IMT School for Advanced Studies
Computer Science and Engineering

Plan Synthesis in Explicit-input Knowledge and Action Bases

Doctoral Dissertation of:
Michele Stawowy

December 2016

Publications

1. Stawowy, M. (2015).
   Optimizations for decision making and planning in description logic dynamic knowledge bases.
   In *Proceedings of the 28th International Workshop on Description Logics*
2. Calvanese, D., Montali, M., Patrizi, F., and Stawowy, M. (2016).
   Plan synthesis for knowledge and action bases.
   In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*

Abstract

In this Thesis we study plan synthesis for data-centric domains, where the interest is not only upon the actions the system performs to reach its desired goal, but also on how the knowledge defining the domain evolves with the aforementioned actions.

We first introduce a rich, dynamic framework named *Explicit-input Knowledge and Action Bases* (eKABs), where states are *Description Logic* (DL) Knowledge Bases, whose extensional part is manipulated by actions that possibly introduce new objects from an infinite domain. We show that plan existence over eKABs is undecidable even under severe restrictions.

We then focus on state-bounded eKABs, a class for which plan existence is decidable, and provide sound and complete plan synthesis algorithms, which combine techniques based on standard planning, DL query answering, and finite-state abstraction. All results hold for any DL with decidable query answering.

We finally show that for lightweight DLs, plan synthesis can be compiled into standard planning, and we provide two translations: translation to STRIPS for a restricted version of lightweight DL eKABs, and translation to ADL for full lightweight DL eKABs. For the STRIPS setting, we provide an additional technique to optimize Knowledge Base satisfiability check inside the translation. We also provide a technique showing how it is possible to transform any full lightweight DL eKAB to an equivalent restricted lightweight DL eKAB.

Contents

List of Acronyms **vii**

1 Introduction **1**

2 Preliminaries **5**
2.1 Description Logic 6
2.1.1 Interpretation of a Knowledge Base . 9
2.1.2 Query Answering 12
2.2 DL-Lite Family 16
2.2.1 First Order Rewritability 18
2.3 Planning 22
2.3.1 STanford Research Institute Problem Solver 25
2.3.2 Action Description Language 27
2.4 Description Logic-based Dynamic Systems . 29

3 Explicit-input Knowledge and Action Bases **33**

3.1 Parametric actions 35
3.2 Condition-Action Rules 37
3.3 Execution Semantics 38

4 Planning with eKABs: Plan existence and Plan synthesis 43
4.1 Plan Existence 44
4.2 Plan Synthesis 56
4.2.1 Plan Synthesis for eKABs with Finite Domain 58
4.2.2 Plan Synthesis for State-Bounded eKABs 60
4.2.3 Plan Templates and Online Instantiation 64

5 Plan Synthesis for Lightweight eKABs 69
5.1 Translation to STRIPS 70
5.1.1 Action Rewriting 79
5.2 Translation to ADL 83
5.3 From eKABs to reKABs 90

6 Proof of Concept 103
6.1 Robot on a Grid 104

7 Conclusions 111
7.1 Summary 111
7.2 Future Work 113

Bibliography 115

List of Acronyms

ADL Action Description Language 27

AI Artificial Intelligence 22

CQ Conjunctive Query 12

DL Description Logic ii

DLDS Description Logic-based Dynamic System 2

ECQ Extended Conjunctive Query 14

eKAB Explicit-input Knowledge and Action Base ii

FO First Order 2

FOL First-Order Predicate Logic 5

KAB Knowledge and Action Base 2

KB Knowledge Base 2

PDDL Planning Domain Definition Language 25

reKAB Reduced Explicit-input Knowledge and Action Base 36

RDBMS Relational Database Management System . . 18

SQL Structured Query Language 18

STRIPS STanford Research Institute Problem Solver . 4

TS Transition System 38

UCQ Union of Conjunctive Queries 12

Introduction 1

Classically, management of business processes always focused on workflows and the actions/interactions that take part in them, an approach called *process-centric*. One of the most prominent operations related to business processes is *planning* [Ghallab et al., 2004b], namely finding a sequence of operations/actions that allows to reach a desired goal. Lately, such approach has been call into question, as the sole focus on the workflow leaves out the *informational context* in which the workflow is executed.

Recently, there has been an increasing interest in *Artifact-centric models for business processes* [Bhattacharya et al., 2007, Cohn and Hull, 2009], as they integrate static struc-

tural knowledge (i.e. data related), with action-based mechanisms in a seamless way, thus overcoming the limits of the process-centric approach. Combining these two aspects into a single logical system is known to be difficult and easily leading to undecidability, even for simple forms of inference about system dynamics, when the logics used are rather limited [Wolter and Zakharyaschev, 1999, Gabbay et al., 2003].

In this context, we can see the development of the framework called *Knowledge and Action Bases* (KABs) [Hariri et al., 2013], the later higher formalization of it named *Description Logic-based Dynamic Systems* (DLDSs) [Calvanese et al., 2013b], and the Golog-based work of [Baader and Zarrieß, 2013]. These works all share the same concept: handle the data-layer through a *DL ontology*, while the process-layer, since DLs are only able to give a static representation of the domain of interest, is defined as *actions* that update the ontology. This is the so-called "functional view of knowledge bases" [Levesque, 1984]: actions evolve the system by querying the current state using logical inference (ASK operation), and then using the derived facts to assert new knowledge in the resulting state (TELL operation).

A prominent feature of KABs is that actions allow one to incorporate into the *Knowledge Base* (KB) external input provided by fresh objects taken from an infinite domain. This gives rise, in general, to an infinite-state system, in which reasoning is undecidable. Nevertheless, decidability of verification of *First Order* (FO) temporal

properties has been obtained for KABs that are *state-bounded* [Bagheri Hariri et al., 2013a, Bagheri Hariri et al., 2014], i.e., in which the number of objects in each single state is bounded a-priori, but is unbounded along single runs, and in the whole system.

The main contribution of this Thesis is the study of the problem of planning, specifically plan existence and synthesis [Ghallab et al., 2004a, Cimatti et al., 2008], for knowledge-intensive dynamic systems over infinite domains. To achieve this goal, we first introduce a variation of KABs, termed *Explicit-input Knowledge and Action Bases* (eKABs), more suited for our purposes, in which the input-related information for an action is made explicit in its signature, and not hidden in its conditional effects. In fact, eKABs can be considered as a concrete instantiation of the more abstract framework of DLDS [Calvanese et al., 2013c], and inherit its possibility of abstracting away the specific DL formalism used to capture the underlying KB.

We show that, in line with previous work on planning in rich settings [Erol et al., 1995], plan existence is undecidable even for severely restricted eKABs. We thus focus on a non-trivial subset of eKABs, namely state-bounded eKABs, for which we prove that plan existence is decidable in PSpace data complexity, by combining techniques based on standard planning, DL query answering, and finite-state abstractions for DLDSs. After proving plan existence, we move on plan synthesis, and present a sound and complete algorithm for state-bounded, DL agnostic eKABs; more over, the returned plans represent templates

for the original synthesis problem, defined over an eKAB with an infinite domain.

We then concentrate on eKABs based on lightweight DLs of the *DL-Lite* family [Calvanese et al., 2007b].In this case, we demonstrate that plan synthesis can be tackled by compilation into standard *STanford Research Institute Problem Solver* (STRIPS) or Action Description Language planning problems, which can then be processed by any off-the-shelf planner. For the STRIPS setting, we also show a special technique to insert consistency control of the KB directly into the actions.

Our work is similar in spirit to the one by [Hoffmann et al., 2009], as combining a rich knowledge-based setting with the possibility of incorporating external input. However, to the best of our knowledge, our setting is the first one where planning is decidable without severe restrictions, in particular on how the external input is handled. These results are based on the following publications:

- Stawowy, M. (2015).
  Optimizations for decision making and planning in description logic dynamic knowledge bases.
  In *Proceedings of the 28th International Workshop on Description Logics*
- Calvanese, D., Montali, M., Patrizi, F., and Stawowy, M. (2016).
  Plan synthesis for knowledge and action bases.
  In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*

Preliminaries 2

In this Chapter we introduce the main theoretical elements that constitute the base of this Thesis. As the main subject is knowledge manipulation, we first deal with how we intend to represent this knowledge; we need, of course, a formal framework where we can express facts about the modelled world of interest, and tools to work with them.

We start with *Description Logic* (Section 2.1), a very famous branch of *First-Order Predicate Logic* (FOL), widely used for its expressivity while still being decidable (in contrast with FOL, which is not), and comprised of many different dialects. Of these dialects, we focus on one of them, namely the *DL-Lite* family (Section 2.2), appreciated for

its balance between expressivity and computational properties.

Given an introduction to how we intend to represent knowledge, we introduce the main aspects of *Planning* (Section 2.3), a branch of Artificial Intelligence devoted to the process of computationally find strategies or action sequences that satisfy a given goal as best as possible. We do this as the thesis focuses on planning-oriented knowledge manipulation, where we want to achieve a state where specific facts hold true. Many types of planning exist, mainly related with how detailed is, and how many aspects we consider into, the representation of our domain of interest; of these, we have a look at *Classical Planning*, and two formalisms to represent a classical planning problem, namely *STanford Research Institute Problem Solver* (Section 2.3.1) and *Action Description Language* (Section 2.3.2).

2.1 Description Logic

Description Logic (DL) is a family of formal knowledge representation languages widely used in ontological modelling [Baader et al., 2003].

DLs can be seen as fragments of *First-Order Predicate Logic* (FOL). It is well known that general FOL is undecidable, so research has focused on tailoring fragments of it that are decidable, to such an extent that now decidability is perceived as a necessary condition to call a formalism a DL.

The DL semantic model represents the domain of interest in terms of *individuals* coming from a (possibly) infinite *domain*, that participate to *concepts* and *roles*. This representation is captured by a DL *Knowledge Base* (KB), also often called *ontology*, which is based on a vocabulary of concept, role, and individual names, and is composed of two parts: a *TBox*, that represents the *intentional knowledge* of the KB through universal assertions over concepts and roles, and an *ABox*, that represents the *extensional knowledge* of the KB through membership assertions (or facts) about the participation of individuals to concepts and roles. As stated before, there is not only just a single Description Logic, but many DL languages which differ in the type of assertions one is allowed to use in the KB; the more assertions are allowed, the more expressive the language becomes, as well as more complex. It is upon the user choose the right DL for her intended application.

In the following, while not referring to any specific DL language, we give a more formal definition of the syntax and semantics of a generic DL KB, starting from the domain. Let Δ be a countably infinite universe of individual names (referred also as individuals or objects), acting as standard names [Levesque and Lakemeyer, 2001]. The ABox is a *finite set of membership assertions*, i.e., atomic formulas of the form $\mathsf{N}(\mathsf{d})$ and $\mathsf{P}(\mathsf{d}, \mathsf{d}')$, where N is a concept name, P is a role name, and d, d' are individual names; with $\textsc{adom}(A)$ we denote the set of objects from Δ occurring in A, e.g. d and d'.

For example, we can state that Mark is an engineer

through the following assertion:

$$\mathsf{Engineer}(\mathsf{mark}) \tag{2.1}$$

Depending on the language, also other types of assertions could be present, such as equality assertions between individuals (e.g., $\mathsf{d} = \mathsf{d}'$).

The TBox instead contains axioms describing relationships between concepts and roles, such as the *general concept inclusion* axiom $\mathsf{A} \sqsubseteq \mathsf{B}$ (often called also subsumption relation), which states that all the individuals related to A belong also to the concept B. For example, we can use the inclusion to model the fact that all engineers are employees through the axiom

$$\mathsf{Engineer} \sqsubseteq \mathsf{Employee} \tag{2.2}$$

Again, the general concept inclusion is only one of the many available expressions in DLs; other notable expressions are the negation $\neg$, intersection $\sqcap$, existential quantification over roles $\exists.R$, etc.

Example 1 *We can model the different roles inside a company, such as the hierarchy of the job types. For example, we can state that all engineers and all designers are employees, but one cannot be both:*

$$\begin{gathered}\mathsf{Engineer} \sqsubseteq \mathsf{Employee} \\ \mathsf{Designer} \sqsubseteq \mathsf{Employee} \\ \mathsf{Designer} \not\sqsubseteq \mathsf{Engineer}\end{gathered}$$

Additionally, we can express the fact that employees can be linked to a task, but engineers cannot be assigned to assembling tasks:

$$\exists hasTask \sqsubseteq \mathsf{Employee}$$
$$\mathsf{Engineer} \not\sqsubseteq \forall \mathsf{hasTask.AssemblyTask}$$

We can also add statements regarding individuals, thus defining the ABox:

$$\mathsf{Engineer(mark)}, \mathsf{Employee(david)}, \mathsf{AssemblyTask(task1)}, ...$$

In the rest of the Thesis, to maintain a clear separation between the intensional and the extensional levels, we disallow *nominals* (concepts interpreted as singletons) in the TBox T.

2.1.1 Interpretation of a Knowledge Base

The combination of TBox and ABox assertions describes a particular situation in the modelled domain, although this description is, intentionally, not fully specified. First of all, there is no formal relationship between the symbols used and the real objects that they represent. Secondly, the information in a KB is typically implicit; for example, given the previous statements 2.1 and 2.2, it is obvious for us that Mark is also a human, but this is not explicitly said in the KB, and we have to formally define how this *inference* works. Lastly, DLs are designed with the so-called *open-world semantic*, which allows DLs to deal with

incomplete information. For example, if an individual's membership to a concept is neither stated in the ABox nor excluded, then there are different possible (and valid) interpretations, in contrast to the *closed-world semantic*, where an individual would be assumed to not belong to a concept if this is not explicitly asserted.

These points lead us to the notion of *interpretation*. Informally, an interpretation can be seen as potential "world" in which all the KB assertions hold true. Formally, an interpretation, normally denoted with $\mathcal{I}$, consists of: i) a set $\Delta^{\mathcal{I}}$ called the *interpretation domain* and representing all the individuals that exist in the "world" that $\mathcal{I}$ represents (the domain $\Delta^{\mathcal{I}}$ is not required to be finite, but can also be an infinite set); ii) an *interpretation function* $\cdot^{\mathcal{I}}$ that connects each individual name d to an element $\mathsf{d}^{\mathcal{I}} \in \Delta^{\mathcal{I}}$, each atomic concept A to a set $\mathsf{A}^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ (as opposed to the domain itself, $\mathsf{A}^{\mathcal{I}}$ is allowed to be empty), each atomic role R to a binary relation $\mathsf{R}^{\mathcal{I}} \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ (also possibly empty). The last part missing is to determine the interpretation of complex concepts and roles (e.g., a negated concept $\neg\mathsf{C}$, or intersection $\mathsf{C} \sqcap \mathsf{D}$), and axioms (e.g., a general concept inclusion axiom $\mathsf{C} \sqsubseteq \mathsf{D}$). For complex terms, we follow the principle of compositional semantics, and define them starting from the semantics of its constituents. For example, the term $\neg\mathsf{C}$ denotes the set of all those individuals that are not contained in the extension of C, thus its interpretation function is defined as $(\neg\mathsf{C})^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus \mathsf{C}^{\mathcal{I}}$. The term $\mathsf{C} \sqcap \mathsf{D}$ denotes the concept formed by all individuals that are simultaneously in C and D, thus the interpreta-

tion function is defined as $(\mathsf{C} \sqcap \mathsf{D})^{\mathcal{I}} = \mathsf{C}^{\mathcal{I}} \cap \mathsf{D}^{\mathcal{I}}$. For axioms, instead, the interpretation function has to determine their *satisfaction*, which means determine when an axiom α is true (holds); for example, the general concept inclusion $\mathsf{C} \sqsubseteq \mathsf{D}$ is satisfied by $\mathcal{I}$, if every instance of C is also an instance of D, which, formally, can be written as $\mathsf{C}^{\mathcal{I}} \subseteq \mathsf{D}^{I}$. If this is the case, we also say that $\mathcal{I}$ is a model of α (or that $\mathcal{I}$ satisfies α), and write $\mathcal{I} \models \alpha$.

The interpretation $\mathcal{I}$ is a *model* of a given knowledge base KB (alternatively, $\mathcal{I}$ satisfies KB, written $\mathcal{I} \models KB$), if it satisfies all the axioms of KB ($\mathcal{I} \models \alpha$ for every $\alpha \in KB$). A knowledge base KB is called *satisfiable* (or *consistent*) if it has at least a model. If no interpretation $\mathcal{I}$ can satisfy the given KB, then the ontology is called *unsatisfiable* (or *inconsistent*); in this case every axiom is vacuously satisfied by all of the (none) interpretations that models the KB.

Further, we say that $\langle T, A\rangle$ *logically implies* an ABox assertion α, written $\langle T, A\rangle \models \alpha$, if every model of $\langle T, A\rangle$ is also a model of α. Given two ABoxes A_1, A_2, and a bijection $h : S_1 \to S_2$, where $\textsc{adom}(A_1) \subseteq S_1$ and $\textsc{adom}(A_2) \subseteq S_2$, A_1 and A_2 are said to be *logically equivalent modulo renaming h w.r.t. a TBox T*, written $A_1 \cong^h_T A_2$, if:

1. for each fact α_1 in A_1, $\langle T, A_2\rangle \models h(\alpha_1)$;
2. for each fact α_2 in A_2, $\langle T, A_1\rangle \models h^{-1}(\alpha_2)$;

where $h(\alpha_1)$ (resp. $h^{-1}(\alpha_2)$) is a new assertion obtained from α_1 (resp., α_2), by replacing each occurrence of an object $d \in S_1$ (resp., $d \in S_2$) with $h(d)$ (resp., $h^{-1}(d)$). We say that A_1 and A_2 are *logically equivalent modulo*

renaming w.r.t. T, written $A_1 \cong_T A_2$, if $A_1 \cong_T^h A_2$ for some bijection h. We omit T when clear from the context.

2.1.2 Query Answering

Knowledge Base Satisfiability is one of the most (if not the most) important reasoning tasks offered by DLs that require elaborate inferencing; many other tasks (such as axiom entailment, concept satisfiability, and others) can be reduced to Knowledge Base Satisfiability. A prominent reasoning task extensively used, and not reducible to Knowledge Base Satisfiability, is *Query Answering*, that is the ability to query a KB and retrieve the desired data. To perform this task we resort to a well known formalism in the database community [Chandra and Merlin, 1977]: *Conjunctive Query* (CQ) and *Union of Conjunctive Queries* (UCQ). CQs and UCQs are a restricted form of first-order queries, and they constitute an expressive query language with capabilities that go beyond standard reasoning tasks in DLs.

A CQ q over a KB is a FOL formula constructed from atomic formulae and using only the conjunction and existential quantification operators, while disjunction, negation, or universal quantification operators are not allowed. It has the following form:

$$\overrightarrow{x}.\exists\overrightarrow{y}.\phi_1 \wedge ... \wedge \phi_n$$

where $\overrightarrow{x}$ is a set of free variables (also called distinguished), $\overrightarrow{y}$ is a set of existentially quantified variables (also called

non-distinguished), and ϕ an atomic formula of the type $\mathsf{C}(z)$, $\mathsf{R}(z, z')$ where C and P respectively denote a concept and a role name occurring the KB, and z, z' are individual names or variables in $\overrightarrow{x}$ or $\overrightarrow{y}$. An UCQ q over a KB is a FOL formula comprised of, as the name suggest, the OR-union of CQs, and has the form:

$$\exists \overrightarrow{y}_1.conj(\overrightarrow{x}, \overrightarrow{y}_1) \vee ... \vee \exists \overrightarrow{y}_n.conj(\overrightarrow{x}, \overrightarrow{y}_n)$$

where $\exists \overrightarrow{y}_i.conj(\overrightarrow{x}, \overrightarrow{y}_i)$ (for $1 \leqslant i \leqslant n$) is a CQ as detailed before, with the free variables $\overrightarrow{x}$ shared by each CQ, and possibly different existential variables $\overrightarrow{y}_i$ each.

Given the syntax of a (U)CQ q, we now define its semantics, that is how the formula q is evaluated against a KB in order to retrieve the desired data. (U)CQs employ the so called *certain answers* model: a certain answer to q over the KB $\langle T, A\rangle$ is a substitution ϑ of the free variables $\overrightarrow{x}$ of q with individual names in A, such that $q\vartheta$ evaluates to true in every model $\mathcal{I}$ of the KB (denoted $\langle T, A\rangle \models q\vartheta$). The certain answers is then the set $\textsc{ans}(q, T, A)$ of all valid substitutions ϑ. If q has no free variables $\overrightarrow{x}$, then it is a *boolean query* and its certain answers are either the *empty substitution* $\{\}$ denoting true, or the empty set $()$ denoting false. *Query Answering* is thus the problem of finding all certain answer answers.

Example 2 *Given the KB defined in Example 1, we can query it in order to retrieve individuals hat respect specific values. We could ask for all the employees that are assigned to a certain task and who are designers, obtaining*

the following CQ:

$$q = \mathsf{hasTask}(x, \mathsf{task1}) \wedge \mathsf{Designer}(x)$$

Another type of query is the so-called *Extended Conjunctive Query* (ECQ). ECQs are an extension of UCQs, based upon the query language *EQL-Lite(UCQ)* [Calvanese et al., 2007a], that is, the FOL query language whose atoms are UCQs evaluated according to the certain answer semantics explained above. An ECQ over a KB $\langle T, A\rangle$ is a (possibly open) formula of the form [1]:

$$Q \longrightarrow [q] \mid [x = y] \mid \neg Q \mid Q_1 \wedge Q_2 \mid \exists x.Q$$

In the formula, we have that logical operators have the usual meaning, the existential quatification ranges over elements of the domain $\textsc{adom}(\mathrm{A})$, $[q]$ represents the certain answers of the UCQ q over $\langle T, A\rangle$, while $[x = y]$ denotes the certain answers $\textsc{ans}(x = y, T, A)$ (which is the set $\{\langle x, y\rangle \in \textsc{adom}(A) \mid \langle T, A\rangle \models (x = y)\}$). Given an ECQ Q, we define a certain answer to Q over the KB $\langle T, A\rangle$ as a substitution ϑ of the free variables $\overrightarrow{x}$ of Q with individual names in A such that:

$T, A, \vartheta \models [q]$	if	$\langle T, A\rangle \models q\vartheta$
$T, A, \vartheta \models [x = y]$	if	$\langle T, A\rangle \models (x = y)\vartheta$
$T, A, \vartheta \models \neg Q$	if	$T, A, \vartheta \not\models Q$
$T, A, \vartheta \models Q_1 \wedge Q_2$	if	$T, A, \vartheta \models Q_1$ and $T, A, \vartheta \models Q_2$
$T, A, \vartheta \models \exists x.Q$	if	exists $t \in \textsc{adom}(A)$ such that $T, A, \vartheta[x/t] \models Q$

[1] In this thesis we consider ECQs that are *domain-independent* and $\langle T, A\rangle$-range restricted [Calvanese et al., 2007a].

where $\vartheta[x/t]$ denotes the substitution obtained from ϑ by assigning to x the constant/term t; if x is already present in ϑ, then its value is replaced by t, otherwise the pair x/t is added to ϑ.

Given the definition of a single certain answer, we define the *certain answers to* Q *over* $\langle T, A\rangle$ ($\text{ANS}(Q, T, A)$) as before, i.e., the set of all possible substitutions ϑ for the free variables in Q so that $\langle T, A\rangle \models Q\vartheta$. The *certain answers* $\text{ANS}(Q, T, A)$ *of an ECQ* Q *over* $\langle T, A\rangle$ are obtained by first computing, for each atomic ECQ $[q]$, the certain answers of q, and then composing the obtained answers through the FOL constructs in Q. Hence $[q]$ acts as a *minimal knowledge operator*, and negation and quantification applied over UCQs are interpreted epistemically [Calvanese et al., 2007a]. Under this semantics, ECQs are *generic*, in the sense of *genericity* in databases [Abiteboul et al., 1995]: a query evaluated over two logically equivalent ABoxes returns the same answer, modulo object renaming.

Example 3 *Given the KB defined in Example 1, we could now ask if exists any employee that is assigned to a any task and who is not an engineer, obtaining the following ECQ:*

$$q = \exists x.[\exists y.\mathsf{hasTask}(x, y)] \wedge \neg[\mathsf{Engineer}(x)]$$

As customary, we consider DLs for which query answering (and hence the standard reasoning tasks of KB satisfiability and logical implication) is decidable.

2.2 DL-Lite Family

Given the formal introduction to Description Logics in Sec. 2.1, we now detail a specific subset denoted *DL-Lite* [Calvanese et al., 2007b]. DL-Lite is a family of DL dialects specifically designed for data-intensive scenarios, where ontologies are used as a high-level, conceptual view over large data repositories; typical scenarios for this are those of Information and Data Integration Systems, the Semantic Web, and ontology-based data access. Current reasoners for expressive DLs, although having good performances, are not able to deal with large amounts of data (e.g., a large ABox), and are impractical for the afore-mentioned situations. The DL-Lite family, on the contrary, offers a low complexity of reasoning and of answering complex queries (in particular w.r.t. data complexity), while still being able to capture basic ontology and conceptual data modelling languages (such as UML class diagrams).

Of the whole DL-Lite family, we introduce one of its most prominent member, *DL-Lite*$_A$ [Calvanese et al., 2009]. In DL-Lite$_A$, concepts are differentiated in *atomic*, *basic*, and *general* as detailed in the following grammar:

$$\begin{array}{ll} \mathsf{B} \longrightarrow \mathsf{A} \mid \exists \mathsf{Q} & \text{basic concept} \\ \mathsf{C} \longrightarrow \top_c \mid \mathsf{B} \mid \neg \mathsf{B} \mid \exists \mathsf{Q}.\mathsf{C} & \text{general concept} \end{array}$$

where: A is an atomic concept (i.e., a concept denoted by a name), $\exists \mathsf{Q}$ (also called *unqualified existential restriction* concept) is the concept that denotes the domain of a role

Q (i.e., the set of individuals that Q relates to some individuals), $\neg\mathsf{B}$ is the *negation* of concept B, $\exists\mathsf{Q}.\mathsf{C}$ (also called *qualified existential restriction* concept) is the concept that denotes the qualified domain of a role Q w.r.t. the concept C (i.e., the set of individuals that Q relates to some individuals in C), and $\top_c$ is the universal concept. DL-Lite$_A$ roles are defined similarly:

$$\begin{array}{ll} \mathsf{Q} \longrightarrow \mathsf{P} \mid \mathsf{P}^- & \text{basic role} \\ \mathsf{R} \longrightarrow \mathsf{Q} \mid \neg\mathsf{Q} & \text{general role} \end{array}$$

where: P is an atomic role (i.e., a role denoted by a name), P^- is the *inverse* of the atomic role P, and $\neg\mathsf{Q}$ is the *negation* of role Q.

Given the possible DL-Lite$_A$ expressions, we show now what type of axioms are available in a DL-Lite$_A$ TBox:

$$\begin{array}{ll} \mathsf{B} \sqsubseteq \mathsf{C}\ ,\ \mathsf{Q} \sqsubseteq \mathsf{R} & \text{inclusion assertions} \\ (\mathsf{funct}\ \mathsf{Q}) & \text{functionality assertions} \end{array}$$

A functionality assertion states that the binary relation represented by a role is a function.

A DL-Lite$_A$ ABox is of a set of membership assertions of the following form:

$$\mathsf{A}(\mathsf{d}) \qquad \mathsf{P}(\mathsf{d}_1, \mathsf{d}_2)$$

where: A is an atomic concept, P is an atomic role, d, d_1, d_2 are individuals.

2.2.1 First Order Rewritability

One of the most remarkable features that DL-Lite$_A$ offers, is the so-called *First Order Rewritability* (or *FOL-rewritability*). FOL-rewritability allows to reduce query answering over a KB $\langle T, A\rangle$ to evaluating a FOL query q_{rew} as a standard *Structured Query Language* (SQL) query over a relational database equivalent to A. This is done as a two step process: in the first step, called *query reformulation*, the query q is reformulated using the intensional knowledge of the TBox, obtaining an UCQ q_{rew} that can be directly evaluated over the ABox; second, assuming that the ABox is maintained by an *Relational Database Management System* (RDBMS) in secondary storage, the evaluation can be done using a standard SQL engine, thus taking advantage of well established query optimization strategies. Since the query reformulation does not depend on the data but only on the TBox, and in the second step the query q_{rew} is evaluated against a relational database, the whole query answering process is in AC^0 in the size of the data, i.e., the same complexity as the plain evaluation of a conjunctive query over a relational database. Also satisfiability checking of a KB can be reduced to evaluating a special boolean FOL query Q^T_{unsat} through FOL-rewritability.

Before defining *FOL-rewritability*, we have to introduce the minimal model $DB(A)$ for a given ABox A, which is the interpretation $\langle \Delta^{DB(A)}, \cdot^{DB(A)}\rangle$ defined as follows:

- $\Delta^{DB(A)}$ is the non-empty set consisting of the union

of the set of all individual constants occurring in A;

- $\mathsf{a}^{DB(A)} = \mathsf{a}$ for each object constant a;
- $\mathsf{A}^{DB(A)} = \{\mathsf{a} | \mathsf{A}(\mathsf{a}) \in A\}$ for each atomic concept A;
- $\mathsf{P}^{DB(A)} = \{(\mathsf{a}_1, \mathsf{a}_2 | \mathsf{P}(\mathsf{a}_1, \mathsf{a}_2) \in P\}$ for each atomic concept P;

Secondly, we introduce the *NI-closure* of T (denoted by $cln(T)$), defined inductively as follows:

1. all functionality assertion in T are also in $cln(T)$;
2. all negative inclusion assertion in T are also in $cln(T)$;
3. if $\mathsf{B}_1 \sqsubseteq \mathsf{B}_2$ is in T and $\mathsf{B}_2 \sqsubseteq \neg\mathsf{B}_3$ or $\mathsf{B}_3 \sqsubseteq \neg\mathsf{B}_2$ are in $cln(T)$, then also $\mathsf{B}_1 \sqsubseteq \neg\mathsf{B}_3$ is in $cln(T)$;
4. if $\mathsf{Q}_1 \sqsubseteq \mathsf{Q}_2$ is in T and $\exists\mathsf{Q}_2 \sqsubseteq \neg\mathsf{B}$ or $\mathsf{B} \sqsubseteq \neg\exists\mathsf{Q}_2$ are in $cln(T)$, then also $\exists\mathsf{Q}_1 \sqsubseteq \neg\mathsf{B}$ is in $cln(T)$;
5. if $\mathsf{Q}_1 \sqsubseteq \mathsf{Q}_2$ is in T and $\exists\mathsf{Q}_2^- \sqsubseteq \neg\mathsf{B}$ or $\mathsf{B} \sqsubseteq \neg\exists\mathsf{Q}_2^-$ are in $cln(T)$, then also $\exists\mathsf{Q}_1^- \sqsubseteq \neg\mathsf{B}$ is in $cln(T)$;
6. if $\mathsf{Q}_1 \sqsubseteq \mathsf{Q}_2$ is in T and $\mathsf{Q}_2 \sqsubseteq \neg\mathsf{Q}_3$ or $\mathsf{Q}_3 \sqsubseteq \neg\mathsf{Q}_2$ are in $cln(T)$, then also $\mathsf{Q}_1 \sqsubseteq \neg\mathsf{Q}_3$ is in $cln(T)$;
7. if one of the assertions $\exists\mathsf{Q} \sqsubseteq \neg\exists\mathsf{Q}$, $\exists\mathsf{Q}^- \sqsubseteq \neg\exists\mathsf{Q}^-$, or $\mathsf{Q} \sqsubseteq \neg\mathsf{Q}$ is in $cln(T)$, then all three such assertions are in $cln(T)$.

We can now define the boolean UCQ with inequalities Q^T_{unsat}, which allows to verify whether $DB(A)$ is a model of $\langle cln(T), A\rangle$ by simply evaluating Q^T_{unsat} over $DB(A)$ itself. A translation function δ is defined from assertions in $cln(T)$ to boolean CQs with inequalities, as follows:

$$\begin{aligned}
\delta((\mathsf{funct\ P})) &= \exists x, y_1, y_2.\mathsf{P}(x, y_1) \wedge \mathsf{P}(x, y_2) \wedge y_1 \neq y_2 \\
\delta((\mathsf{funct\ P^-})) &= \exists x_1, x_2, y.\mathsf{P}(x_1, y) \wedge \mathsf{P}(x_2, y) \wedge x_1 \neq x_2 \\
\delta(\mathsf{B}_1 \sqsubseteq \neg \mathsf{B}_2) &= \exists x.\gamma_1(\mathsf{B}_1, x) \wedge \gamma_2(\mathsf{B}_2, x) \\
\delta(\mathsf{Q}_1 \sqsubseteq \neg \mathsf{Q}_2) &= \exists x, y.\rho(\mathsf{Q}_1, x, y) \wedge \rho(\mathsf{Q}_2, x, y)
\end{aligned}$$

where in the last two equations:

$$\gamma_i(\mathsf{B}, x) = \begin{cases} \mathsf{A}(x) & \text{if } \mathsf{B} = \mathsf{A} \\ \exists y_i.\mathsf{P}(x, y_i) & \text{if } \mathsf{B} = \exists \mathsf{P} \\ \exists y_i.\mathsf{P}(y_i, x) & \text{if } \mathsf{B} = \exists \mathsf{P}^- \end{cases}$$

$$\rho(\mathsf{Q}, x, y) = \begin{cases} \mathsf{P}(x, y) & \text{if } \mathsf{Q} = \mathsf{P} \\ \mathsf{P}(y, x) & \text{if } \mathsf{Q} = \mathsf{P}^- \end{cases}$$

Q^T_{unsat} is then defined with the following steps (the symbol $\bot$ indicates a predicate whose evaluation is *false* in every interpretation):

1. $Q^T_{\text{unsat}} := \bot$;
2. for each $\alpha \in cln(T)$ do: $Q^T_{\text{unsat}} := Q^T_{\text{unsat}} \cup \{\delta(\alpha)\}$.

We report here the formal definitions of FOL-rewritability taken directly from [Calvanese et al., 2009]:

Definition 2.2.1 *Satisfiability in a DL $\mathcal{L}$ is FOL-rewritable, if for every TBox T expressed in $\mathcal{L}$, there exists a boolean*

FOL query Q^T_{unsat}*, over the alphabet of* T*, such that for every non-empty ABox* A*, the ontology* $\langle T, A\rangle$ *is satisfiable if and only if* Q^T_{unsat} *evaluates to* false *in* $DB(A)$*.*

Definition 2.2.2 *Answering UCQs in a DL* $\mathcal{L}$ *is FOL-rewritable, if for every UCQ* q *and every TBox* T *expressed over* $\mathcal{L}$*, there exists a FOL query* q'*, over the alphabet of* T*, such that for every non-empty ABox* A *and every tuple of constants* a *occurring in* A*, we have that* $a \in \text{ANS}(q, T, A)$ *if and only if* $a^{DB(A)} \in q^{DB(A)}$*.*

The latter definition means that every UCQ Q expressed over a *DL-Lite*$_\mathcal{A}$ TBox T can be effectively rewritten into a FOL query $rew(Q, T)$ s.t., for every ABox A, the certain answers $\text{ANS}(Q, T, A)$ can be computed by *evaluating* $rew(Q, T)$ over A seen as a database under the *closed-world assumption*.

FOL-rewritability extends also to answering ECQs over a KB, as we detailed (in Section 2.1.2) that the certain answers of an ECQ Q over $\langle T, A\rangle$ are obtained by first computing the certain answers of the inner UCQs.

Example 4 *Let us consider the KB in Example 1 (which is expressible in DL-Lite, apart from the axiom* $\mathsf{Engineer} \not\sqsubseteq \forall \mathsf{hasTask}.\mathsf{AssemblyTask}$*), and the query that request all the employees:*

$$\mathsf{Employee}(x)$$

This query can be rewritten in order to consider the contribution of the TBox axioms $\mathsf{Engineer} \sqsubseteq \mathsf{Employee}$ *and*

$\mathsf{Designer} \sqsubseteq \mathsf{Employee}$, *becoming:*

$$\mathsf{Employee}(x) \vee \mathsf{Engineer}(x) \vee \mathsf{Designer}(x)$$

We can now instead build the query to check the satisfiability of an ABox w.r.t. the Tbox:

$$\exists x.\mathsf{Engineer}(x) \wedge \mathsf{Designer}(x)$$

deriving from the only negative axiom left in the TBox $\mathsf{Designer} \not\sqsubseteq \mathsf{Engineer}$.

2.3 Planning

Automated planning and scheduling (usually denoted as planning) is a branch of *Artificial Intelligence* (AI) [Ghallab et al., 2004a] devoted to the process of computationally find strategies or action sequences that satisfy a given goal as best as possible.

Slightly more formally, given as input a description of the domain of interest (typically given as the possible initial state(s) of the domain), the set of executable actions, and the desired goal (usually given a set of final states or a formula to satisfy), the planning problem is to find a valid plan that leads to a state in which the goal is met. The complexity of the planning problem is directly related to how detailed (or, vice-versa, simplified) is the representation we choose to use, such as:

- *deterministic* or *non deterministic* actions;
- *full observability* or *partial observability* of the states;
- modelling *temporal aspects* such as the *duration* of actions;

- *complex goals* such as recurrent-goals or maximizing a reward function;
- *multi-agent* environment;
- etc.

The chosen details identify several classes of planning problems, among which *Classical Planning Problem* is the simplest and, as it can be considered historically the first, the most studied. In a Classical Planning Problem, the domain of interest is represented using a directed transition system defined by a function-free first-order language with finitely many predicate symbols and constant symbols; states are sets of ground atoms that are considered to be `true`, while edges represent the actions performed. Here we see the first important assumption of classical planning: the symbols used to represent a state are finite, thus the transition system is *finite* as well. Note that such transition system represents all the possible states and evolutions of the domain of interest, without considering any goal state, nor an eventual initial state representing the current situation of the domain.

The transition system, though, is not given in its complete form, as the explicit graph is exponential w.r.t. the symbols available in the language, making it too cumbersome to use and store. It is preferred to use a more compact and practical way, by giving only the *initial state* and actions as *functions*. An action is applicable to a state when the action's precondition is satisfied, i.e., some variables have certain values in the state. When applicable, the action applies its *effect* to the state, i.e., it will change

the values of certain (possibly different) state variables. Other assumptions thus emerge: in a classical planning problem there is a unique, fully-observable initial state, and actions are *deterministic* (i.e. there is only one possible effect, and it's certainly applied), *durationless*, and *sequential* (i.e. only one action can be performed at a time).

A *classical planning domain* is a triple $\mathcal{D} = \langle S, A, \rho \rangle$, where S is a finite set of *states*, A is a finite set of *actions*, and $\rho : S \times A \to S$ is a *transition function*. Domain states are propositional assignments, and actions are operators that change them, according to ρ. A *classical planning problem* is a triple $\mathcal{P} = \langle \mathcal{D}, s_0, G \rangle$, where $\mathcal{D}$ is a planning domain, $s_0 \in S$ is the *initial state*, and $G \subseteq S$ is a set of *goal states*.

A problem $\mathcal{P}$ induces a labelled graph $\mathcal{G} = \langle S, E \rangle$, where E contains a (labelled) edge $s \xrightarrow{a} s'$ if and only if $s' = \rho(s, a)$. Essentially, planning algorithms amount to searching (in an effective way) a path in $\mathcal{G}$ from s_0 to a *goal* state. A *plan* for a planning domain is a finite sequence $\pi = a_1 \cdots a_n$ of actions in A. We call $\rho_\pi = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_n$ the run *induced* by π, where: $s_0, \ldots, s_n \in S$, for $i = 0, \ldots, n-1$, $s_{i+1} = \rho(s_i, a_{i+1})$, and s_n is the *final state* of ρ_π. A plan π is a *solution to a planning problem* $\mathcal{P}$ if there exist a (unique) run ρ_π, so that the final state $s_n \in G$.

2.3.1 STanford Research Institute Problem Solver

To formally define a planning problem, we need a specific language. In the case of classical planning, there are many options, such as STRIPS, ADL, SAT, SAS, and few others. Of the aforementioned languages, the *STanford Research Institute Problem Solver* (STRIPS) [Ghallab et al., 2004a] is the oldest and most famous; born as a problem-solving program, its propositional version is widely used in the planning community for its simplicity in describing planning problems. We will base our formalization upon the so-called *STRIPS-style planning* adopted in *Planning Domain Definition Language* (PDDL)[Bacchus, 2000], which is the standard de-facto in the planning community for representing planning problems.

Notice that PDDL describes the world in a schematic way relative to a set of objects (the domain), in order to make the encoding small and easy to write. This schematic input is then usually translated into (propositional) STRIPS through *grounding*, i.e., by instantiating the variables in all possible ways. In the following, we will equivalently use the expressions "STRIPS-style" and "STRIPS".

A *STRIPS planning problem* can be formalized as a tuple $\langle \mathcal{C}, \mathcal{C}_0, \mathcal{F}, \mathcal{A}, \varphi, \psi \rangle$, where:

- $\mathcal{C}$ is a *finite object domain*;
- $\mathcal{C}_0 \subseteq \mathcal{C}$ is the set of *initial objects*;
- $\mathcal{F}$ is a finite set of *fluents*, i.e., predicates whose extension can vary over time;

- $\mathcal{A}$ is a finite set of *STRIPS operators*;
- φ is the *initial state*, i.e., a conjunction of ground literals using predicates in $\mathcal{F}$ and objects in $\mathcal{C}_0$;
- ψ is the *goal description*, i.e., a closed FO formula using predicates in $\mathcal{F}$ and objects in $\mathcal{C}_0$.

A state s is a set of ground literals, which represents the things that are true in the considered state, while any literal that is not included is considered false.

Each STRIPS operator in $\mathcal{A}$ is a tuple $\langle \textsc{n}, \vec{x}, \rho(\vec{x}), \varepsilon(\vec{x}) \rangle$, where:

- $\textsc{n}$ is the *name*;
- variables $\vec{x}$ are the *parameters*;
- $\rho(\vec{x})$ is the *precondition*, i.e., a FO formula over $\mathcal{F}$, and whose terms are quantified variables, objects in $\mathcal{C}_0$, and parameters $\vec{x}$;
- $\varepsilon(\vec{x})$ is the *effect*, i.e., a conjunction of (possibly negated) literals over $\mathcal{F}$ and whose terms are objects in $\mathcal{C}_0$, or parameters in $\vec{x}$.

Given an operator $\textsc{n}$, we call *action* a ground instance of $\textsc{n}$ (denoted $\textsc{n}\vartheta$), done by using a substitution function ϑ to substitute all the variables $x_1, ..., x_k$ in $\textsc{n}$ with constant symbol in $\mathcal{C}$ (e.g., $\vartheta = \{x_1 \mapsto \mathsf{a_1}, ..., x_k \mapsto \mathsf{a_k}\}$). An action $\textsc{n}\vartheta$ is said to be *applicable* to a state s if and only if the precondition $\rho(\vec{x})$ is true in s under the given binding ϑ:

$$\rho(\vec{x})^{+}\vartheta \subseteq s,\ \rho(\vec{x})^{-}\vartheta \cap s = \emptyset$$

where $\rho(\vec{x})^{+}\vartheta$ denotes the positive atoms in the grounded preconditions, and $\rho(\vec{x})^{-}\vartheta$ denotes the negated atoms. If $\textsc{n}\vartheta$ is applicable to s, then the resulting state s' is:

$$s' = (s \setminus \varepsilon(\vec{x})^{-}\vartheta) \cup \varepsilon(\vec{x})^{+}\vartheta$$

where $\varepsilon(\vec{x})^{+}\vartheta$ denotes the positive atoms in the effects, and $\varepsilon(\vec{x})^{-}\vartheta$ denotes the negated atoms.

A planning problem $\langle \mathcal{C}, \mathcal{C}_0, \mathcal{F}, \mathcal{A}, \varphi, \psi \rangle$ induces a transition system $\Upsilon = (S, A, \gamma)$, where: S is the set of all possible states, A is the set of actions, and γ is the state-transition function (i.e., $\gamma \subseteq S \times A \to S$). The state-transition function $\gamma(s, \text{N}\vartheta)$ of an action $\text{N}\vartheta$ performed in the state s is defined as:

- If $\text{N}\vartheta$ is applicable to s, then:
$$\gamma(s, \text{N}\vartheta) = (s \setminus \varepsilon(\vec{x})^{-}\vartheta) \cup \varepsilon(\vec{x})^{+}\vartheta = s'$$
- If $\text{N}\vartheta$ is not applicable to s, then $\gamma(s, \text{N}\vartheta)$ is undefined.

Finally, given the goal formula ψ, a state s is a *goal state* if s satisfies ψ, i.e., $\exists\vartheta.s \models \psi\vartheta$.

2.3.2 Action Description Language

Another possible way to define a planning problem is to use the *Action Description Language* (ADL) [Pednault, 1989]. ADL aims to be a more expressive and flexible language than STRIPS, while still keeping the computation complexity manageable. In terms of computational efficiency, it can be located between STRIPS and the Situation Calculus. Although any ADL problem can be translated into a STRIPS instance, the translation techniques

are worst-case exponential (ADL is strictly more brief than STRIPS). ADL planning is a PSPACE-complete problem, with most of the algorithms being polynomial in space even if the preconditions and effects are complex formulae.

We introduce the formal notion of ADL planning problem, by relying on and the ADL-fragment of standard PDDL. An *ADL planning problem* [Drescher and Thielscher, 2008] is a tuple $\langle \mathcal{C}, \mathcal{C}_0, \mathcal{F}, \mathcal{A}, \varphi, \psi \rangle$, where:

- $\mathcal{C}$ is a *finite object domain*;
- $\mathcal{C}_0 \subseteq \mathcal{C}$ is the set of *initial objects*;
- $\mathcal{F}$ is a finite set of *fluents*, i.e., predicates whose extension can vary over time;
- $\mathcal{A}$ is a finite set of *ADL operators*;
- φ is the *initial state*, i.e., a conjunction of ground literals using predicates in $\mathcal{F}$ and objects in $\mathcal{C}_0$; and
- ψ is the *goal description*, i.e., a closed FO formula using predicates in $\mathcal{F}$ and objects in $\mathcal{C}_0$.

Each ADL operator in $\mathcal{A}$ is a tuple $\langle \textsc{n}, \vec{x}, \rho(\vec{x}), \varepsilon(\vec{x}) \rangle$, where:

- n is the *name*;
- variables $\vec{x}$ are the *parameters*;
- $\rho(\vec{x})$ is the *precondition*, i.e., a FO formula over $\mathcal{F}$, and whose terms are quantified variables, objects in $\mathcal{C}_0$, and parameters $\vec{x}$;
- $\varepsilon(\vec{x})$ is the *effect*, i.e., the universal closure of a FO conjunction built from admissible components, inductively defined as follows:
  - Fluent literals over $\mathcal{F}$ and whose terms are variables, objects in $\mathcal{C}_0$, or parameters in $\vec{x}$, are ad-

missible.
- If $\phi_1(\vec{y}_1)$ and $\phi_2(\vec{y}_2)$ are admissible, then $\phi_1(\vec{y}_1) \wedge \phi_2(\vec{y}_2)$ and $\forall \vec{y}_1.\phi_1(\vec{y}_1)$ are also admissible.
- Let $\phi_1(\vec{y}_1)$ be a FO formula over $\mathcal{F}$, whose terms are quantified variables, objects in $\mathcal{C}_0$, variables $\vec{y_1}$, or parameters $\vec{x}$. If $\phi_2(\vec{y}_2)$ is admissible and does not contain any occurrence of $\rightarrow$ nor $\forall$, then $\phi_1(\vec{y}_1) \rightarrow \phi_2(\vec{y}_2)$ is also admissible. This is used to tackle so-called *ADL conditional effects*.

2.4 Description Logic-based Dynamic Systems

Description Logic-based Dynamic Systems (DLDSs) [Calvanese et al., 2013c] provide a general, abstract framework to formally capture dynamic systems that operate over a DL Knowledge Base, by evolving its extensional part in accordance with Levesque's functional approach. The KB is evolved by actions that query its content, and that use the obtained certain answers, together with additional (possibly fresh) input objects, to determine the facts holding in the new state. The introduction of fresh objects is a distinctive feature of DLDSs, and it is essential to tackle (business) processes, whose instances typically results in the on-the-fly creation of new objects and the establishment of relationships among them.

Technically, a DLDS is a tuple $\mathcal{S} = \langle T, A_0, \Lambda \rangle$, where

$\langle T, A_0 \rangle$ is a DL KB, and Λ is a finite set of parametric *actions*. We use $\mathcal{A}_T$ to denote the set of all ABoxes containing facts that are constructed by using concept and role names in T, and objects in Δ. Each action in Λ has the form $\langle \pi, \tau \rangle$, where

- $\pi : \mathcal{A}_T \to 2^{\mathcal{P}}$ is a *parameter selection function* that, given an ABox A, returns the *finite* set $\pi(A) \subseteq \mathcal{P}$ of parameters of interest for τ in A;
- $\tau : \mathcal{A}_T \times \Delta^{\mathcal{P}} \mapsto \mathcal{A}_T$, is a (partial) *effect function* that, given an ABox A and a *parameter assignment* $m : \pi(A) \to \Delta$ ($\Delta^{\mathcal{P}}$ denotes the set of such parameter assignments), returns (if defined) the ABox $A' = \tau(A, m)$, which: i) is consistent w.r.t. T, and ii) contains only constants in $\textsc{adom}(A) \cup \textsc{im}(m)$. [2]

In [Calvanese et al., 2013c], several decidability results related to verification and synthesis of DLDSs are shown, under the hypothesis that the DLDS of interest is *generic* and *state-bounded*. The first condition extends the previously mentioned well-known notion of genericity in databases to the case of dynamic systems. Specifically, it intuitively states that the behavior of the DLDS is invariant under renaming of objects, i.e., only depends on the classes and roles relating such objects, but not on the objects themselves. In other words, if the current state is replaced with a logically equivalent state, the system induces evolutions that are pairwise identical modulo renaming of the involved objects. The second condition requires

[2] We denote with $\textsc{dom}(\cdot)/\textsc{im}(\cdot)$ the domain/image of a function.

that the number of objects in each single state generated by the DLDS is bounded a-priori, but is still unbounded along single runs and in the whole system.

Explicit-input Knowledge and Action Bases 3

We present a model which is a variant of Knowledge and Action Bases [Bagheri Hariri et al., 2013b]. In broad terms, KABs concretize Description Logic-based Dynamic Systems by introducing an action formalism for explicitly describing updates over the DL Knowledge Base, as well as condition-action rules declaratively describing when certain actions can be executed. KABs employ the TBox not only for answering queries over the current state according to the certain answer semantics, but also to check the consistency of an action application (Section 2.1).

In their original form, KABs have two distinctive fea-

tures that make them somehow unsuitable for planning. On the one hand, they do not make any assumption regarding the frame problem, and how objects not affected by an action have to be preserved. Specifically, they work under the assumption of destructive assignment: at every step, the entire ABox needs to be reconstructed, and facts that are not explicitly (re)asserted are lost. On the other hand, the introduction of new objects into the system state is hidden inside the conditional effects of the actions, following the paradigm of service calls that are internally issued and resolved during the application of an action.

To tackle the first issue, we build on the approach by [Montali et al., 2014]. They provide a high-level surface syntax for actions, which supports (conditional) additions and deletions together with the frame assumption that facts that are neither added nor deleted are implicitly maintained. To tackle the second issue, we extract the input-related information from the conditional effects, and make it explicit in the action signature. This is why we call our formalism *Explicit-input Knowledge and Action Bases* (eKABs).

We define eKABs parametrically with respect to a DL $\mathcal{L}$. An $\mathcal{L}$-eKAB $\mathcal{K}$ is a tuple $\langle \mathcal{C}, \mathcal{C}_0, T, A_0, \Lambda, \Gamma \rangle$, where:

- $\mathcal{C} \subseteq \Delta$ is the (possibly infinite) *object domain* of $\mathcal{K}$;
- $\mathcal{C}_0 \subset \mathcal{C}$ is a *finite* set of *distinguished objects*;
- T is an $\mathcal{L}$-TBox;
- A_0 is an $\mathcal{L}$-ABox all of whose objects belong to $\mathcal{C}_0$;
- Λ is a finite set of *parametric actions*;

- Γ is a finite set of *condition-action rules*.

When the specific DL $\mathcal{L}$ is irrelevant, we omit it.

3.1 Parametric actions

A *parametric action* α has the form $a(\vec{p}) : \{e_1, \ldots, e_n\}$, where: a is the action's *name*, $\vec{p}$ are the *input parameters*, and $\{e_1, \ldots, e_n\}$ is the finite set of *conditional effects*. Each *effect* e_i has the form $Q_i \rightsquigarrow \mathbf{add}\ F_i^+, \mathbf{del}\ F_i^-$, where:

- Q_i is the *effect condition*, i.e., an ECQ over*, whose terms can be action parameters $\vec{p}$ (acting as variables), additional free variables $\vec{x}_i$, or objects from $\mathcal{C}_0$;
- F_i^+ and F_i^- are two sets of atoms over the vocabulary of T, with terms from $\vec{p} \cup \vec{x}_i \cup \mathcal{C}_0$.

Intuitively, an action is applied by grounding its parameters with objects in $\mathcal{C}$. Its effects are then evaluated in parallel, and its corresponding assertions instantiated with all the answers extracted from conditions Q_i. The instantiated assertions are finally added to/removed from the current ABox, giving priority to those that have to be added.

Given an action α and a substitution θ assigning objects of $\mathcal{C}$ to $\vec{p}$, $\alpha\theta$ denotes the *action instance* of α obtained by assigning values to $\vec{p}$ according to θ. We write $\alpha(\vec{p})$ to explicitly name the input parameters of α. The

ABox resulting from the *application* of an action instance $\alpha\theta$ on an ABox A, denoted $\text{DO}(\alpha\theta, A, T)$, is the ABox $(A \setminus A^-_{\alpha\theta}) \cup A^+_{\alpha\theta}$, where:

- $A^+_{\alpha\theta} = \bigcup_{i\in\{1,\ldots,n\}} \bigcup_{\sigma\in\text{ANS}(Q_i\theta,T,A)} F_i^+\sigma$;
- $A^-_{\alpha\theta} = \bigcup_{i\in\{1,\ldots,n\}} \bigcup_{\sigma\in\text{ANS}(Q_i\theta,T,A)} F_i^-\sigma$.

Importantly, $\alpha\theta$ is *applicable* to A only if the resulting ABox $\text{DO}(\alpha\theta, A, T)$ is consistent with T. Like for standard KABs, we limit the applicability of an action by accepting only those actions that do not lead to an inconsistency.

We note that the problem of actions that lead to inconsistent states is a well know issue, and has been treated in detail in Reasoning about Action in AI. When an action cannot be executed because of the inconsistency generated by adding the effects to the current KB, we get the so called "qualification problem" [Ginsberg and Smith, 1988]: we considered only the preconditions explicitly written in the specification, while there is at least an implicit one that we can derive from the axioms that would have blocked the performing of the action. We address this problem in a specific setting, namely *DL-Lite$_\mathcal{A}$-Reduced Explicit-input Knowledge and Action Bases* (reKABs) (Section 5.1.1), although it requires a deeper study in order to solve the qualification problem in the generic case (as we allow the use of very expressive TBox axioms), and we leave it as a future development. We believe that we could leverage the approach presented by [Calvanese et al., 2013d] and [Calvanese et al., 2015], based on the notion

of *repair*, thus effectively equipping eKABs with the ability to deal with the resulting inconsistent KB instead of simply ignoring it.

3.2 Condition-Action Rules

A *condition-action rule* γ_α for an action α has the form $Q_\alpha(\vec{x}) \mapsto \alpha(\vec{p})$, where Q_α is an ECQ mentioning only objects from $\mathcal{C}_0$ and whose free variables $\vec{x}$ come from $\vec{p}$. We assume that each action α in Λ has exactly one corresponding condition-action rule γ_α in Γ (multiple rules can be combined into a single disjunctive rule). The ECQ Q_α is used to constrain the set of action instances potentially applicable to a certain Abox A. Specifically, let θ be a parameter substitution for $\vec{p}$, and let $\theta[\vec{x}]$ denote the parameter substitution obtained by projecting θ on $\vec{x}$. Then, θ is a *$\mathcal{K}$-legal parameter substitution in A for α*, if:

- $\theta : \vec{p} \to \mathcal{C}$;
- $\theta[\vec{x}] \in \text{ANS}(Q_\alpha, T, A)$;
- $\text{DO}(\alpha\theta, A, T)$ is T-consistent.

When this is the case, $\alpha\theta$ is a *$\mathcal{K}$-legal action instance in A*. Any variable in $\vec{p} \setminus \vec{x}$ is also referred to as an *external input parameter* of α. Note that, when present, such parameters are not constrained by Q_α and can be assigned to any object, including fresh ones not occurring in $\text{ADOM}(A)$.

This last point is an important feature of the proposed framework, as it allows, given a state represented by a KB, to introduce new individuals. This would allow to

simulate, among others, the following scenarios:

- users input of new data, e.g., inserting the data of new potential clients, not known before;
- augmentation of the possessed knowledge through the creation of new primary keys to insert new tuples inside a relationship.

3.3 Execution Semantics

The *semantics* of eKABs is defined in terms of a *Transition System* (TS) with states and transitions labelled by ABoxes and action instances, respectively [Calvanese et al., 2013c]. A TS Υ is a tuple $\langle \mathcal{C}, T, \Sigma, s_0, abox, \rightarrow \rangle$, where:

- $\mathcal{C}$ is the *object domain*;
- T is a TBox;
- Σ is a (possibly infinite) *set of states*;
- $s_0 \in \Sigma$ is the *initial state*;
- $abox$ is the *labelling function*, mapping states from Σ into T-consistent ABoxes with terms from $\mathcal{C}$ only;
- $\rightarrow \subseteq \Sigma \times L \times \Sigma$ is a labelled *transition relation*, with L the (possibly infinite) set of labels.

We write $s \xrightarrow{l} s'$ as a shortcut for $\langle s, l, s' \rangle \in \rightarrow$.

An eKAB $\mathcal{K} = \langle \mathcal{C}, \mathcal{C}_0, T, A_0, \Lambda, \Gamma \rangle$ *generates* a TS $\Upsilon_{\mathcal{K}} = \langle \mathcal{C}, T, \Sigma, s_0, abox, \rightarrow \rangle$, where:

- $abox$ is the identity function;
- $s_0 = A_0$;
- $\rightarrow \subseteq \Sigma \times L_{\mathcal{K}} \times \Sigma$ is a labelled transition relation with $L_{\mathcal{K}}$ the set of all possible action instances;

- Σ and $\rightarrow$ are defined by mutual induction as the smallest sets s.t. if $A \in \Sigma$, then for every $\mathcal{K}$-legal action instance $\alpha\theta$ in A, we have that $\textsc{do}(\alpha\theta, A, T) \in \Sigma$ and $A \xrightarrow{\alpha\theta} \textsc{do}(\alpha\theta, A, T)$.

In general, $\Upsilon_{\mathcal{K}}$ is infinite, if $\mathcal{C}$ is so.

We close this section by showing that eKABs can be seen as concrete models for generic DLDSs (Section 2.4).

Lemma 3.3.1 *eKABs are generic DLDSs.*

Proof 3.3.1 *Consider an eKAB action α with its corresponding condition-action rule $Q_\alpha(\vec{x}) \mapsto \alpha(\vec{y})$. This action can be seen as a corresponding DLDS action $\langle \gamma, \tau \rangle$, where:*

- *$\gamma = |\vec{y}|$ is the constant parameter selection function that returns the fixed number of parameters in α;*
- *given an ABox A and a parameter assignment m, the effect function τ checks whether m restricted to $\vec{x}$ corresponds to one of the answers in $\textsc{ans}(Q_\alpha(\vec{x}), T, A)$, and, if so, produces a new ABox A' according to the specification of α (which guarantees that only objects in $\textsc{adom}(A) \cup \textsc{im}(m)$ are used to produce A').*

Finally, genericity is directly obtained from the observation that the evolution of an eKAB is completely driven by ECQ query answering, which is generic.

Example 5 *An eKAB $\mathcal{K} = \langle \mathcal{C}, \mathcal{C}_0, T, A_0, \Lambda, \Gamma \rangle$ is used to support decision making in a company, where employees work on tasks. $\mathcal{C}$ contains infinitely many objects. T is expressed in $\mathcal{ALCQIH}$, and contains the following axioms:*

- $\mathsf{Engineer} \sqsubseteq \mathsf{Employee}$ *and* $\mathsf{Technician} \sqsubseteq \mathsf{Employee}$ *(engineers and technicians are employees);*
- $\exists \mathsf{worksOn}^- \sqsubseteq \mathsf{Employee}$ *and* $\exists \mathsf{worksOn} \sqsubseteq \mathsf{Task}$ *(*worksOn *role links employees to tasks),*
- *similar axioms can be used to express that* worksIn *links employees to company branches;*
- $\mathsf{Employee} \sqsubseteq (= 1\, \mathsf{worksIn})$ *(employees work in exactly one one branch);*
- $\mathsf{hasResp} \sqsubseteq (\leq 1\, \mathsf{Employee})$ *and* $\mathsf{hasResp} \sqsubseteq \mathsf{worksOn}^-$ *(a task may have at most one responsible, among the employees associated to the task);*
- $\exists \mathsf{hasResp}^- \sqsubseteq \neg \mathsf{Technician}$ *(technicians are not task responsible).*

As for the dynamic aspects, to model that a new engineer can be hired in a branch provided that the planning agent does not know whether there already exists an engineer there, $\mathcal{K}$ contains the rule

$$\mathsf{Branch}(b) \wedge \neg[\exists x.\mathsf{Engineer}(x) \wedge \mathsf{worksIn}(x, b)] \mapsto \mathit{hireEng}(x, b)$$

where:

$$\mathsf{hireEng}(e, b) : \{\mathsf{true} \rightsquigarrow \boldsymbol{add}\ \{\mathsf{Engineer}(e), \mathsf{worksIn}(e, b)\}\}$$

A similar action $\mathsf{HireTech}(t, b)$ *can be used to hire a technician. Rule*

$$\mathsf{Task}(t) \wedge \mathsf{Employee}(e) \mapsto \mathit{makeResp}(t, e)$$

states that an employee can be made responsible of a task, where

$$makeResp(t, e) : \left\{ \begin{array}{r} \mathsf{hasResp}(t, p) \rightsquigarrow \boldsymbol{del}\{\mathsf{hasResp}(t, p)\} \\ \mathsf{true} \rightsquigarrow \boldsymbol{add}\{\mathsf{hasResp}(t, e)\} \end{array} \right\}$$

removes the previous task responsible, if any, and makes the selected employee the new responsible. Finally, rule

$$\mathsf{Employee}(e) \mapsto Anonymize(e)$$

models that an employee can be anonymized, where action

$$Anonymize(e) : \{\mathsf{worksIn}(e, b) \rightsquigarrow \boldsymbol{del}\{\mathsf{worksIn}(e, b)\}\}$$

models anonymization by removing the explicit information on the branch to which the selected employee belongs.

Note that there is a complex interplay between the TBox and the dynamic component of $\mathcal{K}$. E.g., the last TBox axioms forbids to hire a technician and make it responsible for a task. However, notice that this is not explicitly forbidden in the condition-action rule that defines the (potential) applicability of the makeResp action.

Planning with eKABs: Plan existence and Plan synthesis 4

We can now define the problem of *eKAB planning*. Let $\mathcal{K}$ be an Explicit-input Knowledge and Action Base and $\Upsilon_{\mathcal{K}}$ its generated Transition System.

Following the definitions in Section 2.3, we say that a *plan for* $\mathcal{K}$ is a finite sequence $\pi = \alpha_1\theta_1 \cdots \alpha_n\theta_n$ of action instances over $L_{\mathcal{K}}$ (we cannot use only actions, as they are generic and need to be grounded). Moreover, a plan π is *executable* on $\mathcal{K}$ if there exists a (unique) run $\rho_\pi = A_0 \xrightarrow{\alpha_1\theta_1} A_1 \xrightarrow{\alpha_2\theta_2} \cdots \xrightarrow{\alpha_n\theta_n} A_n$ of $\mathcal{K}$.

An *eKAB planning problem* is a pair $\langle \mathcal{K}, G \rangle$, with $\mathcal{K}$

an eKAB, and G the *goal*, a boolean ECQ mentioning only objects from $\mathcal{C}_0$. A plan π for $\mathcal{K}$ *achieves* G, if $\textsc{ans}(G, A_n,) = \mathsf{true}$, for A_n the final state of the run ρ_π.

Example 6 *Given the eKAB in Example 5, a goal G could express the intention to have an engineer and a technician working for a given task* t*, provided that, for privacy reasons, it is not known to the planning agent whether the two work in the same branch. Formally:*

$$G = \exists e_1, e_2.\mathsf{Technician}(e_1) \wedge \mathsf{Engineer}(e_2) \wedge \mathsf{worksOn}(e_1, \mathsf{t}) \wedge \mathsf{worksOn}(e_2, \mathsf{t}) \wedge \neg[\exists b.\mathsf{worksIn}(e_1, b) \wedge \mathsf{worksIn}(e_2, b)$$

where negation is in fact interpreted epistemically, in accordance with the semantics of ECQs.

4.1 Plan Existence

The initial step we take is to study *plan existence*, i.e., the problem of checking whether, for a planning problem $\langle \mathcal{K}, G \rangle$, an executable plan π for $\mathcal{K}$ exists that achieves G. The decidability of plan existence for eKABs strongly depends on the cardinality of the object domain. We consider both the cases of finite and infinite object domain, and observing how this influences the complexity of the framework. We start with the finite domain case, where plan existence is decidable.

Theorem 4.1.1 *Let $\mathcal{L}$ be a DL for which answering ECQs*[1] *is in* coNP *in data complexity. Then plan existence for $\mathcal{L}$-eKABs with a finite object domain is decidable in* PSpace *in the size of the object domain.*

Proof 4.1.1 *Let $\mathcal{K} = \langle \mathcal{C}, \mathcal{C}_0, T, A_0, \Lambda, \Gamma \rangle$ be an $\mathcal{L}$-eKAB with TS $\Upsilon_\mathcal{K}$, and G a goal. The existence of a plan achieving G can be directly reformulated as the following reachability problem: is it true that, in $\Upsilon_\mathcal{K}$, the initial state A_0 can reach a state A such that* $\text{ANS}(G, A,) = \mathsf{true}$ *This problem is decidable, because since $\mathcal{C}$ is finite, so is the number of states in $\Upsilon_\mathcal{K}$. More specifically, such states contain facts constructed by inserting objects from $\mathcal{C}$ into the extension of concepts in T, and pairs of objects from $\mathcal{C}$ into the extension of roles in T. Hence, the number of states in $\Upsilon_\mathcal{K}$ corresponds, in the worst case, to $2^{n_c \cdot |\mathcal{C}| + n_r \cdot |\mathcal{C}|^2}$, where n_c and n_r respectively denote the number of concept and role names in the vocabulary of T. On the other hand, to solve the reachability problem, $\Upsilon_\mathcal{K}$ can be constructed on-the-fly, checking whether G holds in the currently constructed state and, if this is not the case, guessing a successor state to be explored. By the assumption on $\mathcal{L}$, the goal check as well as the construction of the successor are in* coNP *in*

[1] For a significant class of Description Logics, UCQ-query answering is known to be in coNP in data complexity [Glimm et al., 2008, Ortiz et al., 2008, Calvanese et al., 2013a]. ECQs inherit this result, since they simply combine the certain answers returned by the embedded UCQs with the evaluation of the FOL operators present in the query.

the size of $\mathcal{C}$, since they respectively require to compute the certain answers of G and those of the queries involved in the application of the action used to generate the successor state. As a consequence, we get that reachability can be solved in PSpace *in the size of $\mathcal{C}$.*

We remark that KB satisfiability checking is never harder that query answering in DLs [Rudolph, 2011], thus, although we need to perform the satisfiability check at every step, the complexity is not affected by it.

When the object domain is infinite, instead, plan existence is undecidable even by considering severely limited eKABs. The following result is not surprising, and is similar in spirit to the undecidability result by [Erol et al., 1995] in the classical planning setting but with an infinite object domain. Also, a similar result was presented in [Zarrieß and Claßen, 2015], in a knowledge-based setting that is radically different from ours, since in their case the TBox is considered only in the initial state, while subsequent states may violate its assertions. This makes the proof technique significantly more complex in our case.

Theorem 4.1.2 *Plan existence for eKABs with an infinite object domain is undecidable, even if the goal is a ground CQ, and the input eKAB has: (i) an empty TBox; (ii) actions/rules employing UCQs only.*

Proof 4.1.2 *The proof is via a reduction from the termination problem of Minsky 2-counter machines, well-known*

to be undecidable [Minsky, 1967]. The two-counter machine is simulated using two stacks, in turn represented using a chain of objects. The chain contains a special element that is used as a separator for the two stacks, so that the semi-chain on the right (resp., left) of the separator simulates the first (resp., second) stack. With this structure: i) increment of counter 1 is simulated by extending the chain with a new object inserted at the extreme right; ii) decrement of counter 1 is simulated by trimming the chain on its extreme right; iii) zero testing of counter 1 is simulated by checking whether the extreme right of the chain coincides with the separator. All these operations can be implemented using UCQs (i.e., without using negation). In addition, the control states of the two-counter machine are manipulated using simple effects that remove the current state and insert the new one. The only challenging operation is increment, because there is no way of ensuring that an inserted element is effectively new, or coincides with an already existing one. To solve this issue, we proceed as follows. First, we insert an "ok" flag in the initial ABox. Whenever an increment operation is executed, a special effect checks whether the chain now contains a lasso. If this is the case, then it means that the inserted object is not new, and that the increment has not been properly executed. An error is then signalled by removing the "ok". With this formalization, we obtain that the original two-counter machine halts if and only if there exists a plan in the corresponding eKAB that achieves a state containing the "ok" flag and the halting control state.

The presence of the flag witnesses that the halting state is achieved without encountering increment errors.

Formally, a 2-counter machine $\mathcal{C}$ *is a tuple* $\langle S, s_0, s_f, \Pi \rangle$, *where:*

- S *is a finite set of states,*
- $s_0 \in S$ *is the initial state,*
- $s_f \in S$ *is the final state, and*
- Π *a finite state of instructions.*

Each instruction in Π *is of one of the following two forms:*

- $\langle s, c+, s' \rangle$, *where* $s, s' \in S$ *and* $c \in \{1, 2\}$, *is an increment instruction, which increments the value of* c *by 1 and triggers a transition from* s *to* s'.
- $\langle s, c-, s', s'' \rangle$, *where* $s, s', s'' \in S$ *and* $c \in \{1, 2\}$, *is a conditional decrement intstruction, which tests whether the value of* c *is zero; if so, it triggers a transition from* s *to* s'', *if not, it decrements the value of* c *by* 1 *and triggers a transition from* s *to* s'.

We adopt the standard execution semantics for 2-counter machines. In this setting, the halting problem asks whether there exists a run of $\mathcal{C}$ *that eventually leads to reach state* s_f.

A 2-counter machine Π *is encoded as a corresponding eKAB* $\mathcal{K}_\mathcal{C} = \langle \mathcal{T}_\mathcal{C}, A_{0,\mathcal{C}}, \Lambda_\mathcal{C}, \Gamma_\mathcal{C} \rangle$ *whose runs all simulate the single run of* $\mathcal{C}$. *The TBox* $\mathcal{T}_\mathcal{C}$ *is empty (i.e., does not contain any assertion), and provides the following vocabulary:*

- *Concept State keeps the program counter of* $\mathcal{C}$.
- *Concepts* Top_1 *and* Top_2 *store the two objects that are respectively at the top of the first and second stack.*

- *Concept Ok is used as a special flag, so that the presence of an object in Ok signals that the computation is currently correct.*
- *Concept Win is used as a special flag, so that the presence of an object in Win signals that the computation is correct, and led to the halting state.*
- *Role Next stores which objects come after each other in the two stacks, where* $Next(o_1, o_2)$ *indicates that* o_2 *immediately follows (resp., immediately precedes)* o_1 *in the first (resp., second) stack.*

$A_{0,\mathcal{C}}$ *populates the two (initially empty) stacks with a special "bottom" object bot, and initializes the state according to* $\mathcal{C}$*:*

$$A_{0,\mathcal{C}} = \{Top_1(bot), Top_2(bot), New(bot), State(s_0), Ok(c_0)\}$$

It also signals that the computation is initially correct, by introducing an arbitrary constant c_0 *in Ok.*

Increment and decrement of the two counters are reconstructed by using six dedicated actions in $\Lambda_{\mathcal{C}}$*. Increment of counter* 1 *is modelled by an action with two parameters, where the first represent the new element to be inserted into the first stack, while the second indicates the new state. The action extends the first stack with the new element, and updates the state:*

$$inc_1(t_n, s_n) : \left\{ \begin{array}{lll} Top_1(t_o) \rightsquigarrow & \textbf{\textit{del}} & \{Top_1(t_o)\}, \\ & \textbf{\textit{add}} & \{Next(t_o, t_n), Top_1(t_n)\} \\ State(s_o) \rightsquigarrow & \textbf{\textit{del}} & \{State(s_o)\}, \\ & \textbf{\textit{add}} & \{State(s_n)\} \end{array} \right\}$$

Notice, however, that this formulation of inc_1 is not correct. In fact, due to the eKAB execution semantics, there is no guarantee that the object assigned to the t_n parameter is indeed new, or instead corresponds to an object that is already part of one of the two stacks. Consequently, depending on the actual object selected for t_n, two situations may arise:

- *t_n is assigned to a new, fresh object. In this case, inc_1 correctly simulates the requested increment, and the structure induced by Next continues to be a linear chain.*
- *t_n is assigned to an object that is already mentioned in a Next fact. In this case, inc_1 does not simulate a proper increment, and the structure induced by Next now contains a lasso.*

Freshness of input parameters is simulated by adding an additional effect that recognizes whether the structure induced by Next is indeed a linear chain and, if not, signals the problem by removing the $Ok(c_0)$ fact from the current ABox. For the first counter, the presence of a lasso can be detected by simply checking whether the top element has a successor in the chain. Hence, the control effect ε_{ok,c_1} for the first counter has the following form:

$$Top_1(x) \wedge \exists y.Next(x, y) \rightsquigarrow \textbf{\textit{del}}\{Ok(c_0)\}$$

This control effect is inserted in the definition of inc_1 as well as that of the other actions. This is required because the presence of an error can be detected only after

the action causing the error is applied, and consequently the removal of the flag becomes responsibility of the next action.

The condition-action rule for inc_1 *needs to ensure that: i)* inc_1 *can be applied only in those states foreseen by* $\mathcal{C}$ *for the increment of counter* 1*; ii) the first parameter is an external input; iii) the second parameter is bound to the next state as foreseen by* $\mathcal{C}$*. Specifically, we get:*

$$\bigvee_{\langle s,1+,s'\rangle\in\Pi} State(s) \wedge y = s' \mapsto inc_1(x,y)$$

Decrement of counter 1 *is instead modeled by two distinct actions, respectively capturing the case where the counter is* 0 *or not. In the zero case, we just need to update the state with the given new state. This is done through the following simple action:*

$$decz_1(s_n) : \left\{ \begin{array}{c} State(s_o) \rightsquigarrow \textbf{\textit{del}}\ \{State(s_o)\}, \\ \textbf{\textit{add}}\ \{State(s_n)\} \\ \varepsilon_{ok,c_1} \end{array} \right\}$$

The condition-action rule for $decz_1$ *needs to ensure that: i)* $decz_1$ *can be applied only in those states foreseen by* $\mathcal{C}$ *for the decrement of counter* 1*, and only when counter* 1 *is zero; ii) the unique parameter is bound to the next state as foreseen by* $\mathcal{C}$*. Specifically, we get:*

$$Top_1(bot) \wedge \bigvee_{\langle s,1-,s'\rangle\in\Pi} State(s) \wedge y = s' \mapsto decz_1(y)$$

where zero testing is reconstructed by checking that the special bot element is at the top of the first stack.

In the nonzero case, decrement is applied by removing the element at the top of the first stack, i.e., the rightmost element in the chain formed by Next. This is done as follows:

$$decnz_1(s_n):$$
$$\left\{ \begin{array}{rl} Top_1(t_o) \wedge Next(t_n, t_o) \rightsquigarrow & \textbf{\textit{del}} \ \{Next(t_n, t_o), Top_1(t_o)\}, \\ & \textbf{\textit{add}} \ \{Top_1(t_n)\} \\ State(s_o) \rightsquigarrow & \textbf{\textit{del}} \ \{State(s_o)\}, \textbf{\textit{add}}\{State(s_n)\} \\ \varepsilon_{ok,c_1} & \end{array} \right\}$$

The condition-action rule for $decnz_1$ needs to ensure that: i) $decnz_1$ can be applied only in those states foreseen by $\mathcal{C}$ for the decrement of counter 1, and only when counter 1 is non-zero; ii) the unique parameter is bound to the next state as foreseen by $\mathcal{C}$. Specifically, we get:

$$\exists o.Next(bot, o) \wedge \bigvee_{\langle s, 1-, s' \rangle \in \Pi} State(s) \wedge y = s' \mapsto decnz_1(y)$$

where non-zero testing is reconstructed by checking that the special bot element has at least one element on its right in the chain, i.e., whether the first stack is non-empty.

Increment and decrement for counter 2 mirror the three actions used for counter 1, but using Top_2 in place of Top_1, and $Next^-$ in place of Next.

We finally introduce a special action raiseWin that raises the Win flag when the computation reaches the halting state, i.e., $State(s_f)$ is contained in the current ABox,

and at the same time the computation is still judged as correct, i.e., $Ok(c_0)$ is contained in the current ABox. [2]

The formalization of raiseWin together with its condition-action rule is then as follows:

$$State(s_f) \wedge Ok(c_0) \mapsto raiseWin()$$

$$raiseWin() : \{\mathsf{true} \rightsquigarrow \boldsymbol{add}\{Win(c_o)\}\}$$

It is now easy to show that each run of $\mathcal{K}_\mathcal{C}$ simulates a run of $\mathcal{C}$, and that each run of $\mathcal{C}$ is simulated by infinitely many runs in $\mathcal{K}_\mathcal{C}$ (each one differing from the others only in terms of the actual objects inserted in the stacks, but equivalent as far as the size of the stacks is concerned). This, in turn, means that $\mathcal{C}$ halts if and only if the plan existence problem for eKAB $\mathcal{K}_\mathcal{C}$ and the ground, atomic goal $Win(c_0)$ has a positive answer. It is immediate to check that $\mathcal{K}_\mathcal{C}$ and this goal satisfy the requirements of the theorem.

This result can be further extended to an even more restrictive type of eKABs, where not only the TBox is empty and actions/rules employ solely UCQs, but actions only have a single, non-conditional effect of the type $True \rightsquigarrow$ *effects*. This specific setting transforms eKABs to a STRIPS-style planning domain which is still *undecidable* [Erol et al.,

[2] Notice that the very last action could set as the top element a non-fresh individual. Anyway, this event doesnt' affect the validity of the evolution of the eKAB, as it is anyway the last step and no further actions can be performed.

1995], as it has infinitely many costants (the domain is infinite), and allows delete effects. To achieve a decidable setting while still preserving the infinite domain, we have to even further restrict the framework to allow only addition effects in the actions[Erol et al., 1995].

Lemma 4.1.3 *Plan existence for eKABs with an infinite object domain is undecidable, even if the goal is a ground CQ, and the input eKAB has: (i) an empty TBox; (ii) actions/rules employing UCQs only; (iii) actions only allow one non-conditional effect.*

These options are, of course, unsatisfactory in light of the proposed goal to use eKABs for planning with knowledge. On the other side, we also do not want to fall back to resorting to a finite domain, as such setting is trivial and doesn't add anything to the well studied area of planning. We thus attack undecidability by focusing on infinite-domain eKABs that are *state-bounded* [Bagheri Hariri et al., 2013a, Calvanese et al., 2013c, Belardinelli et al., 2014]. Specifically, an eKAB $\mathcal{K}$ is *b-bounded* if its generated TS $\Upsilon_{\mathcal{K}} = \langle T, \Sigma, s_0, abox, \rightarrow \rangle$ is so that for every state $s \in \Sigma$, we have $|\textsc{adom}(abox(s))| \leq b$, that is, every state (or ABox) of $\Upsilon_{\mathcal{K}}$ contains at most b distinct objects. Note that a b-bounded eKAB still has, in general, infinitely many states, as the domain $\mathcal{C}$ is still infinite, from which one can obtain infinitely many distinct ABoxes, each containing a bounded number of ob-

jects. This makes b-bounded eKABs an interesting and non-trivial setting in which studying plan synthesys.

Following from the verification results by [Calvanese et al., 2013c], we have that for state-bounded eKABs, checking plan existence is not more difficult than in the standard setting of propositional planning [Bylander, 1994].

Theorem 4.1.4 *Plan existence over state b-bounded eKABs is decidable in* PSPACE *in the bound b.*

We observe that while boundedness is undecidable in general (by reduction to checking whether a Turing Machine uses a bounded number of cells on a given input), checking whether an eKAB is bounded *for a given bound* is decidable, as proven in [De Giacomo et al., 2014], although in the Situation Calculus framework. The PSPACE bound on b holds as the complexity of KB satisfiability checking (needed at every step of the computation) is EXPTIME-complete for the Description Logics considered, but data complexity is CONP.

Proof 4.1.3 *Let $\mathcal{K} = \langle \mathcal{C}, \mathcal{C}_0, T, A_0, \Lambda, \Gamma \rangle$ be a b-bounded eKAB such that $\mathcal{C}$ is infinite, and let G be a goal. It can be easily shown that $\mathcal{K}$ is a state-bounded, generic DLDS [Calvanese et al., 2013c]. Thanks to Theorem 2 by [Calvanese et al., 2013c], this in turn means that verification of the μDL_p logic over $\mathcal{K}$ is decidable, and reducible to standard model checking of propositional μ-calculus over*

a finite-state transition system $\Theta_{\mathcal{K}}$ whose size is at most exponential in b. Intuitively, μDL_p is a first-order variant of μ-calculus in which states are queried using ECQs. It is then immediate to check whether $\mathcal{K}$ achieves G by model checking $\Theta_{\mathcal{K}}$ against the μDL_p reachability property $\Phi_G = \mu Z.(G \vee \langle - \rangle Z)$.

As for Theorem 4.1.1, to check Φ_G the TS $\Theta_{\mathcal{K}}$ can be constructed on-the-fly, using space that is polynomial in b.

4.2 Plan Synthesis

We now consider plan synthesis for eKABs. We first introduce Algorithm 1, which is based on the schema of a basic, depth-first progressive planning algorithm from Classical Planning (Section 2.3), that synthesizes a plan through a forward search on the induced graph.

The auxiliary functions initialState($Prob$), goal($Prob$), holds(G, s), and successors($Prob, s$) are self-explicative. Note that the schema above abstracts from the specific interpretation of states, that is, it is applicable no matter how states and actions are represented, once the functions above are instantiated on the case at hand. For instance, for a classical planning problem $\mathcal{P} = \langle \mathcal{D}, s_0, G \rangle$ with $\mathcal{D} = \langle S, A, \rho \rangle$, we have initialState($\mathcal{P}$) $= s_0$, goal($\mathcal{P}$) $= G$, holds(G, s) $=$ true iff $s \in G$, and successors($\mathcal{P}, s$) $= \{\langle a, s' \rangle \mid s' = \rho(s, a)\}$.

Next, we show how this schema can be lifted to handle plan synthesis for eKABs. The lifting we propose results

Algorithm 1: Forward planning algorithm schema

```
Function FindPlan
    input  : A planning problem Prob
    output : A plan that solves Prob, or fail if there is
             no solution
    V := ∅          // Global set of visited states
    return FwSearch(Prob,initialState(Prob),ϵ)
end
Function FwSearch
    input  : A planning problem Prob, the current state
             s, the sequence π of actions that led to s
    output : A plan that solves Prob, or fail if there is
             no solution
    if s ∈ V then
        return fail                                  // Loop!
    end
    V := V ∪ {s}
    if holds(goal(Prob), s) then
        return π
    end
    forall ⟨a, s'⟩ ∈ successors(Prob, s) do
        π_n := FwSearch(Prob, s', π · a)
        if π_n ≠ fail then
            return π_n
        end
    end
    return fail
end
```

in an algorithm that is *correct*, i.e., sound and complete, in the following sense:

1. terminates,
2. preserves plan existence,
3. produces proper plans, i.e., if it does not fail, the returned result corresponds to a proper solution to the input planning problem, and
4. it finds plans that are representative of classes of plans, i.e., the instances used in the plan can be substituted with other "equivalent" ones, thus obtaining infinite valid plans.

As we can see from the point above, the definition of complete differs from the classical one, but this is also an intended consequence of working with a framework where there could be possibly infinite plans.

We stress that, while we present the lifting on Algorithm 1, the same approach applies to any other, possibly optimized, algorithm. Indeed, the lifting strategy is agnostic with respect to how the search space is traversed.

We now consider the two classes of eKABs for which decidability of plan existence has been established in Section 4.1.

4.2.1 Plan Synthesis for eKABs with Finite Domain

This case differs from classical planning in that eKABs have ABoxes as states, and they provide an implicit representation of successor states in terms of actions and

condition-action rules. The former aspect requires to replace propositional entailment with ECQ query answering when checking whether the goal has been reached. The latter requires to use DO to compute a state's successors. Specifically, given an eKAB planning problem $\mathcal{E} = \langle \mathcal{K}, G \rangle$, where $\mathcal{K} = \langle \mathcal{C}, \mathcal{C}_0, T, A_0, \Lambda, \Gamma \rangle$, with finite $\mathcal{C}$, Algorithm 1 can be instantiated as follows:

- INITIALSTATE$(\mathcal{E}) = A_0$;
- GOAL$(\mathcal{E}) = G$;
- HOLDS$(\mathcal{E}, A) =$ ANS(G, T, A);
- SUCCESSORS$(\mathcal{E}, A)$ returns the set of pairs $\langle \alpha\theta, A' \rangle$, where:
  - $\alpha \in \Lambda$,
  - θ is a $\mathcal{K}$-legal parameter substitution in A for α, and
  - $A' =$ DO$(\alpha\theta, A, T)$.

We call the resulting algorithm FINDPLAN-EKABFD. By this instantiation, the search space of FINDPLAN-EKABFD is exactly $\Upsilon_{\mathcal{K}}$, which is finite, so being the eKAB domain. Thus, we have:

Theorem 4.2.1 FINDPLAN-EKABFD *is correct.*

Proof 4.2.1 FINDPLAN-EKABFD *corresponds to a recursive, depth-first visit of the TS* $\Upsilon_{\mathcal{K}}$*, which is finite-state. The visit positively terminates along a branch if the goal query* G *holds in its last state, and negatively terminates along a branch if the branch achieves an already visited*

state (this automatically handles the case of loops), or if the branch is a dead-end.

FindPlan-eKabFD can be directly implemented on top of existing, off-the-shelf planners, provided that the native goal check and successor generation are replaced with ECQ-query answering and the computation of do.

4.2.2 Plan Synthesis for State-Bounded eKABs

We now consider state-bounded eKABs over infinite object domains. In this case, the search space is potentially infinite, thus FindPlan-eKabFD is not readily applicable, as termination is not guaranteed. To tackle this problem, instead of visiting the original, infinite, search space, we work on a finite-state abstraction of the eKAB. We first argue that the execution semantics of eKABs has two properties:

- it is driven by ECQ-query answering, which is generic (cf. Section 2.1.2);
- all allowed configurations of external input parameters are considered when applying an action.

These imply that eKABs are *generic DLDSs* in the sense of [Calvanese et al., 2013c], which, together with state-boundedness, allows us to apply the same abstraction technique used by [Calvanese et al., 2013c]. In particular,

Algorithm 2: Plan synthesis for state-bounded eKABs.

Function FindPlan-eKabSB

input : An eKAB planning problem $\mathcal{E} = \langle \mathcal{K}, G \rangle$, where $\mathcal{K} = \langle \mathcal{C}, \mathcal{C}_0, T, A_0, \Lambda, \Gamma \rangle$ is b-bounded and $\mathcal{C}$ is infinite

output : A plan that solves $\mathcal{E}$, or fail if there is no solution

$n := max\{k \mid \text{there is } \alpha \in \Lambda \text{ with } k \text{ parameters}\}$

pick $\widehat{\mathcal{C}}$ s.t. $\begin{cases} \mathcal{C}_0 \subseteq \widehat{\mathcal{C}} \subset \mathcal{C} \\ |\widehat{\mathcal{C}}| = b + n + |\mathcal{C}_0| \end{cases}$ `// Abstract dom.`

$\widehat{\mathcal{K}} := \langle \widehat{\mathcal{C}}, \mathcal{C}_0, T, A_0, \Lambda, \Gamma \rangle$

return FindPlan-eKabFD($\langle \widehat{\mathcal{K}}, G \rangle$)

reachability is preserved if the infinite-state TS induced by the input eKAB is shrunk into a finite-state TS over a finite, but sufficiently large, object domain. We leverage this technique as the core of the FindPlan-eKabSB procedure shown in Algorithm 2, which reduces the original planning problem over an infinite search space to a classical planning problem.

It can be checked that all correctness conditions are satisfied: i) is a consequence of Theorem 4.2.1; ii) holds because the finite-state abstraction preserves reachability; iii) holds because the search space of the finite-state abstraction is contained into that of the original eKAB. Thus, we have:

Theorem 4.2.2 FindPlan-eKabSB *is correct.*

Proof 4.2.2 *We separately consider the two conditions for correctness.*

As for the preservation of plan existence, we notice that, by Lemma 3.3.1, the input eKAB $\mathcal{K}$ is a generic, state-bounded DLDS. Hence, Theorem 2 in [Calvanese et al., 2013c] applies to $\mathcal{K}$, telling us that model checking a μDL_p property over $\mathcal{K}$ can be reduced to model checking μDL_p over a finite abstraction of $\mathcal{K}$. This abstraction is produced by considering the same DLDS, but with a finite, sufficiently large domain. The required size of this domain is exactly the one used in FindPlan-eKabSB *to pick the object domain $\widehat{\mathcal{C}}$. Consequently, the application of* FindPlan-eKabFD *to $\widehat{\mathcal{K}}$ leads to generate the desired abstraction. Since the existence of a plan achieving G can be formulated in μDL_p (cf. the proof of Theorem 4.1.4), the result follows.*

As for the generation of proper plans, it is sufficient to observe that, since $\widehat{\mathcal{C}} \subset \mathcal{C}$, every run explored by FindPlan-eKabFD *over $\widehat{\mathcal{K}}$ is actually also a run for the original input eKAB $\mathcal{K}$. Finally, the results returned by* FindPlan-eKabFD *over $\widehat{\mathcal{K}}$ are correct due to Theorem 4.2.1.*

Example 7 *Consider again the eKAB $\mathcal{K}$ in Example 5, together with goal G in Example 6. Notice that $\mathcal{K}$ is state-bounded, as the only way to increase the number of objects present in the system is by hiring someone, but the number*

of hirings is in turn bounded by the number of company branches (which is fixed once and for all in the initial state) as the rule that activates the action hireEng *is:*

$$\mathsf{Branch}(b) \wedge \neg[\exists x.\mathsf{Engineer}(x) \wedge \mathsf{worksIn}(x, b)] \mapsto \mathsf{hireEng}(x, b)$$

Now assume that the initial state indicates the known existence of a main and a subsidiary branch for the company, as well as the fact that task t *has an assigned technician from the main branch:*

$$A_0 = \{\mathsf{Branch}(\mathsf{main}), \mathsf{Branch}(\mathsf{sub}), \mathsf{Technician}(123), \\ \mathsf{worksIn}(123, \mathsf{main}), \mathsf{worksOn}(123, \mathsf{t})\}$$

A possible plan leading from A_0 *to a state where* G *holds is*

$$\pi_1 = \mathsf{hireEng}(452, \mathsf{sub})\ \mathsf{makeResp}(\mathsf{t}, 452)$$

This plan achieves G *by hiring an engineer in a different branch from that of* 123*, with whom the engineer shares task* t*. Observe, again, the interplay between the actions and the TBox: while no action explicitly indicates that* 452 *works on task* t*, this is implicitly obtained from the fact that such an engineer is made responsible for* t*.*

Another, quite interesting plan achieving G *is:*

$$\pi_2 = \mathsf{hireEng}(521, \mathsf{main})\ \mathsf{makeResp}(\mathsf{t}, 521)\ \mathsf{Anonymize}(521)$$

In this case, an engineer is hired in the same branch of technician 123*, and made responsible for task* t*. These two actions do not suffice to achieve* G*, since the planning agent*

knows that the two employees work in the same branch. This knowledge is somehow "retracted" by anonymizing the hired engineer: after the execution of Anonymize(521)*, the planning agent still knows that* 521 *must work in some branch (this is enforced by a dedicated TBox axiom), but does not know which one (due to the open-world semantic, in one model it could work in the main branch, while in another it could be related to an unknown branch), thus satisfying also the (epistemic) negative part of the goal.*

4.2.3 Plan Templates and Online Instantiation

FindPlan-eKabSB returns plans with ground actions mentioning only objects in $\widehat{\mathcal{C}}$, as those in $\mathcal{C} \setminus \widehat{\mathcal{C}}$ are not used in $\widehat{\mathcal{K}}$. Such plans can be regarded as *templates* from which we can obtain regular plans for $\mathcal{K}$. This is a consequence of genericity, which yields that a plan keeps achieving a goal even under consistent renaming of the objects it mentions.

To formalize this intuition, we recast the notion of equality commitment [Calvanese et al., 2013c] in this setting. Let B be a set of objects, and I a set of external input parameters. An *equality commitment* H over a *finite* set $S \subseteq B \cup I$ is a partition $\{H_1, \ldots, H_n\}$ of S s.t. each H_j contains at most *one* object from B. A substitution $\theta : I \to B$ is *compatible* with H if:

- for every pair of parameters $i_1, i_2 \in I$, we have that $i_1\theta = i_2\theta$ iff i_1 and i_2 belong to the same H_j;

- whenever H_j contains an object $d \in B$, then $i\theta = d$ for every $i \in H_j$.

Intuitively, θ is compatible with H if it maps parameters from the same class into the same object, and parameters from different classes into distinct objects. Finally, given a finite set $B' \subset B$ and a substitution $\theta : I \to B$, fix an ordering $O = \langle d_1, \ldots, d_n \rangle$ over the set $B_{cur} = B' \cup \text{IM}(\theta)$ of the objects that are either mentioned in B' or assigned to I by θ. The equality commitment H *induced by* θ *over* B' (under O) is the partition $H = \{H_1, \ldots, H_n\}$, s.t., for $j \in \{1, \ldots, n\}$:

- H_j contains d_j iff $d_j \in B'$, i.e., objects mentioned by θ but not present in B' are discarded;

- H_j contains $i \in \text{DOM}(\theta)$ iff $\theta(i) = d_j$, i.e., all parameters mapped to the same, j-th object are included in the j-th equivalence class.

Example 8 *Let* $B = \{d_1 \ldots, d_5\}$, $B' = \{d_1, d_2, d_3\}$, $I = \{i_1, \ldots, i_4\}$, *and* $\theta : I \to B$, *s.t.:* $\theta(i_1) = d_1, \theta(i_2) = \theta(i_3) = d_5, \theta(i_4) = d_4$. *The equality commitment induced by* θ *over* $B \cup \{d_4, d_5\}$ *is* $H = \{H_1, \ldots, H_5\}$, *where:* $H_1 = \{d_1, i_1\}, H_2 = \{d_2\}, H_3 = \{d_3\}, H_4 = \{i_4\}, H_5 = \{i_2, i_3\}$. *Notice that* $H_i \subseteq B$.

We can now state the following key result.

Lemma 4.2.3 *Let* $\mathcal{K} = \langle \mathcal{C}, \mathcal{C}_0, T, A_0, \Lambda, \Gamma \rangle$ *be an eKAB, and* A_1, A_1' *ABoxes over* T, *s.t.* $A_1 \cong_T^h A_1'$ *for some object*

renaming h. Let $\alpha(\vec{p}, \vec{i})$ be an action in Λ with external input parameters $\vec{i}$, and θ a $\mathcal{K}$-legal substitution in A_1 for α. Let H be the equality commitment induced by θ over $\textsc{adom}(A_1)$, and $H' = h(H)$ the equality commitment obtained from H by renaming each $d \in \textsc{adom}(A_1)$ as $h(d) \in \textsc{adom}(A_2)$. If θ' is a parameter substitution for $\vec{p}$ and $\vec{i}$ compatible with H', then:

- *θ' is a $\mathcal{K}$-legal parameter for α in A'_1;*
- *$\textsc{do}(\alpha\theta, A_1, T) \cong^{h'}_{T} \textsc{do}(\alpha\theta', A'_1, T)$, where h' extends h as follows: for every parameter i of α s.t. $\theta(i) \notin \textsc{adom}(A_1)$, we have $h'(\theta(i)) = \theta'(i)$.*

Proof 4.2.3 *The proof is directly obtained from the definition of* DO*, and the fact that eKABs are generic.*

Intuitively, Lemma 4.2.3 states that, modulo object renaming consistent with a parameter substitution that induces the same equality commitment, the same action can be applied to two logically equivalent ABoxes. Furthermore, such action induces the same update, modulo renaming of the objects mentioned in the two ABoxes and the involved parameters. In Algorithm 3, we exploit this result to build, in an online fashion, a plan for $\mathcal{K}$ starting from one for $\widehat{\mathcal{K}}$. This provides the freedom of dynamically choosing which actual objects to use when actions are executed, provided that the choice induces the same equality commitment induced by the parameter substitution in the original plan. By Lemma 4.2.3, we obtain:

Algorithm 3: Online instantiation of a plan template.

Procedure ONLINEEXEC

input : An eKAB $\mathcal{K} = \langle \mathcal{C}, \mathcal{C}_0, T, A_0, \Lambda, \Gamma \rangle$, and a plan $\pi = \alpha_1\theta_1 \cdots \alpha_m\theta_m$

$A_o := A_0$ `// old, effective state`

$A_n := A_0$ `// current, effective state`

$h : \mathcal{C}_0 \to \mathcal{C}_0$ s.t. $h(d) = d$ for each $d \in \mathcal{C}_0$ `// cur. bijection`

for $k \in \{1, \ldots, m\}$ **do**

$H :=$ eq. commitment induced by θ_i over $\text{ADOM}(A)$

pick θ'_k that is compatible with $h(H)$ `// Agent choice`

$A'_o := \text{DO}(\alpha_i\theta_k, A_o, T)$

$A'_n := \text{DO}(\alpha_i\theta'_k, A_n, T)$

$h_n : \mathcal{C}_0 \cup \text{ADOM}(A_o) \to \mathcal{C}_0 \cup \text{ADOM}(A_n)$ s.t.

$$\begin{cases} h_n(d) = d, & \text{for } d \in \mathcal{C}_0 \\ h_n(d) = h(d), & \text{for } d \in \text{ADOM}(A'_o) \cap \text{ADOM}(A_o) \\ h_n(\theta_k(i)) = \theta'_k(i), & \text{for } i \text{ parameter of } \alpha_k \\ & \text{s.t. } \theta_k(i) \notin \text{ADOM}(A_o) \end{cases}$$

$h := h_n$, $A_o := A'_o$, $A_n := A'_n$

end

Theorem 4.2.4 *Let $\mathcal{E} = \langle \mathcal{K}, G \rangle$ be an eKAB planning problem. If π is a plan that achieves G, then* ONLINEEXEC$(\mathcal{K}, \pi)$ *is guaranteed to achieve G for each possible choice.*

Proof 4.2.4 *Each iteration in* ONLINEEXEC *obeys to the assumption of Lemma 4.2.3. As a consequence, at each*

iteration we have that $A'_o \cong^{h_n}_T A'_n$. *This also holds for the last pair of ABoxes that, by definition, are both guaranteed to satisfy* G.

Example 9 *Consider plan* π_2 *of Example 7. This plan can be lifted online by the planning agent as follows: when hiring the engineer, the planning agent can freely inject a fresh employee identifier in place of* 521*, provided that the chosen identifier is then consistently used in the subsequent actions. In other words,* π_2 *acts as a footprint for the infinite family of plans of the form*

$$\mathsf{hireEng}(Id, \mathsf{main})\ \mathsf{makeResp}(\mathsf{t}, Id)\ \mathsf{Anonymize}(Id)$$

where Id is selected on-the-fly when the planning agent executes hireEng*, and is s.t. it is different from all the objects present in* A_0 *(this reconstructs the same equality commitment as in the case of* 521*). All such infinite plans are guaranteed to achieve* G.

Plan Synthesis for Lightweight eKABs
5

Given the planning techniques devised in Section 4.2, we now want to take a step further, and see if and how standard planning techniques, with no modifications, could be applied to eKABs. This goal is interesting for different reasons: under a theoretical point of view, this would allow to apply to planning problems expressed through eKABs the same optimizations (such as heuristic functions) that have been widely studied in the planning community, while under a practical point of view, it would mean to be able to use off-the-shelf planners.

Closing the gap we just mentioned, requires a restric-

tion in the DL chosen to express the eKAB. We consider plan synthesis for state-bounded eKABs over the lightweight DL *DL-Lite*$_{\mathcal{A}}$ (see Section 2.2), and devise two techniques to translate an eKAB planning problem into, respectively, a STRIPS-style (Section 2.3.1) and an ADL planning problem (Section 2.3.2).

5.1 Translation to STRIPS

The first translation we introduce aims at encoding an eKAB planning problem into a corresponding STRIPS planning problem. To do so, we have to define some limitations to the expresivness of eKABs, as the original framework cannot be matched to the expressivity of STRIPS.

The first main limitation is on the domain, as STRIPS planning problems only deal with finite domains of individuals; this is easily overcome by considering only eKABs that have either a finite domain or are state bounded, and thus can be reduced to an equivalent finite domain eKAB (Section 4.2.2).

Secondly, since in STRIPS, FOL formulae are directly *evaluated* over FOL structures (without any form of ontological reasoning), we have to suitably consider the contribution of T. Indeed (see Section 3), T is used both during query answering and to check whether the ABox resulting from an action instance is T-consistent. This restricts the choice of the DL languages we can use to represent the eKAB; it must be a *lightweight* DL Language

that allows somehow to compile away the TBox contributions. We tackle this problem by relying on *DL-Lite*$_\mathcal{A}$'s *FO rewritability* of both ECQs and T-consistency checks (see Section 2.2).

The last limitation we impose is on the definition of actions and their related rules. Since in STRIPS actions only have one non-conditional effect, which is instantiated by considering only one of the possible substitutions coming from the precondition, we now limit eKABs in the following way:

- actions can only have one, non-conditional effect;
- the precondition of the effect must be the boolean value true, meaning it is always activated.

In the following, we will refer to an eKAB that abides to such limitations as a reKAB $\mathcal{K}_r$.

We thus define a syntactic, modular translation procedure leKab2Strips that takes as input a *DL-Lite*$_\mathcal{A}$-reKAB planning problem $\mathcal{E}_r = \langle \mathcal{K}_r, G \rangle$ with $\mathcal{K}_r = \langle \mathcal{C}, \mathcal{C}_0, T, A_0, \Lambda, \Gamma \rangle$, and produces a corresponding STRIPS planning problem $\mathcal{P}_\mathcal{K} = \langle \mathcal{C}_\mathcal{K}, \mathcal{C}_0, \mathcal{F}_\mathcal{K}, \mathcal{A}_\mathcal{K}, \varphi_\mathcal{K}, \psi_G \rangle$.

Object domain. As in Section 4.2, if $\mathcal{C}$ is finite, so is $\mathcal{C}_\mathcal{K}$. If instead $\mathcal{C}$ is infinite but $\mathcal{K}$ is b-bounded, we fix $\mathcal{C}_\mathcal{K}$ to contain $\mathcal{C}_0$ plus $n + b$ objects from $\mathcal{C}$.

Fluents. $\mathcal{F}_\mathcal{K}$ is obtained by encoding concept and role names in T into corresponding unary and binary fluents. It also contains two special nullary fluents: *ChkCons*, distinguishing *normal execution modality* from *check consistency modality* of $\mathcal{P}_\mathcal{K}$, and *Error*, marking when the con-

sistency check fails.

Operators. $\mathcal{A}_\mathcal{K}$ is obtained by transforming every action in Λ, with its related condition-action rule in Γ, into a STRIPS operator: The condition of the rule is used as the precondition, while the single action's effect defines the effects in the STRIPS operator (since the effect's condition is the boolean value true, it is simply ignored).

To embed the T-consistency checks into STRIPS, we force $\mathcal{A}_\mathcal{K}$ to alternate between two phases: the *normal execution modality*, where a "normal" STRIPS operator is applied, mirroring the execution of an action instance of $\mathcal{K}$; and the *check consistency modality*, where a special STRIPS operator checks if the obtained state is T-consistent. Alternation is realized by toggling the fluent *ChkCons*, and activating *Error* when the consistency check fails, thus blocking operator application.

We now detail how a "normal" action is translated. Technically, consider an action $\alpha = a(\vec{p}) : \{e_1\}$ in Λ and its corresponding condition-action rule $Q_\alpha(\vec{x}) \mapsto a(\vec{p})$ in Γ. Let $\vec{z} = \vec{p} \setminus \vec{x}$. We produce a corresponding STRIPS operator $a = \langle \rho_\alpha, \varepsilon_\alpha \rangle$. Its precondition ρ_α corresponds to the FO formula $\mathit{rew}(Q_\alpha(\vec{x}), T) \wedge \neg \mathit{ChkCons} \wedge \neg \mathit{Error}$, which leaves the external input parameters $\vec{z}$ unconstrained . Notice how we embed the contribution T during query answering by rewriting the condition ECQ ($\mathit{rew}(Q_\alpha(\vec{x}), T)$).

The operator effect ε_α is the FO conjunction of the translation of $e_1 = \mathsf{true} \rightsquigarrow \mathbf{add}\ F_1^+, \mathbf{del}\ F_1^-$, which generates the following conjunct:

$$\bigwedge_{P_j(\vec{p},\vec{x_i})\in F_1^+} P_j(\vec{p},\vec{x_i}) \wedge \bigwedge_{P_k(\vec{p},\vec{x_i})\in F_1^-} \neg P_k(\vec{p},\vec{x_i}) \wedge \mathit{ChkCons}$$

Finally, the special STRIPS operators used to check T-consistency are:

- $\mathit{checkOk} = \langle \rho_{\mathit{chkOk}}, \varepsilon_{\mathit{chkOk}} \rangle$, where: $\rho_{\mathit{chkOk}} = \mathit{ChkCons} \wedge \neg Q^T_{\text{unsat}}$ checks whether the check flag is on and if is true, and check if the state is consistent ($\neg Q^T_{\text{unsat}}$ has to be true); $\varepsilon_{\mathit{chkOk}} = \neg \mathit{ChkCons}$ takes care of toggling off the check flag. In other words, *checkOk* activates its effects only if there are no inconsistencies;

- $\mathit{checkNo} = \langle \rho_{\mathit{chkNo}}, \varepsilon_{\mathit{chkNo}} \rangle$, is almost the same as *checkOk*, but activates its effects if there are no inconsistencies, thus requiring to trigger the error flag. The precondition and effects are, respectively, $\rho_{\mathit{chkNo}} = \mathit{ChkCons} \wedge Q^T_{\text{unsat}}$ and $\varepsilon_{\mathit{chkNo}} = \neg \mathit{ChkCons} \wedge \mathit{Error}$.

Initial state specification. $\varphi_{\mathcal{K}}$ is by constructing the conjunction of all facts contained in A_0: $\varphi_{\mathcal{K}} = \bigwedge_{P_i(\vec{o})\in A_0} P_i(\vec{o})$.

Goal description. The goal description ψ_G is obtained from goal G in $\mathcal{E}$ as $\psi_G = \mathit{rew}(G,T) \wedge \neg \mathit{ChkCons} \wedge \neg \mathit{Error}$.

The two-fold contribution of T is taken care of, as in the operators, by rewriting G (via $\mathit{rew}(G,T)$) and ensuring that the ending state is T-consistent (by requiring the absence of the consistency check and error flags in ψ_G).

We close by considering the following algorithm, called FindPlan-leKabStrips:

1. take as input a *DL-Lite*$_\mathcal{A}$-eKAB planning problem $\mathcal{E}$;
2. translate $\mathcal{E}$ into an STRIPS planning problem using leKab2Strips;
3. invoke an off-the-shelf STRIPS planner;
4. if the planner returns fail, return fail as well;
5. if the planner returns a plan π, filter away all *check* operators from π, and return it as a result.

Theorem 5.1.1 FindPlan-leKabStrips *is correct.*

Proof 5.1.1 *The proof is given in two steps. In the first step, we show that, thanks to FO-rewritability, each DL-Lite*$_\mathcal{A}$*-eKAB planning problem (of which reKABs are a subset) can be transformed into an equivalent planning problem whose DL-Lite*$_\mathcal{A}$*-eKAB has an empty TBox, and employs standard FO query evaluation to progress the ABox. In the second step, we show that such an "intermediate" reKAB problem is correctly mirrored by the STRIPS translation obtained through* FindPlan-leKabStrips*.*

Consider a DL-Lite$_\mathcal{A}$*-eKAB planning problem* $\mathcal{E} = \langle \mathcal{K}, G \rangle$ *with* $\mathcal{K} = \langle \mathcal{C}, \mathcal{C}_0, T, A_0, \Lambda, \Gamma \rangle$*. As pointed out before, we assume* $T = T_p \uplus T_n \uplus T_f$*. We construct a corresponding DL-Lite*$_\mathcal{A}$*-eKAB planning problem* $eKABprob_r = \langle \mathcal{K}_r, G_r \rangle$ *as follows. The eKAB* $\mathcal{K}_r$ *is obtained from* $\mathcal{K}$ *by rewriting all ECQs, so as to compile away the positive assertions of the TBox* T*. Specifically,* $\mathcal{K}_r = \langle \mathcal{C}, \mathcal{C}_0, T_n \uplus T_f, A_0, \Lambda_r, \Gamma_r \rangle$*, where:*

- *For each action* $a(\vec{p}) : \{e_1, \ldots, e_n\}$ *in* Λ, Λ_r *contains a corresponding action* $a_r(\vec{p}) : \{e_1^r, \ldots, e_n^r\}$, *where each effect* e_i *of the form* $Q_i(\vec{p}, \vec{x}_i) \rightsquigarrow \textbf{\textit{add}}\, F_i^+, \textbf{\textit{del}}\, F_i^-$ *becomes a corresponding effect* $rew(Q_i(\vec{p}, \vec{x}_i), T) \rightsquigarrow \textbf{\textit{add}}\, F_i^+, \textbf{\textit{del}}\, F_i^-$.
- *For each condition-action rule* $Q_a(\vec{x}) \mapsto a(\vec{y})$ *in* Γ, Γ_r *contains a corresponding condition-action rule* $rew(Q_a(\vec{x}), T) \mapsto a_r(\vec{y})$.

Similarly to the case of the eKAB, the goal G_r *corresponds to* $rew(G, T)$.

Let A *be an ABox, and* $\theta : \vec{y} \to \mathcal{C}$ *be a parameter assignment for actions* $a(\vec{y})$ *and* $a_r(\vec{y})$. *The following two key properties hold:*

1. $a(\vec{y})\theta$ *is a* $\mathcal{K}$*-legal action instance in* A *if and only if* $a_r(\vec{y})\theta$ *is a* $\mathcal{K}_r$*-legal action instance in* A.
2. *If* θ *is a* $\mathcal{K}$*-legal parameter substitution in* A *for* $a(\vec{y})$ *(equivalently,* θ *is a* $\mathcal{K}_r$*-legal parameter substitution in* A *for* $a_r(\vec{y})$*), then* $\textsc{do}(a(\vec{y})\theta, A, T) = \textsc{do}(a_r(\vec{y})\theta, A, \emptyset)$.

In fact, from FO-rewritability we directly have that $\theta[\vec{x}] \in \textsc{ans}(Q_a, T, A)$ *if and only if* $\theta[\vec{x}] \in \textsc{ans}(rew(Q_a, T), T_n \uplus T_f, A)$ *if and only if* $\theta[\vec{x}] \in \textsc{ans}(rew(Q_a, T), \emptyset, A)$ *(recall that disjointness and functionality assertions do not participate in query answering, but only matter for* T-

consistency). Furthermore, we have that

$$\begin{aligned}
&\textsc{do}(a_r(\vec{y})\theta, A, T_n \uplus T_f)\\
=\ &\textsc{do}(a_r(\vec{y})\theta, A, \emptyset)\\
=\ &(A \setminus \bigcup_{i\in\{1,\ldots,n\}} \bigcup_{\sigma_j\in \textsc{ans}(rew(A,T),\emptyset,Q_i\theta)} F_i^+\sigma_j)\\
&\cup \bigcup_{i\in\{1,\ldots,n\}} \bigcup_{\sigma_j\in \textsc{ans}(rew(A,T),\emptyset,Q_i\theta)} F_i^-\sigma_j\\
=\ &(A \setminus \bigcup_{i\in\{1,\ldots,n\}} \bigcup_{\sigma_j\in \textsc{ans}(A,T,Q_i\theta)} F_i^+\sigma_j)\\
&\cup \bigcup_{i\in\{1,\ldots,n\}} \bigcup_{\sigma_j\in \textsc{ans}(A,T,Q_i\theta)} F_i^-\sigma_j\\
=\ &\textsc{do}(a(\vec{y})\theta, A, T)
\end{aligned}$$

Since T is a DL-Lite$_\mathcal{A}$ TBox, we finally obtain that $\textsc{do}(a(\vec{y})\theta, A, T)$ *is T-consistent if and only if it is $(T_n \uplus T_f)$-consistent if and only if* $\textsc{do}(a_r(\vec{y})\theta, A, T_n \uplus T_f)$ *is $(T_n \uplus T_f)$-consistent. This concludes the proof of claims (1) and (2) above.*

From such two claims, we directly obtain, by induction, that $\mathcal{K}$ and $\mathcal{K}_r$ generate identical TSs, i.e., $\Upsilon_\mathcal{K} = \Upsilon_{\mathcal{K}_r}$. Hence, the two DL-Lite$_\mathcal{A}$-eKAB planning problems $\langle \mathcal{K}, G\rangle$ and $\langle \mathcal{K}_r, G_r\rangle$ are equivalent.

Let us now consider the translation of the planning problem $\langle \mathcal{K}, G\rangle$, with $\mathcal{K}$ a reKAB, into the corresponding STRIPS planning $\mathcal{P}_\mathcal{K} = \langle \mathcal{C}_\mathcal{K}, \mathcal{C}_0, \mathcal{F}_\mathcal{K}, \mathcal{A}_\mathcal{K}, \varphi_\mathcal{K}, \psi_G\rangle$, as defined by FindPlan-leKabStrips. *It is easy to see that this translation is the same as the one obtained from $\langle \mathcal{K}_r, G_r\rangle$.*

Furthermore, the evolutions induced by $\mathcal{K}_r$ are obtained by applying standard FO query evaluation, by considering the underlying ABox as a database of facts, interpreted with the closed-world assumption. Hence, query answering for $\mathcal{K}_r$ is the same as that adopted by STRIPS for $\mathcal{P}_\mathcal{K}$

(identical to $\mathcal{P}_{\mathcal{K}_r}$). Furthermore $\mathcal{P}_{\mathcal{K}}$ represents a direct syntactic reformulation of $\mathcal{K}_r$ in STRIPS, with the only key difference that $\mathcal{K}_r$ checks whether the ABox resulting from the application of an action instance is $(T_n \uplus T_f)$-consistent, while $\mathcal{P}_{\mathcal{K}}$ simulates this check by introducing the two-step approach discussed in the translation.

Let A be an ABox, and $\alpha\theta$ be a $\mathcal{K}_r$ action instance, with $\alpha = a(\vec{y})$. Let D be the STRIPS representation of A (i.e., the formula that corresponds to the conjunction of facts in A), and $\langle a, \vec{y}, \rho_\alpha(\vec{y}), \varepsilon_\alpha(\vec{y})\rangle$ the STRIPS operator corresponding to α. We have three cases to discuss for θ.

If θ is not an answer to the guard of the condition-action rule for α in A, then $\alpha\theta$ is not applicable. In this case, $\rho_\alpha(\vec{y})\theta$ does not hold in D, and hence the STRIPS operator corresponding to α, grounded with θ, is not applicable in D neither.

If instead θ is an answer to the guard of the condition-action rule for α in A, we have two sub-cases to discuss, corresponding to the situation where $\textsc{do}(\alpha\theta, A, \emptyset)$ is $(T_n \uplus T_f)$-consistent, and that where it is not.

In the first sub-case, $\alpha\theta$ is $\mathcal{K}_r$-legal in A. In this situation, we have have that D' is the STRIPS representation of A', where:

- *$A' = \textsc{do}(\alpha\theta, A, \emptyset)$.*
- *D' results from D by applying the sequence of the STRIPS operator $\langle a, \vec{y}, \rho_\alpha(\vec{y}), \varepsilon_\alpha(\vec{y})\rangle$ instantiated with θ, followed by the application of the STRIPS operator $\langle checkOk, \emptyset, \rho_{chk}, \varepsilon_{chk}\rangle$, which has the only effect of removing the check flag. In fact, thanks to the*

FO-rewritability of consistency check in DL-Lite$_\mathcal{A}$, we have that Q^T_{unsat} holds in D' if and only if A' is $(T_n \uplus T_f)$-inconsistent.

Hence, the progression induced by $\alpha\theta$ corresponds to that induced by the sequence of the corresponding STRIPS operator, followed by the check operator.

In the second sub-case, $A' = \textsc{do}(\alpha\theta, A, \emptyset)$ is $(T_n \uplus T_f)$-inconsistent, and hence $\alpha\theta$ is not is not applicable in A. In STRIPS, instead, the operator $\langle a, \vec{y}, \rho_\alpha(\vec{y}), \varepsilon_\alpha(\vec{y})\rangle$, instantiated with θ, is applicable, and leads to a state D' in which only the check operator $\langle checkNo, \emptyset, \rho_{chk}, \varepsilon_{chk}\rangle$ can be executed. Since $D' \setminus \{ChkCons\}$ corresponds to A', which is $(T_n \uplus T_f)$-inconsistent, the application of the check operator leads to introduce the error flag, i.e., leads to the STRIPS state $D'' = (D' \setminus \{ChkCons\}) \cup \{Error\}$. The error flag is never removed, and therefore this spurious path cannot lead to satisfy the desired STRIPS goal $\psi_G = G_r \wedge \neg ChkCons \wedge \neg Error$.

By induction, we consequently obtain that non-spurious runs induced by the STRIPS operators in $\mathcal{P}_\mathcal{K}$ exactly replicate those in $\Upsilon_{\mathcal{K}_r}$, if one projects away the intermediate check states (which do not affect the possibility to satisfy the given goal).

5.1.1 Action Rewriting

We now devise a technique to remodel actions in order to remove the need of the operators *checkOk* and *checkNo*. We note that this is essentially a technical discussion, and no correctness theorems are stated.

The technique builds upon the work presented in [Stawowy, 2015], and, informally speaking, embeds the satisfiability check provided by Q^T_{unsat} inside the rules' precondition; it does so by analysing all possible inconsistencies that could arise by performing the action related to the rule, tailoring Q^T_{unsat} to consider only those specific cases, and add the resulting control query Q_{check} to the condition of the rule.

Before proceeding, though, we have to notice that the aforementioned additional control query could be asked to "judge" the contribution of any of the parameters used in the effects; this could raise an invalid query in the condition-action rule, as the parameters $\vec{x}$ available in the rule are (possibly) a subset of the action's parameters $\vec{p}$, leaving out the external input parameters. In order to be able to address them in the condition-action rule, we have to translate the given reKAB $\mathcal{K}$ into an equivalent one $\mathcal{K}_{Dom}$ where we modify the semantic of the actions in the following way:

1. we introduce the special concept Domain in the KB, and add to it all the individuals of the domain (if it is finite, otherwise if is infinite but state-bounded, we calculate the equivalent finite domain), thus giving

a representation of the domain directly inside the ABox. Such concept is immutable, thus cannot be used in the definition of the effects;

2. we modify each condition-action rule γ_α to incorporate the external input parameters assignment of the related action α. For each external input p_{ext} in $\vec{p}$, we add to the condition of the rule the CQ $\mathsf{Domain}(p_{ext})$.

With this modifications, we effectively move the assignment of the external input parameters into the condition-action rule, and link them, as in the original reKAB $\mathcal{K}$, to a random value in the domain through its "image" Domain in the ABox. We can thus conclude that the reKABs $\mathcal{K}$ and $\mathcal{K}_{Dom}$ are equivalent (they generate the same TS), and resume the building of Q_{check}.

By definition, Q^T_{unsat} checks for all possible inconsistencies by means of queries. When we perform an instantiated action $\alpha\vartheta$ in state A and get the successor state A_{next}, we can be sure that the only source of inconsistencies comes from the set of instantiated positive effects $F^+\vartheta$ of the action α; this is easily proved by the fact that, if we assume the state A to be consistent, then removing statements from it (as the set of negative effects F^- does), does not alter its consistency. We thus concentrate on the set F^+, without any substitution ϑ, as the technique must be applicable in every situation and for every substitution.

We consider the rule γ related to the action α, and extend its condition's ECQ by adding the query Q_{check}, which is created as follows (we add a step-by-step example

along). We consider Q^T_{unsat} to be saturated with respect to the positive axioms of the TBox (we can achieve this by applying the query reformulation to Q^T_{unsat}). For every atomic effect $(f_i^+, \vec{x}_{f+}) \in F^+$ (e.g., $\mathsf{A}(y)$):

1. check if the term f_i^+ appears in any CQ q_u in Q^T_{unsat} (we denote with $\vec{x}_q$ the vector of existential variables that appear in q_u, as every CQ in Q^T_{unsat} is boolean). E.g., we have $q_u = \exists x.\mathsf{A}(x) \wedge \mathsf{B}(x)$;
2. given q_u, we proceed to unify f_i^+ with one atom in q_u with the same term as f_i^+ (it could be case of functional axioms, where the related CQ q_u contains two atoms using the same term). E.g., $q_u[x/y] = \mathsf{A}(y) \wedge \mathsf{B}(y)$;
3. we substitute f_i^+ in q_u with the vale true, as it is the added effect and thus forcibly evaluates always to true. E.g., $q_u[x/y] = \mathsf{true} \wedge \mathsf{B}(y)$;
4. we evaluate the contribution of the negative effects F^-. For every atomic effect $(f_i^-, \vec{x}_{f-}) \in F^-$, we check if the term f^- appears in any atom $\delta(\vec{x}_q)[\vec{x}_q/\vec{x}_{f+}]$ of q_u (every CQ in Q^T_{unsat} contains at most two atoms, plus an inequality in case of a functionality assertion). In such case, we want to express the fact that, when $\delta(\vec{x}_q)[\vec{x}_q/\vec{x}_{f+}] = (f_i^-, \vec{x}_{f-})$, the negative effect effectively remove the possibility of an inconsistency. We thus add the inequalities $\bigwedge x_{q,i} \neq x_{f-,j}$ for $x_{q,i} \in \vec{x}_q, x_{f+,j} \in \vec{x}_{f+}$. E.g., if $(f_i^-, \vec{x}_{f-}) = \mathsf{B}(z)$, then $q_u[x/y] = \mathsf{true} \wedge \mathsf{B}(y) \wedge y \neq z$;
5. we evaluate the contribution of the remaining positive effects in F^+. For every atomic effect $(f_i^r, \vec{x}_{fr}) \in$

F^+, we check if the term f^r appears in any atom $\delta(\vec{x}_q)[\vec{x}_q/\vec{x}_{f+}]$ of q_u (every CQ in Q^T_{unsat} contains at most two atoms, plus inequalities in case of a functionality assertion and negative effects). In such case, we want to express the fact that, when $\delta(\vec{x}_q)[\vec{x}_q/\vec{x}_{f+}] = (f^r_i, \vec{x}_{fr})$, the addition of both effects generates an inconsistency. We thus add the equality $\bigwedge x_{q,i} = x_{fr,j}$ for $x_{q,i} \in \vec{x}_q, x_{fr,j} \in \vec{x}_{fr}$. E.g., if $(f^r_i, \vec{x}_{fr}) = \mathsf{B}(u)$, then $q_u[x/y] = \mathsf{true} \wedge \mathsf{B}(y) \wedge y \neq z \wedge y = u$;

6. q_u now effectively contains all the elements to block possible inconsistencies related to the atomic effect $(f^+_i, \vec{x}_{f+})$. Since it is derived from Q^T_{unsat}, if it evaluates to true, than it means that $(f^+_i, \vec{x}_{f+})$ generates an inconsistency. We thus add $\neg q_u$ to Q_{check}.

Definition 5.1.1 *Given an action $\alpha \in \Lambda$ and its corresponding rule $\gamma \in \Gamma$ ($\gamma : Q \mapsto \alpha$), we call extended rule γ_{ext} the rule defined as:*

$$\gamma_{ext} : Q_{ext} \mapsto \alpha$$

with $Q_{ext} = Q_{rew} \wedge Q_{check}$, and Q_{check} the ECQ obtained by tailoring Q^T_{unsat} w.r.t. the positive effects F^+.

The extended rules effectively catch possible inconsistencies beforehand, and thus eliminating the need of the *check consistency modality* (and its related predicates) in the translation to STRIPS. The union of all extended rules defines the set Γ_{ext}.

5.2 Translation to ADL

Given the translation of a reKAB to a STRIPS planning problem (Section 5.1), we now try to move onto a more expressive framework, and translate a full *DL-Lite*$_\mathcal{A}$-eKAB into an ADL planning problem (Section 2.3.2).

Before showing the translation, we note that, unfortunately, the possibility to embed the satisfiability check directly into the rules' conditions (Section 5.1.1) cannot be applied to *DL-Lite*$_\mathcal{A}$-eKABs, due to the nature of actions' effects. First, the embedding should take place in the condition of the effect, as there could be many different conditional effects in one action. Secondly, in a normal eKAB, an effect works on a set of answers, provided by the effect's condition, which can contain variables not in the parameters' set; in a reKAB this doesn't happen as we impose the only effect not to have any condition, using only a single substitution for the parameters. To show how this affects the embedding of the satisfiability check, we propose a simple example.

Example 10 *Consider the following elements:*

- *a TBox T with the disjointness axiom* $\mathsf{A} \sqsubseteq \neg \mathsf{B}$
- *an ABox A with the assertions* $\{\mathsf{P}(\mathsf{a},\mathsf{b}), \mathsf{P}(\mathsf{b},\mathsf{c})\}$
- *an action α with only two external input parameters and with one effect*
  $e : \mathsf{P}(x, y) \rightsquigarrow \boldsymbol{add}\{\mathsf{A}(x), \mathsf{B}(y)\}$

If we apply the embedding and extend the condition of the effect, we would obtain:

$$e_{ext} : \mathsf{P}(x, y) \wedge \neg\mathsf{B}(x) \wedge \neg\mathsf{A}(y) \rightsquigarrow \boldsymbol{add}\{\mathsf{A}(x), \mathsf{B}(y)\}$$

Retrieving the certain answers of e_{ext} *in the state* A *would yield the tuples:*

$$\{(x \mapsto \mathsf{a}, y \mapsto \mathsf{b}), (x \mapsto \mathsf{b}, y \mapsto \mathsf{c})\}$$

and produce the state A'*:*

$$A' = \{\mathsf{P}(\mathsf{a}, \mathsf{b}), \mathsf{P}(\mathsf{b}, \mathsf{c}), \mathsf{A}(\mathsf{a}), \mathsf{B}(\mathsf{b}), \mathsf{A}(\mathsf{b}), \mathsf{B}(\mathsf{c})\}$$

which is clearly inconsistent.

The problem arises from the fact that the embedded consistency check works on a single certain answer per time, and fails at considering the combined effects of the whole set. Although we cannot apply this technique in eKABs, we gain in expressivity, thus balancing the loss.

We now define a syntactic, modular translation procedure leKab2Adl that takes as input a *DL-Lite*$_\mathcal{A}$-eKAB planning problem $\mathcal{E} = \langle \mathcal{K}, G \rangle$ with $\mathcal{K} = \langle \mathcal{C}, \mathcal{C}_0, T, A_0, \Lambda, \Gamma \rangle$, and produces a corresponding ADL planning problem $\mathcal{P}_\mathcal{K} = \langle \mathcal{C}_\mathcal{K}, \mathcal{C}_0, \mathcal{F}_\mathcal{K}, \mathcal{A}_\mathcal{K}, \varphi_\mathcal{K}, \psi_G \rangle$.

Object domain. As in Section 4.2, if $\mathcal{C}$ is finite, so is $\mathcal{C}_\mathcal{K}$. If instead $\mathcal{C}$ is infinite but $\mathcal{K}$ is b-bounded, we fix $\mathcal{C}_\mathcal{K}$ to contain $\mathcal{C}_0$ plus $n + b$ objects from $\mathcal{C}$.

Fluents. $\mathcal{F}_\mathcal{K}$ is obtained by encoding concept and role names in T into corresponding unary and binary fluents. It also contains two special nullary fluents: *ChkCons*, distinguishing *normal execution modality* from *check consistency modality* of $\mathcal{P}_\mathcal{K}$, and *Error*, marking when the consistency check fails.

Operators. $\mathcal{A}_\mathcal{K}$ is obtained by transforming every action in Λ, with its condition-action rule in Γ, into an ADL operator: each action's effect produces a conditional effect in the ADL operator, and the condition-action rule its precondition.

As for the translation to STRIPS (Section 5.1), we embed the TBox contribution by relying on *DL-Lite*$_\mathcal{A}$'s *FO rewritability* of both ECQs and T-consistency checks (Section 2.2). We force $\mathcal{A}_\mathcal{K}$ to alternate between two phases: the *normal execution modality*, where a "normal" ADL operator is applied, mirroring the execution of an action instance of $\mathcal{K}$; and the *check consistency modality*, where a special ADL operator checks if the obtained state is T-consistent. Alternation is realized by toggling the fluent *ChkCons*, and activating *Error* when the consistency check fails, thus blocking operator application.

Technically, consider an action $\alpha = a(\vec{p}) : \{e_1, \ldots, e_n\}$ in Λ and its corresponding condition-action rule $Q_\alpha(\vec{x}) \mapsto a(\vec{p})$ in Γ. Let $\vec{z} = \vec{p} \setminus \vec{x}$. We produce a corresponding ADL operator $\langle a, \vec{p}, \rho_\alpha(\vec{p}), \varepsilon_\alpha(\vec{p}) \rangle$. Its precondition $\rho_\alpha(\vec{p})$ corresponds to the FO formula $rew(Q_\alpha(\vec{x}), T) \wedge \neg ChkCons \wedge \neg Error$, which leaves the external input parameters $\vec{z}$ unconstrained. The operator effect $\varepsilon_\alpha(\vec{p})$ is the FO conjunction of the translation of $e_1, \ldots, e_n$. Each $e_i = Q_i(\vec{p}, \vec{x}_i) \rightsquigarrow$ **add** F_i^+, **del** F_i^- generates the following conjunct:

$$rew(Q_i(\vec{p}, \vec{x}_i), T) \rightarrow \bigwedge_{P_j(\vec{p},\vec{x}_i) \in F_i^+} P_j(\vec{p}, \vec{x}_i) \wedge \bigwedge_{P_k(\vec{p},\vec{x}_i) \in F_i^-} \neg P_k(\vec{p}, \vec{x}_i) \wedge ChkCons$$

Finally, the special ADL operator used to check T-consistency

is $\langle check, \emptyset, \rho_{chk}, \varepsilon_{chk} \rangle$, where $\rho_{chk} = ChkCons$ just checks whether the check flag is on, while ε_{chk} takes care of toggling the check flag, as well as of triggering the error flag if an inconsistency is detected:

$$\varepsilon_{chk} = \neg ChkCons \wedge (Q^T_{\text{unsat}} \rightarrow Error)$$

Initial state specification. $\varphi_{\mathcal{K}}$ is by constructing the conjunction of all facts contained in A_0: $\varphi_{\mathcal{K}} = \bigwedge_{P_i(\vec{o}) \in A_0} P_i(\vec{o})$.

Goal description. The goal description ψ_G is obtained from goal G in $\mathcal{E}$ as $\psi_G = rew(G, T) \wedge \neg ChkCons \wedge \neg Error$.

The two-fold contribution of T is taken care of, as in the operators, by rewriting G (via $rew(G, T)$) and ensuring that the ending state is T-consistent (by requiring the absence of the consistency check and error flags in ψ_G).

Example 11 *Consider an eKAB with the following TBox:*
$\mathsf{Engineer} \sqsubseteq \mathsf{Employee}$, $\mathsf{Designer} \sqsubseteq \mathsf{Employee}$, $\mathsf{ElectronicEng} \sqsubseteq \mathsf{Engineer}$, $\mathsf{Designer} \sqsubseteq \neg\mathsf{Engineer}$
Γ *contains the following rule* $\gamma_{assignTest}$*:*
$[\mathsf{Employee}(x)] \mapsto assignTest(x)$
while the action $assignTest$ *is defined as:*
$assignTest(x) = \{e_{assignTest}(x)\}$
$e_{assignTest}(x) = \mathsf{true} \rightsquigarrow \boldsymbol{add}\{\mathsf{TestingAgent}(x)\}$

We can rewrite Q_α *in order to compile away the TBox, and transform* $\gamma_{assignTest}$ *in:*
$[\mathsf{Employee}(x) \vee \mathsf{Engineer}(x) \vee \mathsf{Designer}(x) \vee \mathsf{ElectronicEng}(x)] \mapsto assignTest(x)$

while the query Q^T_{unsat} *is:*
$\exists x.((\mathsf{Designer}(x) \wedge \mathsf{Engineer}(x)) \vee$
$(\mathsf{Designer}(x) \wedge \mathsf{ElectronicEng}(x)))$

We show the translation to ADL through PDDL of the action assignTest:

```
: action assignTest
    : parameters (?x)
    : precondition (and
        (not(CheckConsistency)) (not (Inconsistent))
        (or (Employee ?x) (Eng ?x) (ElectronicEng ?x))
        ))
    : effect (and (CheckConsistency) (TestingAgent ?x))
```

We close by considering the following algorithm, called FindPlan-leKabAdl:

1. take as input a *DL-Lite*$_\mathcal{A}$-eKAB planning problem $\mathcal{E}$;
2. translate $\mathcal{E}$ into an ADL planning problem using leKab2Adl;
3. invoke an off-the-shelf ADL planner;
4. if the planner returns fail, return fail as well;
5. if the planner returns a plan π, filter away all *check* operators from π, and return it as a result.

Theorem 5.2.1 FindPlan-leKabAdl *is correct.*

Proof 5.2.1 *The proof is given in two steps. In the first step, we show that, thanks to FO-rewritability, each DL-Lite$_\mathcal{A}$-eKAB planning problem can be transformed into an equivalent planning problem whose DL-Lite$_\mathcal{A}$-eKAB has an empty TBox, and employs standard FO query evaluation to progress the ABox. In the second step, we show that such an "intermediate" problem is correctly mirrored by the ADL translation obtained through* FindPlan-leKabAdl. *The first step is already proven in Proof 5.1.1, so we concentrate only on the second step.*

Given the eKAB $\mathcal{K}_r$ obtained from $\mathcal{K}$ by rewriting all ECQs, so as to compile away the positive assertions of the TBox T, the evolutions induced by $\mathcal{K}_r$ are obtained by applying standard FO query evaluation, by considering the underlying ABox as a database of facts, interpreted with the closed-world assumption. Hence, query answering for $\mathcal{K}_r$ is the same as that adopted by ADL for $\mathcal{P}_\mathcal{K}$ (identical to $\mathcal{P}_{\mathcal{K}_r}$). Furthermore $\mathcal{P}_\mathcal{K}$ represents a direct syntactic reformulation of $\mathcal{K}_r$ in ADL, with the only key difference that $\mathcal{K}_r$ checks whether the ABox resulting from the application of an action instance is $(T_n \uplus T_f)$-consistent, while $\mathcal{P}_\mathcal{K}$ simulates this check by introducing the two-step approach discussed in the translation.

Let A be an ABox, and $\alpha\theta$ be a $\mathcal{K}_r$ action instance, with $\alpha = a(\vec{y})$. Let D be the ADL representation of A (i.e., the formula that corresponds to the conjunction of facts in A), and $\langle a, \vec{y}, \rho_\alpha(\vec{y}), \varepsilon_\alpha(\vec{y})\rangle$ the ADL operator corresponding to α. We have three cases to discuss for θ.

If θ is not an answer to the guard of the condition-

action rule for α in A, then $\alpha\theta$ is not applicable. In this case, $\rho_\alpha(\vec{y})\theta$ does not hold in D, and hence the ADL operator corresponding to α, grounded with θ, is not applicable in D neither.

If instead θ is an answer to the guard of the condition-action rule for α in A, we have two sub-cases to discuss, corresponding to the situation where $\text{DO}(\alpha\theta, A, \emptyset)$ is $(T_n \uplus T_f)$-consistent, and that where it is not.

In the first sub-case, $\alpha\theta$ is $\mathcal{K}_r$-legal in A. In this situation, we have have that D' is the ADL representation of A', where:

- *$A' = \text{DO}(\alpha\theta, A, \emptyset)$.*
- *D' results from D by applying the sequence of the ADL operator $\langle a, \vec{y}, \rho_\alpha(\vec{y}), \varepsilon_\alpha(\vec{y})\rangle$ instantiated with θ, followed by the application of the ADL operator $\langle check, \emptyset, \rho_{chk}, \varepsilon_{chk}\rangle$, which has the only effect of removing the check flag. In fact, thanks to the FO-rewritability of consistency check in DL-Lite$_\mathcal{A}$, we have that Q^T_{unsat} holds in D' if and only if A' is $(T_n \uplus T_f)$-inconsistent.*

Hence, the progression induced by $\alpha\theta$ corresponds to that induced by the sequence of the corresponding ADL operator, followed by the check operator.

In the second sub-case, $A' = \text{DO}(\alpha\theta, A, \emptyset)$ is $(T_n \uplus T_f)$-inconsistent, and hence $\alpha\theta$ is not is not applicable in A. In ADL, instead, the operator $\langle a, \vec{y}, \rho_\alpha(\vec{y}), \varepsilon_\alpha(\vec{y})\rangle$, instantiated with θ, is applicable, and leads to a state D' in which only the check operator $\langle check, \emptyset, \rho_{chk}, \varepsilon_{chk}\rangle$ can be executed. Since $D' \setminus \{ChkCons\}$ corresponds to A', which is

$(T_n \uplus T_f)$-inconsistent, the application of the check operator leads to introduce the error flag, i.e., leads to the ADL state $D'' = (D' \setminus \{ChkCons\}) \cup \{Error\}$. The error flag is never removed, and therefore this spurious path cannot lead to satisfy the desired ADL goal $\psi_G = G_r \wedge \neg ChkCons \wedge \neg Error$.

By induction, we consequently obtain that non-spurious runs induced by the ADL operators in $\mathcal{P}_\mathcal{K}$ exactly replicate those in $\Upsilon_{\mathcal{K}_r}$, if one projects away the intermediate check states (which do not affect the possibility to satisfy the given goal).

5.3 From eKABs to reKABs

In Section 5.1 we shown a translation from reKABs, a subset of eKABs, to STRIPS, while in 5.2 we translated eKABs to ADL. In Figure 5.1 we depict the inter-relationships existing among these different way to represent a planning domain; the full arrows correspond to the translations provided in the previous sections. It is well know that it is possible to translate an ADL planning problem to a STRIPS one [Gazen and Knoblock, 1997], and we are thus interested to close the same gap also between eKABs and reKABs.

The method to translate an eKAB in a reKAB leverages on the work done in [Abdulla et al., 2016], which is about transforming actions with bulk effects (i.e., the action generates a set of instantiated effects all applied

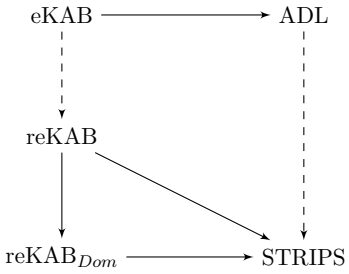

FIGURE 5.1: Translations map

at the same time) in a database context, in a sequence of action with only a single effect. To achieve the same result, we need to consider an action $\alpha(\vec{p})$ together with its related condition-action rule γ, and decompose them into a set of reKAB actions and rules that reproduces the execution semantic of γ and α.

We now describe and formally define how we model the various phases of the execution semantic, detailing as well the special concepts and roles we introduce for the purpose.

1. Action Lock
in this phase we select the action α we want to perform, and set a lock in order that no other action can start its execution. We introduce the concept $\mathsf{HasLock}$ that is used to represent which action is being performed at the moment (e.g. $\mathsf{HasLock}(\alpha)$).
Rule:

$rule\text{-}perform_\alpha : \neg\exists x.\mathsf{HasLock}(x) \wedge \exists\vec{x}\mathbf{Q}(\vec{x}) \mapsto lock_\alpha$;

Action:
$perform_\alpha : \mathsf{true} \rightsquigarrow \mathbf{add}\{\mathsf{HasLock}(\alpha), \mathsf{RetrieveParametersPhase}(\alpha)\}$

2. Condition-action Rule Parameters Retrieval
In this phase we execute the query $Q(\vec{x})$ in the condition-action rule γ, and save the answer used to ground part of the parameters $\vec{p}$ of the action α (the external parameters $\vec{y} = \vec{p} \setminus \vec{x}$ are taken care by the next phase). The answer is saved inside the set of special concepts $\mathsf{Parameter}^i_\alpha$, one for every parameter in $\vec{x}$.
We use the concept RetrieveParametersPhase to recognize this phase.
Rule:
$rule\text{-}setParameters_\alpha : \mathsf{HasLock}(\alpha) \wedge \mathsf{RetrieveParametersPhase}(\alpha) \wedge \mathbf{Q}(\vec{x}) \mapsto setParameters_\alpha(\vec{x})$

Action:
$setParameters_\alpha(\vec{x}) : \mathsf{true} \rightsquigarrow$
$\mathbf{del}\{\mathsf{RetrieveParametersPhase}(\alpha)\}$
$\mathbf{add}\{\bigwedge_{x_i \in \vec{x}} \mathsf{Parameter}^i_\alpha(x_i), \mathsf{RetrieveExtInputPhase}(\alpha)\}$

3. External Input Parameters Retrieval
We set a value for the external parameters input $\vec{y}$. The external input parameters are saved inside the set of special concepts $\mathsf{ExtInput}^i_\alpha$, one for every parameter in $\vec{y}$.
We use the concept RetrieveExtInputPhase to recognize this phase. We also create a lock for each effect in the form of the role HasLockEff, where the domain is an action name; since the next phase is the retrieval of the answers for each

effect's condition, we add the lock $\mathsf{HasLockEff}(\alpha, \mathsf{effect1})$ in order to start from the first one.
Rule:
rule-retrieveExt$_\alpha$: $\mathsf{HasLock}(\alpha) \wedge \mathsf{RetrieveExtInputPhase}(\alpha)$
$\mapsto$ *retrieveExt*$_\alpha(\vec{y}, ansId)$

Action:
retrieveExt$_\alpha(\vec{y}, ansId)$: $\mathsf{true} \rightsquigarrow$
del$\{\mathsf{RetrieveExtInputPhase}(\alpha)\}$
add$\{\bigwedge_{y_i \in \vec{y}} \mathsf{ExtInput}^i_\alpha(y_i), \mathsf{RetrieveAnswersPhase}(\alpha),$
$\mathsf{HasLockEff}(\alpha, \mathsf{effect1}), \mathsf{NewAnsId}(ansId)\}$

4. Effects' Answers Retrieval
We retrieve and save each certain answer for each effect's condition query $Q_{eff^1}(\vec{p}, \vec{z}_1)$, where $\vec{z}_1$ are the additional free variable appearing in Q_{eff^1}. Since DLs don't support n-ary predicates (we would need one to save the whole answer with only an assertion), we need to resort to *reification*: we decompose the needed n-ary (with $n = |\vec{p} \cup \vec{z}_1|$) predicate into n binary predicates (i.e., roles) of the type $\mathsf{AnsDel}^{\mathsf{var_i}}_{\alpha-\mathsf{effJ}}$, where the first term is the unique identifier of the answer, while the second is used to sore the value to which the variable var_i is mapped to.

To identify the answers, we have to be sure that each ID used is unique: we introduce for this purpose the disjoint concepts NewAnsId and OldAnsIds (NewAnsId $\not\sqsubseteq$ OldAnsIds). NewAnsId stores the ID we can use to save the next answer, while OldAnsIds is used to track the IDs that have already being used.
We use the concept RetrieveAnswersPhase to recognize this

phase. We introduce 3 rules and 3 actions, dedicated, respectively, to: retrieve an answer for a given effect, switch from an effect to the next one, switch from the last effect to the next phase.

Rule (presented for the first effect):
$rule\text{-}addAnsEff1_{\alpha}$: $\mathsf{HasLock}(\alpha) \wedge \mathsf{RetrieveAnswersPhase}(\alpha) \wedge \mathsf{HasLockEff}(\alpha, \mathsf{effect1}) \wedge \mathbf{Q_{eff1}}(\vec{p}, \vec{z}_1) \wedge \bigwedge_{x_i \in \vec{x}} \mathsf{Parameter}^{i}_{\alpha}(x_i)$
$\neg \exists ansIdTemp. \bigwedge_{1..n} \mathsf{AnsDel}^{\mathsf{var_i}}_{\alpha-\mathsf{eff1}}(ansIdTemp, var_i) \wedge \mathsf{NewAnsId}(ansId) \mapsto addAnsEff1_{\alpha}(\vec{p}, \vec{z}_1, andId, andIdNew)$
with $var_1, ..., var_n$ the variables appearing in Q_{eff1}.

Action (presented for the first effect):
$addAnsEff1_{\alpha}(\vec{p}, \vec{z}_1, andId, andIdNew)$: $\mathsf{true} \rightsquigarrow$
$\mathbf{del}\{\mathsf{RetrieveExtInputPhase}(\alpha), \mathsf{NewAnsId}(ansId)\}$
$\mathbf{add}\{\mathsf{AnsDel}^{\mathsf{var_i}}_{\alpha-\mathsf{eff1}}(ansId, var_i), \mathsf{OldAnsIds}(ansId), \mathsf{NewAnsId}(ansIdNew)\}$

Rule (presented for the switch between the first and second effect):
$rule\text{-}fromEff1toEff2_{\alpha}$: $\mathsf{HasLock}(\alpha) \wedge \mathsf{RetrieveAnswersPhase}(\alpha) \wedge \mathsf{HasLockEff}(\alpha, \mathsf{effect1}) \wedge \forall_{var_i \in \vec{p} \cup \vec{z}_1} (\mathbf{Q_{eff1}}(\vec{p}, \vec{z}_1) \wedge \bigwedge_{x_i \in \vec{x}} \mathsf{Parameter}^{i}_{\alpha}(x_i)) \rightarrow (\exists ansId. \mathsf{AnsDel}^{\mathsf{var_i}}_{\alpha-\mathsf{eff1}}(ansId, var_i))$
$\mapsto fromEff1toEff2_{\alpha}$

Action (presented for the switch between the first and second effect):
$fromEff1toEff2_{\alpha}$: $\mathsf{true} \rightsquigarrow$
$\mathbf{del}\{\mathsf{HasLockEff}(\alpha, \mathsf{effect1})\}$
$\mathbf{add}\{\mathsf{HasLockEff}(\alpha, \mathsf{effect2})\}$

Rule:
$rule\text{-}fromEffNtoDel_{\alpha}$: $\mathsf{HasLock}(\alpha) \wedge \mathsf{RetrieveAnswersPhase}(\alpha) \wedge$
$\mathsf{HasLockEff}(\alpha, \mathsf{effectN}) \wedge \mathsf{NewAnsId}(ansId) \wedge$
$\forall_{var_i \in \vec{p} \cup \vec{z}_n} (\mathbf{Q}_{\mathbf{effN}}(\vec{p}, \vec{z}_n) \wedge \bigwedge_{x_i \in \vec{x}} \mathsf{Parameter}^{i}_{\alpha}(x_i)) \rightarrow$
$(\exists ansId.\mathsf{AnsDel}^{\mathsf{var_i}}_{\alpha-\mathsf{effN}}(ansId, var_i))$
$\mapsto fromEffNtoDel_{\alpha}(ansId)$

Action:
$fromEffNtoDel_{\alpha}(ansId)$: $\mathsf{true} \rightsquigarrow$
$\mathbf{del}\{\mathsf{HasLockEff}(\alpha, \mathsf{effectN}), \mathsf{RetrieveAnswersPhase}(\alpha),$
$\mathsf{NewAnsId}(ansId)\}$
$\mathbf{add}\{\mathsf{DeletePhase}(\alpha), \mathsf{HasLockEff}(\alpha, \mathsf{effect1})\}$

5. Delete Effects
In this phase we apply the deletion effects for each effect in α. We do so by cycling through all the saved answers in each $\mathsf{AnsDel}^{\mathsf{var_i}}_{\alpha-\mathsf{effJ}}$, apply the proper variable substitutions to the deletion effects, and move the answer to the concept $\mathsf{AnsAdd}^{\mathsf{var_i}}_{\alpha-\mathsf{effJ}}$.
We use the concept DeletePhase to recognize this phase. We introduce 3 rules and 3 actions, dedicated, respectively, to: delete an atomic effect for a given effect, switch from an effect to the next one, switch from the last effect to the next phase.

Rule (presented for the first effect):
$rule\text{-}delEff1_{\alpha}$: $\mathsf{HasLock}(\alpha) \wedge \mathsf{DeletePhase}(\alpha)$
$\wedge \mathsf{HasLockEff}(\alpha, \mathsf{effect1}) \wedge \bigwedge_i \mathsf{ExtInput} - \mathsf{i}_{\alpha}(extInp_i)$
$\wedge \bigwedge_{var_i \in \overrightarrow{var}} \mathsf{AnsDel}^{\mathsf{var_i}}_{\alpha-\mathsf{eff1}}(ansId, var_i) \mapsto delEff1_{\alpha}(\overrightarrow{var}, ansId)$
with $\overrightarrow{var}$ the variables appearing in the answer $\mathsf{AnsDel}^{\mathsf{var_i}}_{\alpha-\mathsf{eff1}}$ with id $ansId$.

Action (presented for the first effect):
$delEff1_{\alpha}(\overrightarrow{var}, ansId)$: $\mathsf{true} \rightsquigarrow$
$\mathbf{del}\{\mathbf{F_1^-}(\vec{p}), \bigwedge_{var_i \in \overrightarrow{var}} \mathsf{AnsDel}^{\mathsf{var_i}}_{\alpha-\mathsf{eff1}}(ansId, var_i),$
$\mathsf{OldAnsIds}(ansId)\}$
$\mathbf{add}\{\mathsf{AnsAdd}^{\mathsf{var_i}}_{\alpha-\mathsf{eff1}}(ansId, var_i)\}$

Rule (presented for the switch between the first and second effect):
$rule\text{-}fromDelEff1toDelEff2_{\alpha}$: $\mathsf{HasLock}(\alpha) \wedge \mathsf{DeletePhase}(\alpha) \wedge$
$\mathsf{HasLockEff}(\alpha, \mathsf{effect1}) \wedge \neg\exists ansId, var_1.\mathsf{AnsDel}^{\mathsf{var_1}}_{\alpha-\mathsf{eff1}}(ansId, var_1)$
$\mapsto fromDelEff1toDelEff2_{\alpha}$

Action (presented for the switch between the first and second effect):
$fromDelEff1toDelEff2_{\alpha}$: $\mathsf{true} \rightsquigarrow$
$\mathbf{del}\{\mathsf{HasLockEff}(\alpha, \mathsf{effect1})\}$
$\mathbf{add}\{\mathsf{HasLockEff}(\alpha, \mathsf{effect2})\}$

Rule:
$rule\text{-}fromDelEffNtoAdd_{\alpha}$: $\mathsf{HasLock}(\alpha) \wedge \mathsf{DeletePhase}(\alpha) \wedge$
$\mathsf{HasLockEff}(\alpha, \mathsf{effectN}) \wedge \neg\exists ansId, var_1.\mathsf{AnsDel}^{\mathsf{var_1}}_{\alpha-\mathsf{effn}}(ansId, var_1)$
$\mapsto fromDelEffNtoAdd_{\alpha}$

Action:
$fromDelEffNtoAdd_{\alpha}$: $\mathsf{true} \rightsquigarrow$
$\mathbf{del}\{\mathsf{HasLockEff}(\alpha, \mathsf{effectN}), \mathsf{DeletePhase}(\alpha)\}$
$\mathbf{add}\{\mathsf{AddPhase}(\alpha)\}$

6. Add Effects

In this phase we apply the add effects for each effect in α. We do so by cycling through all the saved answers in each $\mathsf{AnsAdd}^{\mathsf{var_i}}_{\alpha-\mathsf{effJ}}$, apply the proper variable substitutions to the add effects, and erase the answer.

We use the concept AddPhase to recognize this phase. We introduce 3 rules and 3 actions, dedicated, respectively, to: add an atomic effect for a given effect, switch from an effect to the next one, switch from the last effect to the next phase.

Rule (presented for the first effect):
$rule\text{-}addEff1_{\alpha}$: $\mathsf{HasLock}(\alpha) \wedge \mathsf{AddPhase}(\alpha)$
$\wedge\, \mathsf{HasLockEff}(\alpha, \mathsf{effect1}) \wedge \bigwedge_{1..n} \mathsf{ExtInput} - \mathsf{i}_{\alpha}(extInp_i)$
$\wedge \bigwedge_{var_i \in \overrightarrow{var}} \mathsf{AnsAdd}^{\mathsf{var_i}}_{\alpha-\mathsf{eff1}}(ansId, var_i) \mapsto addEff1_{\alpha}(\overrightarrow{var}, ansId)$
with $\overrightarrow{var}$ the variables appearing in the answer $\mathsf{AnsAdd}^{\mathsf{var_i}}_{\alpha-\mathsf{eff1}}$ with id $ansId$.

Action (presented for the first effect):
$addEff1_{\alpha}(\overrightarrow{var}, ansId)$: $\mathsf{true} \rightsquigarrow$
$\mathbf{del}\{\bigwedge_{var_i \in \overrightarrow{var}} \mathsf{AnsAdd}^{\mathsf{var_i}}_{\alpha-\mathsf{eff1}}(ansId, var_i),$
$\mathsf{OldAnsIds}(ansId)\}$
$\mathbf{add}\{\mathbf{F_1^+}(\vec{p})\}$

Rule (presented for the switch between the first and second effect):
$rule\text{-}fromAddEff1toAddEff2_{\alpha}$: $\mathsf{HasLock}(\alpha) \wedge \mathsf{AddPhase}(\alpha) \wedge$
$\mathsf{HasLockEff}(\alpha, \mathsf{effect1}) \wedge \neg\exists ansId, var_1.\mathsf{AnsAdd}^{\mathsf{var_1}}_{\alpha-\mathsf{eff1}}(ansId, var_1)$
$\mapsto fromAddEff1toAddEff2_{\alpha}$

Action (presented for the switch between the first and second effect):
$fromAddEff1toAddEff2_{\alpha}$: $\mathsf{true} \rightsquigarrow$
$\mathbf{del}\{\mathsf{HasLockEff}(\alpha, \mathsf{effect1})\}$
$\mathbf{add}\{\mathsf{HasLockEff}(\alpha, \mathsf{effect2})\}$

Rule:
$rule\text{-}fromAddEffNtoClean_{\alpha}$: $\mathsf{HasLock}(\alpha) \wedge \mathsf{AddPhase}(\alpha) \wedge$

$\mathsf{HasLockEff}(\alpha, \mathsf{effectN}) \wedge$
$\neg\exists ansId, var_1.\mathsf{AnsAdd}^{\mathsf{var_1}}_{\alpha-\mathsf{effN}}(ansId, var_1)$
$\mapsto \mathit{fromAddEffNtoClean}_\alpha$

Action:
$\mathit{fromAddEffNtoClean}_\alpha : \mathsf{true} \rightsquigarrow$
del$\{\mathsf{HasLockEff}(\alpha, \mathsf{effectN}), \mathsf{AddPhase}(\alpha)\}$
add$\{\mathsf{CleanPhase}(\alpha)\}$

7. Clean Up
This phase is used to empty the special concepts and roles used during the previous phases, in order to leave a clean state, ready to perform another action.

We use the concept $\mathsf{CleanUpPhase}$ to recognize this phase.

Rule:
$\mathit{rule\text{-}clean}_\alpha : \mathsf{HasLock}(\alpha) \wedge \mathsf{CleanPhase}(\alpha) \wedge$
$\bigwedge_{extInp_i \in \overrightarrow{extInp}} \mathsf{ExtInput}^i_\alpha(extInp_i) \mapsto \mathit{clean}_\alpha(\overrightarrow{extInp})$

Action:
$\mathit{clean}_\alpha(\overrightarrow{extInp}) : \mathsf{true} \rightsquigarrow$
del$\{\bigwedge_{extInp_i \in \overrightarrow{extInp}} \mathsf{ExtInput}^i_\alpha(extInp_i),$
$\mathsf{CleanPhase}(\alpha), \mathsf{HasLock}(\alpha)\}$
add$\{\}$

The last element to be able to translate an eKAB into a reKAB, is, given planning problem $\langle \mathcal{K}, G \rangle$, creating the proper goal query G_r. We have to be sure that, given the translation α_r of an action α, in the transition system $\Upsilon_{\mathcal{K}_r}$ the goal is not met in the middle of the execution of α_r, but only at the very end. We achieve this by setting the

goal G_r as:

$$G_r = G \wedge \neg \exists x.\mathsf{HasLock}(x)$$

Given the translation of a generic action and its related condition-action rule, we now state two theorems and sketch their proofs: such theorems are necessary to asses the correctness of the translation of a lightweight DL, state-bounded eKAB into a reKAB.

Theorem 5.3.1 *Given a lightweight DL, state-bounded eKAB $\mathcal{K}$, the resulting reKAB $\mathcal{K}_r$ obtained by the translation of $\mathcal{K}$ is still state-bounded.*

Proof 5.3.1 (Sketch) *Given the initial state-bounded eKAB, we have that the translation introduces actions with external parameters, and thus the possibility that these actions, if repeated infinitely many times, break the state-boundedness.*

The actions that present external input parameters are the one related to the retrieval and storage of query answers for the effects' condition; more precisely the additional external parameter is the individual used as a unique ID to store the answer. The interested actions are: $retrieveExt_\alpha(\vec{y}, ansId)$, and $addAnsEffI_\alpha(\vec{p}, \vec{z}_1, andId, andIdNew)$ (with I denoting the i-th effect in the action)

As the original eKAB is state-bounded (meaning in each state at most b individuals are present), and that the number of concepts and roles is constant, we can calculate, for each condition query, the maximum number maxAns of possible answers returned. Having maxAns, we can extend

the bound of the original eKAB, in order to assure that there are enough distinct individuals that could be used as IDs for the answers storage.

Theorem 5.3.2 *Given a lightweight DL, state-bounded eKAB $\mathcal{K}$, and the resulting reKAB $\mathcal{K}_r$ obtained by the translation of $\mathcal{K}$, we have that, for every plan π_r obtained for $\mathcal{K}_r$, it exists an equivalent plan π for $\mathcal{K}$ obtained by removing spurious actions from π_r.*

Proof 5.3.2 (Sketch) *We prove the theorem by showing two points:*

- *given a state A (both $\mathcal{K}$ and $\mathcal{K}_r$ share the same TBox, and states are ABoxes), an action α (and its related rule γ) can be performed in A with the parameter substitution ϑ and creating the state A_{next}, if and only if also the translation α_r of α for $\mathcal{K}_r$ can be performed as well in A with the same substitution ϑ and reaching at the end the same state A_{next}.*
- *when the goal G_r is met, then also G is met.*

The first point can be proved by using contradiction. Assume that an action α can be performed in A with substitution ϑ, obtaining state A_{next}, while the combined actions of α_r cannot. This can have the following reasons:

1. *α_r cannot be performed;*
2. *α_r cannot retrieve the same parameters substitution ϑ;*
3. *α_r cannot retrieve the same answers for the effects;*
4. *α_r removes or add different assertions.*

The first case we can simply check that the rule rule-perform$_\alpha$ simply checks if no other action is being performed (which is not the case), and see if the condition $q(\vec{x})$ of the rule γ returns any answer. If this is not the case, then even the original action α couldn't be performed, raising a contradiction.

Also the other points follow a similar pattern, showing that the various elements of α_r simply reproduce the behaviour of α one step at a time. With the last phase of clean up, α_r removes all additional elements that have been used to regulate the different phases, effectively leaving the final state reached by α_r the same as α.

The point about the goal is easily proved by looking at how the different phases of α_r are defined: the only time when the concept HasLock *is empty is at the beginning (in the state A), and at the end (state A_{next}). In all the intermediate states is always present the assertion* HasLock(α)*, thus making the goal query G_r automatically false.*

Proof of Concept 6

Given the translation methods proposed in Chapter 5, we now try to asses their usability under a practical point of view. Of the two methods, we concentrate on the most expressive one, namely the translation of an eKAB planning problem to an ADL one (Section 5.2).

The test consists in running an example and measure its scalability. Unfortunately, the example is not taken from any standard planning benchmark (e.g., from the ICAPS challenges[1]), although it is frequently used in the literature. Also, the example does not use the full expressive power of eKABs, but nonetheless shows how a plan-

[1] `http://www.icaps-conference.org/`

ning problem can be modelled and performs. Additionally, our goal is to give a generic implementation, avoiding using specific optimizations (such as manually fine tuning the resulting translations) which would alter significantly the performance.

6.1 Robot on a Grid

The example is about a robot positioned on a grid, with the caveat that the robot does not know its exact position. Goal of the robot is to reach a specific cell, and to do so it needs first to derive exactly the cell it is in. Such example represent a classical planning problem under uncertainty, as the robot does not have enough knowledge to assert its exact position, unless it starts exploring the grid.

The grid is represented in Fig. 6.1, and consists of m columns and n rows. When the robot is sure about its position, we say that the robot is in column i and/or row j. When the absolute position hasn't been yet determined, we use a relative one, such as "left of line i", to indicate that the robot could be in any column from 0 to i (same for the rows). The relative positions available are: *Left Of* (LO), *Right Of* (RO), *Above Of* (AO), *Below Of* (BO). The robot can freely move on the grid in the 4 directions (up, down, left, and right), and the only feedback it has is when it hits the border of the grid.

To represent the described setting through an eKAB, we introduce the following concepts: i) we represent each

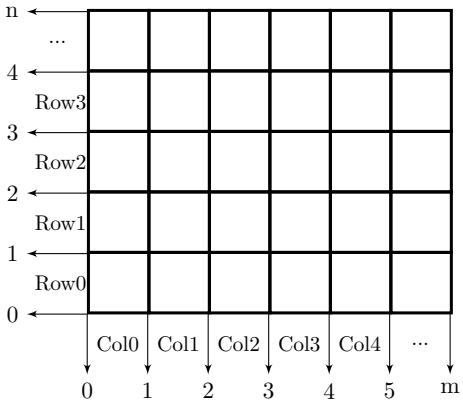

FIGURE 6.1: Grid representation

column and row with concepts $\mathsf{Col_i}$ $(0 \leq i < m)$ and $\mathsf{Row_j}$ $(0 \leq j < n)$, respectively; ii) relative positions are modelled by the concepts LOi $(0 < i \leq m$, as the robot cannot be on the left of column 0), ROi $(0 \leq i < m)$, AOj $(0 \leq j < n)$, BOj $(0 < j \leq n)$;

We also model the inferences the robot can apply given his knowledge. First of all, if the robot knows to be at the right of a certain column (e.g., $\mathsf{RO2(robot)}$), then it can infer also that it is on the right of all previous columns (e.g., $\mathsf{RO1(robot)}$ and $\mathsf{RO0(robot)}$); we model this through general inclusion axioms of the type $\mathsf{ROi} \sqsubseteq \mathsf{ROi - 1}$, for $0 < i < m$. The same goes for all the other relative positions. We give a visual representation of the concepts about columns in Figure 6.2 (the rows hierarchy is equiv-

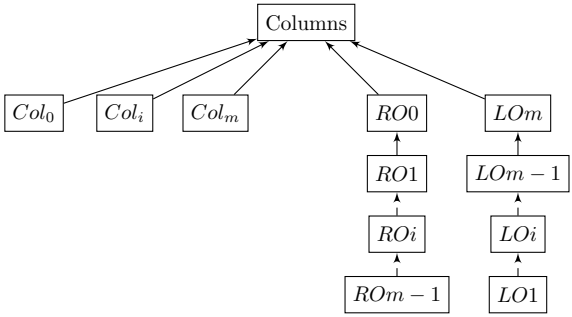

Figure 6.2: Columns hierarchy

alent).

The robot also knows that it cannot be on the right and on the left of the same line i; this is modelled through disjoint axioms of the type $\mathsf{ROi} \sqsubseteq \neg\mathsf{LOi}$, for $0 < i < m$. Same goes for the concepts AO and BO.

Last, we can model the fact that, when the robot is on the right of line i, and on the left on line $i+1$, then it can infer that it is certainly in column i. We resort to a type of axiom available in the dialect *DL-Lite*$_{Horn}$:

$$\mathsf{ROi} \sqcap \mathsf{LOi} + 1 \sqsubseteq \mathsf{Col_i} \text{ for } 0 \leq i \leq m$$

The complete list of TBox axioms is:
$\mathsf{ROi} \sqsubseteq \mathsf{ROi} - 1$ for $0 < i < m$
$\mathsf{LOi} \sqsubseteq \mathsf{LOi} + 1$, for $0 < i < m$
$\mathsf{AOj} \sqsubseteq \mathsf{AOj} - 1$ for $0 < j < n$
$\mathsf{BOj} \sqsubseteq \mathsf{BOj} + 1$, for $0 < j < n$
$\mathsf{ROi} \sqsubseteq \neg\mathsf{LOi}$ for $0 < i < m$

$\mathsf{AOj} \sqsubseteq \neg\mathsf{BOj}$ for $0 < j < n$
$\mathsf{ROi} \sqcap \mathsf{LOi} + 1 \sqsubseteq \mathsf{Col_i}$ for $0 \leq i < m$
$\mathsf{AOj} \sqcap \mathsf{BOj} + 1 \sqsubseteq \mathsf{Row_j}$ for $0 \leq j < n$

The initial ABox A_0 contains simply the fact that the robot is know to be inside the grid:
$A_0 = \{\mathsf{RO0(robot)}, \mathsf{LOm(robot)}, \mathsf{AO0(robot)}, \mathsf{BOn(robot)}\}$

We now focus on the actions (and the related rules) that govern the robot's knowledge. They are quite self-explanatory, as the effects of the rules update the knowledge of the robot in accordance to its (relative and/or absolute) position:
$ruleRight : \mathsf{Columns}(x) \mapsto moveRight(x)$
$moveRight(x) : \{e_{RO0}, ..., e_{LO1}, ..., e_{Col0}, ...\}$
$e_{ROi} : K(\mathsf{ROi}(x)) \rightsquigarrow \mathbf{add}\{\mathsf{ROi} + 1(x)\}$ for $0 \leq i < m$
$e_{LOi} : K(\mathsf{LOi}(x)) \rightsquigarrow \mathbf{add}\{\mathsf{LOi} + 1(x)\}, \mathbf{del}\{\mathsf{LOi}(x)\}$ for $0 < i \leq m$
$e_{Coli} : K(\mathsf{Col_i}(x)) \rightsquigarrow \mathbf{add}\{\mathsf{Col_{i+1}}(x)\}, \mathbf{del}\{\mathsf{Col_i}(x)\}$ for $0 \leq i < m$

$ruleLeft : \mathsf{Columns}(x) \mapsto moveLeft(x)$
$moveLeft(x) : \{e_{RO1}, ..., e_{LO1}, ..., e_{Col1}, ...\}$
$e_{LOi} : K(\mathsf{LOi}(x)) \rightsquigarrow \mathbf{add}\{\mathsf{LOi} - 1(x)\}$ for $0 < i \leq m + 1$
$e_{ROi} : K(\mathsf{ROi}(x)) \rightsquigarrow \mathbf{add}\{\mathsf{ROi} - 1(x)\}, \mathbf{del}\{\mathsf{ROi}(x)\}$ for $0 < i \leq m$
$e_{Coli} : K(\mathsf{Col_i}(x)) \rightsquigarrow \mathbf{add}\{\mathsf{Col_{i-1}}(x)\}, \mathbf{del}\{\mathsf{Col_i}(x)\}$ for $0 < i \leq m$

$ruleUp : \mathsf{Rows}(x) \mapsto moveUp(x)$

$moveUp(x) : \{e_{AO0}, ..., e_{BO1}, ..., e_{Row0}, ...\}$
$e_{AOi} : K(\mathsf{AOi}(x)) \rightsquigarrow \mathbf{add}\{\mathsf{AOi}+1(x)\}$ for $0 \leq i < n$
$e_{BOi} : K(\mathsf{BOi}(x)) \rightsquigarrow \mathbf{add}\{\mathsf{BOi}+1(x)\}, \mathbf{del}\{\mathsf{BOi}(x)\}$ for $0 < i \leq n$
$e_{Rowi} : K(\mathsf{Row_i}(x)) \rightsquigarrow \mathbf{add}\{\mathsf{Row_{i+1}}(x)\}, \mathbf{del}\{\mathsf{Row_i}(x)\}$ for $0 \leq i < n$

$ruleDown : \mathsf{Rows}(x) \mapsto moveDown(x)$
$moveDown(x) : \{e_{AO1}, ..., e_{BO1}, ..., e_{Row1}, ...\}$
$e_{BOi} : K(\mathsf{BOi}(x)) \rightsquigarrow \mathbf{add}\{\mathsf{BOi}-1(x)\}$ for $0 < i \leq n+1$
$e_{AOi} : K(\mathsf{AOi}(x)) \rightsquigarrow \mathbf{add}\{\mathsf{AOi}-1(x)\}, \mathbf{del}\{\mathsf{AOi}(x)\}$ for $0 < i \leq n$
$e_{Rowi} : K(\mathsf{Col_i}(x)) \rightsquigarrow \mathbf{add}\{\mathsf{Row_{i-1}}(x)\}, \mathbf{del}\{\mathsf{Row_i}(x)\}$ for $0 < i \leq n$

Unfortunately, although *DL-Lite*$_{Horn}$ axioms are FOL-rewritable, the current available tools do not support them. To overcome this, we introduce an additional effects of the type:
$e_{sure\ i} : K(\mathsf{ROi}(x) \wedge \mathsf{LOi}+1(x)) \rightsquigarrow \mathbf{add}\{\mathsf{Col_{i+1}}(x)\}$ for $0 \leq i < m$
for each action. We also have to make sure that the initial ABox is saturated under the axioms $\mathsf{ROi} \sqcap \mathsf{LOi}+1 \sqsubseteq \mathsf{Col_i}$. To simplify things, we consider only initial ABoxes in which all the membership axioms that could be derived from the axioms $\mathsf{ROi} \sqcap \mathsf{LOi}+1 \sqsubseteq \mathsf{Col_i}$ are explicitly included in the ABox.

The goal we want to reach is to have the robot in a specific cell, e.g., $G = \mathsf{Col3}(\mathsf{robot}) \wedge \mathsf{Row4}(\mathsf{robot})$. We now have all the elements to evaluate how the translation

Grid Size	Time (s)
3x3	0.43
4x4	0.50
5x5	3.51
6x6	61.22
7x7	>1800

Table 6.1: Solving times for the robot problem

of such eKAB planning problem to an ADL one (and its resolution through an off-the-shelf planner) performs.

Having defined the eKAB planning problem, we translated it, and fed it to an ADL planner (we used *Fast Downward* [2]) ran over a machine equipped with an Intel Core i7-4600U CPU and 16 GB of RAM. The goal was to run the example multiple times, and increase the grid size every time while noting down how the timings for finding a plan get affected, considering a 30 minutes time-out. We report the results in Table 6.1.

Unfortunately, already with a grid size of 7x7, the planner reaches the time-out limit (it solved the problem in around 2.210 seconds), which is clearly not an acceptable result. This can be traced down to the structure of the planning problem and how the planner deals with its translation to ADL. eKABs are a very expressive framework, and, as we can see from this example, we can easily define actions with multiple effects, and queries that uses quan-

[2] `http://www.fast-downward.org/`

tification, negation, etc. The planner translates this setting to a STRIPS one, generating a problem exponential in size, which, consequently, burdens the planner execution. This problem is not peculiar to the chosen planner: almost every major planner adopts this technique to deal with ADL planning problems.

Of course, it would be possible to tailor the planning domain and generate a more efficient encoding, but this can be done either manually (which defies the scope of our work), or by studying generic optimizations (which is left as future work).

Conclusions 7

7.1 Summary

This Thesis addressed the problem of plan synthesis over rich dynamic data-centric systems. We presented a framework named *Explicit-input Knowledge and Action Base* (eKAB) (Chapter 3), where data is taken care by a full-fledged Description Logic Knowledge Base, while a set of actions governs its evolution by adding or removing assertions. The expressive power and reasoning services of DLs are very helpful to describe and manage the domain knowledge, but constitute a difficult environment to deal with when it comes to the dynamics of the processes.

Additionally, as the actions have the possibility to introduce new instances, leads to a framework where checking plan existence is undecidable even under severe restrictions on the eKAB of interest. Nonetheless, we demonstrated that, under the state-boundedness assumption for eKABs, plan existence is PSpace in data complexity (Section 4.1). In this setting, we presented sound, complete and terminating algorithms for plan synthesis by fusing together classic planning algorithms with DL reasoning services (Section 4.2).

We then focused our attention on eKABs equipped with a lightweight DL of the *DL-Lite* family, and proposed a technique to compile plan synthesis for state-bounded, lightweight *Reduced Explicit-input Knowledge and Action Bases* (reKABs) into a standard STRIPS planning problem (Section 2.3.1), and for state-bounded, lightweight eKABs into a standard ADL planning problem (Section 2.3.2). More over, in the case of reKABs, we provided a technique to embed the consistency check of the generated ABoxes directly into the condition of the actions. These techniques allow to transform an highly expressive DL-based setting into standard planning, where the reasoning services (namely, consistency check and query answering) contribution is embedded into the translation, and planning tools and optimizations can be seamlessly applied to the problem.

To test the feasibility of our proposal for knowledge-intensive planning, we ran an empirical evaluation of the framework (Chapter 6), where we considered two different

examples, translated them into ADL planning problems written in PDDL, and fed them as input to an off-the-shelf planner. Unsurprisingly, performances of the resolution of vanilla translations are poor; to the best of our knowledge, planners that support ADL planning problems, transform them into equivalent STRIPS problem, an operation that is worst-case exponential. We have to remark, though, that the work of this Thesis was aimed at providing the theoretical foundation for plan synthesis in eKABs and the possibility to reduce it to standard planning; we consider possible optimizations (such as the use of heuristics, and other well know planning optimization techniques) as future work.

7.2 Future Work

Data-intensive dynamic systems are an open and vibrant research topic, and the ideas and contributions about this topic proposed in this Thesis can be refined and extended. Here follow some considerations on possible future work.

First of all, the plan synthesis process for DL-agnostic eKABs can be improved by adopting better algorithms, both adapting existing ones and creating novel ones (as we are dealing with a non-standard planning setting). In this respect, a promising line of research is to combine our approach with that of [Fan et al., 2012]. There, the authors study a knowledge-intensive variant of Golog operating over an infinite object domain, and where incom-

plete knowledge is captured using *proper KBs* instead of DLs. In particular, they propose a semi-decidable technique to handle progression and query evaluation. In spite of the key differences between that approach and ours, their approach is reminiscent to the abstraction technique we adopt towards decidability. We intend to leverage this similarity with the aim of understanding if, and how, the optimization strategies proposed in [Fan et al., 2012] can be lifted to our setting.

Another line of work is, of course, improve the performance of plan synthesis for the translations into STRIPS and ADL planning problems: this can be done by exploring few ways, such as implementation of heuristics that take advantage of the structured knowledge inside the eKABs, and improve the translation procedures by optimizing the generated elements (most of all, the queries that govern the actions).

Bibliography

[Abdulla et al., 2016] Abdulla, P. A., Aiswarya, C., Atig, M. F., Montali, M., and Rezine, O. (2016).
Recency-bounded verification of dynamic database-driven systems (extended version).
CoRR, abs/1604.03413.

[Abiteboul et al., 1995] Abiteboul, S., Hull, R., and Vianu, V. (1995).
Foundations of Databases.
Addison Wesley Publ. Co.

[Baader et al., 2003] Baader, F., Calvanese, D., McGuinness, D., Nardi, D., and Patel-Schneider, P. F., editors (2003).
The Description Logic Handbook: Theory, Implementation and Applications.
Cambridge University Press.

[Baader and Zarrieß, 2013] Baader, F. and Zarrieß, B. (2013).

Verification of Golog programs over description logic actions.
Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 8152 LNAI:181–196.

[Bacchus, 2000] Bacchus, F. (2000).
Subset of PDDL for the AIPS 2000 Planning Competition.

[Bagheri Hariri et al., 2013a] Bagheri Hariri, B., Calvanese, D., De Giacomo, G., Deutsch, A., and Montali, M. (2013a).
Verification of relational data-centric dynamic systems with external services.
In *Proc. of PODS 2013*, pages 163–174.

[Bagheri Hariri et al., 2014] Bagheri Hariri, B., Calvanese, D., Deutsch, A., and Montali, M. (2014).
State-boundedness in data-aware dynamic systems.
In *Proc. of KR 2014*. AAAI Press.

[Bagheri Hariri et al., 2013b] Bagheri Hariri, B., Calvanese, D., Montali, M., De Giacomo, G., De Masellis, R., and Felli, P. (2013b).
Description logic Knowledge and Action Bases.
J. of Artificial Intelligence Research, 46:651–686.

[Belardinelli et al., 2014] Belardinelli, F., Lomuscio, A., and Patrizi, F. (2014).
Verification of agent-based artifact systems.
J. of Artificial Intelligence Research, 51:333–376.

[Bhattacharya et al., 2007] Bhattacharya, K., Gerede, C., and Hull, R. (2007).
Towards formal analysis of artifact-centric business process models.
Business Process Management, pages 288–304.

[Bylander, 1994] Bylander, T. (1994).
The computational complexity of propositional STRIPS planning.
Artificial Intelligence, 69(1–2):165–204.

[Calvanese et al., 2009] Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Poggi, A., Rodriguez-Muro, M., and Rosati, R. (2009).
Ontologies and databases: The *DL-Lite* approach.
In Tessaris, S. and Franconi, E., editors, *RW 2009 Tutorial Lectures*, volume 5689 of *LNCS*, pages 255–356. Springer.

[Calvanese et al., 2007a] Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., and Rosati, R. (2007a).
EQL-Lite: Effective first-order query processing in description logics.
In *Proc. of IJCAI 2007*, pages 274–279.

[Calvanese et al., 2007b] Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., and Rosati, R. (2007b).
Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family.
J. of Automated Reasoning, 39(3):385–429.

[Calvanese et al., 2013a] Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., and Rosati, R. (2013a).
Data complexity of query answering in description logics.
Artificial Intelligence, 195:335–360.

[Calvanese et al., 2013b] Calvanese, D., De Giacomo, G., Montali, M., and Patrizi, F. (2013b).
Verification and Synthesis in Description Logic Based Dynamic Systems, volume 7994 of *Lecture Notes in Computer Science*.
Springer Berlin Heidelberg, Berlin, Heidelberg.

[Calvanese et al., 2013c] Calvanese, D., De Giacomo, G., Montali, M., and Patrizi, F. (2013c).
Verification and synthesis in description logic based dynamic systems.
In *Proc. of RR 2013*, volume 7994 of *LNCS*, pages 50–64. Springer.

[Calvanese et al., 2013d] Calvanese, D., Kharlamov, E., Montali, M., Santoso, A., and Zheleznyakov, D. (2013d).
Verification of inconsistency-aware knowledge and action bases.
In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, pages 810–816. AAAI Press.

[Calvanese et al., 2016] Calvanese, D., Montali, M., Patrizi, F., and Stawowy, M. (2016).

Plan synthesis for knowledge and action bases.
In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*.

[Calvanese et al., 2015] Calvanese, D., Montali, M., and Santoso, A. (2015).
Verification of generalized inconsistency-aware knowledge and action bases (extended version).
CoRR, abs/1504.08108.

[Chandra and Merlin, 1977] Chandra, A. K. and Merlin, P. M. (1977).
Optimal implementation of conjunctive queries in relational data bases.
In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 77–90. ACM.

[Cimatti et al., 2008] Cimatti, A., Pistore, M., and Traverso, P. (2008).
Automated planning.
In *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*, pages 841–867. Elsevier.

[Cohn and Hull, 2009] Cohn, D. and Hull, R. (2009).
Business artifacts: A data-centric approach to modeling business operations and processes.
IEEE Data Eng. Bull, 32(3):3–9.

[De Giacomo et al., 2014] De Giacomo, G., Lespérance, Y., Patrizi, F., and Vassos, S. (2014).

Progression and verification of situation calculus agents with bounded beliefs.
In *Proc. ofAAMAS'14*.

[Drescher and Thielscher, 2008] Drescher, C. and Thielscher, M. (2008).
A fluent calculus semantics for ADL with plan constraints.
In *Proc. of JELIA 2008*, volume 5293 of *LNCS*, pages 140–152. Springer.

[Erol et al., 1995] Erol, K., Nau, D. S., and Subrahmanian, V. S. (1995).
Complexity, decidability and undecidability results for domain-independent planning.
Artificial Intelligence, 76(1–2):75–88.

[Fan et al., 2012] Fan, Y., Cai, M., Li, N., and Liu, Y. (2012).
A first-order interpreter for knowledge-based golog with sensing based on exact progression and limited reasoning.
In *Proc. of AAAI 2012*. AAAI Press.

[Gabbay et al., 2003] Gabbay, D., Kurusz, A., Wolter, F., and Zakharyaschev, M. (2003).
Many-dimensional Modal Logics: Theory and Applications.

[Gazen and Knoblock, 1997] Gazen, B. C. and Knoblock, C. A. (1997).

Combining the Expressivity of UCPOP with the Efficiency of Graphplan.
Proc. 4Th European Conference on Planning, pages 221—-233.

[Ghallab et al., 2004a] Ghallab, M., Nau, D. S., and Traverso, P. (2004a).
Automated planning – Theory and Practice.
Elsevier.

[Ghallab et al., 2004b] Ghallab, M., Nau, D. S., and Traverso, P. (2004b).
Automated planning - theory and practice.
Elsevier.

[Ginsberg and Smith, 1988] Ginsberg, M. L. and Smith, D. E. (1988).
Reasoning about action ii: The qualification problem.
Artif. Intell., 35(3):311–342.

[Glimm et al., 2008] Glimm, B., Lutz, C., Horrocks, I., and Sattler, U. (2008).
Conjunctive query answering for the description logic $\mathcal{SHIQ}$.
J. of Artificial Intelligence Research, 31:151–198.

[Hariri et al., 2013] Hariri, B. B., Calvanese, D., Montali, M., De Giacomo, G., Masellis, R. D., and Felli, P. (2013).
Description Logic Knowledge and Action Bases.
J. Artif. Intell. Res. (JAIR), 46:651–686.

[Hoffmann et al., 2009] Hoffmann, J., Bertoli, P., Helmert, M., and Pistore, M. (2009).
Message-based web service composition, integrity constraints, and planning under uncertainty: A new connection.
J. of Artificial Intelligence Research, 35:49–117.

[Levesque, 1984] Levesque, H. J. (1984).
Foundations of a Functional Approach to Knowledge Representation.
Artif. Intell., 23(2):155–212.

[Levesque and Lakemeyer, 2001] Levesque, H. J. and Lakemeyer, G. (2001).
The Logic of Knowledge Bases.
The MIT Press.

[Minsky, 1967] Minsky, M. L. (1967).
Computation: Finite and Infinite Machines.
Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Montali et al., 2014] Montali, M., Calvanese, D., and De Giacomo, G. (2014).
Verification of Data-Aware Commitment-Based Multiagent System.
(Aamas):157–164.

[Ortiz et al., 2008] Ortiz, M., Calvanese, D., and Eiter, T. (2008).
Data complexity of query answering in expressive description logics via tableaux.
J. of Automated Reasoning, 41(1):61–98.

[Pednault, 1989] Pednault, E. P. D. (1989).
Adl: Exploring the middle ground between strips and the situation calculus.
In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pages 324–332, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Rudolph, 2011] Rudolph, S. (2011).
Foundations of description logics.

[Stawowy, 2015] Stawowy, M. (2015).
Optimizations for decision making and planning in description logic dynamic knowledge bases.
In *Proceedings of the 28th International Workshop on Description Logics*.

[Wolter and Zakharyaschev, 1999] Wolter, F. and Zakharyaschev, M. (1999).
Temporalizing description logic.
In Gabbay, D. and de Rijke, M., editors, *Frontiers of Combining Systems*, pages 379–402. Studies Press/Wiley.

[Zarrieß and Claßen, 2015] Zarrieß, B. and Claßen, J. (2015).
Verification of knowledge-based programs over description logic actions.
In *Proc. of IJCAI'15*.