Consistency Checking of Re-engineered UML Class Diagrams via Datalog+/-

Georg Gottlob[1], Giorgio Orsi[1], and Andreas Pieris[2(✉)]

[1] Department of Computer Science, University of Oxford, Oxford, UK
{georg.gottlob,giorgio.orsi}@cs.ox.ac.uk
[2] Institute of Information Systems, Vienna University of Technology, Vienna, Austria
pieris@dbai.tuwien.ac.at

Abstract. UML class diagrams (UCDs) are a widely adopted formalism for modeling the intensional structure of a software system. Although UCDs are typically guiding the implementation of a system, it is common in practice that developers need to recover the class diagram from an implemented system. This process is known as reverse engineering. A fundamental property of reverse engineered (or simply re-engineered) UCDs is consistency, showing that the system is realizable in practice. In this work, we investigate the consistency of re-engineered UCDs, and we show is PSPACE-complete. The upper bound is obtained by exploiting algorithmic techniques developed for conjunctive query answering under guarded Datalog+/-, that is, a key member of the Datalog+/- family of KR languages, while the lower bound is obtained by simulating the behavior of a polynomial space Turing machine.

1 Introduction

Models play a central role in computer science by providing two fundamentally different representational functions: they can be used to capture interesting aspects of the real world, and they can also be employed to represent axioms of abstract theories. System designers use models for representing the requirements and the architecture of software systems. The urge for model construction, maintenance and manipulation becomes evident as soon as systems and data grow in size and complexity.

1.1 UML Class Diagrams

UML class diagrams (UCDs) are a widely adopted formalism for modeling the intensional structure of a software system, and are commonly employed in CASE tools for system design, maintenance and analysis. In fact, UCDs are used to represent classes (entities) of a domain of interest with their attributes (fields) and operations (methods). Classes can be related to each other by means of associations representing relationships among their instances. Due to their simplicity, UCDs are frequently used also for data modeling, de-facto replacing traditional

N. Bassiliades et al. (Eds.): RuleML 2015, LNCS 9202, pp. 35–53, 2015.
DOI: 10.1007/978-3-319-21542-6_3

formalisms like the ER model. Although the usual procedure is to go from a class diagram to a system, it is common that developers need to follow the opposite route, i.e., to recover the class diagram from an implemented system. This process is known as *reverse engineering* [14].

Apart from guiding the implementation of a software system, class diagrams can be used to verify relevant properties so as to assess the quality of a specification to objective criteria. The typical property of interest is *consistency*, proving that the system is realizable in practice, namely its classes can be populated without violating any of the imposed constraints.

1.2 Research Challenges

It is apparent that consistency checking is a key algorithmic task that is relevant for re-engineered class diagrams. UCDs for complex systems usually become very large, and the various constraints may interact in an arbitrary way. This makes the study of the above task urgent, and at the same time very challenging. While consistency checking has been heavily investigated in the past in different scenarios (see, e.g., [3,4,8,12]), nothing is known in the case of re-engineered class diagrams. It is the precise aim of this work to pinpoint the computational complexity of this problem under re-engineered class diagrams.

Towards this direction, we first need to answer the following key question: which fragment of UCDs can be recovered by existing reverse engineering tools? To answer this question, we set up a simple experiment to determine which constructs appear in re-engineered class diagrams. We observed that the constructs that can be recovered are: (1) classes with attributes and operations, where different classes may have attributes/operations with the same name; (2) generalization hierarchies but without completeness assertions; and (3) associations with mandatory or functional participation of classes. This led us to the formalization of the syntax and the semantics of the fragment of UCDs, dubbed RevEng, which can be re-engineered.

After formalizing RevEng diagrams, we proceed with the investigation of the computational complexity of our problem. One may claim that the desired complexity results can be immediately inherited from existing results on UML class diagrams, for instance in [4] which shows that consistency of UCDs is EXPTIME-complete, or results on knowledge representation formalisms such as, e.g., DL-Lite [9], $\mathcal{EL}$ [2] and Horn-$\mathcal{FL}^-$ [13]. This is not true since always the candidate formalism is either not expressive enough to capture RevEng class diagrams, or gives an upper bound which is not optimal. Therefore, RevEng class diagrams form a totally novel formalism w.r.t. complexity, and novel decision procedures beyond the state of the art must be developed.

We exploit algorithmic techniques developed for conjunctive query answering under guarded Datalog$^{\pm}$, that is, a key member of the Datalog$^{\pm}$ family of KR languages [5,6]. Given a RevEng class diagram $\mathcal{C}$, the problem of deciding whether $\mathcal{C}$ is consistent can be naturally reduced to conjunctive query answering under a fragment of guarded Datalog$^{\pm}$. In particular, we construct the following three components: a database D, which stores a witness atom for each class of $\mathcal{C}$; a

set of guarded Datalog$^{\pm}$ rules Σ, which represents $\mathcal{C}$; and a union of conjunctive queries Q that encodes the disjointness assertions among classes, which form the only source of inconsistency occurring in $\mathcal{C}$. The consistency problem of a diagram is then tantamount to the problem of deciding whether D and Σ do not entail the query Q, which in turn implies that there are no inconsistencies. The latter is tackled by exploiting a classical algorithmic tool from the database literature, in particular the *chase algorithm* (see, e.g., [11]), and a novel chase-like decision procedure is proposed.

1.3 Summary of Contributions

Our contribution can be summarized as follows:

1. We set up a simple experiment in Section 2 with the aim of understanding which UML constructs can be recovered by existing reverse engineering algorithms. In particular, we collect a number of Java open-source software packages, mostly taken from the literature on the benchmarking of UML reverse engineering tools. We then consider several prominent CASE tools for software engineering, and we reverse engineer the packages in the benchmark into UCDs. We observe that the UML constructs that can be recovered are: classes with attributes and operations; generalization hierarchies but without completeness assertions; and associations with multiplicities 0..1, 1..1, 0..∞ and 1..∞. Based on the above observation, we then provide a formalization of the syntax and the semantics of the fragment of UCDs, called RevEng, which can be recovered.
2. We consider the problem of deciding the consistency of RevEng diagrams in Section 3. We reduce our problem to query answering under a fragment of guarded Datalog$^{\pm}$, which in turn is shown to be PSPACE-complete. The upper bound is obtained via a novel nondeterministic chase-like algorithm, while the lower bound is shown by simulating the behavior of a polynomial space Turing machine by means of a RevEng diagram.

2 Reverse Engineering

We set up a simple experiment to determine which fragment of UCDs, called RevEng, can be recovered via reverse engineering, and then we provide a formalization of the syntax and the semantics of RevEng.

2.1 Our Experiment

We collected a number of Java open-source software packages, listed in Figure 1, mostly taken from the DaCapo benchmark[1] and the web. We then considered a list of prominent CASE tools with reverse engineering capabilities, given in

[1] http://www.dacapobench.org/

Software	Version
antlr	3.5.1
eje	3.2
fop	0.93
hsqldb	2.31
jamaleon	3.3
jolden	N/A
junit	4.11
pcj	1.2
pmd	5.05
Vuze	5.2

CASE Tool	Version	Vendor
ARGOUML	0.34	Tigris
ASTAH	6.7	Astah
BOUML	6.4.7	Bouml
ENTREPRISE ARCHITECT	10	Sparx Systems
ESS MODEL	2.2	ESSModel
MAGICDRAW	17.0.2	LTR NoMagic
METAMILL	6.1	Metamill Software
POSEIDON UML	6.0.2	Gentleware
UMODEL	2014 SP1	Altova
VISUALPARADIGM	11	VisualParadigm

Fig. 1. Software packages and CASE tools

Figure 1. We re-engineered the packages in the benchmark into class diagrams in XMI format for automated processing. Whenever multiple options for reverse engineering were available, e.g., for fields, we used the option that would result in the more general diagram. We observed, in fact, that interpreting fields as attributes leads to simpler diagrams. We noticed that every single re-engineered class diagram consists of the following: (1) Classes with attributes and operations, where different classes may have attributes/operations with the same name; (2) Generalization hierarchies (is-a) but without completeness assertions; and (3) Associations with multiplicities with one of the following forms: 0..1, 1..1, 0..∞ and 1..∞.

Interestingly, when recovering fields as associations, the tools are often unable to recover the exact multiplicity of the association. A possible explanation for this unexpected behavior is that tools tend not to constrain the upper multiplicity when collections and arrays are involved. This seems not to affect fields referencing another class, where a simple check on the assignment of these fields in either the class constructor or in the field declaration provides enough information to determine the correct multiplicity. Another interesting observation is on the lower bounds of the associations that are often recovered as 1 despite having no evidence of that happening from the code.

2.2 Formalizing Reverse Engineered UCDs

Based on the above observations, we proceed to formalize the syntax of UCDs, called RevEng, that can be obtained by reverse engineering, and also give their formal semantics in terms of first-order logic.

Syntax. A *class*, possibly with *attributes* and *operations*, represents a set of objects with common features, and is graphically represented as shown in Figure 2(a); notice that both the middle and the bottom part are optional.

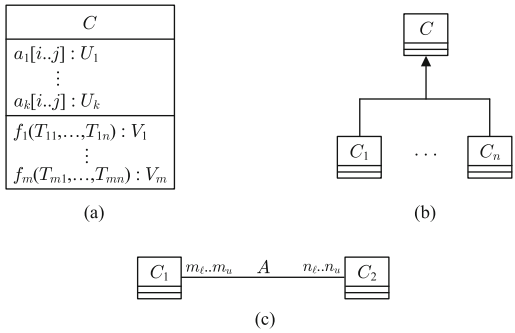


Fig. 2. RevEng UML class diagram constructs

An attribute assertion of the form $a[i..j] : T$, where $i \in \{0, 1\}$ and $j \in \{1, \infty\}$, states that the class C has an attribute a of *type*[2] T, where the optional multiplicity $[i..j]$ specifies that a associates to each instance of C at least i and at most j instances of T. Notice that attributes are unique within a class. However, different classes may have attributes with the same name, possibly with different types. An operation of a class C is a function from the instances of C (and possibly additional parameters) to objects and values. An operation assertion of the form $f(T_1, \ldots, T_n) : V$ asserts that the class C has an operation f with $n \geqslant 0$ parameters, where its i-th parameter is of type T_i and its result is of type V. Let us clarify that the class diagram represents only the *signature*, that is, the name of the functions, the number and the types of their parameters, and the type of their result. Notice that operations are unique within a class. However, different classes may have operations with the same name, possibly with different signature but the same number of parameters. One can use *class generalization* to assert that each instance of a child class is also an instance of the parent class. Several generalizations can be grouped together to form a class hierarchy, as shown in Figure 2(b).

An *association* is a relation between the instances of two classes, that are said to *participate* in the association. Names of associations are unique in the diagram. An association A between two classes C_1 and C_2 is graphically represented as in Figure 2(c). The multiplicity $n_\ell..n_u$, where $n_\ell \in \{0, 1\}$ and $n_u \in \{1, \infty\}$, specifies that each instance of class C_1 can participate at least n_ℓ times and at most n_u times to A; analogously we have $m_\ell..m_u$ for C_2.

Sometimes, in UML class diagrams, it is assumed that all classes not in the same hierarchy are *disjoint*. In this work, we do not enforce this assumption, and we allow two classes to have common instances. When needed, disjointness can be enforced by means of assertions of the form $\{C_1, \ldots, C_n\}$, stating that the classes $C_1, \ldots, C_n$ do not have a common instance. Another standard assumption in UML class diagrams is the *most specific class* assumption, stating that objects in a hierarchy must belong to a single most specific class. We do not enforce this

[2] For simplicity, data types, i.e., collections of values such as integers, are considered as classes, i.e., as collections of objects.

assumption, and two classes in a hierarchy may have common instances, even though they may not have a common subclass. When needed, the existence of the most specific class can be enforced by means of disjointness assertions and most specific class assertions of the form $(\{C_1, \ldots, C_n\}, C_{n+1})$ stating that, if $C_1, \ldots, C_n$ have a common instance c, then c is also an instance of C_{n+1}, i.e., C_{n+1} is a most specific class for $C_1, \ldots, C_n$.

Let $class(\mathcal{C})$ be the set of classes occurring in the diagram $\mathcal{C}$. A RevEng *specification* is a triple $(\mathcal{C}, \mathsf{DISJ}, \mathsf{MSC})$, where $\mathcal{C}$ is a RevEng UML class diagram, $\mathsf{DISJ} \subseteq 2^{class(\mathcal{C})}$ is a set of disjointness assertions, and $\mathsf{MSC} \subseteq 2^{class(\mathcal{C})} \times class(\mathcal{C})$ is a set of most specific class assertions. Notice that DISJ and MSC can be seen as sets of constraints expressed using the *object constraint language (OCL)*[3]. OCL is an expressive language that allows us to impose additional constraints which are not diagrammatically expressible in a UCD. Although OCL has its own syntax, for brevity, we consider the simpler syntax presented above.

Semantics. The formal semantics of RevEng specifications is given in terms of first-order logic (FOL). Given a RevEng specification $\mathcal{S} = (\mathcal{C}, \mathsf{DISJ}, \mathsf{MSC})$, we first define the translation τ of $\mathcal{S}$ into FOL. The semantics of $\mathcal{S}$ is defined as certain models of the first-order theory $\tau(\mathcal{S})$. The formalization adopted here is based on the one presented in [4,12]. For brevity, let $[n] = \{1, \ldots, n\}$, for $n > 0$.

A class C occurring in $\mathcal{C}$ is represented by a unary predicate C, while an attribute a for class C corresponds to a binary predicate a. The attribute assertion $a[i..j] : T$ is translated into:

$$\forall X \forall Y\, (C(X) \wedge a(X, Y) \rightarrow T(Y)),$$
$$\forall X\, (C(X) \rightarrow \exists Y\, a(X, Y)), \text{ if } i = 1,$$
$$\forall X \forall Y \forall Z\, (C(X) \wedge a(X, Y) \wedge a(X, Z) \rightarrow Y = Z), \text{ if } j = 1.$$

The first one asserts that for each instance c of C, an object c' related to c by the attribute a is an instance of T. The second and the third assertions state that for each instance c of C, there exist at least one and at most one different objects, respectively, related to c by a. An operation f, with $m \geqslant 0$ parameters, for class C corresponds to an $(m+2)$-ary predicate f, and the operation assertion $f(T_1, \ldots, T_m) : T$ is translated into:

$$\forall X \forall Y_1 \ldots \forall Y_m \forall Z\, (C(X) \wedge f(X, Y_1, \ldots, Y_m, Z) \rightarrow T_i(Y_i)), \text{ for each } i \in [m],$$
$$\forall X \forall Y_1 \ldots \forall Y_m \forall Z\, (C(X) \wedge f(X, Y_1, \ldots, Y_m, Z) \rightarrow T(Z)),$$
$$\forall X \forall Y_1 \ldots \forall Y_m \forall Z \forall W\, (C(X) \wedge f(X, Y_1, \ldots, Y_m, Z) \wedge f(X, Y_1, \ldots, Y_m, W) \rightarrow Z = W).$$

The first two impose the correct typing for the parameters and the result, and the third one asserts that the operation f is a function from the instances of C and the parameters to the result. A class hierarchy, as the one in Figure 2(b), is translated into:

$$\forall X\, (C_i(X) \rightarrow C(X)), \text{ for each } i \in [n],$$

[3] http://www.omg.org/spec/OCL/

which assert that each instance of C_i is an instance of C. An association A occurring in $\mathcal{C}$ corresponds to a binary predicate A. If A is among classes C_1 and C_2 with multiplicities $m_\ell..m_u$ and $n_\ell..n_u$, then we have the FOL assertions:

$$\begin{aligned}
&\forall X \forall Y\,(A(X,Y) \rightarrow C_1(X)),\\
&\forall X \forall Y\,(A(X,Y) \rightarrow C_2(Y)),\\
&\forall X\,(C_1(X) \rightarrow \exists Y\,A(X,Y)),\ \text{if } n_\ell = 1,\\
&\forall X \forall Y \forall Z\,(C_1(X) \wedge A(X,Y) \wedge A(X,Z) \rightarrow Y = Z),\ \text{if } n_u = 1,\\
&\forall X\,(C_2(X) \rightarrow \exists Y\,A(Y,X)),\ \text{if } m_\ell = 1,\\
&\forall X \forall Y \forall Z\,(C_2(X) \wedge A(Y,X) \wedge A(Z,X) \rightarrow Y = Z),\ \text{if } m_u = 1.
\end{aligned}$$

An assertion $\{C_1, \ldots, C_n\} \in \mathsf{DISJ}$ is translated into

$$\forall X\,(C_1(X) \wedge \ldots \wedge C_n(X) \rightarrow \bot),$$

where $\bot$ denotes the truth constant *false*, while an assertion $(\{C_1, \ldots, C_n\}, C_{n+1}) \in \mathsf{MSC}$ is translated into

$$\forall X\,(C_1(X) \wedge \ldots \wedge C_n(X) \rightarrow C_{n+1}(X)).$$

We are now ready to define the semantics of RevEng specifications via FOL. We consider the following pairwise disjoint sets of symbols: a set $\mathbf{C}$ of *constants* and a set $\mathbf{N}$ of *labeled nulls* (used as placeholders for unknown values, and thus can be also seen as globally existentially quantified variables). Different constants represent different values (*unique name assumption*), while different nulls may represent the same value. An *interpretation* $\mathcal{I} = (\Delta, \mu)$ consists of a non-empty *interpretation domain* $\Delta \subseteq \mathbf{C} \cup \mathbf{N}$, and an *interpretation function* μ for a first-order language. Let $\mathcal{S} = (\mathcal{C}, \mathsf{DISJ}, \mathsf{MSC})$ be a RevEng specification. A *UML-model* of $\mathcal{S}$ is an interpretation $\mathcal{I} = (\Delta, \mu)$ such that *(i)* $\mathcal{I}$ satisfies the first-order theory $\tau(\mathcal{S})$, written $\mathcal{I} \models \tau(\mathcal{S})$; and *(ii)* for each $C \in \mathit{class}(\mathcal{C})$, $\mu(C) \neq \varnothing$. The first condition above implies that $\mathcal{I}$ is a *first-order model* (or simply *FO-model*) of the theory $\tau(\mathcal{S})$, while the second condition indicates that each class in $\mathcal{I}$ is non-empty, i.e., an instance of each class exists without violating any of the requirements imposed by the specification.

3 Consistency Check of Diagrams

The fact that RevEng specifications can be translated into FOL allows one to formally check relevant properties so as to assess the quality of a specification to objective quality criteria. The typical property of interest is consistency: a RevEng specification $\mathcal{S}$ is *consistent* if there exists at least one UML-model of $\mathcal{S}$. We proceed to pinpoint the exact complexity of the problem of deciding whether a RevEng specification is consistent.

Fix a RevEng specification $\mathcal{S} = (\mathcal{C}, \mathsf{DISJ}, \mathsf{MSC})$. To check the consistency of $\mathcal{S}$ it suffices to add to the first-order theory $\tau(\mathcal{S})$ a witness for each class of

class($\mathcal{C}$), and then check whether the obtained theory has at least one FO-model, i.e., is satisfiable. In other words, we can reduce our problem to the satisfiability problem of a first-order theory. Assuming that *class*($\mathcal{C}$) = $\{C_1, \ldots, C_n\}$, let $\mathcal{W}_\mathcal{S}$ be the conjunction of atomic formulas $(C_1(c_1) \wedge \ldots \wedge C_n(c_n))$, where $c_1, \ldots, c_n$ are arbitrary constants of $\mathbf{C}$, and let $\Phi_\mathcal{S}$ be the sentence $(\mathcal{W}_\mathcal{S} \wedge \tau(\mathcal{S}))$. It is not difficult to show that:

Lemma 1. *$\mathcal{S}$ is consistent iff $\Phi_\mathcal{S}$ is satisfiable.*

In the following, we investigate the satisfiability of $\Phi_\mathcal{S}$. Observe that, if $\Phi_\mathcal{S}$ is satisfiable, then it has an FO-model $\mathcal{I} = (\Delta, \mu)$ where $\mu(f) = \varnothing$, for each operation f in $\mathcal{S}$, since the absence of an operation atom cannot lead to a violation of $\Phi_\mathcal{S}$. This implies that the conjuncts that appear in $\tau(\mathcal{S})$ because of an operation assertion are irrelevant for satisfiability purposes and can be safely ignored; in the rest of this section, we exclude from $\tau(\mathcal{S})$ those formulas. By definition, $\tau(\mathcal{S})$ can be equivalently rewritten (by simply reordering its conjuncts) as the conjunction $(\mathcal{X}_\mathcal{S} \wedge \mathcal{E}_\mathcal{S} \wedge \mathcal{F}_\mathcal{S})$, where:

- $\mathcal{X}_\mathcal{S}$ is a conjunction of formulas of the form $\forall \mathbf{X}\,(\varphi(\mathbf{X}) \rightarrow \exists Y\, \alpha(\mathbf{X}, Y))$ (possibly without existentially quantified variables);
- $\mathcal{E}_\mathcal{S}$ is a conjunction of formulas of the form $\forall \mathbf{X}\,(\varphi(\mathbf{X}) \rightarrow X_i = X_j)$; and
- $\mathcal{F}_\mathcal{S}$ is a conjunction of formulas of the form $\forall \mathbf{X}\,(\varphi(\mathbf{X}) \rightarrow \bot)$.

The following technical result follows immediately:

Lemma 2. *$\Phi_\mathcal{S}$ is satisfiable iff the following hold:*

1. *$(\mathcal{W}_\mathcal{S} \wedge \mathcal{X}_\mathcal{S} \wedge \mathcal{E}_\mathcal{S})$ is satisfiable; and*
2. *there exists an FO-model $\mathcal{I}$ of $(\mathcal{W}_\mathcal{S} \wedge \mathcal{X}_\mathcal{S} \wedge \mathcal{E}_\mathcal{S})$ such that $\mathcal{I} \models \mathcal{F}_\mathcal{S}$.*

3.1 A Database-Theoretic Approach

Interestingly, the two decision problems stated in Lemma 2 can be tackled following a database-theoretic approach:

- The conjunction $\mathcal{W}_\mathcal{S} = (\alpha_1 \wedge \ldots \wedge \alpha_n)$ can be seen as the relational database $D_\mathcal{S} = \{\alpha_1, \ldots, \alpha_n\}$;
- The conjunction $\mathcal{X}_\mathcal{S} = (\sigma_1 \wedge \ldots \wedge \sigma_m)$ can be conceived as the set $T_\mathcal{S} = \{\sigma_1, \ldots, \sigma_m\}$ of *tuple-generating dependencies (TGDs)*;
- The conjunction $\mathcal{E}_\mathcal{S} = (\eta_1 \wedge \ldots \wedge \eta_k)$ can be seen as the set $E_\mathcal{S} = \{\eta_1, \ldots, \eta_k\}$ of *equality-generating dependencies (EGDs)*; and
- The conjunction $\mathcal{F}_\mathcal{S} = (\nu_1 \wedge \ldots \wedge \nu_\ell)$ can be conceived as the *union of conjunctive queries (UCQs)* $Q_\mathcal{S} = (q_{\nu_1} \vee \ldots \vee q_{\nu_\ell})$, where, assuming that ν is of the form $\forall \mathbf{X}\,(\varphi(\mathbf{X}) \rightarrow \bot)$, q_ν is the conjunctive query $\exists \mathbf{X}\,(\varphi(\mathbf{X}))$.

Tuple- and equality-generating dependencies are well-known in the database world as a unifying framework for classical database dependencies such as inclusion and functional dependencies [1], and form the basis of the Datalog$^\pm$ family

of KR languages [7]. Conjunctive queries correspond to the select-project-join fragment of relational algebra, and form one of the most natural and commonly used languages for querying relational databases [1].

An FO-model of $(\mathcal{W}_\mathcal{S} \wedge \mathcal{X}_\mathcal{S} \wedge \mathcal{E}_\mathcal{S})$ can be equivalently defined as a relational instance I, called a *model* of $D_\mathcal{S}$ w.r.t. $T_\mathcal{S} \cup E_\mathcal{S}$, such that $I \supseteq D_\mathcal{S}$ and I satisfies $T_\mathcal{S} \cup E_\mathcal{S}$ (written as $I \models T_\mathcal{S} \cup E_\mathcal{S}$); I satisfies $\forall \mathbf{X}\,(\varphi(\mathbf{X}) \rightarrow \exists Y \alpha(\mathbf{X}, Y))$ if, whenever there exists a homomorphism h such that $h(\varphi(\mathbf{X})) \subseteq I$, then there exists an extension h' of h such that $h(\alpha(\mathbf{X}, Y)) \subseteq I$, while I satisfies $\forall \mathbf{X}\,(\varphi(\mathbf{X}) \rightarrow X_i = X_j)$ if the existence of h such that $h(\varphi(\mathbf{X})) \subseteq I$ implies $h(X_i) = h(X_j)$. Let $mods(D_\mathcal{S}, T_\mathcal{S} \cup E_\mathcal{S})$ be the set of models of $D_\mathcal{S}$ w.r.t. $T_\mathcal{S} \cup E_\mathcal{S}$. It is clear that $(\mathcal{W}_\mathcal{S} \wedge \mathcal{X}_\mathcal{S} \wedge \mathcal{E}_\mathcal{S})$ is satisfiable iff $mods(D_\mathcal{S}, T_\mathcal{S} \cup E_\mathcal{S}) \neq \varnothing$. A conjunctive query $\exists \mathbf{X}\,(\varphi(\mathbf{X}))$ is entailed by an instance I if there exists a homomorphism h such that $h(\varphi(\mathbf{X})) \subseteq I$. $Q_\mathcal{S}$ is entailed by I, written $I \models Q_\mathcal{S}$, if at least one of its disjuncts is entailed by I. It is easy to show that there exists an FO-model of $(\mathcal{W}_\mathcal{S} \wedge \mathcal{X}_\mathcal{S} \wedge \mathcal{E}_\mathcal{S})$ that satisfies $\mathcal{F}_\mathcal{S}$ iff the following *does not* hold: for every $I \in mods(D_\mathcal{S}, T_\mathcal{S} \cup E_\mathcal{S})$, $I \models Q_\mathcal{S}$.

In general, $mods(D_\mathcal{S}, T_\mathcal{S} \cup E_\mathcal{S})$ is infinite, and thus not explicitly computable. To overcome this difficulty, we employ a classical algorithmic tool from the database literature called the *chase procedure*, which repairs $D_\mathcal{S}$ w.r.t. $T_\mathcal{S} \cup E_\mathcal{S}$ so that the result, denoted $chase(D_\mathcal{S}, T_\mathcal{S} \cup E_\mathcal{S})$, satisfies $T_\mathcal{S} \cup E_\mathcal{S}$. It works on $D_\mathcal{S}$ through the $\exists$-*chase step*, which aims at satisfying TGDs by adding atoms, and the $=$-*chase step*, which aims at satisfying EGDs by unifying terms; if constants of $\mathbf{C}$ must be unified, then we have a *hard violation* of an EGD and the chase *fails*; for details, see, e.g., [6]. It is implicit in [10] that $mods(D_\mathcal{S}, T_\mathcal{S} \cup E_\mathcal{S}) \neq \varnothing$ iff $chase(D_\mathcal{S}, T_\mathcal{S} \cup E_\mathcal{S})$ does not fail. Moreover, if $chase(D_\mathcal{S}, T_\mathcal{S} \cup E_\mathcal{S})$ does not fail, then $chase(D_\mathcal{S}, T_\mathcal{S} \cup E_\mathcal{S})$ is a *universal model* of $D_\mathcal{S}$ w.r.t. $T_\mathcal{S} \cup E_\mathcal{S}$, i.e., for each $I \in mods(D_\mathcal{S}, T_\mathcal{S} \cup E_\mathcal{S})$, there exists a homomorphism h such that $h(chase(D_\mathcal{S}, T_\mathcal{S} \cup E_\mathcal{S})) \subseteq I$. The next technical result can be established.

Lemma 3. *It holds that:*

1. $(\mathcal{W}_\mathcal{S} \wedge \mathcal{X}_\mathcal{S} \wedge \mathcal{E}_\mathcal{S})$ *is satisfiable iff* $chase(D_\mathcal{S}, T_\mathcal{S} \cup E_\mathcal{S})$ *does not fail; and*
2. *there exists an FO-model* $\mathcal{I}$ *of* $(\mathcal{W}_\mathcal{S} \wedge \mathcal{X}_\mathcal{S} \wedge \mathcal{E}_\mathcal{S})$ *such that* $\mathcal{I} \models \mathcal{F}_\mathcal{S}$ *iff* $chase(D_\mathcal{S}, T_\mathcal{S} \cup E_\mathcal{S}) \not\models Q_\mathcal{S}$.

Thus, the above lemma, combined with Lemmas 1 and 2, suggests the following:

Corollary 1. $\mathcal{S}$ *is consistent iff the following hold:*

1. $chase(D_\mathcal{S}, T_\mathcal{S} \cup E_\mathcal{S})$ *does not fail; and*
2. $chase(D_\mathcal{S}, T_\mathcal{S} \cup E_\mathcal{S}) \not\models Q_\mathcal{S}$.

3.2 Chase Failure

It can be shown that $E_\mathcal{S}$ can be safely ignored and proceed only with $T_\mathcal{S}$. In particular, we can show that the initial segment of $chase(D_\mathcal{S}, T_\mathcal{S})$ obtained starting

from $D_{\mathcal{S}}$ and applying the $\exists$-chase step i times, satisfies $E_{\mathcal{S}}$, for each $i \geqslant 0$; this can be established by induction on i. Therefore, during the construction of $chase(D_{\mathcal{S}}, T_{\mathcal{S}} \cup E_{\mathcal{S}})$ the $=$-chase step is not applied, and the next lemma follows:

Lemma 4. *$chase(D_{\mathcal{S}}, T_{\mathcal{S}} \cup E_{\mathcal{S}}) = chase(D_{\mathcal{S}}, T_{\mathcal{S}})$.*

As an immediate consequence we get that:

Proposition 1. *$chase(D_{\mathcal{S}}, T_{\mathcal{S}} \cup E_{\mathcal{S}})$ does not fail.*

3.3 Query Entailment

Although the problem of deciding whether the chase fails is trivial, the problem of deciding whether $chase(D_{\mathcal{S}}, T_{\mathcal{S}} \cup E_{\mathcal{S}}) \not\models Q_{\mathcal{S}}$ is rather challenging. By Lemma 4, we can focus on the problem of deciding whether $chase(D_{\mathcal{S}}, T_{\mathcal{S}}) \not\models Q_{\mathcal{S}}$. It turned out that it is more convenient to study the complement of the problem under consideration. We present a novel nondeterministic algorithm which decides whether $chase(D_{\mathcal{S}}, T_{\mathcal{S}}) \models Q_{\mathcal{S}}$. Before we proceed further, let us give some auxiliary terminology. We denote by $I\langle\sigma, h\rangle I'$ a single $\exists$-chase step, which means that during the chase we apply the TGD σ of the form $\forall \mathbf{X}\,(\varphi(\mathbf{X}) \rightarrow \exists Y\,\alpha(\mathbf{X}, Y))$ due to the existence of a homomorphism h such that $h(\varphi(\mathbf{X})) \subseteq I$, and $I' = I \cup h'(\alpha(\mathbf{X}, Y))$, where h' is an extension of h, and $h'(Y)$ is a "fresh" null of $\mathbf{N}$. Interestingly, the TGDs of $T_{\mathcal{S}}$ enjoy a crucial syntactic property: for each $\sigma \in T_{\mathcal{S}}$, the left-hand side of σ, denoted $body(\sigma)$, has a *guard* atom, denoted $guard(\sigma)$, that contains all the universally quantified variables of σ; such TGDs are known as *guarded* TGDs [6]. The guarded chase forest is a tree-like representation of the instance constructed by the chase; the formal definition follows:

Definition 1. *The* guarded chase forest *of $D_{\mathcal{S}}$ and $T_{\mathcal{S}}$, denoted $\mathsf{gcf}(D_{\mathcal{S}}, T_{\mathcal{S}})$, is a labeled directed forest (N, E, λ), where $\lambda : N \rightarrow chase(D_{\mathcal{S}}, T_{\mathcal{S}})$, defined as follows:* (i) *for each $\alpha \in D_{\mathcal{S}}$, there exists exactly one $v \in N$ with $\lambda_1(v) = \alpha$;* (ii) *for each step $I\langle\sigma, h\rangle I'$ applied during the construction of $chase(D_{\mathcal{S}}, T_{\mathcal{S}})$: for every atom $\alpha \in \{h(guard(\sigma))\} \cup (I' \setminus I)$, there exists exactly one node $v \in N$ such that $\lambda(v) = \alpha$, and for every $\alpha \in I' \setminus I$, there exists an edge $(v, u) \in E$, where $\lambda(v) = h(guard(\sigma))$ and $\lambda(u) = \alpha$; and* (iii) *no other nodes and edges occur in N and E, respectively. Let $\mathsf{gcf}^k(D_{\mathcal{S}}, T_{\mathcal{S}})$ be the initial part of $\mathsf{gcf}(D_{\mathcal{S}}, T_{\mathcal{S}})$ up to depth $k \geqslant 0$.* □

Based on $\mathsf{gcf}(D_{\mathcal{S}}, T_{\mathcal{S}})$ we define the notion of the guarded chase of $D_{\mathcal{S}}$ and $T_{\mathcal{S}}$ up to a certain depth:

Definition 2. *The* guarded chase *of $D_{\mathcal{S}}$ and $T_{\mathcal{S}}$ of depth up to $k \geqslant 0$ is the instance $gchase^k(D_{\mathcal{S}}, T_{\mathcal{S}}) = \{\lambda(v)\}_{v \in N^k}$ assuming that $\mathsf{gcf}^k(D_{\mathcal{S}}, T_{\mathcal{S}}) = (N^k, E, \lambda)$.* □

Interestingly, for our purposes, we can focus on an initial part of the guarded chase; the following is implicit in [6]:

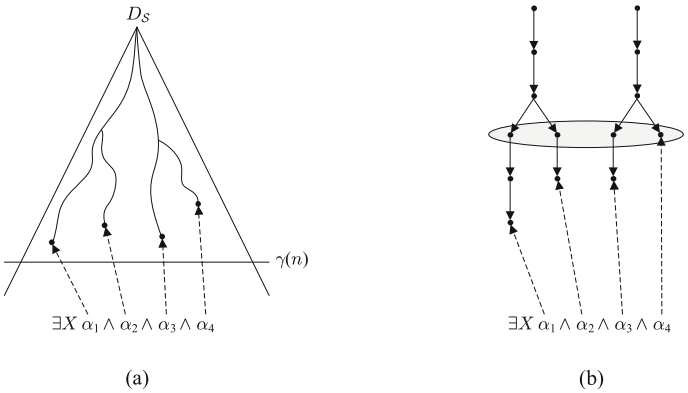


Fig. 3. Proof of a disjunct of $Q_{\mathcal{S}}$

Lemma 5. *There exists $\gamma(n) \in \mathcal{O}(2^n)$, where n is the number of predicates in $T_{\mathcal{S}}$, such that $chase(D_{\mathcal{S}}, T_{\mathcal{S}}) \models Q_{\mathcal{S}}$ iff $gchase^{\gamma(n)}(D_{\mathcal{S}}, T_{\mathcal{S}}) \models Q_{\mathcal{S}}$.*

Therefore, one can simply build $gchase^{\gamma(n)}(D_{\mathcal{S}}, T_{\mathcal{S}})$, and then check whether there exists a homomorphism that maps at least one disjunct of $Q_{\mathcal{S}}$ to it. This naive approach shows that our problem is in 2EXPTIME. However, this upper bound is not optimal. A more clever procedure, which needs only polynomial space, can be designed. Let us first give an informal description of this procedure.

An Informal Description. Assume that $Q_{\mathcal{S}}$ is entailed by $chase(D_{\mathcal{S}}, T_{\mathcal{S}})$. By Lemma 5, there exists a disjunct q of $Q_{\mathcal{S}}$ that can be mapped via a homomorphism h to $gchase^{\gamma(n)}(D_{\mathcal{S}}, T_{\mathcal{S}})$. Let P be the subforest of $\mathsf{gcf}^{\gamma(n)}(D_{\mathcal{S}}, T_{\mathcal{S}})$ that is obtained by keeping only the paths from the root nodes to the nodes which are labeled by the atoms of $h(q)$; in other words, P is the proof of q w.r.t. $D_{\mathcal{S}}$ and $T_{\mathcal{S}}$. Observe that, for each $0 \leqslant i \leqslant \gamma(n)$, the number of nodes occurring at the i-th level of P is at most $|q|$, i.e., the number of conjuncts in q. An abstract example is depicted in Figure 3 — the general shape of the subforest P is given in (a), while its actual structure is shown in (b). It is clear that at each level of P, at most $|q|$ atoms may appear; e.g., in the third level (see shaded nodes) there are exactly $|q|$ atoms. The key idea underlying our algorithm is to nondeterministically construct, in a *level-by-level* fashion, the atoms of each level of P until we reach $h(q)$. In other words, our intention is to generate, by applying some $\exists$-chase steps, the $(i+1)$-th level of P from the i-th level of P, and thus we do not need to store more than $2 \cdot |q|$ atoms at each step. A crucial notion, necessary for this construction, is the type of an atom which is defined as follows:

Definition 3. *The* type *of an atom $\alpha \in chase(D_{\mathcal{S}}, T_{\mathcal{S}})$, denoted $type(\alpha, D_{\mathcal{S}}, T_{\mathcal{S}})$ (or simply $type(\alpha)$), is the set of atoms of the form $C(t)$, where $C \in class(\mathcal{S})$[4], occurring in $chase(D_{\mathcal{S}}, T_{\mathcal{S}})$ such that t appears in α.* □

[4] By abuse of notation, we refer to the set of classes occurring in the UCD of the specification $\mathcal{S}$ by $class(\mathcal{S})$.

Let us explain the importance of the notion of type. Consider a node v occurring at the i-th level of P which is labeled by the atom α. Assume now that there exists a TGD $\sigma \in T_{\mathcal{S}}$ such that $guard(\sigma)$ is mapped via a homomorphism μ to α, and also μ maps the rest of the body of σ, denoted φ_σ, to $chase(D_{\mathcal{S}}, T_{\mathcal{S}})$. This implies that v has a child node u at the $(i+1)$-th level of P which is labeled by the atom obtained after applying σ. Since the TGD σ is guarded, all the variables occurring in $guard(\sigma)$ appear also in φ_σ, and thus μ necessarily maps φ_σ to $type(\alpha)$. From the above informal discussion, we conclude that the level-by-level construction proposed above is feasible, providing that we are also able to construct the type of the generated atoms. Notice that, at each step of the procedure, apart from the $2 \cdot |q|$ atoms, we also need to store their types. However, the size of the type of an atom is at most the cardinality of $class(\mathcal{S})$, and thus overall we need only polynomial space. In what follows, we discuss in depth how the type of an atom can be effectively computed, and we formalize the level-by-level construction sketched above.

Computing the Type of an Atom. In general, the problem of computing the type of an atom is not easier than the problem of query entailment itself. However, in our case, it is possible to construct the type of an atom $\alpha \in chase(D_{\mathcal{S}}, T_{\mathcal{S}})$ by exploiting α and the type of its parent in the guarded chase forest. Let us first formally define what we mean by saying the type of the parent of an atom in the guarded chase forest. To this end, we need the notion of the parent-type function defined as follows:

Definition 4. *The* parent-type function *pt from* $chase(D_{\mathcal{S}}, T_{\mathcal{S}})$ *to* $2^{chase(D_{\mathcal{S}}, T_{\mathcal{S}})}$ *is defined as follows:*

$$pt(\alpha) = \begin{cases} \varnothing, & \alpha \in D_{\mathcal{S}}, \\ type(h(guard(\sigma))), I\langle\sigma, h\rangle(I \cup \{\alpha\}). \end{cases}$$

Let also $pt^+(\alpha) = \{\alpha\} \cup pt(\alpha)$. □

We also need the notion of the distinguished term of an atom $\alpha \in chase(D_{\mathcal{S}}, T_{\mathcal{S}})$, which is crucial for the computation of $type(\alpha)$. In fact, to compute $type(\alpha)$, it suffices to add to the common part between the type of α and the type of its parent the atoms of $chase(D_{\mathcal{S}}, T_{\mathcal{S}})$ which contain only the distinguished term of α.

Definition 5. *The* distinguished term *of an atom* $\alpha \in chase(D_{\mathcal{S}}, T_{\mathcal{S}})$*, denoted* $d(\alpha)$*, is defined as follows: if* $\alpha = C(t)$*, where* $C \in class(\mathcal{S})$*, then* $d(\alpha) = t$*; otherwise,* $d(\alpha)$ *is the null of* $\mathbf{N}$ *invented in* α. □

Finally, we define the so-called projection set of $T_{\mathcal{S}}$, which will allows us to complete the common part between the type of α and the type of its parent, and thus computing $type(\alpha)$, by starting from $pt^+(\alpha)$. Let $A[i]$, where $i \in \{1, 2\}$, be an auxiliary predicate which is used to store the projection to the i-th argument of the predicate A.

Input: An atom α, an instance I, a term t, a specification $\mathcal{S}$.
Output: A finite instance.

1. $J := \varnothing$.
2. $K := chase((\{\alpha\} \cup I)_{\downarrow}, T^{\pi}_{\mathcal{S}})$.
3. For each $C \in class(\mathcal{S})$: if $C(t)_{\downarrow} \in K$, then $J := J \cup \{C(t)\}$.
4. If $\alpha = C(t)$, where $C \in class(\mathcal{S})$, then return J; otherwise, return $(I_{|t'} \cup J)$, where $t' \neq t$ and t' occurs in α.

Fig. 4. The Procedure Type

Definition 6. *Consider a TGD $\sigma \in T_{\mathcal{S}}$ of the form $\varphi \rightarrow \alpha$, and an atom β occurring in σ. If $\beta = A(X, Y)$, where A is an association class, and the variable X (resp., Y) occurs in both φ and α, then $\tau_{\pi}(\beta, \sigma) = A[1](X)$ (resp., $A[2](Y)$); otherwise, $\tau_{\pi}(\beta, \sigma) = \beta$. The* projection set *of $T_{\mathcal{S}}$, denoted $T^{\pi}_{\mathcal{S}}$, is obtained as follows: for each $\sigma \in T_{\mathcal{S}}$ of the form $\varphi \rightarrow \alpha$ where the predicate of α is either a class or an association, and for each atom β in σ, replace β by $\tau_{\pi}(\beta, \sigma)$.* □

An example of a projection set follows:

Example 1. Let $\mathcal{S} = (\mathcal{C}, \varnothing, \varnothing)$, where $\mathcal{C}$ is the diagram in Figure 2(c) with $n_{\ell}..n_u = m_{\ell}..m_u = 1..\infty$, expressing that there is an association A between the classes C_1 and C_2, and each instance of C_1 and C_2 participates at least once in A. $T^{\pi}_{\mathcal{S}}$ is as follows:

$$\forall X\,(C_1(X) \rightarrow \exists Y\,\tau_{\pi}(A(X,Y))) = \forall X\,(C_1(X) \rightarrow A[1](X)),$$
$$\forall X\,(C_2(X) \rightarrow \exists Y\,\tau_{\pi}(A(Y,X))) = \forall X\,(C_2(X) \rightarrow A[2](X)),$$
$$\forall X \forall Y\,(\tau_{\pi}(A(X,Y)) \rightarrow C_1(X)) = \forall X\,(A[1](X) \rightarrow C_1(X)),$$
$$\forall X \forall Y\,(\tau_{\pi}(A(X,Y)) \rightarrow C_2(Y)) = \forall Y\,(A[2](Y) \rightarrow C_2(Y)).$$

For brevity, the second parameter of τ_{π} is omitted. □

We are now ready to give our key technical lemma. Henceforth, given an atom α, we denote by $\alpha_{\downarrow}$ the atom obtained by *freezing* α, i.e., replacing each null $z \in \mathbf{N}$ occurring in α with a new constant $c_z \in \mathbf{C}$; this notation naturally extends to sets of atoms.

Lemma 6. *For each atom $\alpha \in chase(D_{\mathcal{S}}, T_{\mathcal{S}})$, and for each class $C \in class(\mathcal{S})$, $C(d(\alpha)) \in chase(D_{\mathcal{S}}, T_{\mathcal{S}})$ iff $C(d(\alpha))_{\downarrow} \in chase(pt^{+}(\alpha)_{\downarrow}, T^{\pi}_{\mathcal{S}})$.*

The crucial observation in the proof of the above lemma is that in a chase derivation from an atom $\alpha \in chase(D_{\mathcal{S}}, T_{\mathcal{S}})$ to an atom $C(d(\alpha)) \in chase(D_{\mathcal{S}}, T_{\mathcal{S}})$, it is not possible to lose and reintroduce the term $d(\alpha)$; this is because of the fact that the TGDs of $T_{\mathcal{S}}$ are guarded. Therefore, the TGDs that are involved in such a chase derivation are neither of the form $\forall X\,(C'(X) \rightarrow \exists Y\,a(X,Y))$ nor of the form $\forall X \forall Y\,(C'(X) \wedge a(X,Y) \rightarrow T(Y))$; otherwise, we

immediately get a contradiction. Moreover, these TGDs are contributing in such a chase derivation only by projecting out the term $d(\alpha)$; this justifies the definition of $T^{\pi}_{\mathcal{S}}$. Based on Lemma 6, we design the procedure Type, depicted in Figure 4, which computes the type of an atom α by adding to the part of $pt(\alpha)$ that contains only the non-distinguished term t' of α, denoted as $pt(\alpha)_{|t'}$, the set of atoms $J = \{C(d(\alpha)) \mid C \in class(\mathcal{S}) \text{ and } C(d(\alpha)) \in chase(D_{\mathcal{S}}, T_{\mathcal{S}})\}$; clearly, $(pt_{|t'} \cup J) = type(\alpha)$. Since each TGD of $T^{\pi}_{\mathcal{S}}$ does not contain an existentially quantified variable, and also its size is fixed, $chase(pt^{+}(\alpha)_{\downarrow}, T^{\pi}_{\mathcal{S}})$ is finite and can be constructed in polynomial time in the size of $pt^{+}(\alpha)_{\downarrow}$. The instance $pt^{+}(\alpha)_{\downarrow}$ is of polynomial size, and thus $chase(pt^{+}(\alpha)_{\downarrow}, T^{\pi}_{\mathcal{S}})$ can be constructed in polynomial time; hence, the second step of Type terminates after polynomially steps.

Proposition 2. *For each atom $\alpha \in chase(D_{\mathcal{S}}, T_{\mathcal{S}})$,*

1. $\mathsf{Type}(\alpha, pt(\alpha), d(\alpha), \mathcal{S}) = type(\alpha)$*; and*
2. $\mathsf{Type}(\alpha, pt(\alpha), d(\alpha), \mathcal{S})$ *terminates after polynomially many steps.*

The Level-by-level Construction. We have now all the necessary ingredients in order to proceed with our novel algorithm for deciding whether $chase(D_{\mathcal{S}}, T_{\mathcal{S}}) \models Q_{\mathcal{S}}$. The main idea, as sketchily described above, is to nondeterministically construct, in a level-by-level fashion, a segment of $gchase^{\gamma(n)}(D_{\mathcal{S}}, T_{\mathcal{S}})$, which contains at most as many atoms as the biggest disjunct q of $Q_{\mathcal{S}}$, and then check whether there exists a homomorphism that maps q to it. During this procedure, we can compute the children of a node v by exploiting the instance $type(\alpha)$, where α is the label of v, and then forget v and its type. Moreover, the type of an atom α can be constructed by exploiting $pt^{+}(\alpha)$ and the procedure Type. The formal algorithm, called Ent (which stands for entailment), is depicted in Figure 5. Note that D and D' are vectors that hold integer numbers and are used to store the depth of the generated atoms, while P and P' are vectors that hold sets of atoms and are used to store the types of the generated atoms. Moreover, $\gamma(n)$ is the bound on the depth of $\mathsf{gcf}(D_{\mathcal{S}}, T_{\mathcal{S}})$ provided by Lemma 5. A simple example of the execution of Ent follows:

Example 2. Let $S = (\mathcal{C}, \{T_1, T_3\}, \varnothing)$, where $\mathcal{C}$ is the RevEng UCD in Figure 6. The forest $\mathsf{gcf}(D_{\mathcal{S}}, T_{\mathcal{S}})$ is depicted in Figure 6 (for brevity, the atoms $T_1(c_4), T_2(c_5)$ and $T_2(c_6)$ are not shown). A possible execution of $\mathsf{Ent}(\mathcal{S})$, which explores in a level-by-level fashion the shaded nodes of $\mathsf{gcf}(D_{\mathcal{S}}, T_{\mathcal{S}})$, is as follows:

- We choose $(S_1, \prec_1)$ to be $(\{C_3(c_3)\}, \varnothing)$, and the type of $C_3(c_3)$ is stored in P_1;
- We construct $(S_2, \prec_2) = (\{a(c_3, z_3)\}, \varnothing)$ from $C_3(c_3)$ by applying $\forall X\,(C_3(X) \rightarrow \exists Y\, a(X, Y))$, and the type of $a(c_3, z_3)$ is stored in P'_1;
- We assign $(S_2, \prec_2)$ to $(S_1, \prec_1)$ and P'_1 to P_1 — this means that we forget the atom $C_3(c_3)$ and its type;
- We construct $(S_2, \prec_2) = (\{T_1(z_3), T_3(z_3)\}, T_1(z_3) \prec_2 T_3(z_3))$ from the atom $a(c_3, z_3)$ by applying the TGDs $\forall X \forall Y\,(C_1(X) \wedge a(X, Y) \rightarrow T_1(Y))$ and $\forall X \forall Y\,(C_3(X) \wedge a(X, Y) \rightarrow T_3(Y))$ (notice that the crucial atoms $C_1(c_3)$

Input: A RevEng specification $\mathcal{S}$.
Output: *yes* if $chase(D_{\mathcal{S}}, T_{\mathcal{S}}) \models Q_{\mathcal{S}}$; otherwise, *no*.

1. Guess a disjunct q of $Q_{\mathcal{S}}$.
2. $Image := \varnothing$ and $L := \{z_1, \ldots, z_k\} \subset \mathbf{N}$, where $k = 2 \cdot |q|$.
3. Guess a totally ordered set $(S_1, \prec_1)$, where $S_1 \subseteq D_{\mathcal{S}}$ and $|S_1| \in \{1, \ldots, |q|\}$; assume that $\alpha_1 \prec_1 \ldots \prec_1 \alpha_m$.
4. For each $i \in [|S_1|]$: $D[i] := 0$ and $P[i] := \mathsf{Type}(\alpha_i, \varnothing, c, \mathcal{S})$, where c is the constant in α_i.
5. Guess a set of atoms $I \subseteq S_1$; $Image := Image \cup I$.
6. If $|Image| = |q|$, then goto 15.
7. Guess to proceed with the next step or goto 15.
8. Construct a totally ordered set $(S_2, \prec_2)$ as follows:
   a. $(S_2, \prec_2) := (\varnothing, \varnothing)$ and $ctr := 1$.
   b. Guess $\sigma = (\varphi \rightarrow \exists Y \alpha) \in T_{\mathcal{S}}$ for which there exists $i \in [|S_1|]$ and a homomorphism h such that
      – $h(guard(\sigma)) = \alpha_i$,
      – $D[i] < \gamma(n)$, and
      – $h(body(\sigma) \setminus \{guard(\sigma)\}) \subseteq P_i$;
      if there is no such σ, then $\sigma := \epsilon$.
   c. If $\sigma \neq \epsilon$, then do the following:
      – $\beta_{ctr} := h'(\alpha)$, where $h' := h \cup \{Y \rightarrow t \mid t \in L$ and t does not occur in $S_1 \cup S_2\}$.
      – $S_2 := S_2 \cup \{\beta_{ctr}\}$.
      – If $ctr > 1$, then $\beta_{ctr-1} \prec_2 \beta_{ctr}$.
      – $D'[ctr] := D[i] + 1$.
      – $P'[ctr] := \mathsf{Type}(\beta_{ctr}, P[i], h'(Y), \mathcal{S})$.
      – $ctr := ctr + 1$.
   d. If $|S_2| = |q|$ or $\sigma = \epsilon$, then goto 9.
   e. Guess to proceed to the next step or goto 8b.
9. Guess a set $I \subseteq S_2$; $Image := Image \cup I$.
10. If $|Image| = |q|$, then goto 15.
11. Guess to proceed to the next step or goto 15.
12. $(S_1, \prec_1) := (S_2, \prec_2)$; assume that $\alpha_1 \prec_1 \ldots \prec_1 \alpha_m$.
13. $D := D'$ and $P := P'$.
14. Goto 8.
15. If there exists h such that $h(q) \subseteq Image$, then return *yes*; otherwise, return *no*.

Fig. 5. The Nondeterministic Algorithm Ent

and $C_3(c_3)$ occur in $type(a(c_3, z_3))$), and the type of $T_1(z_3)$ and $T_3(z_3)$ are stored in P_1' and P_2', respectively; and
– Finally, we choose to assign $\{T_1(z_3), T_3(z_3)\}$ to *Image*, and then check whether there exists a homomorphism that maps $Q_{\mathcal{S}}$ to *Image*.

Clearly, since such a homomorphism exists, the algorithm returns *yes*, which in turn implies that $chase(D_{\mathcal{S}}, T_{\mathcal{S}}) \models Q_{\mathcal{S}}$. □

By construction, $\mathsf{Ent}(\mathcal{S}) = yes$ iff $gchase^{\gamma(n)}(D_{\mathcal{S}}, T_{\mathcal{S}}) \models Q_{\mathcal{S}}$, where $\gamma(n)$ is the bound provided by Lemma 5, which in turn is equivalent to $chase(D_{\mathcal{S}}, T_{\mathcal{S}}) \models Q_{\mathcal{S}}$. Let us now analyze the space complexity of our algorithm. During the execution of $\mathsf{Ent}(\mathcal{S})$ we need to maintain the following:

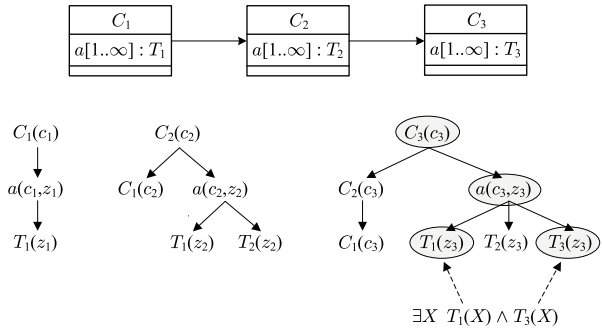


Fig. 6. Execution of the Algorithm Ent

1. The totally ordered sets $(S_1, \prec_1)$ and $(S_2, \prec_2)$;
2. The vectors D, D', P and P'; and
3. The set of atoms *Image*.

It is possible to show that the above structures need $\mathcal{O}(m^3 \cdot \log m)$ space, where $m = |class(\mathcal{S})|$. By Proposition 2, the computation of the types at steps 4 and 8 is feasible in polynomial time, and thus in polynomial space. Finally, since the problem of deciding whether there exists a homomorphism from a query to an instance is feasible in NP (and thus a fortiori in PSPACE), we get that step 15 is feasible in polynomial space. The next result follows:

Proposition 3. *It holds that,*

1. $\mathsf{Ent}(\mathcal{S}) =$ *yes iff* $chase(D_\mathcal{S}, T_\mathcal{S}) \models Q_\mathcal{S}$*; and*
2. *Each step of the computation of* $\mathsf{Ent}(\mathcal{S})$ *uses polynomial space.*

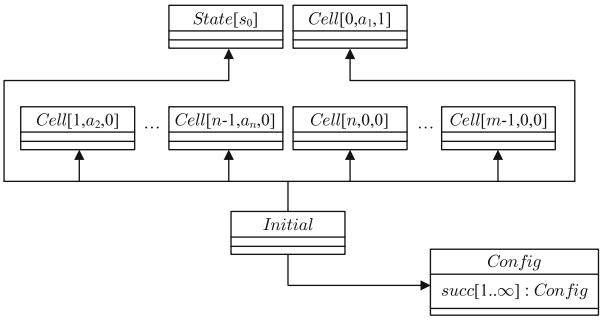
3.4 Pinpointing the Complexity

By using the results established in the previous section, we can now pinpoint the computational complexity of the problem of deciding whether $\mathcal{S}$ is consistent.

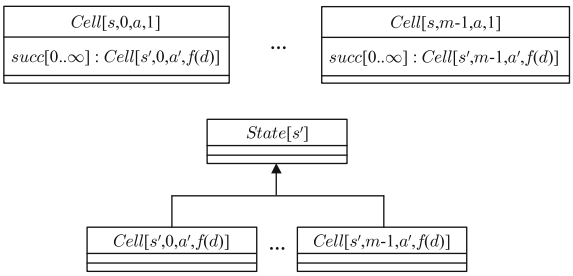
Upper Bound. By Corollary 1 and Proposition 1, we conclude that $\mathcal{S}$ is consistent iff $chase(D_\mathcal{S}, T_\mathcal{S}) \not\models Q_\mathcal{S}$. Since Ent describes a nondeterministic algorithm, Proposition 3 implies that the problem of deciding whether $chase(D_\mathcal{S}, T_\mathcal{S}) \models Q_\mathcal{S}$, that is, the complement of the problem under consideration, is in NPSPACE, and thus in PSPACE since NPSPACE = PSPACE. But PSPACE = coPSPACE, and therefore:

Theorem 1. *The problem of deciding whether* $\mathcal{S}$ *is consistent is in* PSPACE.

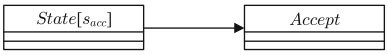
Lower Bound. We show that the upper bound established above is tight. This is done by simulating a polynomial space Turing machine (TM) by means of a RevEng specification. Consider a TM $M = (S, \Lambda, \delta, s_0, F)$, where S is the set of states, Λ is the tape alphabet, $\delta : S \setminus F \times \Lambda \to S \times \Lambda \times \{-1, 0, 1\}$ is the transition



(a) The diagram $\mathcal{C}_{init/config}$



(b) The diagram $\mathcal{C}_{(s,a)}$



(c) The diagram $\mathcal{C}_{acc}$

Fig. 7. Simulating a polynomial space TM

function, s_0 is the initial state, and $F \subseteq S$ is the set of final or accepting states. We assume w.l.o.g. that M has exactly one accepting state, denoted as s_{acc}. We also assume that $\Lambda = \{0, 1\}$, and that each input string has an 1 as the rightmost bit. Consider the computation of M on an input string $I = a_1 a_2 \dots a_n$, and suppose that it halts using $m = n^k$ cells, where $k > 0$. We shall construct a RevEng specification $\mathcal{S} = (\mathcal{C}, \mathsf{DISJ}, \mathsf{MSC})$ such that M accepts I iff $\mathcal{S}$ is not consistent; this shows that the complement of our problem is PSPACE-hard, and thus also our problem is PSPACE-hard.

The initial configuration of M is reflected by the class diagram $\mathcal{C}_{init/config}$ in Figure 7(a). Roughly, $Initial(c)$ states that c is the initial configuration, $State[s](c)$ asserts that the state of the configuration c is s, and $Cell[i, x, y](c)$ says that in the configuration c the i-th cell contains x, and the cursor is on the i-th cell iff $y = 1$.

The auxiliary classes $Cell[s, i, x, y]$, where $(s, i, x, y) \in S \times \{0, \dots, m-1\} \times \{0, 1\} \times \{0, 1\}$, are needed in order to describe the configuration transition via a RevEng class diagram. Roughly,

$$Cell[i, x, y](c) \wedge State[s](c) \;\rightarrow\; Cell[s, i, x, y](c),$$

which is formally defined using the set MSC_{state} of most specific class assertions consisting of: for each $(s, i, x, y) \in S \times \{0, \ldots, m-1\} \times \{0, 1\} \times \{0, 1\}$,

$$(\{Cell[i, x, y], State[s]\}, Cell[s, i, x, y]).$$

The fact that each configuration has a valid configuration as a successor is captured by the diagram $\mathcal{C}_{init/config}$, shown in Figure 7(a); $Config(c)$ expresses that c is a valid configuration, while $succ(c, c')$ states that c' is derived from c.

We now show how the configuration transition can be simulated. Consider an arbitrary pair $(s, a) \in S \setminus F \times \{0, 1\}$, and assume that $\delta((s, a)) = (s', a', d')$. The state transition, as well as the updating of the tape, is reflected by the diagram $\mathcal{C}_{(s,a)}$, shown in Figure 7(b); notice that $f(0) = 1$ and $f(-1) = f(1) = 0$. Eventually, the configuration transition is achieved by the diagram $\mathcal{C}_{trans}$, which is obtained by merging the diagrams $\{\mathcal{C}_{(s,a)}\}_{(s,a)\in S\setminus F\times\{0,1\}}$. It should not be forgotten that those cells which are not changed during the transition keep their old values. This can be ensured by a RevEng UCD, and a set of most specific class constraints. Finally, with the diagram $\mathcal{C}_{acc}$, shown in Figure 7(c), we say that M accepts if it reaches the accepting state.

We define $\mathcal{S}$ to be the specification $(\mathcal{C}, \mathsf{DISJ}, \mathsf{MSC})$, where $\mathcal{C}$ is the UCD obtained by merging the diagrams introduced above, DISJ consists of the single assertion $\{Initial, Accept\}$, and MSC consists of the most specific class assertions introduced above. It is easy to verify that $\mathcal{S}$ is a RevEng specification, and that can be constructed in polynomial time. By providing an inductive argument, we can show that M accepts I iff $chase(D_{\mathcal{S}}, T_{\mathcal{S}}) \models Q_{\mathcal{S}}$ iff $\mathcal{S}$ is inconsistent, and the next result follows:

Theorem 2. *The problem of deciding whether $\mathcal{S}$ is consistent is* PSPACE*-hard*[5].

The following complexity characterization follows from Theorems 1 and 2:

Corollary 2. *The problem of deciding whether $\mathcal{S}$ is consistent is* PSPACE*-complete.*

4 Conclusions

In this work, we focus on the fragment of UML class diagrams that can be recovered from an implemented system. We study the problem of consistency of such diagrams, and we show that is PSPACE-complete. Interestingly, the upper bound is obtained by exploiting algorithmic techniques developed for conjunctive

[5] An alternative way to establish this result is by adapting the construction in the proof of an analogous result for the description logic Horn-$\mathcal{FL}^-$ [13]. However, the resulting diagram is counterintuitive, and does not give any insights about the inherent difficulty of RevEng UCDs. For this reason, and also for self-containedness, we provide a new proof from first principles.

query answering under guarded Datalog$^{\pm}$, that is, a key member of the Datalog$^{\pm}$ family of KR languages. Although the proposed consistency algorithm is theoretically interesting, and allows us to establish a worst-case optimal upper bound for the problem under investigation, it is not very well-suited for a practical implementation. It is unlikely that it will lead to procedures that guarantee the required level of scalability, especially in the presence of very large diagrams. The designing of a more practical consistency algorithm, which will exploit existing database technology, will be the subject of future research.

Acknowledgements. This research has received funding from the EPSRC Programme Grant EP/M025268/ "VADA".

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Baader, F., Brandt, S., Lutz, C.: Pushing the $\mathcal{EL}$ envelope. In: Proc. of IJCAI, pp. 364–369 (2005)
3. Balaban, M., Maraee, A.: Finite satisfiability of UML class diagrams with constrained class hierarchy. ACM Trans. Softw. Eng. Methodol. **22**(3) (2013)
4. Berardi, D., Calvanese, D., De Giacomo, G.: Reasoning on UML class diagrams. Artif. Intell. **168**(1–2), 70–118 (2005)
5. Calì, A., Gottlob, G., Kifer, M.: Taming the infinite chase: Query answering under expressive relational constraints. J. Artif. Intell. Res. **48**, 115–174 (2013)
6. Calì, A., Gottlob, G., Lukasiewicz, T.: A general Datalog-based framework for tractable query answering over ontologies. J. Web Sem. **14**, 57–83 (2012)
7. Calì, A., Gottlob, G., Lukasiewicz, T., Marnette, B., Pieris, A.: Datalog+/-: a family of logical knowledge representation and query languages for new applications. In: Proc. of LICS, pp. 228–242 (2010)
8. Calì, A., Gottlob, G., Orsi, G., Pieris, A.: Querying UML class diagrams. In: Birkedal, L. (ed.) FOSSACS 2012. LNCS, vol. 7213, pp. 1–25. Springer, Heidelberg (2012)
9. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: Tractable reasoning and efficient query answering in description logics: The DL-Lite family. J. Autom. Reasoning **39**(3), 385–429 (2007)
10. Fagin, R., Kolaitis, P.G., Miller, R.J., Popa, L.: Data exchange: Semantics and query answering. Theor. Comput. Sci. **336**(1), 89–124 (2005)
11. Johnson, D.S., Klug, A.C.: Testing containment of conjunctive queries under functional and inclusion dependencies. J. Comput. Syst. Sci. **28**(1), 167–189 (1984)
12. Kaneiwa, K., Satoh, K.: On the complexities of consistency checking for restricted UML class diagrams. Theor. Comput. Sci. **411**(2), 301–323 (2010)
13. Krötzsch, M., Rudolph, S., Hitzler, P.: Complexities of horn description logics. ACM Trans. Comput. Log. **14**(1), 2 (2013)
14. Müller, H.A., Jahnke, J.H., Smith, D.B., Storey, M., Tilley, S.R., Wong, K.: Reverse engineering: a roadmap. In: Proc. of ICSE, pp. 47–60 (2000)