Contextualized Programs for
Ontology-Mediated Probabilistic System Analysis

Clemens Dubslaff, Patrick Koopmann, and Anni-Yasmin Turhan

LTCS-Report 18-09
Contents

1 Introduction 2

2 Preliminaries 3
   2.1 Markov Decision Processes ........................................... 4
   2.2 Stochastic Programs .................................................. 4
   2.3 Description Logics ................................................... 5

3 Contextualized Programs 6
   3.1 Contextualizing Stochastic Programs with Ontologies .......... 6
   3.2 Semantics ............................................................. 7
   3.3 Example .............................................................. 9

4 Analysis of Contextualized Stochastic Programs 11

5 Evaluation 14
   5.1 Multi-Server System Setting ......................................... 14
   5.2 Energy-aware Analysis ............................................... 16

6 Discussion and Future Work 18
Contextualized Programs for
Ontology-Mediated Probabilistic System Analysis

Clemens Dubslaff, Patrick Koopmann, and Anni-Yasmin Turhan

December 3, 2018

Abstract

We introduce the concept of contextualized programs to model and analyze situation-aware systems, i.e. systems that “sense” their context and adapt to it. Within contextualized programs, operational behaviors of and contextual knowledge about the situation-aware system are modeled separately using domain-specific formalisms. The behavior may depend on contextual information resolved by reasoning over the context ontology, for example, using description logic reasoners. We realize our approach in detail for behaviors specified by stochastic guarded command language, a formalism used for example by the PMC tool PRISM, and ontologies formalized in OWL. For this framework, we developed a technique to translate away the context so that we can perform stochastic system analysis using standard PMC tools. Our prototype implementation shows the applicability of the presented concepts, issuing a small case study on a heterogeneous energy-aware multi-server system.

1 Introduction

Context-awareness plays an important role in software and hardware systems and many approaches have been proposed to include contextual information in the system’s engineering process (see, for example, [1]). An entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves.” identified the There is no unified notion of context due to its usage in a wide range of areas. We adopt the notion of a context as a “set of situational elements in which the object being processed is included” [6]. Consequently, context-aware systems exhibit behaviors that depend on properties of the situational elements, usually expressed in the same manner as the specification of the system itself. For instance, following the idea of context-oriented programming, contexts are explicitly incorporated into program code [12]. While these approaches faithfully support behavior variations depending on the context, we argue that they might be not able to describe all the facets of contexts, possibly involving sophisticated dependencies between situational elements. Resolving such dependencies in contexts is a knowledge-intensive task that includes knowledge not known in advance to be included into the behavioral system description [10].

In this paper, we propose contextualized programs as a new approach to describe context-aware systems. The main idea is to separate concerns between behavioral and contextual descriptions, i.e., using different formalisms for the specification of behaviors and contexts, each specialized for their respective needs. Behaviors can be specified by, for example, program code, UML state charts, or control-flow diagrams, where placeholders can refer to elements specific to the context. Contexts themselves are modeled by ontologies defined using description logics (DLs) [2, 3].
Both formalisms are connected through an interface between behaviors and contexts which maps placeholders and state descriptions in the behavioral description to DL expressions. We propose a technique to reduce stochastic analysis of contextualized programs to stochastic analysis of stochastic programs as used in Prism. This technique uses axiom pinpointing for DL ontologies to fully replace all placeholders based on the current context ontology to obtain a stochastic program that uses the same specification formalism as the behavioral description. This approach allows us to reuse existing tools for the program development and analysis in the behavioral specification formalism for the contextualized variants of these programs.

Motivating Example. Let us consider a multi-server system executing jobs of several processes distributed over multiple servers. The behavior of the system includes a protocol that selects the next job to be executed on a server, for example, jobs are scheduled randomized or in a round-robin fashion. Furthermore, processes can be migrated to other servers. The context of the system is given by an ontology that adds further details on the overall system, for example, such as which operating systems run on a server, priorities of processes, and constraints such as the maximal number of processes executable. In this setting, behaviors depend on the context of the system in the following ways. First, the context gives additional details about different instances of the setup, which might differ in the hardware and software configuration of the multi-server system, and different constraints resulting from this. Second, different implementations of the system might follow different migration strategies, which themselves again depend on the context of the program. For example, migration might be invoked by a load-balancing mechanism. Using such a flexible modeling allows to test and compare different migration strategies, and analyse whether certain quality levels can be guaranteed in different configurations of the system, and for which elements of a context specification, better system behaviors can be expected.

Analysis of Contextualized Programs. Our approach is motivated by the challenge of analyzing and verifying context-aware systems, where it is important to avoid costly redesign steps during their development. We exemplify contextualized programs that comprise behavioral descriptions in a stochastic variant of Dijkstra's guarded command language and description logic ontologies. The former is commonly used in state-of-the-art probabilistic model checkers [17] and for the latter description logic reasoners can be used to infer relevant logical entailments in the different states of the system depending on the context. The overall system then can then bridge the gap between programs and the context they are executed in. The semantics of a contextualized program is provided in terms of a Markov decision process (MDP) with annotated costs and rewards, where states are additionally annotated with description logic axioms. To allow for an efficient analysis of contextualized programs using standard PMC tools, we develop a technique to reduce analysis of contextualized programs to the analysis of programs without context. This enables various quantitative analysis and synthesis tasks to evaluate context-aware system behaviors of the contextualized program. We implemented the technique and evaluated it based on the multi-server scenario sketched above. We also show that our approach facilitates the analysis of context-aware systems with varying or evolving behavior and contexts.

2 Preliminaries

Given a countable set $S$, we denote by $\varphi(S)$ the powerset of $S$. A distribution over a set $S$ is a function $\mu : S \rightarrow [0,1]$ with $\sum_{s \in S} \mu(s) = 1$. The set of distributions over $S$ is denoted by $\text{Distr}(S)$. 

2.1 Markov Decision Processes

The operational model used in this paper is given in terms of Markov decision processes (MDPs) (see, e.g., [18]), defined as tuples \( M = (Q, \text{Act}, P, q_0, AP, \lambda) \), where \( Q \) and \( \text{Act} \) are countable sets of states and actions, respectively. \( P : Q \times \text{Act} \rightarrow \text{Distr}(Q) \) is a probabilistic transition function, \( q_0 \in Q \) an initial state, and \( AP \) a set of labels assigned to states via the labeling function \( \lambda : Q \rightarrow \varphi(AP) \). Intuitively, in any state \( q \), an action \( \alpha \in \text{Act} \) is selected non-deterministically, followed by a probabilistic choice selecting a successor state \( q' \) with probability \( P(q, \alpha, q') > 0 \). \( M \) is finite if \( Q \) and \( \lambda(q) \) are finite for all \( q \in Q \). A path in \( M \) is a sequence \( q_0, q_1, q_2, \ldots \), where \( P(q_i, \alpha_i, q_{i+1}) > 0 \) for all \( i \) for which \( q_{i+1} \) is defined. The set of enabled actions \( \text{Act}(q) \) in a state \( q \in Q \) is the set of all actions \( \alpha \in \text{Act} \) for which there is a \( q' \in Q \) with \( P(q, \alpha, q') > 0 \). A path is said to be maximal if it is either infinite or finite and ends in a trap, i.e., a state \( q \in Q \) where \( \text{Act}(q) = \emptyset \). The resolution of non-deterministic choices between actions is formalized via schedulers. For the purposes of this paper, it suffices to consider deterministic, possibly history-dependent schedulers, i.e., partial functions \( \mathcal{S} \) that map finite paths \( \pi \) to an action \( \mathcal{S}(\pi) \in \text{Act}(q_0) \). A finite path \( q_0, q_1, q_2, \ldots, q_n \) is a \( \mathcal{S} \)-path when \( q_0 = \mathcal{S}(q_0) \) for all \( i < n \). The behavior of \( M \) under \( \mathcal{S} \) is purely probabilistic and exhibits a probability measure \( \text{Pr}_\mathcal{S} \) for measurable sets of maximal \( \mathcal{S} \)-paths that start in state \( s \in S \) (for further details, see, e.g., [18]). These paths could for example we quantified using PCTL-formulae over the labels of the MDP. A classical reasoning task for MDPs is to decide whether a given property formulated as a PCTL formula holds for any or all possible schedulers. Amending \( M \) with a weight function \( \text{wgt} : Q \rightarrow \mathbb{Q} \) turns \( M \) into a weighted MDP. The weight of a finite path \( q_0, q_1, q_2, \ldots, q_n \) is defined as \( \text{wgt}(\pi) = \sum_{i=0}^{n} \text{wgt}(q_i) \). For a measurable path property (for example, a PCTL formula) \( \varphi \) and a scheduler \( \mathcal{S} \), we define by \( \text{Exp}_\mathcal{S}(\varphi) \) the expectation of the random variable which assigns to every maximal path that starts in \( q \in Q \) and satisfies \( \varphi \) its weight.

2.2 Stochastic Programs

As a concise representation of certain MDPs, we rely on a probabilistic variant of Dijkstra's guarded-command language [8, 14] that is compatible with the input language of Prism [17]. Intuitively, stochastic programs are concise representations of weighted MDPs with variable assignments as states, and arithmetic constraints on these variables as labels. We define these MDPs at the end of this section as the semantics of stochastic programs.

Throughout the section, we fix a set \( \text{Var} \) of countable set of variables, on which we define evaluations as functions \( \eta : \text{Var} \rightarrow \mathbb{Z} \). We denote the set of evaluations over \( \text{Var} \) by \( \text{Eval}(\text{Var}) \).

Arithmetic Constraints and Boolean Expressions. Let \( z \) range over \( \mathbb{Z} \) and \( v \) range over \( \text{Var} \). The set of arithmetic expressions \( \mathbb{E}(\text{Var}) \) 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 \( \eta(\alpha_1 \cdot \alpha_2) = \eta(\alpha_1) \cdot \eta(\alpha_2) \). \( \mathbb{C}(\text{Var}) \) denotes the set of arithmetic constraints over \( \text{Var} \), that is, terms of the form \( (\alpha \Leftrightarrow z) \) with \( \alpha \in \mathbb{E}(\text{Var}) \). For a given evaluation \( \eta \in \text{Eval}(\text{Var}) \) and constraint \( (\alpha \Leftrightarrow z) \in \mathbb{C}(\text{Var}) \), we write \( \eta \models \alpha \Leftrightarrow z \Leftrightarrow \eta(\alpha) \Leftrightarrow z \) and say that \( \alpha \Leftrightarrow z \) is entailed by \( \eta \). For a given evaluation \( \eta \in \text{Eval}(\text{Var}) \), we denote by \( \mathbb{C}(\eta) \) the constraints entailed by \( \eta \), that is, \( \mathbb{C}(\eta) = \{ c \in \mathbb{C}(\text{Var}) \mid \eta \models c \} \).

For a countable set \( X \) and \( x \) ranging over \( X \), we define Boolean expressions \( \mathbb{B}(X) \) over \( X \) by the grammar \( \phi ::= x \mid \neg \phi \mid \phi \land \phi \). Given a set \( X \), we define the satisfaction relation \( \models \subseteq \mathbb{E}(X) \times \mathbb{B}(X) \) in the usual way by \( X' \models x \) if \( x \in X ' \), \( X' \models \neg \psi \) if \( X' \not\models \psi \), and \( X' \models \psi_1 \land \psi_2 \) if \( X' \models \psi_1 \) and \( X' \models \psi_2 \). For an evaluation \( \eta \in \text{Eval}(\text{Var}) \) and \( \phi \in \mathbb{B}(\mathbb{C}(\text{Var})) \), we write \( \eta \models \phi \) if \( \phi \in \mathbb{B}(\mathbb{C}(\eta)) \).
Stochastic Programs. We call a function \( u: \text{Var} \to \mathbb{E}(\text{Var}) \) update and a distribution \( s \in \text{Distr}(\text{Upd}) \) stochastic update over a given finite set of updates \( \text{Upd} \). The effect of an update \( u: \text{Var} \to \mathbb{E}(\text{Var}) \) onto an evaluation \( \eta \in \text{Eval}(\text{Var}) \) is their composition \( \eta \circ u \in \text{Eval}(\text{Var}) \), that is, \( (\eta \circ u)(v) = \eta(u(v)) \) for all \( v \in \text{Var} \). This notion naturally extends to stochastic updates \( s \in \text{Distr}(\text{Upd}) \) by \( \eta \circ s \in \text{Distr}(\text{Eval}(\text{Var})) \) with \( (\eta \circ s)(\eta') = \sum_{u \in \text{Upd}, \eta \circ u = \eta'} s(u) \) for any \( \eta' \in \text{Eval}(\text{Var}) \). A guarded stochastic command over a finite set of updates \( \text{Upd} \), briefly called command, is a pair \( (g, s) \) where \( g \in \mathbb{B}(\text{C}(\text{Var})) \) is a guard and \( s \in \text{Distr}(\text{Upd}) \) is a stochastic update. Similarly, a weight assignment is a pair \( (g, \lambda) \) where \( g \in \mathbb{B}(\text{C}(\text{Var})) \) is a guard and \( \lambda \in \mathbb{Z} \) a weight. A stochastic program is a tuple \( P = \langle C, W, \eta_0, \text{Var} \rangle \) where \( C \) is a finite set of commands, \( W \) a finite set of weight assignments, and \( \eta_0 \in \text{Eval}(\text{Var}) \) is an initial evaluation. For simplicity, we write \( \text{Upd}(\text{P}) \) for the set of all updates in \( C \). The semantics of a stochastic program \( P = \langle C, W, \eta_0, \text{Var} \rangle \) is defined as the weighted MDP \( \mathcal{M}[P] = \langle S, \text{Act}, P, \eta_0, AP, \lambda, \text{wgt} \rangle \), where

1. \( S = \text{Eval}(\text{Var}) \),
2. \( \text{Act} = \text{Distr}(\text{Upd}(P)) \),
3. \( AP = \mathbb{B}(\text{C}(\text{Var})) \),
4. for all \( \eta \in S \), \( \lambda(\eta) = \mathbb{C}(\eta) \)
5. for any \( \eta, \eta' \in S \) and \( (g, s) \in C \) with \( \lambda(\eta) \models g \), we have \( P(\eta, s, \eta') = (\eta \circ s)(\eta') \), and
6. \( \text{wgt}: S \to \mathbb{Z} \), where \( \text{wgt}(\eta) = \sum_{(g, z) \in W, \lambda(\eta) = g} z \).

Entailment of PCTL formulae and weight expectations \( \text{Exp}^\lambda_{\eta}(\varphi) \) are defined for stochastic programs based on their semantics, that is, a stochastic program satisfies a PCTL formula \( \varphi \) over \( \mathbb{C}(\text{Var}) \) iff its induced MDP satisfies it, and it satisfies \( \text{Exp}^\lambda_{\eta}(\varphi) = x \) iff so does the induced MDP. Note that \( \mathcal{M}[P] \) is indeed a weighted MDP and that \( P(\eta, s) \) is a probability distribution with finite support for all \( \eta \in \text{Eval}(\text{Var}) \) and \( s \in \text{Distr}(\text{Upd}(P)) \).

2.3 Description Logics

We recall basic notions of description logics (DLs) (see, e.g., [2, 3] for more details) and introduce the elements of the DL \( \text{ALCQ} \) relevant to this paper. Let \( \mathbb{N}_c, \mathbb{N}_r \) and \( \mathbb{N}_i \) be pairwise disjoint countable sets of concept names, role names, and individual names, respectively. Based on these sets, we build complex concepts using the following constructors: of the particular DL. For \( A, r, \) and \( n \) ranging over \( \mathbb{N}_c, \mathbb{N}_r \), and \( \mathbb{N}_i \), respectively, \( \text{ALCQ concepts} \) are defined through the grammar

\[
C ::= \top | A | \neg C | C \sqcap C | C \sqcup C | \exists r.C | \forall r.C | \geq nr.C | \leq nr.C
\]

Concept inclusions (CIs) are 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 forms \( A(a) \) and \( r(a, b) \), where \( A \) is a concept, \( r \in \mathbb{N}_r \), and \( a, b \in \mathbb{N}_i \). CIs and assertions are commonly referred to as DL axioms, and we use \( \mathbb{A} \) to denote the set of all DL axioms. An ontology is a tuple \( \langle T, \mathcal{A} \rangle \), where \( T \) is a finite set of axioms, called TBox, and \( \mathcal{A} \) a finite set of assertions, called ABox. By abuse of notation, we often identify ontologies \( \langle T, \mathcal{A} \rangle \) with the sets \( T \cup \mathcal{A} \sqsubseteq \mathbb{A} \).

The semantics of DLs is defined in terms of interpretations, which are tuples \( \langle \Delta^T, \mathcal{I} \rangle \) with a domain \( \Delta^T \) and an interpretation function \( \mathcal{I} \) that maps any \( A \in \mathbb{N}_c \) to some \( A^T \subseteq \Delta^T \), any
r \in \mathbb{N}_r \text{ to some } r^T \subseteq \Delta^T \times \Delta^T, \text{ and any } a \in \mathbb{N}_t \text{ to some } a^T \subseteq \Delta^T. \text{ Interpretation functions are extended to complex concepts: }
\begin{align*}
\top^T &= \Delta^T \\
(-C)^T &= \Delta^T \setminus C^T \\
(C \cap D)^T &= C^T \cap D^T \\
(C \cup D)^T &= C^T \cup D^T \\
(\exists r.C)^T &= \{d \in \Delta^T \mid \exists (d, e) \in r^T : e \in C^T\} \\
(\forall r.C)^T &= \{d \in \Delta^T \mid \forall (d, e) \in r^T : e \in C^T\} \\
(\geq nr.C)^T &= \{d \in \Delta^T \mid \#\{(d, e) \in r^T \mid e \in C^T\} \geq n\} \\
(\leq nr.C)^T &= \{d \in \Delta^T \mid \#\{(d, e) \in r^T \mid e \in C^T\} \leq n\}.
\end{align*}

An interpretation I satisfies an axiom \( C \sqsubseteq D \) iff \( C^T \subseteq D^T \), an assertion of the form \( A(a) \) iff \( a^T \in A^T \), and an assertion of the form \( r(a, b) \) iff \( \langle a^T, b^T \rangle \in r^T \). Let us fix an ontology \( O \). We call I a model of \( O \) iff I satisfies every axiom and assertion in \( O \). If there is a model of \( O \), then \( O \) is called consistent (and inconsistent otherwise). \( O \) entails an axiom or assertion \( \alpha \), in symbols \( O \models \alpha \), if \( I \models \alpha \) for every model I of \( O \). Note that an inconsistent ontology entails every axiom and assertion.

3 Contextualized Programs

Our understanding of contextualized programs follows the concept of separating concerns, complementing standard unified approaches towards context-aware systems such as context-oriented programming [12]. In general, a contextualized program comprises the following three components.

The Program is a specification of the operational behavior of the program, which may use placeholder labels to refer to specific situations that depend on the context.

The Context specifies detailed knowledge about the context in which the program is executed.

The Interface links both elements together, by providing a mapping from the placeholder labels to situation descriptions in the context language, from context vocabulary to corresponding expressions in the program language.

The main idea is that the program specification focuses on the operational behavior, abstracting away from system details that may depend on its context. The context specification provides further information relevant to a specific context in which this operational behaviour is applied, which, together with the interface, results in a full description of the systems behaviour in that context. We advocate the use of different formalisms for the program and context specifications best suited for the task at hand: the program should be specified with a formalism that is best suited for modelling an analysing dynamic systems, while the context should be specified with a formalism best suited for describing and reasoning about complex systems.

This generic idea of contextualized programs can be applied to many kinds of program and context specifications. To illustrate our approach, we focus on contextualized programs in which the operational behavior is specified using a stochastic program extended with placeholder labels, and the context is specified using a DL.

3.1 Contextualizing Stochastic Programs with Ontologies

The program specification of a contextualized program uses place holders to refer to situations specific to the context. To be able to represent these, we fix a set \( \Lambda \) of placeholder labels, labels
for short, and define *abstract stochastic programs* as stochastic programs in which the guards used in the guarded commands and in the weights are elements of \( B(C \cup \Lambda) \), that is, Boolean expressions over arithmetic expressions and labels. For an abstract stochastic program \( P \), we denote by \( \Lambda(P) \) the set of labels appearing in guards of \( P \).

**Definition 1.** A *contextualized stochastic program* is a tuple \( S = (P, C, I) \) where

- \( P = (C, W, \eta_0, \text{Var}_P) \) is an abstract stochastic program,
- \( C \) is an ontology describing the context,
- \( I = (i_q, i_p, \Lambda, B, \text{Var}) \) is a contextual interface over placeholder labels \( \Lambda \), DL axioms \( B \), and variables \( \text{Var} \) such that \( i_q: \Lambda \to \wp(\Lambda) \) maps each label to a set of DL axioms (which do not necessarily have to come from \( B \)), and \( i_p: B \to B(C(\text{Var})) \) maps axioms in \( B \) to Boolean expressions over variables.

and for which we require that \( I \) is *compatible with* \( P \) in the sense that \( \Lambda(P) \subseteq \Lambda \) and \( \text{Var} \subseteq \text{Var}_P \).

The *abstract program* \( P \) describes the operational behavior of the system, where placeholder labels might be used in guards and thus affect the program flow. From the view of the program, each state of the system is identified via variable assignments and context labels. The context specification \( C \) provides for a detailed description of the current system under investigation, without any knowledge about its operational behaviour. From the view of the context, each state of the system is identified via a set of DL axioms from \( B \), translated from the program language via the interface \( i_p \), and from which in addition with the axioms in \( C \) additional inferences can be made. This is in turn relevant for the mapping \( i_q \) which translates the DL view of a state into the label view of the program: in order to decide whether a placeholder label \( L \in \Lambda \) should be assigned to a system state, we have to determine whether the DL axioms of that state entail \( i_q(L) \). \( C \) thus contains a set of DL axioms that can be seen as *static*, as capture the static, time-independent knowledge about the context, while \( B \) contains additional DL axioms which can be seen as *dynamic*, as they are used to reflect the changing system states of the program.

Note that we specifically allow the interface to map also labels which do not occur in the program: these labels might describe situations that are only relevant for the analysis of the contextualized program, so that the same properties can be evaluated for different contexts. For example, we might have a label describing unfortunate system states, and want to compute the probability of getting into such a state for different contexts.

Compatibility of program and context through the interface is a crucial aspect within contextualized programs. Note that compatibility is provided through syntactic criteria. Despite the compatibility constraints given by the interface, program and context are defined independently from each other, implementing the concept of separating concerns.

### 3.2 Semantics

Similar as for stochastic programs, contextualized programs can be seen as concise representations of MDPs, which establish the semantics of the contextualized program. Properties of the system described by a contextualized program are defined with respect to these MDPs.

From the perspective of the program \( P \), a system state is characterized by an evaluation over \( \text{Var} \) and a set of labels from \( \Lambda(P) \), and from the perspective of the context, each system state
corresponds to a set \( C \cup B' \), where \( B' \subseteq B \). To formalize this intuitive semantical understanding of contextualized programs, we present a weighted MDP semantics similar to the one of stochastic programs. While for stochastic programs a state in the MDP only comprises a variable evaluation, for contextualized programs a state \( q \) additionally contains a state ontology \( O_q \).

**Definition 2.** A contextualized state is a tuple of the form \( q = \langle \eta_q, O_q \rangle \), where \( \eta_q \) is an evaluation and \( O_q \) an ontology. Let \( S \) be a contextualized program as in Definition 1. A contextualized state \( q \) conforms to \( S \) if

1. \( O_q \subseteq C \cup B \),
2. \( C \subseteq O_q \), and
3. for every \( \alpha \in B \), \( \alpha \in O_q \) implies \( \eta_q \models i_p(\alpha) \).

Note that \( S \) determines a unique extension of every evaluation \( \eta \) to a contextualized state \( e(S, \eta) \), for which we have \( \eta_{e(S, \eta)} = \eta \) and \( O_{e(S, \eta)} = C \cup \{ \alpha \in B \mid \eta \models i_p(\alpha) \} \). We define the result of applying an update \( u \) on a contextualized state \( q \) as \( u(q) = e(S, u(\eta_q)) \). Note that this corresponds to just applying the update on the evaluation \( \eta_q \) of \( q \), and then computing its unique extension \( e(u(\eta)) \) to a contextualized state.

We are now ready to provide the weighted MDP semantics of contextualized stochastic programs.

**Definition 3.** Let \( S = \langle S, C, I \rangle \) be a contextualized stochastic program as in Definition 1. The MDP induced by \( S \) is defined as

\[ M(S) = \langle Q, Act, q_0, AP, \lambda, wgt \rangle \]

where

- \( Q = \{ e(S, \eta) \mid \eta \in Eval(Var_P) \} \),
- \( Act = Dist(Upd(P)) \),
- \( q_0 = e(S, \eta_0) \),
- \( AP = \Lambda \cup C(Var(P)) \),
- for every \( q \in Q \), \( \lambda(q) = C(\eta_q) \cup \{ \ell \in \Lambda \mid O_q \models i_q(\ell) \} \),
- for every \( q_1, q_2 \in Q \) and every \( \langle g, s \rangle \in C \) with \( \lambda(q_1) \models g \), we have \( P(q_1, s, q_2) = (\eta_{q_2} \circ s)(\eta_{q_2}) \), and
- the weight function \( wgt \) is such that for every \( q \in Q \), we have

\[ wgt(q) = \sum_{(g,z) \in W, \lambda(q) \models g} z. \]

The above definition closely follows the standard semantics for stochastic programs, while amending contextual information to each state in a natural way. Note that the labeling function \( \lambda \) now assigns not only arithmetic constraints, but also labels to each state, which are determined using the interface \( I \) and the ontology \( O_q \) assigned to that state. Because of this, the definition of the probabilistic transition function \( P \) and of the weight function \( wgt \) is in fact almost identical as for MDPs induced by stochastic programs (see Section 2.2).
Remark. Note that it is possible that the MDP induced by a contextualized program can have states whose corresponding ontologies are logically inconsistent. Since by an inconsistent ontology, every axiom is entailed, this consequently means that such states are labelled with all placeholder labels present in the interface. This can be seen as a semantic type of incompatibility of the context with the program that cannot be captured by the syntactic compatibility criterion discussed in Section 3.1. There are different ways in which one could handle such a situation. First, the interface could define a placeholder label $\ell_\bot \in A$ which maps to a contradiction, for example by setting $i_q(\ell_\bot) = (\top \subseteq \bot)$, so that the abstract program can specify an appropriate behaviour for inconsistent states. A disadvantage of this approach is that, without knowledge about the context, the program cannot know how to resolve the inconsistency, so that it is not clear how such a placeholder label could be used in practice. A more refined approach would be to define the semantics of the contextualized program differently, and in such a way that an inconsistent state is never reached. This could for example be done by removing inconsistent states from the MDP, and normalizing all distributions accordingly. Such an approach would necessarily lead to an alteration of the operational behavior, which could have unintended side-effects that the designer of the program did not consider. A third solution would be to provide the user with tool support for deciding whether a given contextualized program can enter an inconsistent state, to point out when a specific context is incompatible with the program specification. This could for example be done using a reachability analysis on the states of the induced MDP. The modeller of the contextualized program could then use this information and for example introduce additional placeholder labels in the interface to allow the program to avoid such situations. A full investigation of all these possibilities is beyond the scope of this paper, which is why we leave them for future work. The contextualized program used in the evaluation and in the running examples however uses such labels to avoid the entering of an inconsistent state.

3.3 Example

We illustrate our formalism by presenting a contextualized stochastic program that follows our motivating example from the introduction, that is, a multi-server system for which we consider instances running $n$ processes on $m$ servers. In the following, we use the same notations as in Definition 1.

Program. The stochastic program $P$ specifies the protocol how processes are scheduled to complete their jobs when running on the same server and how context-dependent migration of processes to other servers is performed. We consider two variants of scheduling the processes on a common server: randomized or round robin. In the randomized case, every process is selected with the same probability, while in the round-robin case the processes are ordered in a queue where in each step the first process is selected and then put at the end of the queue. Each process is assigned an id $i \in [1..m]$, and each server is assigned an id $i \in [1..n]$. For a process with ID $i$ use the variables $\text{server}_\text{proc}i$ to specify which server it is currently running on: e.g., $\text{server}_\text{proc}2 = 3$ means that Process 2 runs on Server 3. We use $\text{server}_\text{proc}i = 0$ to specify that a process is currently idling. We further use variables $\text{sel}_j$ for $j \in [1..m]$ those values range from 1 to $n$ providing the process those job server $j$ will compute in the next step (following the randomized or round-robin scheme). Also here, $\text{sel}_j = 0$ for $j \in [1..m]$ in case no process is active on server $j$. The following guarded stochastic command $\langle g, s \rangle$ in $P$ formalizes the randomized selection of a process on server 2 for $n = 3$, for the case that Processes 1 and 3 are active on Server 2.

\[
g \ = \ \text{server}_\text{proc}1 = 2 \ \land \ \neg\text{server}_\text{proc}2 = 2 \ \land \ \text{server}_\text{proc}3 = 2
\]

\[
s \ : \ s(\text{sel}_2 \mapsto 1) = 1/2 \ \text{and} \ s(\text{sel}_2 \mapsto 3) = 1/2
\]

Here, the stochastic update $s$ is provided by a distribution over updates where $v \mapsto \alpha$ stands for a variable $v \in \text{Var}$ updated by the expression $\alpha \in \mathbb{E}(\text{Var})$. Specifically, provided that the
guard is satisfied in the current program state, the system will either select Server 2 or Server 3 to perform the next step for Process 2, each with a probability of 50%.

In our simplified setting, we use the placeholder labels

\[ \Lambda_P = \{ \text{migrate, critical} \} \cup \{ \text{incons}_{i\text{on}j} \mid i \in [1..n], j \in [1..m] \} \]

to describe situations in which i) the system should consider migrating a process, ii) the system is in a critical situation that should be avoided, and iii) running Process \( i \) on Server \( j \) would lead to an inconsistent system state.

For instance, this can be used by the guarded stochastic command \( \langle \gamma', s' \rangle \) in \( P \) for a migration of Process 2 to Server 1, where

\[ \begin{align*}
\gamma' &= \text{migrate} \land \neg(\text{server}_\text{proc2} = 1) \land \neg\text{incons}_2\text{on}1 \\
\sigma' &= s(\text{server}_\text{proc2} \mapsto 1) = 1.
\end{align*} \]

This guarded command moves Process 2 to Server 1 with a probability of 1, provided that we are in a migrate situation, the process is not already on Server 1, and moving the process there is consistent with the context.

We consider different variants of of weight assignments, which we select based on the analysis we are interested in. These weight assignments represent consumed energy, the number of critical situations entered, and a notion of utility, represented by weight annotations \( W_{\text{egy}} \), \( W_{\text{crit}} \), and \( W_{\text{util}} \), respectively. For instance, each server requires one energy unit when active and one additional energy unit is consumed for each migration operation. This is captured by the following set of weight assignments:

\[ W_{\text{egy}} = \{ \langle \text{migrate}, 1 \rangle, \langle \text{sel}_j \neq 0, 1 \rangle : j \in [1..m] \}. \]

For counting critical situations we simply define \( W_{\text{crit}} = \{ \langle \text{critical}, 1 \rangle \} \), and to calculate the utility we assign one utility unit when completing a standard job and two utility units when completing a prioritized job. Note that all these weight annotations are depending on the context and can be provided as within standard commands using placeholder labels in the weight definitions.

**Context.** The context ontology \( C \) now specifies additional knowledge about the context. We use individuals \( \text{Procs} = \{ \text{proc}_1, \ldots, \text{proc}_n \} \) to denote the different processes and \( \text{Servers} = \{ \text{server}_1, \ldots, \text{server}_m \} \) to denote the servers. For instance, the operating system \( \text{os} \in \{ \square, \Delta \} \) the servers are running and processes are compiled for are specified. The former is done through assertions

\[ \text{runsOS}(\text{server}_i, \text{os}), \quad 1 \leq i \leq m, \]

the latter is formalized by assigning class concepts \( \square \text{Process} \) and \( \Delta \text{Process} \) to processes. Furthermore, a process might have higher priority than others, formalized by assigning the class concept \( \text{Priority} \) to the respective process. The complete system is represented by an object system, which connects to the servers via \( \text{hasServer}(\text{system}, \text{server}_i), 1 \leq i \leq m. \)

\( C \) furthermore contains TBox axioms that put constraints on the system and define concepts relevant for the placeholder labels.

\[ \begin{align*}
\Delta \text{ Server} &
\square \forall \text{runsProcess}. \Delta \text{ Process} \\
\square \text{Server} &
\square \forall \text{hasSystem}. \forall \text{runsProcess}. \square \text{Process} \\
\text{BusyServer} &
\square \geq \text{runsProcess}. \text{Process} \\
\text{ShouldMigrate} &
\square \text{BusyServer} \land \exists \text{runsProcess}. \text{Priority} \\
\text{Migrate} &
\exists \text{hasServer}. \text{ShouldMigrate}
\end{align*} \]
**Interface.** The interface between $\mathbf{P}$ and $\mathbf{C}$ specifies the connection between the “visible” elements of $\mathbf{P}$ in terms of $\mathbf{Var}$ and their representations in $\mathbf{DL}$ as well as a definition of the situation labels in terms of $\mathbf{DL}$ axioms.

The set $\mathbf{B}$ of dynamic axioms provides the connection of processes running on servers.

$$O_p = \{ \text{runsOn}(\text{proc}_i, \text{server}_j) \mid \text{proc}_i \in \text{Procs}, \text{server}_j \in \text{Servers} \}$$

As becomes clear from the program, these are the only aspects of the system that will be changed by the program, so that no further axioms are required for $\mathbf{B}$. The mapping $i_p$ maps the dynamic axioms in $\mathbf{B}$ to conditions on the systems state. In our case, this mapping is provided by

$$i_p(\text{runsOn}(\text{proc}_i, \text{server}_j)) = (\text{server}_\text{proc}_i = j),$$

for $1 \leq i \leq n, 1 \leq j \leq m$.

The mapping $i_p$ maps labels used by the stochastic program to $\mathbf{DL}$ axioms. For instance, the label migrate would be mapped to Migrate(system), which is entailed by the current state’s ontology whenever there is a server which runs at least 4 processes, one of which is a priority process.

### 4 Analysis of Contextualized Stochastic Programs

In order to analyze contextualized programs practically, we make use of a PMC tool such as PRISM, in connection with a $\mathbf{DL}$ reasoner. The naive method for analyzing contextualized programs would be to construct the MDP induced by the program directly, based on Definition 3, where we use a $\mathbf{DL}$ reasoner to decide axiom entailment to determine which labels to assign to a state. However, the set of states of such an MDP can in general be exponential in the size of the program, which means this approach would require us to test for entailment from $\mathbf{DL}$ axioms an exponential number of times, and deciding entailment of $\mathbf{DL}$ axioms is computationally very hard: it ranges from $\text{ExpTime-N2ExpTime}$ for expressive fragments of the $\mathbf{DL}$ underlying the $\text{OWL}$ standard. PMC tools such as PRISM use advanced techniques to reduce the amount of states that have to be explicitly computed, but a tight integration of $\mathbf{DL}$ reasoning into a PMC tool would prove challenging. Instead, we propose a solution that is independent of the PMC tool, and also independent of the ontology language. The main idea is to provide for a translation of the contextualized programs into stochastic programs, so that existing PMC tools can be used for the analysis. A main challenge is to reduce the amount of $\mathbf{DL}$ reasoning necessary by performing this translation in a goal-oriented manner.

To formalise this idea, we define a translation $t$ from contextualized programs $\mathbf{S}$ into ground stochastic programs $\mathbf{t(S)}$, based on an assignment $\mathbf{sf} : \Lambda \rightarrow \mathbb{B}C \mathbf{Var}$ of labels $\ell \in \Lambda$ to corresponding label formulae $\mathbf{sf}(\ell)$, such that the MDPs induced by $\mathbf{S}$ and by $\mathbf{t(S)}$ correspond to each other with respect to their labelings. This correspondence is captured in the following definition.

**Definition 4.** Let $\mathcal{M}_1 = (S_1, \text{Act}_1, P_1, s_0, AP_1, \lambda_1, \text{wgt}_1)$ and $\mathcal{M}_2 = (S_2, \text{Act}_2, P_2, s_0, AP_2, \lambda_2, \text{wgt}_2)$ be two weighted MPDs s.t. $\text{Act}_1 = \text{Act}_2$, and $\mathbf{sf} : \lambda_1 \rightarrow \mathbb{B}(\lambda_2)$ be a partial function mapping labels in $AP_1$ to formulae over $AP_2$. We say $\mathcal{M}_1$ and $\mathcal{M}_2$ are equivalent modulo $\mathbf{sf}$ iff there exists a bijection $b : S_1 \leftrightarrow S_2$ such that

1. $b(s_0) = s_0$,
2. for every $s_1, s_2 \in S_1$ and $a \in \text{Act}$, $P(s_1, a, s_2)$ is defined iff $P(b(s_1), a, b(s_2))$ is defined, and $P(s_1, a, s_2) = P(b(s_1), a, b(s_2))$. 

11
3. for every \( s \in S_1, \text{wgt}_1(s) = \text{wgt}_2(b(s)) \), and
4. for every \( \ell \in AP_1 \) and \( s \in S_1, \ell \in \lambda_1(s) \) iff \( \lambda_2(b(s)) \models \text{sf}(\ell) \).

This notion naturally extends to stochastic programs and contextualized programs via their induced MDPs. Specifically, we say a contextualized program \( S \) and a stochastic program \( P \) are equivalent modulo \( \text{sf} \) iff \( M[S] \) and \( M[P] \) are equivalent modulo \( \text{sf} \).

If a contextualized program \( S \) is equivalent modulo \( \text{sf} \) to a stochastic program \( P \), all analysis tasks on \( S \) can be reduced to analysis on \( P \); specifically, we can reduce entailment of PCTL properties \( \phi \) to entailment of PCTL properties \( \phi' \) in which each label \( \ell \) in the contextualized program is replaced by \( \text{sf}(\ell) \) (in this case: placeholder labels and evaluations), and similarly, we can compute expectation values \( \text{Exp}^S_\pi(\varphi) \). As a result, we can easily use PMC tools that support stochastic programs to evaluate contextualized programs, provided the function \( \text{sf} \) and the corresponding stochastic program can be computed.

We define a function \( \text{sf} \) that can be efficiently computed using DL reasoning, which can also be used to compute the corresponding stochastic program equivalent to the contextualized program. Specifically, for every constraint \( c \in C(\text{Var}) \), we set \( \text{sf}(c) := c \), so that we only need to provide for formulae for the placeholder labels. We then translate the contextualized program \( S \) into a stochastic program \( t(S) \) equivalent to \( S \) modulo \( \text{sf} \) by just taking the abstract stochastic program in \( S \) and replacing all placeholder labels \( \ell \in \Lambda \) by \( \text{sf}(\ell) \). This is sufficient, since by our semantics, the labels assigned to a contextualized state are fully determined by the evaluation of the state: which axioms are part of the state is determined by the mapping \( i_p : B \rightarrow B(\text{C(Var)}) \), and which labels are part of the state is determined by using the mapping \( i_q \), based on which axioms are entailed by the ontology assigned to the state. For a label \( \ell \in \Lambda \), the following formula would therefore fully quantify the contextualized states that have this label, by sole reference to the variable evaluation \( \eta_q \) in that state:

\[
sf'(\ell) := \bigvee_{B' \subseteq B, C \cup B \models i_q(\ell)} \bigwedge_{\alpha \in B} i_p(\alpha). \tag{1}
\]

It is thus easy to see that for every contextualized program \( S \), we can find a function \( \text{sf} \) as required to translate \( S \) into a stochastic program equivalent to \( S \) modulo \( \text{sf} \).

While Equation 1 shows that a function \( \text{sf} \) as required can always be found, computing \( \text{sf}'(\ell) \) for every \( \ell \in \Lambda \) would be too expensive to allow for a practical implementation. Note that for computing this formula, we would need to perform an entailment check for every subset of \( B \), and entailment from DL ontologies is expensive. To compute label formulae in a more goal-oriented manner, we make use of a technique called axiom pinpointing. This technique is defined independently of the DL in question, and there exists standard-tools for various DLs that can be used for it. Axiom pinpointing is concerned with the computation of justifications defined next.

**Definition 5.** Given an ontology \( O \) and a query \( \alpha \) s.t. \( O \models \alpha \), a subset \( J \subseteq O \) is a justification of \( O \models \alpha \) iff \( J \models \alpha \), and for every \( J' \subseteq J \), \( J' \not\models \alpha \). We denote by \( J(O, \alpha) \) the set of all justifications of \( J \models \alpha \).

Justifications were originally developed for ontology-debugging in order to understand entailments of an ontology [15]. Importantly, there exist efficient implementations for axiom pinpointing for various DLs. For example, the OWL API implements a black box algorithm that can be used in connection with any OWL reasoner that supports the OWL API [13]. The reasoner Pellet, which supports a large fragment of OWL DL, furthermore supports efficient computation of justifications using a glass box approach [19].
Intuitively, a justification is a minimal sufficient witness for the entailment of an ontology. For the label formula \( \text{sf}(\ell) \), it is sufficient to consider the justifications \( \mathcal{J} \) of \( O_c \cup O_p \models i_q(\ell) \), as these characterize exactly those subsets \( O' \subseteq O_c \) s.t. \( O_c \cup O' \models i_q(\ell) \). Note that for each such justification \( \mathcal{J} \), only the subset \( \mathcal{J} \setminus O_c \) is relevant. We thus define the situation formula \( \text{sf}(\ell) \) for \( \ell \in \Lambda \) as

\[
\text{sf}(\ell) = \bigvee_{\mathcal{J} \in \mathcal{J}(i_q(\ell),C)} \bigwedge_{a \in \mathcal{J} \setminus O_c} i_p(\alpha).
\]  

(2)

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 \( t(S) \) of the contextualised program \( S = (P, C, I) \) is now the ground stochastic program obtained from \( P \) by replacing every label \( \ell \in \Lambda \) by \( \text{sf}(\ell) \). We have the following theorem.

**Theorem 6.** The contextualized program \( S \) and the stochastic program \( t(S) \) are equivalent modulo \( \text{sf} \).

**Proof.** We take the MDP \( M_1 = M[t(S)] \) induced by the translated program \( t(S) \) and extend it to the MDP \( M_2 = M[S] \) induced by the contextualized program \( S \), such that there exists a bijection as in Definition 4. Specifically, based on the MDP

\[
M_1 = M[t(S)] = (\text{Eval(Var)}, \text{Distr(Upd)}, P, \eta_0, C(Var), \lambda, \text{wgt})
\]

induced by \( t(S) \), we define the MDP

\[
M_2 = (Q', \text{Distr(Upd)}, P', e_S(S, \eta_0), \Lambda \cup C(Var), \lambda', \text{wgt}')
\]

where

- \( Q' = \{e(S, \eta) \mid \eta \in \text{Eval(Var)}\} \) ,
- for every \( \eta_1, \eta_2 \in \text{Eval(Var)} \) and \( d \in \text{Distr(Upd)} \) for which \( P(\eta_1, d, \eta_2) \) is defined, we set \( P'(e(S, \eta_1), d, e(S, \eta_2)) = P(\eta_1, d, \eta_2) \),
- for every \( q \in Q' \), \( \text{wgt}'(q) = \text{wgt}(\eta_q) \), and
- for every \( q \in Q' \), \( \lambda'(q) = \lambda(\eta_q) \cup \{\ell \in \Lambda \mid O_q \models i_q(\ell)\} \).

The bijection \( b : Q' \leftrightarrow \text{Eval(Var)} \) 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 4, so that \( M_1 \) and \( M_2 \) are equivalent modulo \( \text{sf} \). It remains to show that \( M_2 \) is really the MDP induced by \( S \). We start with the following claim.

**Claim 1.** For every \( \ell \in \Lambda \) and \( q \in Q' \), \( \lambda'(q) \models \ell \) iff \( \lambda(q) \models \text{sf}(\ell) \).

**Proof of claim.** Let \( q \in Q' \), and assume \( \lambda'(q) \models \ell \). By construction of \( M_2 \), then \( O_q \models i_q(\ell) \), which means there is a justification \( \mathcal{J} \) for \( O_q \models i_q(\ell) \). Since \( O_q \subseteq C \cup B \), \( \mathcal{J} \) is also a justification for \( C \cup B \models i_q(L) \). Furthermore, by Definition 2, for every \( \alpha \in O_q \setminus C \), \( \eta_q \models i_p(\alpha) \). It follows that \( \eta_q \models \bigwedge_{\alpha \in \mathcal{J} \setminus C} i_p(\alpha) \), and consequently, that \( \eta_q \models \text{sf}(\ell) \) and \( \lambda'(q) \models \text{sf}(\ell) \).

Conversely, assume \( \lambda'(q) \models \text{sf}(\ell) \). Then, by construction of \( \text{sf} \), there is a justification \( \mathcal{J} \) for \( C \cup B \models i_q(\ell) \) s.t. \( \eta_q \models \bigwedge_{\alpha \in \mathcal{J} \setminus C} i_p(\alpha) \). Since \( q \) is a contextualized state, by Definition 2, for
every $\alpha \in \mathcal{J} \setminus \mathcal{C}$, $\eta_q \models i_p(\alpha)$ iff $O_q \models \alpha$. Since also $\mathcal{C} \subseteq O_q$, we obtain that $\mathcal{J} \subseteq O_q$, and since $\mathcal{J} \models i_q(\ell)$, that $O_q \models i_q(\ell)$. Now, by construction of $\mathcal{M}_2$, we obtain that $\ell \in \lambda'(q)$, and consequently that $\lambda'(q) \models \ell$. □

As a consequence of Claim 1, for every Boolean formula $\phi \in \mathbb{B}(\text{Eval}(\text{Var}) \cup \Lambda)$ and $q \in Q'$, $\lambda'(q) \models \phi$ iff $\lambda(b(q)) \models \text{sf}(\phi)$, where $\text{sf}(\phi)$ denotes the result of replacing in $\phi$ every label $\ell \in \Lambda$ by $\text{sf}(\ell)$. We obtain that for every guarded command $\langle g, s \rangle \in C$, where $C$ is the set of guarded commands of the contextualized program $S$, and for every $q \in Q'$, $\lambda'(q) \models \phi$ iff $\lambda'(q_{g, z}) \models \text{sf}(g)$, which means that $P'$ satisfies the condition in Definition 3. Similarly, for $W_1$ the set of weight assignments in $t(S)$, and $W_2$ the set of weight assignments in $S$, we obtain that for every $q \in Q'$,

$$wgt'(q) = wgt(q_{g, z}) = \sum_{(g, z) \in W_1} \sum_{\lambda(q_{g, z}) = \text{sf}(g)} z,$$

which means that also $wgt'$ satisfies the conditions in Definition 3. Hence, $\mathcal{M}_2$ is indeed the MDP induced by $S$. We obtain that the MDP induced by $t(S)$ and the MDP induced by $S$ are equivalent modulo $\text{sf}$, and consequently, that $t(S)$ and $S$ are equivalent modulo $\text{sf}$. □

## 5 Evaluation

We implemented the method described in Section 4 and used it in a tool chain that takes as input a contextualized program, where the abstract stochastic is represented using the input language of the prominent model checker PRISM, and all axiom sets are represented as ontologies in the standard web ontology language OWL. Since the PRISM supports macro definitions, the label assignments provided by the computed function $\text{sf}$ 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 $\text{sf}$, we implemented the method described in Section 4 in Java using the OWL-API [13], where we used the reasoner Pellet [19] 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 2 specifying the situation formula, we only interested in the intersection of the full justification with the set $\mathcal{B}$ of the atoms the program can actually change. Usually, the algorithm computing all justifications would consider all subsets of $\mathcal{C} \subseteq \mathcal{B}$, ignoring the separation into $\mathcal{C}$ and $\text{dynamicAxioms}$. We therefore modified the algorithm so that it ignores axioms in $\mathcal{C}$ when comparing with earlier solution, which reduces the search space a lot of the set of axioms in $\mathcal{C}$ is large. Apart from this optimization, we computed the situation formulae 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 section. Although embedded in a generic tool chain where an arbitrary number of servers, their operating systems, processes and durations of tasks can be investigated, we concentrate here on two particular scenarios. The first comprises one $\text{server}$ and two $\text{Server}$ on which six processes are running, while the second comprises one $\text{server}$ and one $\text{Server}$ 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 program and program specifications for each of the scenarios, we implemented two...
different program specifications and four different contexts. The program specifications are
provided by Prism code implementing a randomized or round-robin strategy for selecting next
jobs to be executed.

We implemented four different contexts, which mostly follow the general idea as in Section 3.3,
but differ in the specification of

- **critical situations** specifying situations that should be avoided,
- **migrate situations** specifying situations in which a server should consider migrating pro-
cesses to another server, and
- **consistency situations** specifying when it is allowed for a specific process to be moved to
  a specific server.

These situations are guided by the concepts OverloadedServer and AlmostOverloadedServer,
which based on the number of running processes specify when a Server is overloaded or al-
most overloaded, respectively.

Specifically, the contexts specify these as follows.

**Context1: Critical Situation** A prioritized process runs on an overloaded server.

- **Migrate Situation** A server is overloaded, or it is almost overloaded and runs a priori-
tized process
- **Migrating Inconsistent** the maximum number of processes of the server is reached

**Context2: Critical Situation** A prioritized process runs on an overloaded server.

- **Migrate Situation** A server is overloaded, or it is almost overloaded and runs a priori-
tized process
- **Migrating Inconsistent** the maximum number of processes of the server is reached, or
  the operating system of the server is incompatible with the operating system of the
  process

**Context3: Critical Situation** A prioritized process runs on an overloaded server.

- **Migrate Situation** A server is almost overloaded and runs a prioritized process
- **Migrating Inconsistent** the maximum number of processes of the server is reached, or
  the operating system of the server is incompatible with the operating system of the
  process

**Context4: Critical Situation** A prioritized process runs on an overloaded server, or two
prioritized processes run on the same server

- **Migrate Situation** A server is overloaded
- **Migrating Inconsistent** the maximum number of processes of the server is reached, or
  the operating system of the server is incompatible with the operating system of the
  process

Note how these contexts provide for a convenient way of not only specifying the specifics of a
given context of the migration 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 16 contextualized programs in total, which we
translated into stochastic programs expressed in the input language of Prism.
5.2 Energy-aware Analysis

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

(1) What is the probability for reaching a critical situation 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 situations before reaching 20 utility?

For each of these properties, we computed the minimal and maximal probabilities determined by resolving the non-deterministic choices of the MDP semantics 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 situation can be seen as costs. For instance, it might be favorable to migrate a priority processes to a different server spending additional energy for increasing the completion rate and gaining more utility. To investigate this trade-off, we considered the following energy-utility quantiles [5].

(ey) 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 critical situations within which at least 20 utility is gained with probability at least 95%?

In the following Table 1, the analysis results of the Reachability Properties (1)-(3) and quantiles are shown. First, we considered the system configuration with two $\triangle$-servers and one $\square$-server running six processes. One can observe that the different contexts have great impact on the results. As two $\triangle$-servers are at hand, $\triangle$-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 $\triangle$-server and one $\square$-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 2-4. Furthermore, the probability of entering a critical situation 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). 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 context definitions in a further refinement step. In all scenarios and contexts we see that a round robin job-selection strategy is superior to the randomized one when the objective concerns minimizing energy consumption, critical situations and their trade-off properties. All the experiments were carried out using the symbolic MTBDD engine of PRISM in the version presented in [16]. The run-time statistics are shown in Table 2, showing a great impact of scenarios, contexts, and program variants on the model characteristics. Noticeable, when different operating systems are not modeled within the context, 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 context-aware analysis using context information 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.

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

<table>
<thead>
<tr>
<th>server/</th>
<th>proc</th>
<th>context</th>
<th>program</th>
<th>prob. crit (1)</th>
<th>exp. egy (2)</th>
<th>exp. crit (3)</th>
<th>quantile</th>
<th>egy</th>
<th>crit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>min</td>
<td>max</td>
<td>min</td>
<td>max</td>
<td>min</td>
<td>max</td>
</tr>
<tr>
<td>/6</td>
<td>1</td>
<td>random</td>
<td>0.6604</td>
<td>0.9983</td>
<td>20.09</td>
<td>58.57</td>
<td>1.73</td>
<td>28.38</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>roundr</td>
<td>0.6392</td>
<td>0.9983</td>
<td>27.92</td>
<td>62.25</td>
<td>1.33</td>
<td>28.72</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>random</td>
<td>0.4558</td>
<td>0.9866</td>
<td>29.17</td>
<td>47.63</td>
<td>0.91</td>
<td>14.05</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>roundr</td>
<td>0.4256</td>
<td>0.9854</td>
<td>27.96</td>
<td>48.17</td>
<td>0.78</td>
<td>13.62</td>
<td>31</td>
</tr>
<tr>
<td>/8</td>
<td>3</td>
<td>random</td>
<td>0.7776</td>
<td>0.7776</td>
<td>29.26</td>
<td>32.04</td>
<td>3.37</td>
<td>7.97</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>roundr</td>
<td>0.7612</td>
<td>0.7612</td>
<td>28.99</td>
<td>31.61</td>
<td>3.06</td>
<td>6.50</td>
<td>32</td>
</tr>
</tbody>
</table>

Table 2: Statistics of the analysis

<table>
<thead>
<tr>
<th>server/</th>
<th>proc</th>
<th>context</th>
<th>program</th>
<th>states</th>
<th>nodes</th>
<th>analysis time [s]</th>
<th>quantiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>/6</td>
<td>1</td>
<td>random</td>
<td>23'072'910</td>
<td>173'841</td>
<td>2'011.52</td>
<td>2'308.40</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>roundr</td>
<td>57'231'023</td>
<td>3'718'247</td>
<td>18'759.57</td>
<td>47'598.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>random</td>
<td>800'814</td>
<td>16'782</td>
<td>164.18</td>
<td>236.93</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>roundr</td>
<td>900'814</td>
<td>15'666</td>
<td>141.45</td>
<td>208.63</td>
<td></td>
</tr>
<tr>
<td>/8</td>
<td>3</td>
<td>random</td>
<td>773'398</td>
<td>157'69</td>
<td>117.88</td>
<td>161.77</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>roundr</td>
<td>425'306</td>
<td>447'244</td>
<td>126.69</td>
<td>156.91</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>random</td>
<td>900'278'882</td>
<td>134'648</td>
<td>4'662.57</td>
<td>18'268.61</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>roundr</td>
<td>66'116'970</td>
<td>2'933'937</td>
<td>56'935.82</td>
<td>384'886.94</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

17
6 Discussion and Future Work

We introduced our methodology for contextualized programs, and presented an instance of that framework that uses the guarded command language as used by Prism together with context definitions based on DL ontologies. In addition, we developed a method for reducing probabilistic analysis of contextualized programs to probabilistic analysis of stochastic programs, and evaluated this technique using the Pellet reasoner and Prism for model checking. Our results show that the approach is feasible and allows for a convenient way of analysis operational behaviour for different contexts. We note that even though we focused on stochastic programs and a semantics based on MDPs, our general technique for computing placeholder replacements using justifications is general enough to be used for any kind of operational formalism that is based on state representations.

There are several open problems that might be interesting to investigate in the future. First, as discussed in Section 3.2, we are currently not addressing inconsistency of states in the contextualized programs directly, while there are various ways in which one could do so. While it is straightforward to use probabilistic analysis of the reduced programs to decide whether a program can enter an inconsistent state (for this we would only have to use a placeholder label marking inconsistency), it would be interesting to investigate different methods to avoid inconsistent states in an automated way. In addition, it would be interesting to develop a closer semantic integration of the two formalisms currently used in contextualized programs. In our current semantics, placeholder labels are assigned to each state independently of each other, based on the sets of axioms entailed in the corresponding state ontology. This means that, if we have two labels $\ell_1$ and $\ell_2$ that express mutually disjoint information (that is, $A(a)$ and $\neg A(a)$), even the disjunction $\ell_1 \lor \ell_2$ is only entailed in the state if one of its disjuncts is entailed explicitly, even though the disjunction $A(a) \lor \neg A(a)$ is entailed in every ontology. It would be interesting to investigate how a semantics would look like that treats entailment of formulae over labels differently, and whether it is still possible to construct a reduced stochastic program that can be used by standard PMC tools. Another interesting feature would be to allow for a dynamic switching of contexts during the execution of the program.

References


