance between our concept of ontology-mediated PMC and *ontology-mediated query answering* (OMQA) [PLC$^+$08, BO15]. OMQA is concerned with the problem of querying a possibly incomplete database, where an ontology is used to provide for additional background knowledge about the domain of the data, so that also information that is only implicit in the database can be queried. Sometimes, additionally a mapping from concept and role names in the ontology to tables in the database is provided, which plays a comparable role to our interface [PLC$^+$08]. Similar to our approach, a common technique for OMQA is to rewrite ontology-mediated queries into queries that can be directly evaluated on the data using standard database systems. However, differently to our approach, this is in general only possible for very restricted DLs. In fact, for expressive DLs, the complexity of OMQA is often an exponential higher than standard DL reasoning tasks [Lut07, NOS16], or not even fully characterized yet [RG10].

8. Discussion and Conclusion

To facilitate the quantitative analysis of knowledge-intensive systems, we introduced ontologized programs where probabilistic guarded command programs specify operational behaviors and DLs are used to describe additional knowledge. We have devised a reduction algorithm for ontology-mediated PMC using ontologized programs to the case of standard PMC, implemented this reduction, and illustrated an application of our approach for a quantitative analysis of a multi-server platform.

From an abstract point of view, the general idea of ontologized programs is to use different domain-specific formalisms for specifying the program and the background knowledge, which are then linked by an interface. We believe that the general idea of specifying operational behavior and static system properties separately, each using a dedicated formalism, would indeed be useful for many other applications. To this end, behaviors could be specified, e.g., by program code of any programming language, UML state charts, control-flow diagrams, etc. that are amended with hooks referring to additional knowledge, e.g., described by databases where hooks are resolved through database queries. Depending on the chosen formalisms, our method for rewriting ontologized programs could still be applicable in such settings.

Note that while we discussed our approach for operational system models in terms of MDPs, it is almost straightforward to apply our methods and implementation to any other formalism that can be described in the input language of Prism. To this end, also (probabilistic) model checking of transition systems, discrete Markov chains, or continuous-time Markov chains could be enhanced with ontology knowledge for quantitative analysis.

Lessons learnt. The presented approach exploits the formalisms and techniques that are well-suited to the respective concerns. This approach made it very easy to benefit from the most recent developments in the respective fields without a dedicated implementation built from scratch, but using standard state-of-the-art tooling. Furthermore, varying behavioral descriptions and ontologies was simple on the conceptual and also on the evaluation side. On initial tryouts not relying on the techniques presented, a direct encoding of knowledge into probabilistic guarded command language turned out to be far more complicated and error-prone. One aspect that was not obvious from the start was the impact of the specification of knowledge on the model sizes and analysis times. Our experiments showed that the more specific knowledge is provided, the more the state-space could be reduced and thus, the analysis could be vastly speed up (cf. Table 5).

Further directions. Besides investigating the applicability of our approach on different formalisms for the program component and ontology component of ontologized programs and their use with different analysis techniques, there is plenty room for further improvements already in the setting presented in this paper. First, our definitions of probability-normalizing and probability-preserving semantics are generative, i.e., they can be algorithmically pursued by a model transformation on the consistency-independent semantics. It would be interesting to implement these transformations and investigate the impact of the different semantics on the resulting model size and analysis speed. Second, one could consider a closer integration between the ontology and the abstract program by means of a richer interface. For example, it might be desirable to exchange numerical values between the components. Such values could be mapped directly into the DL by the use of *concrete domains* [BH91], which would allow to express more numerical constraints in the ontology. Another way to extend the expressivity is to employ non-classical variants of DLs, such as probabilistic DLs. Such DLs can express probabilistic constraints in the ontology. There are several approaches to enrich DL ontologies by probabilities (see, e.g., [LS10, BKT17]) on which reasoning procedures have been devised and even axiom pinpointing has been investigated [PT13]. Third, we plan to investigate dynamic switching of ontologies during program execution, to model complex interaction between ontologies as in [DBK15], exploiting the close connection to feature-oriented systems discussed in Section 7. By doing so, we could achieve a switch from context-dependent PMC on knowledge-intensive systems to context-aware PMC. Fourth, one could further optimize the generation of hook evaluations. Currently, such Boolean expressions are represented in disjunctive normal form (see Equation (10)). These could be optimized via well-known reduction methods from circuit design such as Espresso [BSVMH84].

References

[BBL05] Franz Baader, Sebastian Brandt, and Carsten Lutz. Pushing the $\mathcal{EL}$ envelope. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence IJCAI-05*, Edinburgh, UK, 2005. Morgan-Kaufmann Publishers.

[BCD+18] Christel Baier, Philipp Chrszon, Clemens Dubslaff, Joachim Klein, and Sascha Klüppelholz. Energy-utility analysis of probabilistic systems with exogenous coordination. In *It's All About Coordination*, LNCS, pages 38–56, Cham, 2018. Springer.

[BCM+03] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press, 2003.

[BDD+14] C. Baier, M. Daum, C. Dubslaff, J. Klein, and S. Klüppelholz. Energy-utility quantiles. In *Proc. NASA Formal Methods (NFM'14)*, volume 8430 of *LNCS*, pages 285–299. Springer, 2014.

[BDK+14] Christel Baier, Clemens Dubslaff, Sascha Klüppelholz, Marcus Daum, Joachim Klein, Steffen Märcker, and Sascha Wunderlich. Probabilistic model checking and non-standard multi-objective reasoning. In *Fundamental Approaches to Software Engineering*, pages 1–16, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[BH91] Franz Baader and Philipp Hanschke. A scheme for integrating concrete domains into concept languages. In *Proceedings of IJCAI 1991*, pages 452–457. Morgan Kaufmann, 1991.

[BHLS17] Franz Baader, Ian Horrocks, Carsten Lutz, and Ulrike Sattler. *An Introduction to Description Logic.* Cambridge University Press, 2017.

[BK08] C. Baier and J.-P. Katoen. *Principles of Model Checking.* MIT Press, 2008.

[BKT17] Franz Baader, Patrick Koopmann, and Anni-Yasmin Turhan. Using ontologies to query probabilistic numerical data. In *Frontiers of Combining Systems: 11th International Symposium*, volume 10483 of *LNCS*, pages 77–94. Springer International Publishing, 2017.

[BO15] Meghyn Bienvenu and Magdalena Ortiz. Ontology-mediated query answering with data-tractable description logics. In *Reasoning Web. Web Logic Rules*, pages 218–307, 2015.

[BPS07] Franz Baader, Rafael Peñaloza, and Boontawee Suntisrivaraporn. Pinpointing in the description logic $\mathcal{EL}^+$. In *Proc. of the 30th German Annual Conf. on Artificial Intelligence (KI'07)*, volume 4667 of *Lecture Notes in Computer Science*, pages 52–67, Osnabrück, Germany, 2007. Springer.

[Bra04] Sebastian Brandt. Polynomial time reasoning in a description logic with existential restrictions, GCI axioms, and—what else? In R. López de Mantáras and L. Saitta, editors, *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI-2004)*, pages 298–302. IOS Press, 2004.

[BSVMH84] Robert King Brayton, Alberto L. Sangiovanni-Vincentelli, Curtis T. McMullen, and Gary D. Hachtel. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, USA, 1984.

[BZ13] Franz Baader and Benjamin Zarrieß. Verification of Golog programs over description logic actions. In *Proceedings of FroCos 2013*, volume 8152 of *LNCS*, pages 181–196. Springer, 2013.

[CBDK20] Philipp Chrszon, Christel Baier, Clemens Dubslaff, and Sascha Klüppelholz. From features to roles. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A*, SPLC '20, New York, NY, USA, 2020. Association for Computing Machinery.

[CDB+16] Philipp Chrszon, Clemens Dubslaff, Christel Baier, Joachim Klein, and Sascha Klüppelholz. *Modeling Role-Based Systems with Exogenous Coordination*, pages 122–139. Springer, Cham, 2016.

[CDGLR11] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. 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.

[CDKB18] Philipp Chrszon, Clemens Dubslaff, Sascha Klüppelholz, and Christel Baier. Profeat: feature-oriented engineering for family-based probabilistic model checking. *Formal Aspects of Computing*, 30(1):45–75, 2018.

[CDL+07] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family. *J. Autom. Reasoning*, 39(3):385–429, 2007.

[DBK15] C. Dubslaff, C. Baier, and S. Klüppelholz. Probabilistic model checking for feature-oriented systems. *Trans. on Aspect-Oriented Software Dev.*, 12:180–220, 2015.

[DBRL12] Philippe Dhaussy, Frédéric Boniol, Jean-Charles Roger, and Luka Leroux. Improving model checking with context modelling. *Advances in Software Engineering*, 2012:13, 2012.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[DJKV17] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. A Storm is coming: A modern probabilistic model checker. In *29th Int. Conf. on Computer Aided Verification (CAV)*, volume 10427 of *LNCS*, pages 592–600. Springer, 2017.

[DKT19] Clemens Dubslaff, Patrick Koopmann, and Anni-Yasmin Turhan. Ontology-mediated probabilistic model checking. In Wolfgang Ahrendt and Silvia Lizeth Tapia Tarifa, editors, *Integrated Formal Methods - 15th International Conference, IFM 2019*, volume 11918 of *Lecture Notes in Computer Science*, pages 194–211. Springer, 2019.

[DKT20] Clemens Dubslaff, Patrick Koopmann, and Anni-Yasmin Turhan. Give inconsistency a chance: Semantics for ontology-mediated verification. In *Proceedings of the 33rd International Workshop on Description Logics*, 2020.

[FKNP11] V. Forejt, M. Z. Kwiatkowska, G. Norman, and D. Parker. 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)*, volume 6659 of *LNCS*, pages 53–113. Springer, 2011.

[GHM+14] Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos, and Zhe Wang. HermiT: An OWL 2 reasoner. *J. Autom. Reason.*, 53(3):245–269, 2014.

[HB11] Matthew Horridge and Sean Bechhofer. The OWL API: A Java API for OWL ontologies. *Semantic Web*, 2(1):11–21, 2011.

[HCM+13] Babak Bagheri Hariri, Diego Calvanese, Marco Montali, Giuseppe De Giacomo, Riccardo De Masellis, and Paolo Felli. Description logic knowledge and action bases. *J. Artif. Intell. Res.*, 46:651–686, 2013.

[HCN08] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology, March-April 2008, ETH Zurich*, 7(3):125–151, 2008.

[HKS06] Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The even more irresistible $\mathcal{SROIQ}$. In *Proc. of KR 2006*, pages 57–67. AAAI Press, 2006.

[Hor11] Matthew Horridge. *Justification Based Explanation in Ontologies.* PhD thesis, University of Manchester, UK, 2011.

[JSM97] He Jifeng, K. Seidel, and A. McIver. Probabilistic models for the guarded command language. *Science of Computer Programming*, 28(2):171 – 192, 1997.

[Kaz08] Yevgeny Kazakov. $\mathcal{RIQ}$ and $\mathcal{SROIQ}$ are harder than $\mathcal{SHOIQ}$. In *Proc. of the 11th Intern. Conf. on Principles of Knowledge Representation and Reasoning (KR'08)*, pages 274–284. AAAI Press, 2008.

[KBC+18] Joachim Klein, Christel Baier, Philipp Chrszon, Marcus Daum, Clemens Dubslaff, Sascha Klüppelholz, Steffen Märcker, and David Müller. Advances in probabilistic model checking with PRISM: variable reordering, quantiles and weak deterministic büchi automata. *Intern. J. on Software Tools for Technology Transfer*, 20(2):179–194, Apr 2018.

[KBGA15] Thomas Kühn, Stephan Böhme, Sebastian Götz, and Uwe Aßmann. A combined formal model for relational context-dependent roles. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, Pittsburgh, PA, USA, October 25-27, 2015*, pages 113–124, 2015.

[KKS14] Yevgeny Kazakov, Markus Krötzsch, and Frantisek Simancik. The incredible ELK - from polynomial procedures to efficient reasoning with $\mathcal{EL}$ ontologies. *J. Autom. Reason.*, 53(1):1–61, 2014.

[KNP11] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. of the 23rd Intern. Conf. on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591, 2011.

[KPHS07] Aditya Kalyanpur, Bijan Parsia, Matthew Horridge, and Evren Sirin. Finding all justifications of OWL DL entailments. In Karl Aberer, Key-Sun Choi, Natasha Fridman Noy, Dean Allemang, Kyung-Il Lee, Lyndon J. B. Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Riichiro Mizoguchi, Guus Schreiber, and Philippe Cudré-Mauroux, editors, *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007*, volume 4825 of *Lecture Notes in Computer Science*, pages 267–280. Springer, 2007.

[LS10] Carsten Lutz and Lutz Schröder. Probabilistic description logics for subjective uncertainty. In *Proceedings of the 12th International Conference on Principles of Knowledge Representation and Reasoning (KR2010)*. AAAI Press, 2010.

[Lut07] Carsten Lutz. Inverse roles make conjunctive queries hard. In *Proc. of the 20th Intern. Workshop on Description Logics (DL'07)*, volume 250 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[MCH$^+$09] B. Motik, B. Cuenca Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz. OWL 2 web ontology language profiles. W3C Recommendation, 27 October 2009. `http://www.w3.org/TR/2009/REC-owl2-profiles-20091027/`.

[MNSY16] Jacopo Mauro, Michael Nieke, Christoph Seidl, and Ingrid Chieh Yu. Context aware reconfiguration in software product lines. In *Proc. of the 10th Intern. Workshop on Variability Modelling of Software-intensive Systems (VaMoS'16)*, pages 41–48. ACM, 2016.

[MP04] A. S. Miner and D. Parker. Symbolic representations and analysis of large probabilistic systems. In *Validation of Stochastic Systems - A Guide to Current Research*, volume 2925 of *LNCS*, pages 296–338, 2004.

[NOS16] Nhung Ngo, Magdalena Ortiz, and Mantas Simkus. Closed predicates in description logics: Results on combined complexity. In *Proceedings of KR 2016*, pages 237–246. AAAI Press, 2016.

[PLC$^+$08] Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking data to ontologies. *Journal on Data Semantics*, 2008.

[PMG$^+$17] Bijan Parsia, Nicolas Matentzoglu, Rafael S. Gonçalves, Birte Glimm, and Andreas Steigmiller. The OWL reasoner evaluation (ORE) 2015 competition report. *J. Autom. Reasoning*, 59(4):455–482, 2017.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57, 1977.

[PT13] Rafael Peñaloza and Anni-Yasmin Turhan. Instance-based non-standard inferences in $\mathcal{EL}$ with subjective probabilities. In *Uncertainty Reasoning for the Semantic Web II, International Workshops, Revised Selected Papers*, number 7123 in LNCS, pages 80–98. Springer-Verlag, 2013.

[Put94] M. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* John Wiley & Sons, Inc., New York, NY, 1994.

[RG10] Sebastian Rudolph and Birte Glimm. Nominals, inverses, counting, and conjunctive queries or: why infinity is your friend! *J. Artif. Intell. Res.*, 39:429–481, 2010.

[RKGH16] Ana Armas Romero, Mark Kaminski, Bernardo Cuenca Grau, and Ian Horrocks. Module extraction in expressive ontology languages via datalog reasoning. *J. Artif. Intell. Res.*, 55:499–564, 2016.

[SC03] Stefan Schlobach and Ronald Cornet. Non-standard reasoning services for the debugging of description logic terminologies. In Georg Gottlob and Toby Walsh, editors, *Proc. of the 18th Int. Joint Conf. on Artificial Intelligence (IJCAI 2003)*, pages 355–362, Acapulco, Mexico, 2003. Morgan Kaufmann.

[Sch91] Klaus Schild. A correspondence theory for terminological logics: Preliminary report. In John Mylopoulos and Raymond Reiter, editors, *Proc. IJCAI'91*, pages 466–471. Morgan Kaufmann, 1991.

[SLG14] Andreas Steigmiller, Thorsten Liebig, and Birte Glimm. Konclude: System description. *J. Web Semant.*, 27-28:78–85, 2014.

[SPG$^+$07] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: a practical OWL-DL reasoner. *J. Web Sem.*, 5(2):51–53, 2007.

[Tob01] Stephan Tobies. *Complexity results and practical algorithms for logics in knowledge representation.* PhD thesis, RWTH Aachen University, Germany, 2001.

[Var85] Moshe Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *26th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 327–338. IEEE Computer Society, 1985.

[ZC15] Benjamin Zarrieß and Jens Claßen. Verification of knowledge-based programs over description logic actions. In *IJCAI*, pages 3278–3284. AAAI Press, 2015.