TECHNISCHE UNIVERSITÄT DRESDEN

Technische Universität Dresden
Institute for Theoretical Computer Science
Chair for Automata Theory

LTCS-Report

Ontology-Mediated Probabilistic Model Checking (Extended Version)

Clemens Dubslaff, Patrick Koopmann, and Anni-Yasmin Turhan

LTCS-Report 18-09

Postal Address:
Lehrstuhl für Automatentheorie
Institut für Theoretische Informatik
TU Dresden
01062 Dresden

http://lat.inf.tu-dresden.de

Visiting Address:
Nöthnitzer Str. 46
Dresden

Contents

Ontology-Mediated Probabilistic Model Checking (Extended Version) *

Clemens Dubslaff, Patrick Koopmann, and Anni-Yasmin Turhan

July 10, 2019

Abstract

Probabilistic model checking (PMC) is a well-established method for the quantitative analysis of dynamic systems. On the other hand, description logics (DLs) provide a well-suited formalism to describe and reason about static knowledge, used in many areas to specify domain knowledge in an ontology. We investigate how such knowledge can be integrated into the PMC process, introducing *ontology-mediated PMC*. Specifically, we propose a formalism that links ontologies to dynamic behaviors specified by guarded commands, the de-facto standard input formalism for PMC tools such as Prism. Further, we present and implement a technique for their analysis relying on existing DL-reasoning and PMC tools. This way, we enable the application of standard PMC techniques to analyze knowledge-intensive systems. Our approach is implemented and evaluated 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.

1 Introduction

Probabilistic model checking (PMC, see, e.g., [6, 15] for surveys) is an automated technique for the quantitative analysis of dynamic systems. PMC has been successfully applied in many areas, e.g., to ensure the system to meet quality requirements such as low error probabilities or an energy consumption within a given bound. The de-facto standard specification for the dynamic (probabilistic) system under consideration is given by *stochastic programs*, a probabilistic variant of Dijkstra's guarded command language [13, 19] used within many PMC tools such as Prism [22]. Usually, the behavior described by a stochastic program is part of a bigger system, or might be even used within the context of a collection of systems that have an impact on the operational behavior as well. There are different ways in which this can be taken into consideration by using stochastic programs: one could 1) integrate additional knowledge about the surrounding system directly into the stochastic program, or 2) use the concept of nondeterminism that models all possible behaviors of the surrounding system. The second approach might lead to analysis results that are too coarse with respect to desired properties and increase the well known state-space explosion problem. Also the first approach has its drawbacks: although guarded command languages are well-suited for describing dynamic behavior,

*The authors are supported by the DFG through the Collaborative Research Centers CRC 912 (HAEC) and 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).

they are not specialized for describing static knowledge, which makes it cumbersome to describe knowledge-intensive contexts. We therefore propose a third approach, where we separate the specification of the dynamic behavior of a system from the specification of the additional knowledge that influences the behaviors. This allows to use different, specialized formalisms for describing the complex properties of the system analyzed. Further, such an approach adds flexibility, as we can exchange both behavioral and knowledge descriptions, e.g., to analyze the same behavior in different contexts, or to analyze different behaviors in the same context.

A well-established family of formalisms for describing domain knowledge are description logics (DLs), fragments of first-order logic balancing expressivity and decidability [1, 3]. While the worst-case complexity for reasoning in DLs can be very high, modern optimized DL reasoning systems often allow reasoning even for very large knowledge bases in short times [28]. Logical theories formulated in a DL are called ontologies, and may contain both assertional axioms about specific individuals, and universal statements defining and relating sets of individuals.

In this paper, we propose *ontology-mediated probabilistic model checking* as an approach to include knowledge described in a DL ontology into the PMC process. The center of this approach are *ontologized (stochastic) programs* which can be subject of probabilistic model checking. Following the concept of separation of concerns described above, ontologized programs use different formalisms for specifying the operational behavior and the ontology, loosely coupled through an interface. Specifically, ontologized programs are stochastic programs that use *hooks* to refer to the ontology within the behavior description, which are linked to DL expressions via the interface. The semantics of ontologized programs follows a product construction of the operational semantics for the stochastic program, combined with annotations in which states are additionally associated with DL knowledge bases. To analyze ontologized programs, we present a technique to rewrite ontologized programs into (plain) stochastic programs without explicit references to the ontology, preserving those properties of the program that are relevant for the analysis. A similar transformation is done to those analysis properties that depend on the ontology, i.e., include hooks. This translational approach enables the full potential of standard PMC tools such as PRISM [22] including advanced analysis properties [5, 21] also for the analysis of ontologized programs with properties that refer to ontology knowledge.

We implemented the technique in a tool-chain in which the operational behavior is specified in the input language of PRISM, and where the ontology is given as OWL knowledge base [25]. Technically, we use *axiom pinpointing* during the translation to minimize the use of external DL reasoning, and to enable a practical implementation. Since our approach is independent of any particular DL, the implementation supports any OWL-fragment that is supported by modern DL reasoners. We evaluated the 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.

2 Preliminaries

We recall well-known concepts 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 S be a countable set. We denote by $\wp(S)$ the powerset of S. A probability *distribution* over S is a function $\mu\colon S \to [0,1] \cap \mathbb{Q}$ with $\sum_{s\in S} \mu(s) = 1$. The set of distributions over S is denoted by $\mathit{Distr}(S)$.

2.1 Markov Decision Processes

The operational model used in this paper is given in terms of *Markov decision processes (MDPs)* (see, e.g., [30]). MDPs are tuples $\mathcal{M} = \langle Q, Act, P, q_0, \Lambda, \lambda \rangle$ where Q and Act are countable sets of *states* and *actions*, respectively, $P \colon Q \times Act \rightharpoonup Distr(Q)$ is a partial probabilistic transition function, $q_0 \in Q$ an initial state, and Λ a set of labels assigned to states via the labeling function $\lambda \colon Q \to \wp(\Lambda)$. Intuitively, in a state $q \in Q$, we nondeterministically select an action $\alpha \in Act$ for which $P(q, \alpha)$ is defined, and then move to a successor state q' with probability $P(q, \alpha, q')$. Formally, a *path in* $\mathcal{M}$ is a sequence $\pi = q_0 \, \alpha_0 \, q_1 \, \alpha_1 \ldots$ where $P(q_i, \alpha_i, q_{i+1}) > 0$ for all i for which q_{i+1} is defined. The probability of a finite path is the product of its transition probabilities. Resolving nondeterministic choices gives then rise to a probability measure over *maximal paths*, i.e., paths that cannot be extended. Amending $\mathcal{M}$ with a *weight function* $wgt \colon Q \to \mathbb{N}$ turns $\mathcal{M}$ into a *weighted MDP* $\langle \mathcal{M}, wgt \rangle$. The weight of a finite path $\pi = q_0 \, \alpha_0 \, q_1 \ldots q_n$ is defined as $wgt(\pi) = \sum_{i<n} wgt(q_i)$.

MDPs are suitable for a quantitative analysis using probabilistic model checking (PMC, cf. [6]). 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. An *energy-utility quantile* [5] is an advanced property that is used to reason about trade-offs: given a probability bound $p \in [0, 1]$ and a set of goal states, we ask for the minimal (resp. maximal) weight required to reach the goal with probability at least p when ranging over some (resp. all) resolutions of nondeterminism.

2.2 Stochastic Programs

A concise representation of MDPs is provided by a probabilistic variant of Dijkstra's *guarded-command language* [13, 19], compatible with the input language of the PMC tool Prism [22]. Throughout this section, we fix a countable set Var of *variables*, on which we define *evaluations* as functions $\eta \colon Var \to \mathbb{Z}$. We denote the set of evaluations over Var by $Eval(Var)$.

2.2.1 Arithmetic Constraints and Boolean Expressions.

Let z range over $\mathbb{Z}$ and v range over Var. The set of *arithmetic expressions* $\mathbb{E}(Var)$ is defined by the grammar

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

Variable evaluations are extended to arithmetic expressions in the natural way, i.e., $\eta(z) = z$, $\eta(\alpha_1 + \alpha_2) = \eta(\alpha_1) + \eta(\alpha_2)$, and $\eta(\alpha_1 \cdot \alpha_2) = \eta(\alpha_1) \cdot \eta(\alpha_2)$. $\mathbb{C}(Var)$ denotes the set of *arithmetic constraints* over Var, i.e., terms of the form $(\alpha \bowtie z)$ with $\alpha \in \mathbb{E}(Var)$, $\bowtie \in \{>, \geq, =, \leq, <, \neq\}$, and $z \in \mathbb{Z}$. For a given evaluation $\eta \in Eval(Var)$ and constraint $(\alpha \bowtie z) \in \mathbb{C}(Var)$, we write $\eta \models (\alpha \bowtie z)$ iff $\eta(\alpha) \bowtie z$ and say that $(\alpha \bowtie \theta)$ is *entailed by* η. Furthermore, we denote by $\mathbb{C}(\eta)$ the constraints entailed by η, i.e., $\mathbb{C}(\eta) = \{c \in \mathbb{C}(Var) \mid \eta \models c\}$.

For a countable set X and x ranging over X, we define *Boolean expressions* $\mathbb{B}(X)$ over X by the grammar $\phi \ ::= \ x \mid \neg\phi \mid \phi \wedge \phi$. Furthermore, we define the *satisfaction relation* $\models \subseteq \wp(X) \times \mathbb{B}(X)$ in the usual way by $Y \models x$ if $x \in Y$, $Y \models \neg\psi$ iff $Y \not\models \psi$, and $Y \models \psi_1 \wedge \psi_2$ iff $Y \models \psi_1$ and $Y \models \psi_2$, where $Y \subseteq X$. For an evaluation $\eta \in Eval(Var)$ and $\phi \in \mathbb{B}(\mathbb{C}(Var))$, we write $\eta \models \phi$ iff $\mathbb{C}(\eta) \models \phi$. Well-known Boolean connectives such as disjunction $\vee$, implication $\rightarrow$, etc. and their satisfaction relation can be deduced in the standard way using syntactic transformations, e.g., through de Morgan's rule.

2.2.2 Stochastic Programs.

We call a function $u\colon \mathit{Var} \to \mathbb{E}(\mathit{Var})$ *update*, and a distribution $\sigma \in \mathit{Distr}(\mathit{Upd})$ over a given finite set Upd of updates *stochastic update*. The effect of an update $u\colon \mathit{Var} \to \mathbb{E}(\mathit{Var})$ on an evaluation $\eta \in \mathit{Eval}(\mathit{Var})$ is their composition $\eta \circ u \in \mathit{Eval}(\mathit{Var})$, i.e., $(\eta \circ u)(v) = \eta(u(v))$ for all $v \in \mathit{Var}$. This notion naturally extends to *stochastic updates* $\sigma \in \mathit{Distr}(\mathit{Upd})$ by $\eta \circ \sigma \in \mathit{Distr}(\mathit{Eval}(\mathit{Var}))$, where for any $\eta' \in \mathit{Eval}(\mathit{Var})$ we have

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

A *stochastic guarded command* over a finite set of updates Upd, briefly called *command*, is a pair $\langle g, \sigma \rangle$ where $g \in \mathbb{B}(\mathbb{C}(\mathit{Var}))$ is a *guard* and $\sigma \in \mathit{Distr}(\mathit{Upd})$ is a stochastic update. Similarly, a *weight assignment* is a pair $\langle g, w \rangle$ where $g \in \mathbb{B}(\mathbb{C}(\mathit{Var}))$ is a guard and $w \in \mathbb{N}$ a *weight*. A *stochastic program* over Var is a tuple $\mathbf{P} = \langle \mathit{Var}, C, W, \eta_0 \rangle$ where C is a finite set of commands, W a finite set of weight assignments, and $\eta_0 \in \mathit{Eval}(\mathit{Var})$ is an initial variable evaluation. For simplicity, we write $\mathit{Upd}(\mathbf{P})$ for the set of all updates in C.

The semantics of $\mathbf{P}$ is now defined as the weighted MDP

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

where

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

Note that $\mathcal{M}[\mathbf{P}]$ is indeed a weighted MDP and that $P(\eta, \sigma)$ is a probability distribution with finite support for all $\eta \in \mathit{Eval}(\mathit{Var})$ and $\sigma \in \mathit{Distr}(\mathit{Upd}(\mathbf{P}))$.

2.3 Description Logics

We recall basic notions of description logics (DLs) (see, e.g., [1, 3] for more details). Our approach presented in this paper is general enough to be used with any expressive DL, and our implementation supports the expressive DL $\mathcal{SROIQ}$ underlying the web ontology standard OWL-DL [18]. For illustrative purposes, we present here a small yet expressive fragment of this DL called $\mathcal{ALCQ}$. Let $\mathsf{N_c}$, $\mathsf{N_r}$ and $\mathsf{N_i}$ be pairwise disjoint countable sets of *concept names*, *role names*, and *individual names*, respectively. For $A \in \mathsf{N_c}$, $r \in \mathsf{N_r}$, and $n \in \mathbb{N}$, $\mathcal{ALCQ}$ *concepts* are then defined through the grammar

$$C \ ::= \ A \mid \neg C \mid C \sqcap C \mid \exists r.C \mid \geq n r.C \ .$$

Further concept constructors are defined as abbreviations: $C \sqcup D = \neg(\neg C \sqcap \neg D)$, $\forall r.C = \neg\exists r.\neg C$, $\leq n r.C = \neg \geq (n+1) r.C$, $\bot = A \sqcap \neg A$ (for any A), and $\top = \neg\bot$. *Concept inclusions* (CIs) are statements of the form $C \sqsubseteq D$, where C and D are concepts. A common abbreviation is $C \equiv D$ for $C \sqsubseteq D$ and $D \sqsubseteq C$. *Assertions* are of the form $A(a)$ or $r(a, b)$, where A is a

concept, $r \in \mathsf{N_r}$, and $a, b \in \mathsf{N_i}$. CIs and assertions are commonly referred to as *DL axioms*, and we use $\mathbb{A}$ to denote the set of all DL axioms. A *knowledge base* $\mathcal{K}$ is a finite set of DL axioms.

Assertions are used to describe the facts that hold for particular objects from the application domain. CIs model background knowledge on notions and categories from the application domain.

Example 1. We can define the state of a multi-server platform, in which different servers run processes with different priorities, using the following assertions:

$$\mathsf{hasServer}(\mathsf{platform}, \mathsf{server1}) \qquad \mathsf{hasServer}(\mathsf{platform}, \mathsf{server2}) \tag{1}$$

$$\mathsf{runsProcess}(\mathsf{server2}, \mathsf{process1}) \qquad \mathsf{runsProcess}(\mathsf{server2}, \mathsf{process2}) \tag{2}$$

$$\mathsf{hasPriority}(\mathsf{process1}, \mathsf{highP}) \quad \mathsf{hasPriority}(\mathsf{process2}, \mathsf{highP}) \quad \mathsf{High}(\mathsf{highP}), \tag{3}$$

and specify further domain knowledge using the following CIs:

$$\geq 4 \mathsf{runsProcess}.\top \sqsubseteq \mathsf{Overloaded} \tag{4}$$

$$\geq 2 \mathsf{runsProcess}.\exists \mathsf{hasPriority}.\mathsf{High} \sqsubseteq \mathsf{Overloaded} \tag{5}$$

$$\mathsf{PlatformWithOverload} \equiv \exists \mathsf{hasServer}.\mathsf{Overloaded}\ . \tag{6}$$

These CIs express that a server that runs more than 4 processes is overloaded (4), that it is already overloaded when it runs 2 processes with a high priority (5), and that PlatformWithOverload is a platform that has an overloaded server (6).

The semantics of DLs is defined in terms of *interpretations*, which are tuples $\langle \Delta^{\mathcal{I}}, \cdot^{\mathcal{I}} \rangle$ of a set $\Delta^{\mathcal{I}}$ of *domain elements*, and an *interpretation function* $\cdot^{\mathcal{I}}$ that maps every $A \in \mathsf{N_c}$ to some $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, every $r \in \mathsf{N_r}$ to some $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, and every $a \in \mathsf{N_i}$ to some $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$. Interpretation functions are extended to complex concepts in the following way:

$$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}} \qquad (C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$$

$$(\exists r.C)^{\mathcal{I}} = \{d \in \Delta^{\mathcal{I}} \mid \exists e : \langle d, e \rangle \in r^{\mathcal{I}} \wedge e \in C^{\mathcal{I}}\}$$

$$(\geq n r.C)^{\mathcal{I}} = \{d \in \Delta^{\mathcal{I}} \mid \#\{\langle d, e \rangle \in r^{\mathcal{I}} \mid e \in C^{\mathcal{I}}\} \geq n\}$$

Satisfaction of a DL axiom α in an interpretation $\mathcal{I}$, in symbols $\mathcal{I} \models \alpha$, is defined as $\mathcal{I} \models C \sqsubseteq D$ iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$, $\mathcal{I} \models A(a)$ iff $a^{\mathcal{I}} \in A^{\mathcal{I}}$, and $\mathcal{I} \models r(a, b)$ iff $\langle a^{\mathcal{I}}, b^{\mathcal{I}} \rangle \in r^{\mathcal{I}}$. An interpretation $\mathcal{I}$ is a *model* of a DL knowledge base $\mathcal{K}$ iff $\mathcal{I} \models \alpha$ for all $\alpha \in \mathcal{K}$. $\mathcal{K}$ is *inconsistent* if it does not have a model, and it *entails an axiom or assertion* α, in symbols $\mathcal{K} \models \alpha$, iff $\mathcal{I} \models \alpha$ for all models $\mathcal{I}$ of $\mathcal{K}$.

In the above example, we have $\mathcal{K} \models \mathsf{Overloaded}(\mathsf{server2})$ because the server server2 runs two prioritized processes, and $\mathcal{K} \models \mathsf{PlatformWithOverload}(\mathsf{platform})$ as platform has server2 as an overloaded server.

3 Ontologized Programs

We introduce our notion of ontologized programs. In general, an ontologized program comprises the following three components:

The Program is a specification of the operational behavior given as an abstract stochastic program, which may use *hooks* to refer to knowledge that depend on the ontology the program is executed in.

The Ontology is a DL knowledge base representing additional knowledge that may influence the behaviors of the program.

The Interface links program and ontology by providing mappings between the language used in the program and the DL of the knowledge base.

We provide a formal definition of ontologized programs (Section 3.1) and define their semantics in terms of weighted MDPs (Section 3.2). To illustrate these definitions, we extend Example 1 towards a probabilistic model-checking scenario: we consider a generic multi-server platform on which processes can be assigned to servers, scheduled to complete a given number of jobs. The program specifies the dynamics of this scenario, i.e., how jobs are executed, how processes are assigned to servers or moved, and when processes terminate and when they are spawned. The ontology gives details and additional constraints for a specific multi-server platform. In this setting, probabilistic model checking can then be used to analyze different aspects of the system, depending on the operational behavior and the different hardware and software configurations specified by the ontology.

3.1 Ontologizing Stochastic Programs

We introduce ontologized programs formally and illustrate their concepts by our running example. In preparation of the definition, we fix a set H of labels called *hooks*. We define *abstract stochastic programs* as an extension of stochastic programs where the guards used in guarded commands and in weights can be picked from the set $\mathbb{B}(\mathbb{C}(\mathit{Var}) \cup H)$. For instance, with a hook $\texttt{migrate} \in H$, the following guarded command may appear in an abstract stochastic program:

$$(\texttt{migrate} \wedge \texttt{server\_proc1} = 2) \mapsto \begin{cases} 1/2 : \texttt{server\_proc1} := 1 \\ 1/2 : \texttt{server\_proc1} := 3 \end{cases}$$

This command states that, if the hook `migrate` is active and Process 1 runs on Server 2, then we move Process 1 to Server 1 or to Server 3 with a 50% probability each. For a given abstract program $\mathbf{P}$, we refer to its hooks in $\mathbf{P}$ by $H(\mathbf{P})$.

Definition 2. An *ontologized program* is a tuple $\mathbf{O} = \langle \mathbf{P}, \mathcal{K}, \mathbf{I} \rangle$ where

- $\mathbf{P} = \langle \mathit{Var}_{\mathbf{P}}, C, W, \eta_0 \rangle$ is an abstract stochastic program,
- $\mathcal{K}$ is a DL knowledge base describing the *ontology*,
- $\mathbf{I} = \langle \mathit{Var}_{\mathbf{O}}, H_{\mathbf{O}}, \mathcal{F}_{\mathbf{O}}, \mathsf{pD}, \mathsf{Dp} \rangle$ is a tuple describing the *interface*, where $\mathit{Var}_{\mathbf{O}}$ is a set of variables, $H_{\mathbf{O}}$ is a set of hooks, $\mathcal{F}$ is a set of *DL axioms* called *fluent axioms*, and two mappings $\mathsf{pD}\colon H_{\mathbf{O}} \to \wp(\mathbb{A})$ and $\mathsf{Dp}\colon \mathcal{F}_{\mathbf{O}} \to \mathbb{B}(\mathbb{C}(\mathit{Var}_{\mathbf{O}}))$,

and for which we require that $\mathbf{I}$ is *compatible with* $\mathbf{P}$ in the sense that $H(\mathbf{P}) \subseteq H_{\mathbf{O}}$ and $\mathit{Var}_{\mathbf{O}} \subseteq \mathit{Var}_{\mathbf{P}}$. Given an ontologized program $\mathbf{O}$, we refer to its abstract stochastic program by $\mathbf{P}_{\mathbf{O}}$, to its ontology by $\mathcal{K}_{\mathbf{O}}$, and to its interface by $\mathbf{I}_{\mathbf{O}}$.

We illustrate the above definition and its components by our multi-server system example, for which we consider instances running n processes on m servers.

3.1.1 Program.

The stochastic program $\mathbf{P}_{\mathbf{O}}$ specifies the protocol how processes are scheduled to complete their jobs when running on the same server, and when ontology-dependent migration of processes

to other servers should be performed. Job scheduling could be performed, e.g., by selecting processes uniformly via tossing a fair coin or in a round-robin fashion. Here, the hook `migrate` $\in H$ is used to determine when a server should migrate processes to other servers. The program further specifies guarded weights, e.g., amending states marked with `migrate` by the costs to migrate processes.

3.1.2 Ontology.

The knowledge base $\mathcal{K}_{\mathbf{O}}$ models background knowledge about a particular server platform. For instance, it could use the example axioms from Section 2.3 to specify hardware characteristics of the servers using the CIs (4)–(6), architecture specifics using the assertions in (1), and distribute different priorities among a set of predefined processes using the assertions in (3). To establish a link with the hook `migrate` in the interface, we use an additional CI to describe the conditions that necessitate a migration in the platform:

$$\mathsf{NeedsToMigrate} \equiv \mathsf{PlatformWithOverload} \ .$$

In more complex scenarios, migration can depend on a server and can be specified by more complex CIs. This modeling makes it easy to define different migration strategies within the different ontologies. Each of them can be used by simply referring to the `migrate` hook in the program.

Note that the guarded command language uses variables (over integers) to refer to servers and processes, while the knowledge base uses individual names for them. The program and the ontology thus have different views on the system, mapped to each other via the interface.

3.1.3 Interface.

To interpret the states of the program $\mathbf{P}_{\mathbf{O}}$ in DL, the interface specifies a set $\mathcal{F}_{\mathbf{O}}$ of "fluent" DL axioms that describe the dynamics of the system. The function Dp maps each element $\alpha \in \mathcal{F}$ to an expression $\mathsf{Dp}(\alpha) \in \mathbb{B}(\mathbb{C}(\mathit{Var}))$, identifying states in the program language in which α holds. It is thus a mapping from the DL to the abstract program language. In our example, $\mathcal{F}$ would contain assertions of the form $\mathsf{runsProcess}(\mathsf{server}i, \mathsf{process}j)$, which are mapped using $\mathsf{Dp}(\mathsf{runsProcess}(\mathsf{server}i, \mathsf{process}j)) = (\mathtt{server\_proc}j = i)$ to constraints over the states of the abstract program. This allows to represent each program state in the ontology $\mathcal{K}_{\mathbf{O}}$ as a DL knowledge base with axioms from $\mathcal{F}$. Note that the mapping Dp can only refer to variables that are used by the program, as we require $\mathit{Var}_{\mathbf{I}} \subseteq \mathit{Var}(\mathbf{P}_{\mathbf{O}})$ to ensure that for every axiom $\alpha \in \mathcal{F}$. Hence, $\mathsf{Dp}(\alpha)$ has well-defined meaning within the abstract program. However, the program may use additional variables that are only relevant for the operational behavior.

To interpret the hooks in the DL, we additionally need a mapping pD from the program language into the DL. Specifically, pD assigns to each hook $\ell \in H_{\mathbf{O}}$ a set $\mathsf{pD}(\ell)$ of DL axioms. In our running example, the hook `migrate` would, e.g., be mapped as $\mathsf{pD}(\mathtt{migrate}) = \{\mathsf{NeedsToMigrate}(\mathsf{platform})\}$. All hooks in the program are mapped by the interface due to the condition $H(\mathbf{P}_{\mathbf{O}}) \subseteq H$. However, further hooks can be defined that are only relevant for the analysis tasks to be performed. For instance, we might use a hook `critical` to mark critical situations in our system, and analyze the probability of the ontologized program to enter a state in which this hook is activated.

To illustrate the idea of the mappings, consider a virtual communication flow between the program and the ontology. If the ontology wants to know which axioms in $\alpha \in \mathcal{F}$ hold in the current state, it "asks" the abstract program whether the expression $\mathsf{Dp}(\alpha)$ is satisfied. For the program to know which hooks $\ell \in H_{\mathbf{O}}$ are active in the current state, it "asks" the ontology whether an axiom in $\mathsf{pD}(\ell)$ is entailed. In the next section, we formalize this intuition and define the semantics of ontologized programs via induced MDPs.

3.2 Semantics of Ontologized Programs

The semantics is formally defined using *ontologized MDPs*. In order to account for both the program $\mathbf{P_O}$ and the ontology $\mathcal{K}_\mathbf{O}$, the ontologized MDP induced by $\mathbf{P_O}$ has to provide two views on its states. The first view is from the perspective of $\mathbf{P_O}$: for a stochastic programs, a system state is characterized by an evaluation over $Var_\mathbf{P}$. For instance, a state q might be associated with the following evaluation η_q:

$$\texttt{server\_proc1} = 2 \qquad \texttt{server\_proc2} = 2 \qquad \texttt{server\_proc3} = 0 \ ,$$

stating that Process 1 and Process 2 run on Server 3, while Process 3 is currently not running. The second view is from the perspective of the ontology: state q is characterized by a knowledge base $\mathcal{K}_q$ that contains all axioms in $\mathcal{K}_\mathbf{O}$ and

$$\mathsf{runsProcess}(\mathsf{server2}, \mathsf{process1}) \qquad \mathsf{runsProcess}(\mathsf{server2}, \mathsf{process2}) \ .$$

$\mathcal{K}_q$ entails $\mathsf{ShouldMigrate}(\mathsf{platform})$, and therefore the state q should be labeled with the hook `migrate`. We make this intuition formal in the following definition.

Definition 3. An *ontologized state* is a tuple of the form $q = \langle \eta_q, \mathcal{K}_q \rangle$, where η_q is an evaluation and $\mathcal{K}_q$ a DL knowledge base. Let $\mathbf{O}$ be an ontologized program as in Definition 2. An ontologized state q *conforms to* $\mathbf{O}$ iff

1. $\mathcal{K}_q \subseteq \mathcal{K}_\mathbf{O} \cup \mathcal{F}$,
2. $\mathcal{K}_\mathbf{O} \subseteq \mathcal{K}_q$, and
3. for every $\alpha \in \mathcal{F}_\mathbf{O}$, we have $\alpha \in \mathcal{K}_q$ iff $\eta_q \models \mathsf{Dp}(\alpha)$.

Intuitively, an ontologized state *conforms to* $\mathbf{O}$ if it conforms to the mapping Dp provided by the interface, as well as to the axioms specified by the ontology $\mathcal{K}_\mathbf{O}$. It follows from the definition that for every evaluation η and ontologized program $\mathbf{O}$, there is a unique ontologized state q conforming to $\mathbf{O}$ such that $\eta_q = \eta$. We refer to this ontologized state as $q = e(\mathbf{O}, \eta)$, which is defined by $\eta_q = \eta$ and $\mathcal{K}_q = \mathcal{K}_\mathbf{O} \cup \{\alpha \in \mathcal{F}_\mathbf{O} \mid \eta \models \mathsf{Dp}(\alpha)\}$. This observation allows us to define updates on ontologized states in a convenient manner. Specifically, the result of applying an update u on an ontologized state q is defined as $u(q) = e(\mathbf{O}, u(\eta_q))$. Intuitively, we first apply the update on the evaluation η_q of q, and then compute its unique extension to an ontologized state. Our definition naturally extends to stochastic updates, leading to distributions over ontologized states.

Let q denote the ontologized state from above and consider the update $u = \{\texttt{server\_proc1} \mapsto 2\}$. Then denote $q' = u(q)$ and obtain $u(\eta_q) = \eta_{q'}$ as

$$\texttt{server\_proc1} = 1 \qquad \texttt{server\_proc2} = 2 \qquad \texttt{server\_proc3} = 0 \ ,$$

and $\mathcal{K}_q = \mathcal{K}_\mathbf{O} \cup \mathcal{F}'$, where $\mathcal{F}'$ contains

$$\mathsf{runsProcess}(\mathsf{server1}, \mathsf{process1}) \qquad \mathsf{runsProcess}(\mathsf{server2}, \mathsf{process2}) \ .$$

While $\mathcal{K}_q \models \mathsf{ShouldMigrate}(\mathsf{platform})$, there is no such entailment in $\mathcal{K}_{q'}$, so that the hook `migrate` should become inactive in state q'.

In the ontologized MDP, states are labeled with constraints $\mathbb{C}(Var)$ and with hooks H. The hooks $h \in H_\mathbf{O}$ included in the label of a state q are determined by whether $\mathcal{K}_q \models \mathsf{pD}(h)$ is satisfied. This is captured using the labeling function of the MDP, since the labels determine relevant properties of a state for both model checking and update selection.

Definition 4. Let $\mathbf{O} = \langle \mathbf{P}, \mathcal{K}, \mathbf{I} \rangle$ be an ontologized program as in Definition 2. The *weighted MDP induced by* $\mathbf{O}$ is $\mathcal{M}[\mathbf{O}] = \langle Q, Act, P, q_0, \Lambda, \lambda, wgt \rangle$ where

- $Q = \{e(\mathbf{O}, \eta) \mid \eta \in Eval(Var_{\mathbf{P}})\}$,
- $Act = Distr(Upd(\mathbf{P}))$,
- $q_0 = e(\mathbf{O}, \eta_0)$,
- $\Lambda = H \cup \mathbb{C}(Var_{\mathbf{P}})$,
- $\lambda(q) = \mathbb{C}(\eta_q) \cup \{\ell \in H_{\mathbf{O}} \mid \mathcal{K}_q \models \mathsf{pD}(\ell)\}$ for every $q \in Q$,
- $P(q_1, \sigma, q_2) = (\eta_{q_2} \circ \sigma)(\eta_{q_2})$ for any $q_1, q_2 \in Q$ and $\langle g, \sigma \rangle \in C$ with $\lambda(q_1) \models g$, and
- $wgt(q) = \sum_{\langle g, w \rangle \in W, \lambda(q) \models g} w$ for all $q \in Q$.

The above definition closely follows the standard semantics for stochastic programs (see Section 2.2), while amending knowledge information to each state in such a way that hooks are assigned to states as specified by the interface $\mathbf{I}$. It thus can be shown similarly that the weighted MDP induced by a ontologized program is always well defined.

3.2.1 Remark on inconsistent states.

Note that our formalism allows for states of the induced MDP to have logically inconsistent knowledge bases assigned. We call those states *inconsistent*. We can identify and mark inconsistent states easily using a hook $\ell_\perp \in H$ for which we set $\mathsf{pD}(\ell_\perp) = \{\top \sqsubseteq \bot\}$. Depending on the application, inconsistent states might or might not be desirable. In general, there are different ways in which such states can be handled within our framework: 1) Inconsistent states could stem from errors in specification of the operational behavior or in the ontology. We would then want to provide users with tool support for detecting whether the program can enter an inconsistent state. Existing model-checking tools can directly be used for this, as they just have to check whether a state labeled with $\ell_\perp$ is reachable. 2) The stochastic program can detect inconsistent states using the hook $\ell_\perp$, and act upon them accordingly to resolve the inconsistency. This could be useful, e.g., for modeling exception handling or interrupts within the program to deal with unexpected situations. 3) Both the nondeterministic and probabilistic choices in the MDP can be restricted to only enter consistent states. The ontology then has a direct impact on the state space of the MDP. This can be seen as a desirable feature of ontologized programs, as different ontologies may pose different constraints on possible states a system may enter, which can be quite naturally expressed using DL axioms.

4 Analysis of Ontologized Programs

For the quantitative analysis of ontologized programs, we make use of a probabilistic model checking tool in combination with a DL reasoner. Specifically, the DL reasoner is used to decide which hooks are assigned to each state in the MDP. This in turn depends on the axioms entailed by the knowledge base assigned to the state. Constructing the ontologized states explicitly is not feasible in practice, as there can be exponentially many. PMC tools as PRISM use advanced techniques to represent the set of MDP states concisely, e.g., through symbolic representations via MTBDDs [26], which is vital to their performance. However, although such representation can also be useful for exponentially many ontologized states, this representation does not provide a method how to assign the hooks. Furthermore, DL reasoning itself can be

costly: for the DL $\mathcal{SROIQ}$ underlying the OWL-DL standard, it is N2ExpTime-complete [20], and already for its fragment $\mathcal{ALCQ}$ introduced in Section 2.3, it is ExpTime-complete [33]. Even though there exist efficient reasoners that can deal with large OWL-DL ontologies [28], if we want to perform model checking practically, we have to avoid calling the reasoner exponentially many times.

In settings where ontologies are used to enrich queries over databases [8], a common technique is to *rewrite* queries by integrating all relevant information from the ontology. This allows for a direct evaluation of the rewritten query using standard database systems [9]. We propose a similar technique here, where we rewrite the ontologized program into a stochastic program that can be directly evaluated using a probabilistic model checker. To do this efficiently, our technique aims at reducing the amount of reasoning required, as well as to reduce the size of the resulting program.

To formalize this idea, we define a translation t from ontologized programs $\mathbf{O}$ into stochastic programs $t(\mathbf{O})$ that do not contain any hooks in guards. The translation is based on an assignment $\mathsf{hf}\colon H \to \mathbb{B}(\mathbb{C}(\mathit{Var}))$ of hooks $\ell \in H_{\mathbf{O}}$ to corresponding *hook formulas* $\mathsf{hf}(\ell)$, such that the MDPs induced by $\mathbf{O}$ and by $t(\mathbf{O})$ correspond to each other except for the hooks. This correspondence is captured in the following definition.

Definition 5. Given two weighted MDPs,

$$\mathcal{M} = \langle S, \mathit{Act}, P, s_0, \Lambda, \lambda, \mathit{wgt}\rangle$$

and

$$\mathcal{M}' = \langle S', \mathit{Act}', P', s_0', \Lambda', \lambda', \mathit{wgt}'\rangle,$$

such that $\mathit{Act} = \mathit{Act}'$, and a partial function $\mathsf{hf}\colon \Lambda \rightharpoonup \mathbb{B}(\Lambda')$ mapping labels in Λ to formulas over Λ', the weighted MDPs $\mathcal{M}$ and $\mathcal{M}'$ are *equivalent modulo* hf iff there exists a bijection $b\colon S \leftrightarrow S'$ such that

1. $b(s_0) = s_0'$,
2. for every $s_1, s_2 \in S$ and $\alpha \in \mathit{Act}$, $P(s_1, \alpha, s_2)$ is defined iff $P'(b(s_1), \alpha, b(s_2))$ is defined, and $P(s_1, \alpha, s_2) = P'(b(s_1), \alpha, b(s_2))$,
3. for every $s \in S$, $\mathit{wgt}(s) = \mathit{wgt}'(b(s))$, and
4. for every $\ell \in \Lambda$ and $s \in S$ holds that $\ell \in \lambda(s)$ iff $\lambda(b(s)) \models \mathsf{hf}(\ell)$.

This notion extends to stochastic programs and ontologized programs via their induced MDPs: an ontologized program $\mathbf{O}$ and a stochastic program $\mathbf{P}$ are equivalent modulo hf iff $\mathcal{M}[\mathbf{O}]$ and $\mathcal{M}[\mathbf{P}]$ are equivalent modulo hf.

If an ontologized program $\mathbf{O}$ and a stochastic program $\mathbf{P}$ are equivalent modulo hf, all analysis tasks on $\mathbf{O}$ can be reduced to analysis on $\mathbf{P}$, as we just have to replace any label ℓ relevant for the analysis by $\mathsf{hf}(\ell)$. This leads for instance to a straight-forward translation of properties expressed using temporal logics. As a result, we can perform any PMC task that is supported by a PMC tool like Prism on ontologized programs, provided that the translation function hf and the corresponding stochastic program can be computed practically.

Based on $\mathbf{O}$ we define a function hf that can be efficiently computed using DL reasoning and which can be used to compute a corresponding stochastic program equivalent to the ontologized program modulo hf. Specifically, for every constraint $c \in \mathbb{C}(\mathit{Var}_{\mathbf{P}})$ we set $\mathsf{hf}(c) := c$, and for every hook $\ell \in H$, we provide a *hook-formula* $\mathsf{hf}(\ell)$. In other words, we only provide for a translation of the hook formula, and keep the evaluations in the program the same. The stochastic program $t(\mathbf{O})$ is then obtained from $\mathbf{O}$ by replacing every hook $\ell \in H$ by $\mathsf{hf}(\ell)$.

This is sufficient, since the labels assigned to an ontologized state are fully determined by the evaluation of the state: the axioms that are part of the state are determined by the mapping $\mathsf{Dp}\colon \mathcal{F}_{\mathbf{O}} \to \mathbb{B}(\mathbb{C}(\mathit{Var}))$, and the labels that are part of the state are determined by using the mapping $\mathsf{pD}\colon H \to \wp(\mathbb{A})$, based on which axioms are entailed by the ontology assigned to the state.

To compute hf in a goal-oriented manner, we make use of so-called *justifications*. These are defined independently of the DL in question, and there exist tools for computing justifications in various DLs.

Definition 6. Given a knowledge base $\mathcal{K}$ and an axiom α s.t. $\mathcal{K} \models \alpha$, a subset $\mathcal{J} \subseteq \mathcal{K}$ is a *justification of* $\mathcal{K} \models \alpha$ iff $\mathcal{J} \models \alpha$, and for every $\mathcal{J}' \subsetneq \mathcal{J}$, $\mathcal{J}' \not\models \alpha$. We denote by $\mathsf{J}(\mathcal{O}, \alpha)$ the set of all justifications of $\mathcal{J} \models \alpha$.

Intuitively, a justification for $\mathcal{K} \models \alpha$ is a minimal sufficient axiom set witnessing the entailment of α from $\mathcal{K}$. For the hook formula $\mathsf{hf}(\ell)$, we consider the justifications $\mathcal{J}$ of $\mathcal{K}_{\mathbf{O}} \cup \mathcal{F} \models \mathsf{pD}(\ell)$, as these characterize exactly those subsets $\mathcal{F}' \subseteq \mathcal{F}_{\mathbf{O}}$ for which $\mathcal{K}_{\mathbf{O}} \cup \mathcal{F}' \models \mathsf{pD}(\ell)$. Note that for each such justification $\mathcal{J}$, only the subset $\mathcal{J} \setminus \mathcal{K}_{\mathbf{O}}$ is relevant. We thus define the hook formula $\mathsf{hf}(\ell)$ for $\ell \in H$ as

$$\mathsf{hf}(\ell) = \bigvee_{\mathcal{J} \in \mathsf{J}(\mathcal{K}_{\mathbf{O}} \cup \mathcal{F}, \mathsf{pD}(\ell))} \; \bigwedge_{\alpha \in (\mathcal{J} \cap \mathcal{F}_{\mathbf{O}})} \mathsf{Dp}(\alpha) \;\; . \tag{7}$$

Here, we follow the convention that the empty disjunction corresponds to a contradiction $\bot$, while the empty conjunction corresponds to a tautology $\top$.

The final translation $t(\mathbf{O})$ of the ontologized program $\mathbf{O} = \langle \mathbf{P}, \mathcal{K}, \mathbf{I} \rangle$ is then obtained from $\mathbf{P}$ by replacing every hook $\ell \in H$ by $\mathsf{hf}(\ell)$.

Theorem 7. *The ontologized program* $\mathbf{O}$ *and the stochastic program* $t(\mathbf{O})$ *are equivalent modulo* hf.

Proof. We take the MDP $\mathcal{M} = \mathcal{M}[t(\mathbf{O})]$ induced by the translated program $t(\mathbf{O})$ and extend it to the MDP $\mathcal{M}' = \mathcal{M}[\mathbf{O}]$ induced by the ontologized program $\mathbf{O}$, such that there exists a bijection as in Definition 5. Specifically, based on the MDP

$$\mathcal{M} = \mathcal{M}[t(\mathbf{O})] = \langle \mathit{Eval}(\mathit{Var}), \mathit{Distr}(\mathit{Upd}), P, \eta_0, \mathbb{C}(\mathit{Var}), \lambda, \mathit{wgt} \rangle$$

induced by $t(\mathbf{O})$, we define the MDP

$$\mathcal{M}' = \langle Q', \mathit{Distr}(\mathit{Upd}), P', e(\mathbf{O}, \eta_0), \mathbb{C}(\mathit{Var}) \cup H, \lambda', \mathit{wgt}' \rangle,$$

where

- $Q' = \{e(\mathbf{O}, \eta) \mid \eta \in \mathit{Eval}(\mathit{Var})\}$,
- for every $\eta_1, \eta_2 \in \mathit{Eval}(\mathit{Var})$ and $d \in \mathit{Distr}(\mathit{Upd})$ for which $P(\eta_1, d, \eta_2)$ is defined, we set $P'(e(\mathbf{O}, \eta_1), d, e(\mathbf{O}, \eta_2)) = P(\eta_1, d, \eta_2)$,
- for every $q \in Q'$, $\mathit{wgt}'(q) = \mathit{wgt}(\eta_q)$, and
- for every $q \in Q'$, $\lambda'(q) = \lambda(\eta_q) \cup \{\ell \in H \mid \mathcal{K}_q \models \mathsf{pD}(\ell)\}$.

The bijection $b\colon Q' \leftrightarrow \mathit{Eval}(\mathit{Var})$ is then defined by setting $b(q) = \eta_q$ for all $q \in Q'$. Clearly b is a bijection that satisfies Conditions 1–4 in Definition 5, so that $\mathcal{M}$ and $\mathcal{M}'$ are equivalent modulo hf. It remains to show that $\mathcal{M}'$ is the MDP induced by $\mathbf{O}$. We start with the following claim.

Claim. For every $\ell \in H$ and $q \in Q'$, $\lambda'(q) \models \ell$ iff $\lambda'(q) \models \mathsf{hf}(\ell)$.

Proof of claim. ($\Rightarrow$) Assume $\lambda'(q) \models \ell$. By construction of $\mathcal{M}'$ and the fact that λ maps to $\mathbb{C}(Var)$, we have $\mathcal{O}_q \models \mathsf{pD}(\ell)$. Then, there is a justification $\mathcal{J}$ for $\mathcal{O}_q \models \mathsf{pD}(\ell)$. Since $\mathcal{O}_q \subseteq \mathcal{K}_{\mathbf{O}} \cup \mathcal{F}$, $\mathcal{J}$ is also a justification for $\mathcal{K}_{\mathbf{O}} \cup \mathcal{F} \models \mathsf{pD}(\ell)$. Furthermore, since every state in Q' conforms to $\mathbf{O}$, by Definition 3, $\eta_q \models \mathsf{Dp}(\alpha)$ for all $\alpha \in (\mathcal{J} \setminus \mathcal{K}_{\mathbf{O}}) \subseteq (\mathcal{O}_q \setminus \mathcal{K}_{\mathbf{O}})$. It follows that $\eta_q \models \bigwedge_{\alpha \in \mathcal{J} \setminus \mathcal{K}_{\mathbf{O}}} \mathsf{Dp}(\alpha)$, and consequently $\eta_q \models \mathsf{hf}(\ell)$, $\lambda(\eta_q) \models \mathsf{hf}(\ell)$ and $\lambda'(q) \models \mathsf{hf}(\ell)$.

($\Leftarrow$) Conversely, assume $\lambda'(q) \models \mathsf{hf}(\ell)$. Then, by construction of hf, there is a justification $\mathcal{J}$ for $\mathcal{K}_{\mathbf{O}} \cup \mathcal{F} \models \mathsf{pD}(\ell)$ s.t. $\eta_q \models \bigwedge_{\alpha \in \mathcal{J} \setminus \mathcal{K}_{\mathbf{O}}} \mathsf{Dp}(\alpha)$. Since q is an ontologized state that conforms to $\mathbf{O}$, Definition 3 yields that for every $\alpha \in \mathcal{J} \setminus \mathcal{K}_{\mathbf{O}}$, $\eta_q \models \mathsf{Dp}(\alpha)$ iff $\mathcal{O}_q \models \alpha$. Since also $\mathcal{K}_{\mathbf{O}} \subseteq \mathcal{O}_q$, we obtain that $\mathcal{J} \subseteq \mathcal{O}_q$, and since $\mathcal{J} \models \mathsf{pD}(\ell)$, that $\mathcal{O}_q \models \mathsf{pD}(\ell)$. Now, by construction of $\mathcal{M}'$, we obtain that $\ell \in \lambda'(q)$, and consequently that $\lambda'(q) \models \ell$. ■

As a consequence of this claim, for every Boolean formula $\phi \in \mathbb{B}(Eval(Var) \cup H)$ and $q \in Q'$, $\lambda'(q) \models \phi$ iff $\lambda(b(q)) \models \mathsf{hf}(\phi)$, where $\mathsf{hf}(\phi)$ denotes the result of replacing in ϕ every hook $\ell \in H$ by $\mathsf{hf}(\ell)$. We obtain that for every guarded command $\langle g, s \rangle \in C$, where C is the set of guarded commands of the ontologized program $\mathbf{O}$, and for every $q \in Q'$, $\lambda'(q) \models g$ iff $\lambda'(\eta_q) \models \mathsf{hf}(g)$, which means that P' satisfies the condition in Definition 4. Similarly, for W the set of weight assignments in $t(\mathbf{O})$, and W' the set of weight assignments in $\mathbf{O}$, we obtain that for every $q \in Q'$,

$$wgt'(q) \;=\; wgt(\eta_q) \;=\; \sum_{\substack{\langle \mathsf{hf}(g), z \rangle \in W \\ \lambda(\eta_q) \models \mathsf{hf}(g)}} z \;=\; \sum_{\substack{\langle g, z \rangle \in W' \\ \lambda'(q) \models g}} z \;,$$

which means that also wgt' satisfies the conditions in Definition 4. Hence, $\mathcal{M}'$ is indeed the MDP induced by $\mathbf{O}$. We obtain that the MDP induced by $t(\mathbf{O})$ and the MDP induced by $\mathbf{O}$ are equivalent modulo hf, and consequently, that $t(\mathbf{O})$ and $\mathbf{O}$ are equivalent modulo hf. □

5 Evaluation

We implemented the method described in Section 4, where we use the input language of PRISM [22] to specify the abstract program, and the standard web ontology language OWL-DL [25] to specify the context. Specifically, our tool-chain computes a stochastic program based on the ontologized program, on which we can directly perform ontology-dependent PMC using PRISM. Since the PRISM supports macro definitions, the hook assignments provided by the computed function hf could be conveniently used within the program, which we used to generate the translated stochastic program that was finally used by PRISM, and on which we performed several stochastic analysis tasks.

For computing hf, we implemented the method described in Section 4 in Java using the OWL-API [17], where we used the reasoner Pellet [32] for computing the justifications. Pellet supports most of the OWL DL profile, and furthermore comes with an integrated implementation for the computation of justifications using a glass box approach. To further improve the performance, we adapted the main class for computing justifications for our specific needs. Note that in Equation 7 specifying the hook formula, we only interested in the intersection of the full justification with the set $\mathcal{F}$ of the axioms the program can actually change. Usually, the algorithm computing all justifications would consider all subsets of $\mathcal{K}_{\mathbf{O}} \subseteq \mathcal{F}$, ignoring the separation into $\mathcal{K}_{\mathbf{O}}$ and $\mathcal{F}$. We therefore modified the algorithm so that it ignores axioms in $\mathcal{K}_{\mathbf{O}}$ when comparing with earlier solution, which reduces the search space a lot of the set of axioms in $\mathcal{K}_{\mathbf{O}}$ is large. Apart from this optimization, we computed the situation formulas exactly as described in Section 4.

5.1 Multi-Server System Setting

Using the optimizations presented, we are able to analyze the multi-server system that served as running example in the last sections. We embedded our implementation into a generic tool chain that can be parametrized by providing the number of servers, their operating system, the number of processes and prioritized processes, as well as the durations of the tasks. In this evaluation, we focus on two particular scenarios. The first comprises one -server and two -server on which six processes are running, while the second comprises one -server and one -servers running eight processes. For each scenario we assumed that at any time step there is a 50% probability of a new job arriving for some process. To show-case the impact of changing the program and the ontology, we implemented two different program specifications and four different ontologies. The program specifications are provided in PRISM code, where one implements a randomized strategy for to select the next job to be executed, and the other implements a round-robin strategy for this. As we want to evaluate the programs based on energy-consumption and utility, we furthermore used weights to model energy consumption, and variables to store achieved utility. Here, a server consumes energy when one or more processes are assigned to it, and we obtain utility for each successfully executed process.

We implemented four different ontologies that mostly follow the general idea as in our running example, and differ in the specification of the hooks, which are

- a *critical system state hook* to mark states the system should avoid, which we call in the following *critical states*,
- a *migrate hook* describing when the system should schedule the migration of a process, and
- *consistency hooks* specifying when it is allowed for a given process to be moved to a particular server, taking into account both capacity and compatibility limitations.

Table 1 gives an overview of the four ontologies in terms of concepts to be fulfilled to enable the critical system state hook (C), the migrate hook (M), and the inconsistency hooks (I). For instance, for Ontology 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 that does not have the operating system the process is compiled for.

Table 1: Varying different situations for different contexts

context	prioritized runs on overloaded	prioritized runs on almost overloaded	two prioritized on same server	some server overloaded	max. number of procs on server	incompatible operating system
1	C	M		M	I	
2	C	M		M	I	I
3	C	M			I	I
4	C		C	M	I	I

Note how these ontologies provide for a convenient way of not only specifying the specifics of a given multi-server platform in terms of software configuration and quality constraints, but also of specifying different migration strategies, without the designer having to hamper with the specification of the program.

Within all these combinations, we obtained $2 \cdot 4 \cdot 2 = 16$ ontologized programs in total, which we translated into stochastic programs expressed in the input language of PRISM. Note that we only had to employ two different interfaces for contextualized programs: one for each server configuration (3 and 2 servers, respectively).

5.2 Energy-aware Analysis

For the analysis of the ontologized programs described in the last section, we first considered the following standard reachability properties.

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

(2) What is the expected energy consumption for gaining at least 20 utility?

(3) What is the expected number of critical system states before reaching 20 utility?

For each of these properties, we computed the minimal and maximal probabilities determined by resolving the non-deterministic choices in the MDP in a best/worse-case manner.

Note that the weight annotations above give rise to trade-offs between costs and utility, where energy consumption or entering a critical system state can be seen as *costs*. For instance, it might be favorable to migrate a priority process to a different server, which on the one hand might require additional energy, but on the other hand, depending on the ontology, reduces the chances of entering a critical system state, and increases the chance of completing the process and obtaining utility for it. To investigate this trade-off, we considered the following energy-utility quantiles [5].

(egy) What is the minimal energy consumption required for gaining at least 20 utility with probability at least 95%?

(crit) What is the minimal number of occurrences of a critical system state within which at least 20 utility is gained with probability at least 95%?

In following Table 2, the analysis results of the Reachability Properties (1)–(3) and quantiles are shown. The results for the properties that directly depend on critical system state hooks are also depicted in the second graph in Figure 1. Bars span the range of minimal and maximal expected number of critical system states and the critical situation quantile value is depicted by dots. The four different ontologies considered in both hardware/software settings are listed in the x-axis, using the notation "c-s",where "c" stands for the context and "s" for the number of servers. The blue and red bars show the values for random and round-robin scheduling behavior modeled in the stochastic program. In the case of the two-server setup, only in the first context there is some freedom in performing migrations as in this context all software instances are placed on both servers while in the other contexts each server has a different software setup.

First, we considered the system configuration with two 🐧-servers and one ⊞-server running six processes. One can observe that the different ontologies have great impact on the results. As two 🐧-servers are at hand, 🐧-processes can be migrated without reaching an inconsistent state, leading to a wide range of best/worst-case results for properties (1)-(3). This is not the case in the scenario with one 🐧-server and one ⊞-server running eight processes, where (except for the first context disregarding operating systems) no migration can occur. Here, min- and max-values agree for Contexts 2, 3, and 4. Furthermore, the probability of entering a critical system state is much higher than in the first scenario, which also yields a much higher amount of critical situations that are entered until achieving utility (see values for the crit-quantile).

Table 2: Analysis results

server/ proc	context	program	prob. crit (1) min	prob. crit (1) max	exp. egy (2) min	exp. egy (2) max	exp. crit (3) min	exp. crit (3) max	quantile egy	quantile crit
🐧🐧🪟/6	1	random	0.6604	0.9983	29.09	58.57	1.73	28.38	31	4
		roundr	0.6392	0.9983	27.92	62.25	1.53	28.72	30	4
	2	random	0.4558	0.9866	29.17	47.63	0.91	14.05	32	3
		roundr	0.4256	0.9854	27.96	48.17	0.78	13.62	30	2
	3	random	0.4558	0.9866	29.17	47.63	0.91	14.05	32	3
		roundr	0.4256	0.9855	28.73	49.09	0.79	13.66	31	2
	4	random	0.7776	0.7776	29.26	32.04	3.37	7.97	32	8
		roundr	0.7612	0.7612	28.99	31.61	3.06	6.50	32	7
🐧🪟/8	1	random	0.9988	0.9999	32.16	45.80	18.26	30.53	35	23
		roundr	0.9985	0.9999	29.41	48.11	15.56	32.67	32	20
	2	random	0.9991	0.9991	34.06	34.06	27.87	27.87	37	34
		roundr	0.9998	0.9998	32.57	32.57	25.66	25.66	35	30
	3	random	0.9991	0.9991	34.06	34.06	27.87	27.87	37	34
		roundr	0.9998	0.9997	32.76	32.76	25.66	25.66	35	30
	4	random	0.9819	0.9819	28.22	28.22	20.05	20.05	31	28
		roundr	0.9845	0.9845	26.77	26.77	16.57	16.57	29	24

In the latter scenario, the definition of a critical situation might be inappropriate as running four processes on each server is a standard configuration rather than critical. These results thus suggest the developers to adapt ontology definitions in a further refinement step. In all scenarios we see that a round-robin job-selection strategy is superior to the randomized one when the objective concerns minimizing energy consumption, critical system states and their trade-off properties.

All the experiments were carried out[1] using the symbolic MTBDD engine of PRISM in the version presented in [21]. The run-time statistics are shown in Table 3, showing a great impact of scenarios, contexts, and program variants on the model characteristics. Noticeable, when different operating systems are not modeled within the ontology, the arising state-space explosion problem leads to tremendous increase of the analysis times, ranging up to 5 days computation time. This shows the capability of ontology-mediated PMC to reduce the state space and making analysis feasible. The generation of all 16 models considered, including time for DL reasoning and generating PRISM code took only 130 seconds in total. To showcase the analysis times for the computation of minimal and maximal expected number of critical system states and for the critical system state quantile, see the top graph in Figure 1.

6 Related Work

6.1 Model checking context-dependent systems

The idea of using different formalisms for behaviors and contexts to facilitate model checking goes back to [12], where a scenario-based *context description language (CDL)* based on message

[1] Hardware setup: Intel Xeon E5-2680@2.70GHz, 128 GB RAM; Turbo Boost and HT enabled; Debian GNU/Linux 9.1

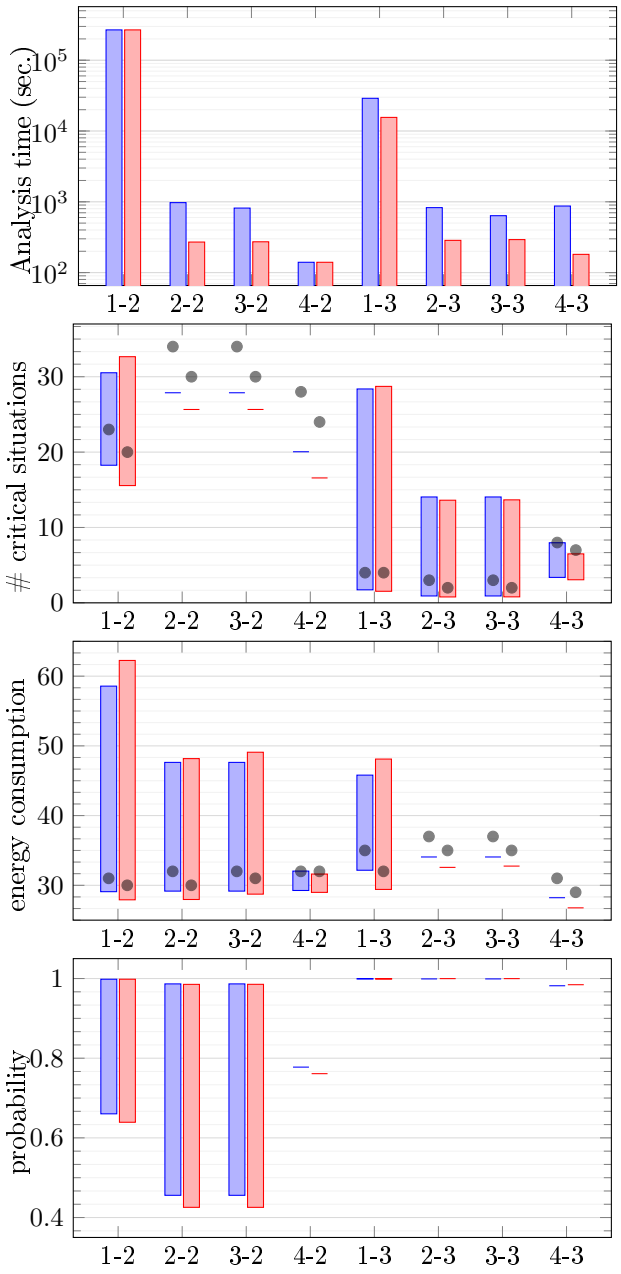

Figure 1: Visualisation of elected analysis results. In order from top to bottom: running times of the experiments (left, logarithmic scale), number of critical system states, energy consumption and probability of entering a critical system state (bars visualise respective min. and max. values for reaching 20 utility, dots the corresponding quantiles).

Table 3: Statistics of the analysis

server/ proc	context	program	states	nodes	analysis time [s] reach	quantiles
🐧🐧🪟/6	1	random	23'072'910	173'841	2'011.52	2'308.40
		roundr	37'231'023	3'718'247	18'759.57	47'598.00
	2	random	800'814	16'782	164.18	236.93
		roundr	967'250	106'897	314.88	404.51
	3	random	800'814	15'666	141.45	208.63
		roundr	975'526	96'030	325.56	412.91
	4	random	773'598	15'769	117.88	161.77
		roundr	425'306	44'724	126.69	156.91
🐧🪟/8	1	random	90'027'882	134'648	4'662.57	18'268.61
		roundr	66'116'970	2'933'937	56'935.82	384'886.94
	2	random	934'122	6'518	79.03	191.39
		roundr	158'368	7'507	49.81	271.52
	3	random	934'122	6'518	69.14	159.78
		roundr	158'432	7'372	46.52	325.56
	4	random	934'122	6'830	35.68	51.79
		roundr	157'472	7'455	45.28	124.75

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 ontologies. Modeling and model checking role-based systems with exploiting exogenous coordination has been detailed in [11, 7]. Here, components may play different roles in specific contexts (modeled through elements called *compartments*). As the approach above, the formalism to specify contexts is the same as for components, and a parallel composition is used for deployment. Feature-oriented systems describe systems comprising features that can be active or inactive (see, e.g., [14]). We can employ similar principles within our framework to combine ontological elements, as show-cased in our evaluation in Section 5. A reconfiguration framework for context-aware feature-oriented systems has been considered in [24]. All the above formalisms use an operational description of contexts, while we intentionally focused on a knowledge-based representation through ontologies that allows for reasoning about complex information and enables the reuse of established knowledge bases.

6.2 Description logics in Golog programs

There is a relation between our work and work on integrating DLs and ConGolog programs [4, 34]. 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. In contrast, we provide a generic approach that allows to employ various PMC tasks using existing tools, and allow for probabilistic programs. Furthermore, ontologies and program statements are not separated as in our approach. However, the main difference is that in the semantics of [4, 34], states are identified with interpretations rather than knowledge bases, which are directly modified by the program. This makes reasoning much more challenging, and easily leads to undecidability if syntactic restrictions are not carefully put. Closer to our semantics are the DL-based programs presented in [16, 10], where actions consist of additions

and removals of assertions in the knowledge base. 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.

6.3 Ontology-mediated query answering

There is a resemblance between the our concept of ontology-mediated PMC and *ontology-mediated query answering* (OMQA) [29, 8], which also inspired the title of this paper. 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 [29]. 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, different to our approach, this is in general only possible for very restricted DLs, while for expressive DLs, the complexity of OMQA is often very high [31, 27, 23].

7 Discussion and Future Work

We introduced ontologized programs, in which stochastic programs specify operational behaviors, and DLs are used to describe additional knowledge, with the aim of facilitating quantitative analysis of knowledge-intensive systems. From an abstract point of view, the general idea is to use different, domain-specific formalisms for specifying the program and knowledge, which are linked through hooks by an interface. We believe that the general idea of specifying operational behaviour 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., 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.

Regarding the specific ontologized programs introduced in this paper, several improvements are possible. First, as discussed in Section 3.2, we are currently not addressing inconsistent states in the ontologized programs directly, but offer various ways to deal with them in the program or analysis. In future work, we want to investigate integrated mechanisms for handling inconsistent states in an automatized way. Second, one could look at closer integrations between the ontology and the abstract program by means of a richer interface. For example, we could map numerical values directly into the DL by use of *concrete domains* [2], which would allow to express more numerical constraints in the ontology. Furthermore, we want to investigate dynamic switching of ontologies during program execution, to model complex interaction between ontologies as in [14], exploiting the close connection to feature-oriented systems discussed in Section 6.

References

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

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

[7] 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.

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

[9] 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.

[10] 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.

[11] P. Chrszon, C. Dubslaff, S. Klüppelholz, and C. Baier. Family-based modeling and analysis for probabilistic systems - Featuring ProFeat. In *Proc. of Fundamental Approaches to Software Engineering (FASE'16)*, volume 9633 of *LNCS*, pages 287–304. Springer, 2016.

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

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

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

[15] 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.

[16] 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.

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

[18] Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The even more irresistible $\mathcal{SROIQ}$. In Patrick Doherty, John Mylopoulos, and Christopher A. Welty, editors, *Proceedings of KR 2006*, pages 57–67. AAAI Press, 2006.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] Deborah L McGuinness and Frank Van Harmelen. Owl web ontology language overview. *W3C recommendation*, 10(10), 2004.

[26] 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.

[27] 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.

[28] 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.

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

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

[31] 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.

[32] 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.

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

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