University of Zurich[UZH]

Benchmarking Incremental Reasoner Systems

Thesis September 12, 2018

Jérôme Oesch
of St. Gallen SG, Switzerland

Student-ID: 11-708-435
jerome.oesch@bluewin.ch

Advisor: **Daniele Dell'Aglio**

Prof. Abraham Bernstein, PhD
Institut für Informatik
Universität Zürich
http://www.ifi.uzh.ch/ddis

Acknowledgements

A very special gratitude goes to my supervisor, Daniele Dell'Aglio, University of Zurich, for the many hours of discourse on the topic of this thesis and for his guiding and helping hand.

I would like to thank Dr. Jeff Z. Pan, Aberdeen University, being an expert on the topic and guiding us onto the right track, answering all of our questions. Many thanks to Romana Pernischová, University of Zurich, for supporting me in Ontology Evolution Mapping, to whom I wish just the best for her upcoming PhD. I would also like to express my gratitude to all people at the chair of Dr. Abraham Bernstein, DDIS, for making my journey a very pleasant one.

Finally, to all my friends that had to bear with some bad temper of mine, my girlfriend and my family, my mother and father for supporting me during my entire life.

Zusammenfassung

Die Durchführung von Benchmarks über Reasoner Systeme ist ein bereits weit verbreiteter Ansatz für den Vergleich von verschiedenen Ontologie-Reasonern. Mit dem Aufkommen von inkrementellen Reasonern wurde jedoch noch kein Benchmark vorgeschlagen, welcher solche konkurrierende inkrementelle Implementationen testen kann. In dieser Arbeit stellen wir nicht nur ein Benchmarking-Framework vor, welches diese Lücke füllen könnte, sondern wir präsentieren auch einen neuen Benchmarking-Ansatz, welcher die automatische Generierung von Abfragen wie auch von Veränderungen an der Ontologie ermöglicht. Hierfür wird als Input nur eine bestehende Ontologie benötigt. Das implementierte Framework, ReasonBench++, benutzt zur Generierung von Abfragen das Konzept von "Competency Question-driven Ontology Authoring", sowie "Ontology Evolution Mapping" zur Generierung von Veränderungen. Durch die Anwendung dieser zwei Konzepte können wir aufzeigen, dass ReasonBench++ lebensnahe Benchmarks erstellen kann, welche einen intensiven Benutzungsfall durch Autoren wie auch Nutzerabfragen wiederspiegelt.

Abstract

Benchmarking reasoner systems is an already wide-spread approach of comparing different ontology reasoners among each other. With the emergence of incremental reasoners, no benchmarks have been proposed so far that are able to test competing incremental implementations. In this thesis, we not only propose a benchmarking framework that could fill this gap, but we present a new approach of a benchmark that is capable of generating both queries and subsequent ontology edits automatically, just requiring an existing ontology as input. Our implemented framework, ReasonBench++, uses the concepts of Competency Question-driven Ontology Authoring to generate queries, as well as Ontology Evolution Mapping to generate edits. With the application of these two concepts, we are able to show that ReasonBench++ is generating close to real-life benchmarks that reflect an ontology intensively used by ontology authors and simultaneous queries of users.

Table of Contents

1 Introduction **1**

2 Ontologies, Competency Question driven Authoring and Evolution Mapping **5**
2.1 Ontologies and Description Logic 5
2.2 Competency Questions and Presuppositions 6
2.3 Authoring Tests 8
2.4 Ontology Evolution Mapping 12
2.5 OWL API 13

3 ReasonBench++ **15**
3.1 Problem Analysis 15
3.1.1 Generating Competency Questions stochastically 18
3.1.2 Mapping Authoring Tests to Competency Question Archetypes . . 20
3.1.3 Generating Ontology Edits stochastically 22
3.2 Requirements 24
3.2.1 OWL API for Java 25
3.2.2 OWL Reasoner 25
3.2.3 Other References 26
3.3 Implementation 27
3.3.1 Inputs 27
3.3.2 Output 31
3.3.3 Benchmark Package 31
3.3.4 Competency Question Package 33
3.3.5 Authoring Test Package 35
3.3.6 Edit Package 37
3.4 Benchmarking Process 39

4 Results **41**
4.1 Methodology and Infrastructure 41
4.1.1 Hardware 41
4.1.2 Configurations of RB++ 42
4.1.3 Ontologies 43
4.1.4 Limitations and Unexpected Behaviour of Reasoners 43

4.2 Benchmark Results 45
4.2.1 Verification of Runtime Results 46
4.2.2 Verification of Reasoner Consistency 50
4.2.3 Results of the Gene Ontology 52

5 Limitations **55**

6 Future Work **57**

7 Conclusions **59**

A Appendix **65**
A.1 Terminal commands for RB++ 65
A.2 Results 66
A.2.1 Additional Figures of Reasoner Behaviour over Time 66
A.2.2 Failed Authoring Tests per Reasoner 69

1

Introduction

A *reasoner* is a software which is able to infer implicit information out of explicitly structured data. The structured data resides in so-called *ontologies*, a standardized repository of information authored by humans. The process of inference of information makes use of predicate logic and is based on set-theory. As an example, if an ontology author explicitly states that Peter is a student, and that a student is a person, the reasoner will infer that Peter is a person.

The development of reasoners has taken a turn in recent years. Previously, the strong assumption that ontologies are static was commonplace. As a reasoner can reach a conclusion in several different ways, a change in the ontology may or may not have consequences on the materialization of a reasoner. Finding approaches to apply specific kinds of changes directly to a materialization of a reasoner is hard and therefore, a reasoner was re-instantiated upon changes in the ontology, restarting the inference process necessary to provide fast response times to queries. This can constrain the application field of non-incremental reasoners to non-live support systems. Benchmarks exist to evaluate such reasoners on their performance, as for example in [Bock et al., 2008, Guo et al., 2005].

As a response to these restrictions, researchers proposed many different approaches, as for example in [Volz et al., 2005, Motik et al., 2015, Ren et al., 2016]. Many reasoners now support incremental updates of their materializations and often do not require a complete re-initialization, preventing downtime and lowering response times. Up to this date however, there does not exist any benchmark that is able to compare these reasoners based on their capability of incrementally updating their materializations. This thesis aims at this shortage - the goal is to design a software framework that allows automated evaluation of different reasoner systems by their capability of incremental reasoning. The possibility to compare different approaches of incremental reasoning can result in gains in overall performance of reasoners as well as the identification of bottlenecks. Additionally, some implementations might fit some specific use cases, while others do not.

As a benchmark should be applicable with many different parameters and inputs, at least partial automation of the process is required. Benchmarking incremental reasoners therefore entails some automation issues. To reflect a real-life application of a reasoner, it requires data in form of an ontology and queries it has to answer. Additionally, changes have to be applied to an ontology to which a reasoner has to adapt. In an ideal

world, queries and changes would both be given, not requiring any form of generation of both. As this however is rarely the case, the queries and changes should be generated automatically and should reflect a real-life scenario - they should be meaningful and a representation of a possible interaction of a human with the machine. Of course, the answers to the queries have to be correct, indicating that the reasoner is consistent in its current state. For this purpose, we are going to use the concepts of *Competency Question Ontology Authoring* (CQOA) by [Ren et al., 2014] to generate queries, as well as *Ontology Evolution Mapping* by [Hartung et al., 2013], allowing us to create artificial changes from the ontology.

QCOA exhibits the usage of *Competency Questions* as a functional requirement of an ontology. By stating these Competency Questions before authoring an ontology, an author defines minimum requirements of information an ontology must contain after supplying new content. Competency Questions can be categorized in *Archetypes* which capture relevant sentence structures and allow to infer *Authoring Tests*, which are used to query the ontology to verify its contents and whether a Competency Question is answerable meaningfully. In the benchmark, the Archetypes are therefore used as templates to create new Competency Questions automatically and the entailed Authoring Tests are used as tasks every reasoner has process.

To generate incremental changes automatically used to alter the ontology, Ontology Evolution Mapping is used. The approach allows to deduce *Change Operations* out of different versions of ontologies, where a set of Change Operations equals the differences between two versions. Distributions of Change Operations between example ontology versions as well as the logic behind any Change Operation can be used to generate edits automatically for any ontology.

Developing such an incremental reasoner benchmarking framework therefore entails the following research questions:

- **RQ1**: How can we automate the generation of tasks and data for a benchmarking framework?
  - **RQ1.1**: Given an ontology, how can we generate Competency Questions automatically in a way that resembles reality?
  - **RQ1.2**: Given a Competency Question, which Authoring Tests can be automatically inferred?
  - **RQ1.3**: Given an ontology, how can we generate a sequence of edits that resembles reality?
- **RQ2**: Does the benchmark allow a meaningful analysis of tested reasoners?

The thesis is structured as follows. Chapter 2 undertakes an in depth look into work related to this thesis, which includes the concepts of Competency Question Ontology Authoring (CQOA, see Section 2.2f) [Ren et al., 2014], as well as Ontology Evolution Mapping [Hartung et al., 2013] (see Section 2.4).

Chapter 3 describes the design and implementation of ReasonBench++ (RB++), the proposed software framework to benchmark incremental reasoner systems. The Chapter contains an implementation-wise problem analysis (see Section 3.1), a definition of requirements (see Section 3.2), a technical description of the framework and its components (see Section 3.3) and finally an outline of the benchmarking process (see Section 3.4).

Chapter 4 discusses the findings of the RB++ when being applied to a variety of ontologies and reasoners with different parameter sets.

Chapters 5 and 6 discuss limitations of this approach as well as future work that could be undertaken to improve the proposed benchmarking framework as well as the tested reasoners.

Finally, Chapter 7 sums up the contents of this thesis.

2

Ontologies, Competency Question driven Authoring and Evolution Mapping

This chapter covers the related work used for this thesis. In Section 2.1, the concept of an ontology and the language of Description Logic are outlined. Section 2.2 refers to Competency Questions and how presuppositions are embedded in such questions. Section 2.3 covers how Authoring Tests can be used to check for above mentioned presuppositions. Section 2.4 covers Ontology Evolution Mapping, an approach allowing to detect changes between different versions of an ontology. Finally, Section 2.5 discusses the OWL API, the Web Ontology Language and its integration into Java.

2.1 Ontologies and Description Logic

An *Ontology* is a knowledge representation of a specific universe of discourse, where often *Description Logic* (DL) [Baader et al., 2003, Colombetti, 2017] is used as a language to specify its contents. Ontologies are built upon set theory and use classes, properties and individuals to describe their contained sets. Every element or thing that is included inside an ontology is called an *entity*. An *individual* is an entity that is contained inside the sets of the ontology, for example a $Porsche$. A *class* denotes an entity that defines a set structure in the universe that can contain individuals, such as a Car. Classes can contain individuals, like $Porsche$, $Volkswagen$ and $Mercedes$, or can be empty (Equation 2.1).

$$Car = \{porsche, volkswagen, mercedes\} \tag{2.1}$$

Properties are used to describe relations between objects, for example $hasEngine$, or to describe relations between classes and data, as $hasWheels$. A statement among multiple entities by using boolean class constructors is again referred to as a class, as it again denotes a new set inside the universe. As an example, a $Vehicle$ that $hasEngine\ Engine$ represents a class (Equation 2.2).

$$Vehicle \sqcap \exists hasEngine.Engine \tag{2.2}$$

All above entities depict "things" inside our universe. These entities can be used to define state of affairs that can - for certain propositions - hold or fail. An instance of

such a state of affair is called a *statement*. Statements that should hold independent from the given situation and interpretation are called *axioms*, and are used as main building blocks inside an ontology.

Boolean class constructors are used to define complex classes. A *class intersection* ($\sqcap$) refers to elements that exist in two differing sets at the same time, e.g. $Car \sqcap Plane$, the set of all entities that are both a car and plane. A *class union* ($\sqcup$) depicts all objects that exist in either one or another class, for example $Vehicle \sqcup House$, all entities that are either a vehicle or a house. The *class complement* ($\neg$) is used to specify all objects that are not in a certain class, $\neg Car$ being all entities that are not a car. In addition there exists *subsumption* ($\sqsubseteq$, $\sqsupseteq$), which defines that all objects inside a subclass are also contained inside a superclass (Equation 2.4, 2.5), as well as *equivalence* ($\equiv$), meaning that two classes are equal (Equation 2.3).

$$Car \equiv Vehicle \sqcap = 4 hasWheels \tag{2.3}$$

$$Car \sqsubseteq Vehicle \tag{2.4}$$

$$SportsCar \sqsubseteq Car \tag{2.5}$$

$$SportsCar \sqsubseteq \forall hasEngine.BoxerEngine \tag{2.6}$$

Axiom (2.3) depicts that *vehicles* with four wheels are *cars*. Axioms (2.4) and (2.5) refer to subsumption; all *cars* are contained in the class of *vehicles* and all *sports cars* are contained in the class of *cars*. In Axiom (2.6) we defined that everything that has a *boxer engine* must be a *sports car* by sub-setting all *sports cars* to the set of things that have a *boxer engine*.

Such axioms represent explicit knowledge inside an ontology and allow inference of implicit knowledge. For example: by using axioms (2.4) and (2.5), one can infer that $SportsCar \sqsubseteq Vehicle$. By using axiom (2.3), we can further infer that $SportsCar \equiv Vehicle \sqcap = 4 hasWheels$, that every sportscar is a vehicle with exactly four wheels.

Inference as in the examples above is conducted by an automatic reasoner. A reasoner is queried whether a certain proposition, given the universe implemented into the ontology, does hold. If that proposition holds, the proposition is called *satisfiable*. A set that is unsatisfiable must be empty, i.e. it is impossible for that set to hold any individuals. Emptiness is modelled as the "bottom class $\bot$". Its counterpart is the "top class $\top$" that contains every set in the ontology.

2.2 Competency Questions and Presuppositions

A *Competency Question* (CQ, [Uschold and Gruninger, 1996]) can be understood as a functional requirement of an ontology. Defined as natural language sentences that express patterns for types of questions people want to be able to answer with the ontology [Ren et al., 2014], they are used to verify the consistency of an ontology by checking whether they can be answered with the explicit and implicit knowledge entailed by an ontology. In contrast to formal requirement specifications, CQs are most useful when used by ontology authors that do not have domain knowledge in DL, but are proficient in

their specific domain to be able to populate an ontology with content [Dennis et al., 2017]. The author would be encouraged to formulate some CQs before starting the authoring process, and the ontology could be counter-checked on whether these CQs are satisfiable or not. This approach, as elaborated by [Ren et al., 2014] is called *Competency Question-driven Ontology Authoring* (CQOA). Using and formalising CQs correctly is however a non-trivial task, as Ren et al. have pointed out by the concept of *presuppositions*. A presupposition is the implicit presumption on a state of the universe, where some knowledge is embedded into a sentence that is not interpretable or understandable by a computer system. Presuppositions originate in linguistic pragmatics and based on Ren et al. are one possible reason why reasoners fail to answer certain queries, as they are unable to encode the implicit information entailed in a CQ. As an example, asking:

Example 1. "What is the type of engine of that car?" implies that this car does have an engine and that engines can have different types - even though this is not explicitly stated.

Based on the idea of presuppositions, Ren et al. performed empirical research on types of CQs that are asked by ontology authors. To gather the most used CQ-types, where also different levels of expertise of the ontology authors were considered, the authors collected 92 CQs from the Software Ontology Project[1] and 76 CQs from the Manchester OWL Tutorials in 2013[2].

After removing invalid questions, for example redundant ones, incomplete sentences, non-real CQs or questions currently not representable by a DL-based ontology language, they grouped CQs into twelve so called *Archetypes*. Each Archetype reflects sentence structure, subjects, objects and relations imposed by the CQs. In Table 2.1, the Archetypes are depicted. In Table 2.2, additional sub-archetypes of Archetype 1 are listed. The main Archetypes appear with different probabilities, an absolute distribution is given in Table 2.3. Ren et al. mention that the list of Archetypes is non-complete, since there might be additional Archetypes that were not available in the set of CQs used for the study.

For the assignment of the different CQs to their fitting Archetypes in Tables 2.1 and 2.2, multiple patterns were identified using a feature-based modelling method [Palmer and Felsing, 2001]. Ren et al. sorted these patterns into primary and secondary patterns, where **Predicate Arity**, **Relation Type**, **Modifier** and **Domain-independent Element** are primary, while **Question Type**, **Element Visibility** and **Question Polarity** were categorized secondary patterns. The primary patterns are used to distinguish the main Archetypes 1 to 12 from each other, while the secondary patters distinguish the sub-archetypes of Archetype 1. The characteristics of these patterns are stated below:

1. **Question Type:** Refers to the kind of answer the question should yield.
   (a) *Selection Question:* From the ontology, all entities that satisfy the constraints given in the question should be returned, being a set of entities.

[1] http://softwareontology.wordpress.com/2011/04/01/user-sourced-competency-questions-for-software/
[2] http://owl.cs.manchester.ac.uk/publications/talks-and-tutorials/fhkbtutorial/

   (b) *Binary Question:* The question specifies a subject that should be checked on satisfiability of stated constraints. The answer is boolean: true or false.

   (c) *Counting Question:* Similar to a selection question, but returning a count of entities of the result set instead of the set itself.

2. **Element Visibility:** States whether the entities inside the question are available explicitly or implicitly.

3. **Question Polarity:** Is concerned with negated and non-negated questions.

4. **Predicate Arity:** Describes the number of of arguments of the main predicate.

   (a) *Unary Predicate:* Whether a certain entity has a certain instance.

   (b) *Binary Predicate:* Whether a certain relation between two entities exist.

   (c) *N-ary Predicate:* Whether relations between n entities exist. N-ary predicates cannot be represented in DL other than using a concept called reification, splitting them up into multiple binary predicates.

5. **Relation Type:** Defines whether a relation is a data property-based or object property-based relation.

6. **Modifier:** Defines either a restriction on the cardinality of a relation expression or limits the value of a data property.

   (a) *Quantity Modifier:* Sets a restriction on the cardinality of a relation expression, both for object property relations and datatype property relations. Can be a concrete value or a value range, e.g. 5, a superlative value, for example "the most" or a comparative value, for instance "more than" .

   (b) *Numeric Modifier:* Imposes a restriction on the value of a datatype property, such as $price > 15\$$.

7. **Domain-independent Element:** An element that does appear in multiple universes - independent from the universes' content. Depending on the question that is asked, the ontology could be required to contain an abstraction of that element.

   (a) *Temporal Element:* An element concerned with the logics of time. For example:"When was version X released?"

   (b) *Spacial Element:* An element referring to a location, not necessarily a geographical one. E.g.:"Where do I get updates?"

2.3 Authoring Tests

Based on the Archetypes defined in Section 2.2, so called *Authoring Tests* (ATs) were elaborated that can be used to check on presuppositions implicitly hidden inside the question, and therefore whether or not a CQ is answerable. In Table 2.4, all ATs are

Table 2.1: CQ Archetypes by [Ren et al., 2014]. (PA = Predicate Arity, RT = Relation Type, M = Modifier, DE = Domain Independent Element; CE = Class Expression, OPE = Object Property Expression, DP = Datatype Property, I = Individual, NM = Numeric Modifier, PE = Property Expression, QM = Quantity Modifier; obj. = Object Property Relation, data. = Data Property Relation, num = Numeric Modifier, quan. = Quantitative Modifier, tem. = Temporal Element, spa = Spatial Element)

ID	Pattern	Example	PA	RT	M	DE
1	Which [CE1] [OPE] [CE2]?	Which pizzas contain pork?	2	obj.		
2	How much does [CE] [DP]?	How much does Margherita Pizza weigh?	2	data.		
3	What type of [CE] is [I]?	What type of software (API, Desktop application etc) is it?	1			
4	Is the [CE1] [CE2]?	Is the software open-source-development?	2			
5	What [CE] has [NM] [DP]?	What pizza has the lowest price?	2	data.	num.	
6	What is the [NM] [CE1] to [OPE] [CE2]?	What is the best/fastest/-most robust software to read-/edit this data?	3	both	num.	
7	Where do I [OPE] [CE]?	Where do I get updates?	2	obj.		spa.
8	Which are [CE]?	Which are gluten free bases?	1			
9	When did/was [CE] [PE]?	When was the 1.0 version released?	2	data.		term.
10	What [CE1] do I need to [OPE] [CE2]?	What hardware do I need to run this software?	3	obj.		
11	Which [CE1] [OPE] [QM] [CE2]?	Which pizza has the most toppings?	2	obj.	quan.	
12	Do [CE1] have [QM] values of [DP]?	Do pizzas have different values of size?	2	data.	quan.	

Table 2.2: CQ Sub-types of Archetype 1 by [Ren et al., 2014]. (QT = Question Type, V = Visibility, QP = Question Polarity; CE = Class Expression, OPE = Object Property Expression; sel. = Selection Question, bin. = Binary Question, cout. = Counting Question, exp. = Explicit, imp. = Implicit, sub. = Subject, pre. = Predicate, pos. = Positive, neg. = Negative)

ID	Pattern	Example	QT	VT	QP
1a	Which [CE1] [OPE] [CE2]?	Which software can read a .cel file ?	sel.	exp.	pos.
1b	Find [CE1] with [CE2]?	Find pizzas with peppers and olives?	sel.	imp. pre.	pos.
1c	How many [CE1] [OPE] [CE2]?	How many pizzas in the menu contain meet?	cout.	exp.	pos.
1d	Does [CE1] [OPE] [CE2]?	Does this software provide XML editing?	bin.	exp.	pos.
1e	Be there [CE1] with [CE2]?	Are there any pizzas with chocolate?	bin.	imp. pre.	pos.
1f	Who [OPE] [CE]?	Who owns the copyright?	sel.	imp. sub.	pos.
1g	Be there [CE1] [OPE]ing [CE2]?	Are there any active forums discussing its use?	bin.	exp.	pos.
1h	Which [CE1] [OPE] no [CE2]?	Which pizza contains no mushroom?	sel.	exp.	neg.

Table 2.3: CQ Archetype absolute Distribution based on [Ren et al., 2014].

Archetype	1	2	3	4	5	6	7	8	9	10	11	12
Software Collection	38	11	1	1	0	4	5	5	3	7	0	0
Pizza Collection	23	7	4	0	5	1	0	22	0	2	5	1
Total	61	18	7	1	5	5	5	27	3	9	5	1

listed. Revisiting the patterns found in CQs by [Ren et al., 2014], each such pattern can be used to define whether an AT is applicable to an Archetype or not.

1. **Occurrence:** The Occurrence AT checks whether a certain entity actually exists in the ontology. For the question "Which pizza has the topping tomato?", this test would check whether entities "Pizza", "hasTopping" and "Tomato" exist in the ontology. Occurence tests do not require a resoner, the ontology itself can be queried for the answer.

2. **Class Satisfiability:** Class Satisfiability checks for every class expression whether it is satisfiable inside the ontology. For the above example, "Pizza" and "Tomato" would be checked to be satisfiable.

3. **Relation Satisfiability:** It is used to check whether a stated relation is actually allowed to exist - with polarity either being positive or negative. Using the above example, "Pizza" must have the possibility to have a relation with "hasTopping", which itself should have the range "Tomato".

4. **Meta-Instance**: Meta-Instance tests whether an entity from a question has the type required to be embedded in such a question. In above example, "Pizza" and "Tomato" both must be a class expression, and "hasTopping" must be an object property expression. A reasoner is not required for this test, the type of any entity is encoded in the ontology rdf / xml format.

5. **Cardinality Satisfiability:** Ensures that a specified exact cardinality used in the question is satisfiable in the ontology. For instance, "Which pizza has three toppings?" requires the ontology to allow a pizza to have exactly three toppings.

6. **Multiple Cardinality:** When a superlative quantity modifier is used, this test checks whether for any $n \in \mathbb{N}^+$, the relation is still possible, e.g. "Which pizza has the most toppings?"

7. **Comparative Cardinality:** If a Comparative Cardinality is used, like "has more than", this test checks whether - given two relations - both relations are allowed to have each one more relation than the other. Using the example: "Does Pizza Margherita have more toppings than Pizza Prosciutto?", this test would check satisfiability of Pizza Prosciutto having one more topping than Pizza Margherita and vice versa.

8. **Multiple Value:** Verifies that a datatype property is able to have different values. In the sentence "Which is the best pizza?", it is implicitly stated that Pizzas must have some score that we call "hasScore". "hasScore" must be allowed to adopt different values, otherwise the question cannot be answered correctly.

9. **Range:** Ensures that, when using datatype properties, their values are comparable. In the "Score" example above, the scores must be present in any numerical format such that comparisons can be made, e.g. *Integer, Float, Double.*

Table 2.4: Authoring Tests by [Ren et al., 2014]. (E: Expression, CE: Class Expression, P: Property, n: Modifier)

AT	Parameter	Checking
Occurrence	[E]	E in ontology vocabulary
Class Satisfiability	[CE]	CE is satisfiable
Relation Satisfiability	[CE1] [P] [E2]	$CE1 \sqcap \exists P.E2$ is satisfiable, $CE1 \sqcap \neg\exists P.E2$ is satisfiable
Meta-Instance	[E1] [E2]	$E1$ has type $E2$
Cardinality Satisfiability	[CE1] [n] [P] [E2]	$CE1 \sqcap = nP.E2$ is satisfiable, $CE1 \sqcap \neg = nP.E2$ is satisfiable
Multiple Cardinality (on superlative quantity modifier)	[CE1] [P] [E2]	$\forall n \geq 0, CE1 \sqcap \neg = nP.E2$ is satisfiable
Comparative Cardinality (on quantity modifier)	[CE1] [P1] [P2] [E1] [E2]	$\exists n \geq 0$, $CE1 \sqcap \leq nP1.E1$ and $CE1 \sqcap (n+1)P2.E2$ are satisfiable, $\exists m \leq 0$, $CE1 \sqcap \geq mP2.E2$ and $CE2 \sqcap \geq (m+1)P1.E1$ are satisfiable
Multiple Value (on superlative numeric modifier)	[CE1] [P]	$\forall D \subseteq range(P)$, $CE1 \sqcap \neg\exists P.D$ is satisfiable
Range	[P] [E]	$\top \sqsubseteq \forall P.E$

Ren et al. does not present a direct mapping from CQ Archetype to ATs. They however envision a system where ATs can be automatically checked for any CQ, returning results of these tests to users. Recent work of [Dennis et al., 2017] tried to validate above theory of Ren et al. empirically, handing out CQs and corresponding ATs to participants and asking them whether these ATs make sense or not. They encountered that authors proficient in ontology authoring actually had learned about the concept of presuppositions and therefore foresaw the implications of the ATs. Trying to help them preventing presuppositions when authoring an ontology is not as useful as doing this for a novice.

2.4 Ontology Evolution Mapping

While Authoring Tests and Competency Questions are targeted at novice users and in some extent as supportive system for professional ontology authors, Ontology Evolu-

tion Mapping targets mostly professionals. Most of today's ontologies are developed by multiple curators at the same time. To allow this collaboration more effectively, an approach by [Hartung et al., 2013] called *COnto-Diff* proposes a new method to calculate changes between versions of an existing ontology. The differences (Diffs) between two given versions of an ontology can be first associated with very basic *Changes Operations* as "insert", "delete" and "update". This initial mapping is enriched by more complex Change Operations that combine multiple basic Change Operations, as for example adding an entire subgraph or moving a node from one location to another. Complex Change Operations can contain complex Change Operations themselves, however currently these embedded Change Operations are not visible and countable by the user. This algorithm can be applied to any ontology, given that old and diverse versions of that ontology exist. The Diffs acquired by this method give insight in how an ontology is developed by professionals.

The state of the approach in the paper of [Hartung et al., 2013] has been refined by the authors in recent years, even though there is no publication existing yet that documents these refinements. Therefore there exist Change Operations in the code base[3] that are not described as such in their paper. This includes for example the Change Operation "addInner" that adds a node in between two existing nodes. Table 2.5 shows some of the detectable complex Change Operations, including ones that are not described in the paper.

As previously stated, the current COnto-Diff version does allow the application of complex Change Operations recursively, but these Change Operations are lost when assembling the results. In her recent, not yet published master thesis, [Pernischova, 2018] proposed an approach to include such embedded complex Change Operations with the result that these Change Operations are now countable as well.

2.5 OWL API

OWL, the Web Ontology Language, was recommended by W3C in 2004 [Patel-Schneider et al., 2004], and later on updated to OWL 2 [W3C OWL Working Group, 2009]. It is used to describe entities of an universe in such a way that it is interpretable by computers. In addition, OWL allows easy sharing of ontologies by using standard formats like RDF and XML. The OWL API for Java was developed by [Horridge and Bechhofer, 2011], and is currently one of the most well known APIs in use. It has the capability to read ontologies and to attach OWL Reasoners (Section 3.2.2) to be able to query ontologies. Many of the most important reasoners, developed by different universities world wide, have added OWL API support to their reasoners, which lowers the effort required in setting up such a software system. OWL API is available open-source on github.com[4].

[3] http://dbserv2.informatik.uni-leipzig.de:8080/webdifftool/WebDiffTool.html?fromOnex=-True&ontology=

[4] https://github.com/owlcs/owlapi

Table 2.5: Excerpt of COnto-Diff operations with descriptions and their inverses by [Hartung et al., 2013] with additional Change Operations not named in their paper (marked by *).

Change Operation	Description	Inverse Change Operation
move(c, C_To, C_From)	Moves a concept c and its subgraph from concept set C_From to the concept set C_To.	move(c, C_From, C_To)
addLeaf(c, C_Parents)	Insertion of a leaf concept c below the concepts in C_Parents.	delLeaf(c, C_Parents)
addSubGraph(c_root, C_Sub)	Inserts a new subgraph with root c_root and concepts C_Sub connected by "is_a" and "is_part_of" relationships.	delSubGraph(c_root, C_Sub)
addInner()*	Adds a node in between two existing nodes.	removeInner()
merge(Source_C, target_c)	Merges multiple source concepts Source_O into one target concept target_C.	split(target_c, Source_C)
toObsolete(c)	Concept c becomes obsolete, i.e., it should not be used any more.	revokeObsolete(c)
split(source_C, Target_C)	Splits one source concept source_c into multiple target concepts Target_C	merge(Target_C, source_c)
substitute(c1, c2)	Concept c1 is replaced by c2.	substitute(c2, c1)
revokeObsolete(c)	The obsolete status of c is revoked, i.e., it becomes active again.	toObsolete(c)

3

ReasonBench++

This chapter outlines the implementation of the *ReasonBench++* (RB++) framework. Section 3.1 is concerned with an analysis of the state of the art theories described in the Related Work Chapter 2 and outlines necessary changes to these theories such that they can be reused in the implementation. Section 3.2 describes the requirements concerning the capabilities and functions of the software, as well as a run-down of third-party software sources. Section 3.3 discusses the details of the implementation of the RB++ software, with a declaration of the most important packages as well as its inputs and outputs. Finally, in Section 3.4, the exact benchmarking process is outlined.

3.1 Problem Analysis

The goal of this software framework is a reasoner benchmarking software that tests different reasoners on their capability to recover from a change made by an ontology author. Recovering from a change of an ontology author has multiple facets, which define the key objectives our benchmark should investigate:

- The time it takes the reasoner to be able to process this change internally (with or without caching)
- Whether the speed of the reasoner degrades / rises after a change
- Whether the reasoner is still able to respond to queries correctly

In a production system, a reasoner should be able to recover from such a change as quickly as possible, as it might return wrong results to other queries it receives at the same time. The benchmark should therefore mimic a user querying the reasoner (tasks), while changes take place in the background. These tasks should be as unbiased as possible, such that no advantages are given to any test subject. To mirror various applications of ontologies - resulting in different ontology structures - the benchmark should also be executable for any ontology. Automation of all these processes is therefore an important aspect of the framework.

To generate queries, CQOA (see Section 2.2) comes into play. First of all, it covers the full process of converting real language questions - being a CQ Archetype - to ATs that

can be executed and cross-checked in a benchmark. The query of the user is mapped to an Archetype, for which specific ATs apply. The ATs are the queries, the CQs and Archetypes are the template the ATs are created from. The answers to ATs allow us to check whether a question - in the current setting and universe of the ontology - is satisfiable or not. A CQ to which all its ATs are satisfiable is interpreted as being satisfiable as a whole and can be assumed to be a reasonable, authentic task - as all necessary information is encoded in the ontology. The satisfiability of CQs and its ATs is therefore a comparable measure that can be cross-checked between different reasoners. It is important to note here that failing some ATs of a CQ does not imply that this question is per se unauthentic, but that the ontology needs more information to answer this CQ. It therefore could also indicate a design flaw of the ontology.

Figure 3.1 depicts the process of generating CQs. The CQs are generated by using the CQ Archetypes as templates. The generation process maps entities available in the input ontology against the templates, yielding a set of CQs per Archetype. Based on the amount of CQs required in the benchmark, the generator will then select n CQs based on the CQ-Archetype distribution. The resulting set of CQs is then used during the benchmarks, where the entailed ATs of the CQs are the queries the reasoners have to answer.

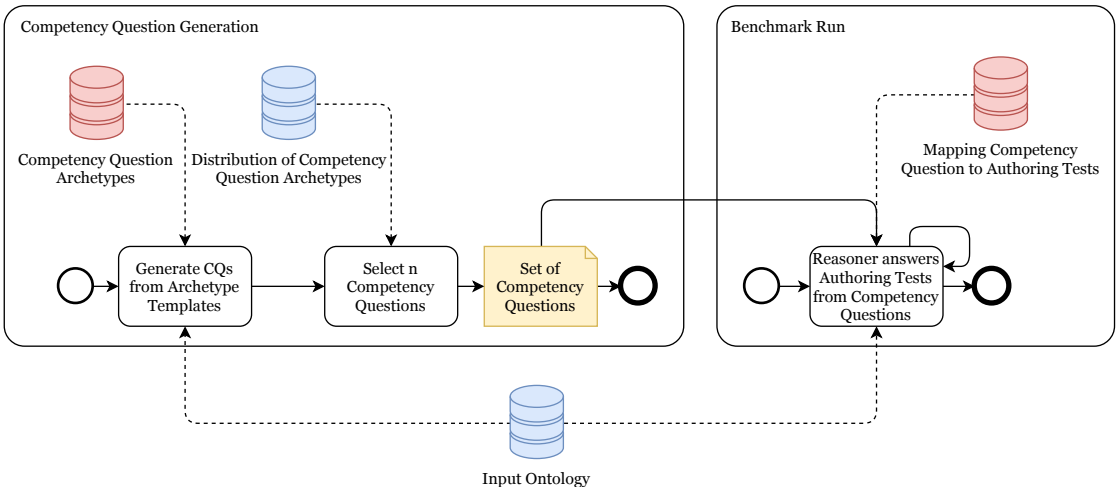

Figure 3.1: Graphical representation of the CQ generation process of RB++.

When editing the ontology, the results of the ATs of a specific CQ are likely to change. This can be detected and evaluated. At the same time, CQOA mimics real world questions and has proven itself doing so quite well [Ren et al., 2014, Dennis et al., 2017], allowing the generation of genuine tasks for our reasoners. Query generation and its problems is further discussed in Section 3.1.1.

Generation of edits makes use of the theory of Ontology Evolution Mapping (see Section 2.4. Generating edits should not be done at random, and the changes have to reflect real-life changes that an ontology author would actually apply. Using the COnto-Diff Algorithm allows us to evaluate existing ontologies onto the changes between versions

to create a distribution of Change Operations. This distribution can then be used to generate real-life ontology edits with a proportional distribution of different kinds of Change Operations. As COnto-Diff also specifies the contents and logical implications of every specific Change Operation and its inverse, we envision a forwards and backwards mode in our framework. As we start with a full ontology (in backwards mode), Change Operations applied in reverse would normally result in a gradually narrowing ontology. If enough changes are applied, the ontology is empty. As we know the inverse of every Change Operation, we can now switch to mode forwards, applying one edit after each other. The resulting ontologies between every edit are now all different versions of the original ontology. In addition, multiple Change Operations per edit should be possible.

Figure 3.2 displays the process of edit generation. Using the Change Operation distribution, the set of Change Operations and the input ontology, new Change Operations are generated and applied to the input ontology. As the sequence of Change Operations grows, the input ontology changes its contents accordingly. The generated sequence of Change Operations is then used during a benchmark run to alter the input ontology after every round. Constraints and restrictions of edit generation are discussed in Section 3.1.3.

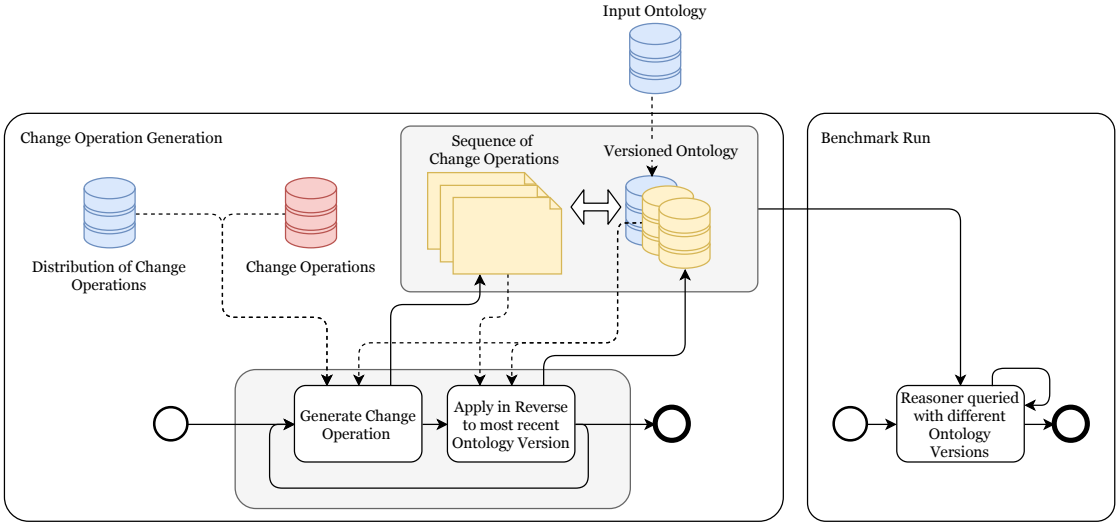

Figure 3.2: Graphical representation of the edit generation process of RB++.

Following, a general process is worked out that should reflect a real life application of a reasoner being queried. The envisioned process of the framework consists of multiple parts (see Figure 3.3). RB++ should first load an existing ontology. Next, it should create n CQs through a stochastic process based on the contents of that ontology, where the CQs should be as close to real life CQs as possible (see Section 3.1.1). A baseline benchmark is now carried out on the original ontology's state where a specified baseline-reasoner has to answer all ATs of the generated CQs, requiring a mapping from CQs to ATs (see Section 3.1.2). In succession, RB++ should generate edits stochastically comprised of the ontology's content and apply all of them in reverse to the ontology,

which most likely will result in a smaller or empty ontology (see Section 3.1.3). The benchmark can now be run, for each reasoner m times. In the first iteration of the benchmark, the reasoners run against the ontology with all edits applied, where it has to answer all ATs that are mapped to all CQs. In the second and every subsequent run, some of the generated edits should be applied to the ontology in forward mode, and in immediate succession, the reasoner should answer the same ATs again he already answered in the previous run. The runs continue until all edits have been applied to the ontology, resulting in the ontologies original state. This run acts as a control group, as the ATs of the CQs should now return the same result as in the baseline run, where we check for differences in the results of the ATs. If one of the reasoners returns different results compared to the baseline run, it can be assumed that the results of the reasoner are inconsistent. Repeating this benchmark m times will allow us to calculate statistics on the speed of the queried reasoners as well as whether they are prone to returning wrong answers.

The described parts of this process are discussed in more detail in the following subsections.

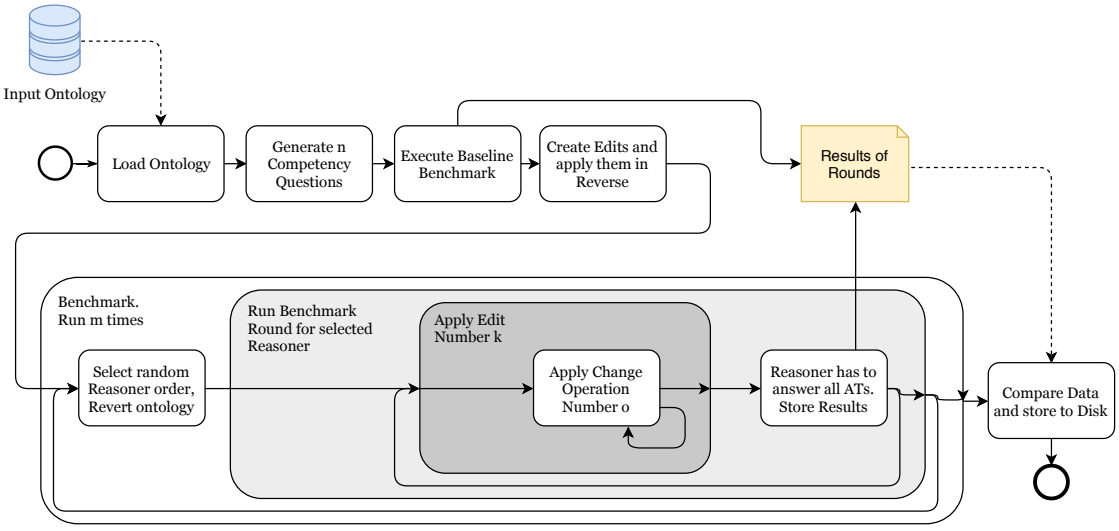

Figure 3.3: Graphical representation of the benchmarking process of RB++.

3.1.1 Generating Competency Questions stochastically

Generating random CQs entails some complications. The Archetypes of CQs have been defined by [Ren et al., 2014] (Table 2.1, 2.2), as well as an approximate distribution of these Archetypes from an analysis of sets of publicly available CQs (see Table 2.3). This information can - with some adjustments - be reused in this approach. The CQs proposed in CQOA were created with linguistic patterns in mind, these however do not necessarily reflect the logic of a computer and how it interacts with that data it

matches to that pattern. Analysing the CQs given from CQOA (Table 2.1) from an implementation perspective returned the following observations:

- Archetypes 7 and 9 both contain a domain independent element. For a reasoner being able to answer queries containing that sort of logic, it is necessary that this logic is encoded into the ontology. When using different ontologies for the process, it is unknown for the reasoner whether this logic is implemented. In most cases, workarounds would be used to reflect these domain independent elements. As an example of a workaround, the question: "When was the 1.0 version released?" requires classes "Version" and "Software", an object property "hasVersion" and a datatype property "hasReleaseDate" to be able to answer that question. These Archetypes are therefore removed.

- Archetypes 1 and 10 both contain two class expression parameters, as well as one object property expression. Implementation-wise, the generated CQs based on these two Archetypes will not differ from each other. The predicate arity is currently not interpretable by this software. Therefore these two Archetypes are treated equivalently. Archetype 10 was therefore removed and its distribution share is added to Archetype 1.

- Archetype 6 can be replaced by an instance of Archetype 1 and Archetype 5. For example: The Archetype 6 question "What is the healthiest pizza that has topping tomato?" can be replaced by a the query (Archetype 1): "Which pizza has topping tomato?" and the query (Archetype 5): "What [CE1] has the highest nutrition value?", where CE1 is the result set of query 1. The distribution share of Archetype 6 is therefore added to Archetype 1 and 5.

- Archetype 11 and 12 are concerned with cardinalities of either superlative or absolute nature. As it is difficult to represent both cases at the same time with machine logic, Archetype 11 is split into 11a, dealing with absolute numeric cardinalities, and 11b, dealing with superlative cardinalities. The same applies for Archetype 12, being split into 12a and 12b. The distribution of both is halved into 11a and 11b, 12a and 12b respectively.

Concerning the sub-archetypes in Table 2.2, it was found that an implementation of these sub-archetypes is unnecessary for our process:

- Sub-archetypes 1a and 1h are equivalent to Archetype 1, 1h being the negated (complement) version of 1a. They both result in a set of class expressions.

- Sub-archetype 1b lacks the object property expression available in 1a. To answer this question, the reasoner has to iterate over all object property expressions in the signature of CE1 and has to inspect all possible class expressions attached to these object property expressions. If CE2 is contained, this instance is added to the returned list. This Archetype is therefore a more complex version of 1a, but the structure is equivalent. In addition, this sub-Archetype can include presuppositions not

detectable by a machine. As an example, let O be an ontology containing the following entities: $\{Parent, FemaleChild, MaleChild, hasChild, hasChildWish\}$. When O is used to answer the question: "Find parents with a female child?", the reasoner would return two sets, the set using relation $hasChild$ as well as $hasChildWish$ - which in most situations would be incorrect presupposing that the parents actually have a child already.

- Sub-archetype 1c is equivalent to 1a, returning the count of the result-set of a question of Archetype 1a.
- Sub-archetype 1d is similar 1a, but it asks for a boolean return value for a specific entity. The process for the reasoner stays the same as in 1a.
- Sub-archetype 1e is similar to 1b, but as in 1d it asks for a boolean return value.
- Sub-archetype 1f lacks one class expression, which would be replaced by the top class $\top$. This is equivalent to 1a.
- Sub-archetype 1g is simliar to 1a, but asks for a boolean return value.

Therefore, the sub-archetypes are not used for this implementation, as all of them are either covered by Archetype 1 or contain for now unsupported, not answerable presuppositions. Table 3.1 shows the final selection of CQs used by RB++.

3.1.2 Mapping Authoring Tests to Competency Question Archetypes

Executing ATs automatically based on the CQ Archetype they stem from requires a mapping from all CQ Archetypes to its applicable ATs. Based on the theory of CQOA and its Authoring Tests (see Sections 2.2, 2.3), the following list was worked out that shows a mapping based on certain peculiarities of CQ Archetypes:

- **Occurrence**: Applies to all entities in an Archetype. Is applicable for every Archetype, as every Archetype must contain at least one entity.
- **Class Satisfiability**: Is applicable for every class expression of an Archetype. Every class can be checked on whether it is satisfiable or not.
- **Relation Satisfiability**: Is applicable for all Archetypes that explicitly contain at least two class expressions and one object property relation. It is currently not applicable to datatype property relations, as OWL 2 lacks an implementation of `OWLDataFactory.getOWLDataComplementOf()` that allows an `OWLClassExpression` as parameter. The test is therefore incomplete and currently removed from the framework.
- **Meta Instance**: Is not applicable to any Archetype. The Meta Instance AT would be required if the input parameters of a query could be set manually, where for example a user could mix and match datatypes with data properties (including

Table 3.1: CQ Archetypes used by RB++. (PA = Predicate Arity, RT = Relation Type, M = Modifier, DE = Domain Independent Element; CE = Class Expression, OPE = Object Property Expression, DP = Datatype Property, I = Individual, NM = Numeric Modifier, PE = Property Expression, QM = Quantity Modifier; obj. = Object Property Relation, data. = Data Property Relation, num = Numeric Modifier, quan. = Quantitative Modifier, abs. = Absolute, sup. = Superlative, tem. = Temporal Element, spa = Spatial Element)

ID	Pattern	Example	PA	RT	M	DE
1	Which [CE1] [OPE] [CE2]?	Which pizzas contain pork?	2	obj.		
2	How much does [CE] [DP]?	How much does Margherita Pizza weigh?	2	data.		
3	What type of [CE] is [I]?	What type of software (API, Desktop application etc) is it?	1			
4	Is the [CE1] [CE2]?	Is the software open-source-development?	2			
5	What [CE] has [NM] [DP]?	What pizza has the lowest price?	2	data.	num.	
8	Which are [CE]?	Which are gluten free bases?	1			
11a	Which [CE1] [OPE] [QM] [CE2]?	Which pizza has the most toppings?	2	obj.	quan. abs.	
11b	Which [CE1] [OPE] [QM] [CE2]?	Which pizza has the most toppings?	2	obj.	quan. sup.	
12a	Do [CE1] have [QM] values of [DP]?	Do pizzas have different values of size?	2	data.	quan. abs.	
12b	Do [CE1] have [QM] values of [DP]?	Do pizzas have different values of size?	2	data.	quan. sup.	

combinations that are impossible). As our queries are generated automatically, with the knowledge embedded in the ontology, no parameter mapping could occur that would match wrong types with each other.

- **Cardinality Satisfiability:** Is applicable for all Archetypes that contain an absolute quantity modifier. An absolute cardinality is an integer number.
- **Multiple Cardinality:** Is applicable for all Archetypes that contain a superlative or comparative quantity modifier, such as "the most", "the fewest" or "different number of".
- **Comparative Cardinality:** Is not applicable to any Archetype. Comparative Cardinality would require an Archetype that compares two relations among each other, but no such Archetype exists.
- **Multiple Value:** Is not applicable to any Archetype due to technical restrictions. To date, there exists no possible way in OWL to check an ontology onto the actual values of a data property relation.
- **Range:** In its original form, it would not be applicable to any Archetype, with the same reasons as formulated for Meta-Instance. An automatically generated query would not use incomparable data types. However, the Range AT is now used in conjunction with object property relations, to check whether the relation of some class CE with some object property OPE is satisfiable. This is applicable to all Archetypes that contain an object property relation and a class that should be in range of that object property.

Based on these findings, Table 3.2 was created that shows direct mappings from every CQ Archetype to its ATs.

3.1.3 Generating Ontology Edits stochastically

To be able to generate ontology edits that are close to real life, the Ontology Evolution Mapping approach by [Hartung et al., 2013] is used. As there exists an inverse for every Change Operation a user would apply, the idea is to use a distribution of an existing, versioned ontology and to apply Change Operations using that distribution of Change Operations in an inverted fashion to a full or complete ontology. There exists one constraint of that approach, being that some of the Change Operations, if applied in reverse, require additional domain knowledge not encoded into the ontology. In consequence, any Change Operation that's inverse requires the addition of some entity to the ontology would result in new, to the executing machine unknown entities being added to the ontology. This is the case for "split" Change Operations, as well as all possible Change Operations that would delete some entity from the ontology.

To get an approximate real life distribution of Change Operations (see Table 2.5, the COnto-Diff algorithm with included refinements of [Pernischova, 2018] was applied on

Table 3.2: Mapping from Archetype ID to applicable Authoring Tests. (CE = Class expression, OPE = Object Property Expression, DP = Datatype Property, I = Individual, NM = Numeric Modifier, PE = Property Expression, QM = Quantity Modifier, * = currently not implemented.)

ID	Pattern	Applicable Authoring Tests
1	Which [CE1] [OPE] [CE2]?	Occurrence, Class Satisfiability, Relation Satisfiability, Range
2	How much does [CE] [DP]?	Occurrence, Class Satisfiability, Relation Satisfiability*
3	What type of [CE] is [I]?	Occurrence, Class Satisfiability
4	Is the [CE1] [CE2]?	Occurrence, Class Satisfiability
5	What [CE] has [NM(absolute)] [DP]?	Occurrence, Class Satisfiability, Relation Satisfiability*
8	Which are [CE]?	Occurrence, Class Satisfiability
11a	Which [CE1] [OPE] [QM(absolute)] [CE2]?	Occurrence, Class Satisfiability, Relation Satisfiability, Cardinality Satisfiability, Range
11b	Which [CE1] [OPE] [QM(superlative)] [CE2]?	Occurrence, Class Satisfiability, Relation Satisfiability, Multiple Cardinality, Range
12a	Do [CE1] have [QM(absolute)] values of [DP]?	Occurrence, Class Satisfiability, Relation Satisfiability*, Cardinality Satisfiability
12b	Do [CE1] have [QM(superlative)] values of [DP]?	Occurrence, Class Satisfiability, Relation Satisfiability*, Multiple Cardinality

Table 3.3: Absolute amount of Change Operations detected in the Gene Ontology [Harris et al., 2008], dating between 2010-01-01 and 2018-04-01. * points to Change Operations that are not implemented either due to constraints or sparse usage.

Change Operation	Amount of Change Operations
move	11657
addLeaf	5748
addSubGraph	2775
addInner	1751
merge	627
toObsolete*	525
split*	284
substitute*	26
revokeObsolete*	5

the Gene Ontology[1] by [Harris et al., 2008]. This yields the results visible in Table 3.3. The compared versions are taken from their archive, dating between 2010-01-01 and 2018-04-01.

Some Change Operations cannot be implemented due to the following reasons:

- **toObsolete:** This Change Operation would require a possible annotation to flag an entity such that it should not be used. This is not the case for all ontologies.
- **split:** Split would require a merge operation when generating edits. Finding two nodes to merge without domain knowledge is currently infeasible.
- **substitute and revokeObsolete:** Are not required due to their sparse usage.

Finally, the framework requires a graph representation of any given ontology to be able to read out graph-based attributes such as being a leaf or being an inner node.

3.2 Requirements

From the process and the analysis outlined in Section 3.1, the following upper-level requirements for RB++ can be defined. In the following sections, required software frameworks are described.

1. The framework must be able to **generate CQs through a stochastic process** based on an existing ontology. These questions should be as close to life and reasonable as possible.
2. The framework must be able to **generate edits through a stochastic process** based on an existing ontology to simulate a change of an ontology author, again as close to life and reasonable as possible.

[1] cf. http://www.geneontology.org/

3. The framework must provide a **mapping from CQ Archetypes to ATs** and must be able to **execute ATs automatically** based on the CQ Archetype.

4. The framework must be **compatible with OWL API**.

5. The framework must be able to **execute the previously outlined process (see Section 3.1)**.

6. The framework should contain an **abstraction of the logic of CQOA**, a logic representation of ATs, CQs and CQ Archetypes, as well as their interaction.

7. The framework should contain an **abstraction of the logic of Ontology Evolution Mapping**, with its most important Change Operations.

8. The framework should contain an **abstraction of Change Operations**.

9. The framework must be able to **export** all generated data of the process to do a state of the art analysis.

3.2.1 OWL API for Java

At the time of creation of this document, OWL API (see Section 2.5) is available in version 5, however this version is not yet well supported by most of the available reasoners. Due to that, OWL API version 4 is used for the development of RB++. Reasoners supporting version 5 include HermiT and JFact. Reasoners supporting version 4 include HermiT, JFact, Pellet and ELK.

3.2.2 OWL Reasoner

An OWL Reasoners is a reasoner that supports the OWL API, which is the case for most today's reasoners (see Section 3.2.1). From the big list of available OWL Reasoners, some were chosen to be part of this benchmark, mostly due to their easy availability online, as well as their version support. What follows is a description of these reasoners.

- **HermiT:** HermiT [Glimm et al., 2014] was developed by the University of Oxford. It is based on hypertableau calculus, as well as other novel optimizations. [Chaussecourte et al., 2013] states that from v1.3.4 Hermit supports a very simplistic form of incremental reasoning. It has expressivity of $\mathcal{SROIQ}$ logic (OWL 2). Used version: v1.3.8.

- **JFact:** JFact[2] is a Java port of FaCT++ and shares its license. FaCT++ is a tableaux-based reasoner, developed by the University of Manchester [Tsarkov and Horrocks, 2006]. It lacks support for key constraints and some datatypes. Very little is known about the JFact implementation. [Dentler et al., 2011] state that FaCT++ does not support incremental reasoning. However a more recent article

[2] cf. http://jfact.sourceforge.net/

by [Tsarkov, 2014] introduces incremental reasoning for FaCT++. [Tsarkov, 2014] does not mention whether these changes have been forwarded to JFact. It has expressivity of $\mathcal{SROIQ}$ logic (OWL 2). Used version: v4.0.4.

- **Pellet:** Pellet [Sirin et al., 2007] is an OWL 2 description logic reasoner developed and maintained by Complexible Inc. It is available open source or under commercial license. Key features include optimizations for nominals, conjunctive query answering and first approaches in incremental reasoning. Incremental reasoning includes incremental classification (addition, removal) [Dentler et al., 2011]. It has expressivity of $\mathcal{SROIQ}$ logic (OWL 2). Used version: v2.4.0.

- **ELK:** ELK [Kazakov et al., 2012] was developed in a collaboration between the University of Oxford and the Ulm University. It is a specialized reasoner for the OWL EL language. It lacks some implementations of interface methods, such as `getObjectPropertyRanges()`. With version v0.4.0, ELK is supposed to support incremental reasoning[3] on the axiom types `SubClassOf`, `EquivalentClasses`, `DisjointClasses`, `ObjectPropertyDomain`, `ObjectPropertyAssertion`, `ClassAssertion`. Changes on other axiom types will trigger a full re-classification (such as sub-property and property relations). A more recent documentation entry on github however states that incremental reasoning is currently unavailable[4] due to a bug. ELK has expressivity of $\mathcal{EL}$. Used version: v0.4.3.

3.2.3 Other References

Main source for external libraries is Maven, which is used to automatically load and handle external libraries. Maven is used for following framework components:

- owlapi-distribution, version 4.3.1, net.sourceforge.owlapi, the OWL API distribution.

- org.semanticweb.hermit, version 1.3.8.431, net.sourceforge.owlapi, the OWL Reasoner "Hermit".

- elk-owlapi-standalone, version 0.4.3, org.semanticweb.elk and elk-reasoner, version 0.4.3, org.semanticweb.elk, the OWL Reasoner "ELK"

- jfact, version 4.0.4, net.sourceforge.owlapi, the OWL Reasoner "JFact"

- junit, version 3.8.1, JUnit Testing Environment

The reasoner "Pellet" is not available via Maven. The library was therefore downloaded directly and added to the extlib folder. It is available on github.com[5].

[3] cf. https://github.com/liveontologies/elk-reasoner/wiki/IncrementalReasoning
[4] cf. https://github.com/liveontologies/elk-reasoner/wiki/ReasoningTasks
[5] cf. https://github.com/ignazio1977/pellet/blob/releases/pellet-2.4.0-ignazio1977-dist.zip

3.3 Implementation

This section is outlining the implementation of ReasonBench++. It is split into different subsections. Subsection 3.3.1 discusses the different inputs and parameters of the framework. Subsection 3.3.2 outlines the outputs of a benchmark run. Subsections 3.3.3, 3.3.4, 3.3.5 and 3.3.6 describe the different packages of RB++ and how they interact.

RB++ is written in Java and uses a set of different classes to model the process of Section 3.1 in software. The classes are interconnected in a similar manner as the theory of CQOA models ATs, CQs and their Archetypes, and how Ontology Evolution Mapping models its Change Operations. The entry point for the RB++ framework resides in class `Main`.

There exist four major packages, being `benchmark`, `competencyquestion`, `authoringtest` and `edit`. `benchmark` contains the main benchmark logic, as described in Section 3.1). `competencyquestion` includes parts of the CQOA logic, being Archetypes, the CQ generators as well as the logic for the probability distribution used to generate the CQs (see Section 2.2). `authoringtest` comprises the implementation of all ATs (see Section 2.3). Finally, `edit` contains the logic of Ontology Evolution Mapping and its Change Operations (see Section 2.4). These packages all entail a "Manager"-class, where its logic is implemented, and at least one data container class ("-Info", "-Set") to store results. The Manager classes are implemented with re-use and stateless design in mind, therefore their constructors are mostly empty and the single methods they supply require all parameters as inputs. The container classes are intertwined to reflect the logic of CQOA and Ontology Evolution Mapping: The `BenchmarkInfo` class contains a field for the CQ container called `CompetencyQuestionSet`, a field for the generated edits of type `EditInfoSet`, as well as a list of maps of `AuthoringTestSet` that include results of the different benchmark runs.

In addition, there exist three minor packages. `ontology` provides instances of ontologies and ontology handling. `reasoner` allocates reasoner instances. `helper` contains supporting methods, used for statistical purposes or to print specific elements to the console. It also includes an `ExportService` class that is used to export results in the .csv format.

In the following subsections, the main components of the software are discussed.

3.3.1 Inputs

The input-parameters for RB++ are stored in the `BenchmarkInfo`-, as well as the `BenchmarkOptions`-class (see Section 3.3.3) which are used as main data storage containers. Parameters requiring additional explanation contain references to other sections where their function is explained in more detail.

The input parameters that are required to setup a benchmark are stored in an instance of the `BenchmarkInfo` class and are the following:

- `ontologyPath`: The local file path or URL to load the ontology. RB++ is capable to detect whether the input is an URL or a path to local storage.

- `reasonerList`: The list of all reasoners that should be benchmarked. The type `Reasoner` is an enum from where the reasoners can be chosen.
- `seed`: A long value that is used as seed for all succeeding operations. This seed is used to generate different benchmarks and allows traceability / re-execution of a specific test. It is used by all packages.
- `cqGenAmount`: The amount of CQs that should be generated (= "selected"). This amount is split among the different Archetypes according to the probability distribution in class `CompetencyQuestionProbabilityDistribution` (see Section 3.3.4).
- `reasonerForCQGen`: The reasoner that is used to generate the CQs. Does not have to be part of the `reasonerList` (see Section 3.3.4).
- `editAmount`: The amount of edits that should be created (see Section 3.3.3).

Additionally, when calling the method `benchmarkN()`, one has to provide `n`, which is the number of consecutive benchmark runs.

Parameters used to customize certain aspects of the benchmark are stored in the static `BenchmarkOptions` class. They include the following:

- `CQ_GEN_TYPE`: The type of CQ Generator that is used by the benchmark (see Section 3.3.4), either `ExpectedTrueCompetencyQuestionGenerator` or `RandomCompetencyQuestionGenerator`. *Default value:* `EXPECTED_TRUE`.
- `CQ_GEN_REMOVE_OWL_THING`: When generating CQs, `OWLThing` is also taken into consideration. By setting this flag to false, `OWLThing` is not allowed for any generated question (see Section 3.3.4). *Default value:* `TRUE`.
- `CQ_GEN_ONLY_SELECT_SATISFIABLE_CQS`: Whether CQs generated have to pass all their ATs to be selected for the benchmark run. If set to `TRUE` prevents CQs from being selected for the benchmark that have not passed their ATs (see Section 3.3.4). *Default value:* `TRUE`.
- `TEST_SELECTED_COMPETENCY_QUESTIONS_ONLY`: This flag sets whether ATs should be run for all created CQs or only the ones selected to be used for the benchmark run. If set to false, a big rise in computing time can be expected (see Section 3.3.5). *Default value:* `TRUE`.
- `USE_BUFFERED_REASONER`: Whether the reasoners created should run in buffered mode or non-buffered mode. A buffered reasoner is allowed to cache data before flushing changes. (see Section 3.3.3). *Default value:* `TRUE`.
- `SHUFFLE_REASONERS_DURING_BENCHMARK`: Flag that sets whether the benchmarked reasoners should be shuffled after every run in their benchmark execution order or not (see Section 3.3.3). *Default value:* `TRUE`.

- `INCLUDING_FLUSHING_IN_RUNTIMES`: Flag that sets whether the flushing call should be included in the timing measurements or not (see Section 3.3.3). `TRUE` includes flushing time. *Default value:* `TRUE`.

- `EDIT_GEN_REMOVE_OWL_THING`: Whether `OWLThing` is taken into consideration when generating edits. It is recommended not to use `OWLThing`, as this could lead to a corrupt ontology (see Section 3.3.6). *Default value:* `TRUE`.

- `CHANGE_OPERATION_SUB_GRAPH_MAX_DIFFERENCE_OUTERMOST_LEVEL`: When generating a Change Operation of type "subgraph", how high the difference in level of the candidate node inside the graph to the outermost leaf is allowed to be (see Section 3.3.6). 0 = all nodes allowed. Minimum level = 2. *Default value:* `2`.

- `CHANGE_OPERATION_MOVE_MAX_DIFFERENCE_OUTERMOST_LEVEL`: When generating a Change Operation of type "move", how high the difference in level of the candidate node inside the graph to the outermost leaf is allowed to be (see Section 3.3.6). 0 = all nodes allowed, minimum level = 1 (will just move leaves). *Default value:* `2`.

- `CHANGE_OPERATION_MOVE_MAX_LEVEL_UP`: When generating a Change Operation of type "move", how high the algorithm is allowed to push the the node from its previous location (see Section 3.3.6). Minimum level = 2. *Default value:* `2`.

Throughout the framework exist different distributions required to get real-life distributions of both CQs and edits. The distributions are located in the `competencyquestion` and `edit` packages. All of them are to be accessed statically, therefore they can be set before a benchmark run and will keep their individual values.

The `CompetencyQuestionProbabilityDistribution` is responsible for a real-life CQ distribution. By setting one of its `[...]Amount` variables to another value than default, the whole distribution will shift according to the newly fitted value. Default values used are listed in Section 2.2.

The `ChangeOperationProbabilityDistribution` is used for a real-life Change Operation distribution. Similar to the distribution above, by changing one of its public fields to another value, the complete distribution will shift accordingly. Default values used are listed in Section 3.1.3.

The `EditSizeDistribution` is extended by a `FixedEditSizeDistribution` as well as a `ExponentialEditSizeDistribution`. They are both used to set the amount of Change Operations in one edit. If the `FixedEditSizeDistribution` is used, an edit will always have a fixed amount of Change Operations. The `ExponentialEditSizeDistribution` on the other hand assignes low values more frequently than higher ones, with a definable upper bound as well as an alpha value that defines the steepness of the curve.

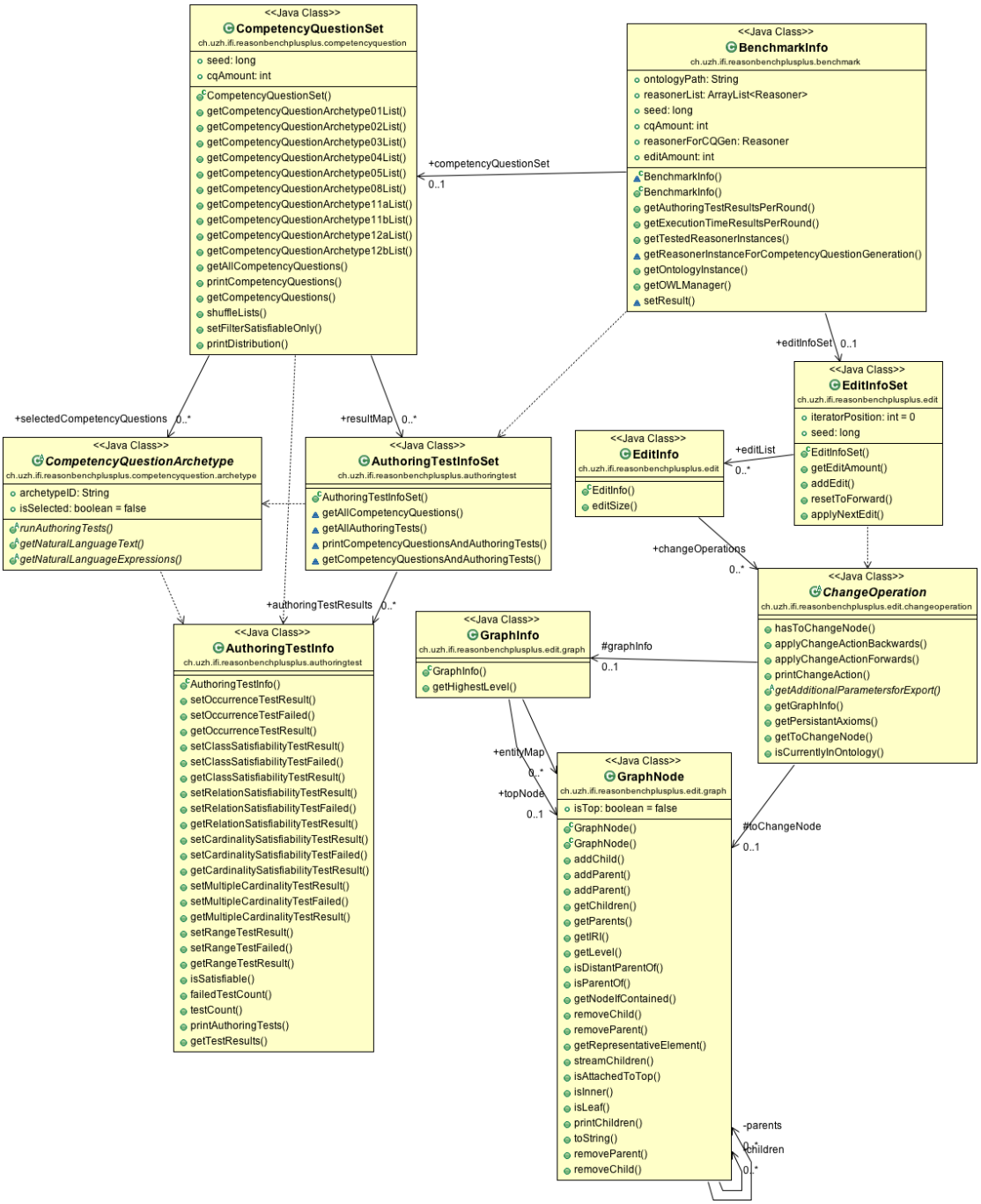

Figure 3.4: Contents of the `BenchmarkInfo` output.

3.3.2 Output

After a benchmark run, `BenchmarkManager` will return an instance of `BenchmarkInfo`, which includes all data of said benchmark run. Figure 3.4 shows the contents of the `BenchmarkInfo` instance.

`CompetencyQuestionSet` (see Section 3.3.4) contains all data of the generated CQs. The single CQs contain links to their respective AT results. `EditInfoSet` (see Section 3.3.6) includes all data of the generated edits. The edits themselves are linked to their representative `GraphNode` in a generated graph of the ontology. All classes mentioned above are serializable and can therefore be exported as a whole. In addition, the package `export` contains methods to export all of this data into `.csv` files.

3.3.3 Benchmark Package

The `benchmark` package contains the logic required to run a benchmark. It consists out of the `BenchmarkManager` class, the `BenchmarkOptions` class as well as the `BenchmarkInfo` data container class. All data required for a n-benchmark is fed into `BenchmarkManager` from class `Main` via the mentioned data container. If only a single run of the benchmark is required, `BenchmarkManager` also offers a constructor that allows the input of all data as single parameters.

There exist three major methods to call. In method `benchmarkN()`, `BenchmarkManager` first creates an instance of `CompetencyQuestionManager` to generate the CQs for this benchmark. In addition, the edits are generated by `EditManager`. Both the edits and CQs are then used for the benchmark runs. The method executes the private method `benchmark()` n times, with n as specified in one of its parameters.

The public method `benchmark()` on the other hand is to be used for a single benchmark run only. Therefore, the edits as well as the CQs have to be provided separately.

Finally, method `runBenchmarkAtEditPosition()` will run a benchmark where only one specific edit-version of the ontology will be used. This allows for comparing measurements of a normal benchmark with changes after every round and measurements taken at specific edit-versions of the ontology where in-between runs no changes are applied to the ontology.

Figure 3.5 shows the dependencies of class `BenchmarkManager` to the other available classes.

`BenchmarkInfo`, used for storing all settings and results for any benchmark, includes additional logic that allows retrieval of specific data that was generated during some benchmark run. In addition, it also contains methods to retrieve ontology and reasoner instances. The packages `ontology` and `reasoner` do the general handling of these parameters. As it is a requirement, all data from any benchmark-run is exportable (= serializable) (see Section 3.2) and is condensed inside `BenchmarkInfo`. Some components of OWL API are not serializable, therefore reasoners and ontologies are not serialized in their current instance, but instead as enums that represent them. The many fields of `BenchmarkInfo` are listed in Section 3.3.1, some of which are used in the `benchmark` package, while others are handed over to the other packages. `BenchmarkInfo` offers a

copy-constructor that allows copying all settings of a previous benchmark iteration - without their respective results. Finally, the class `BenchmarkOptions` provides general settings that are described in Section 3.3.1. `BenchmarkOptions` is static and therefore accessible from everywhere. Settings are applied immediately and globally. Changing settings during a benchmark run is possible but not recommended.

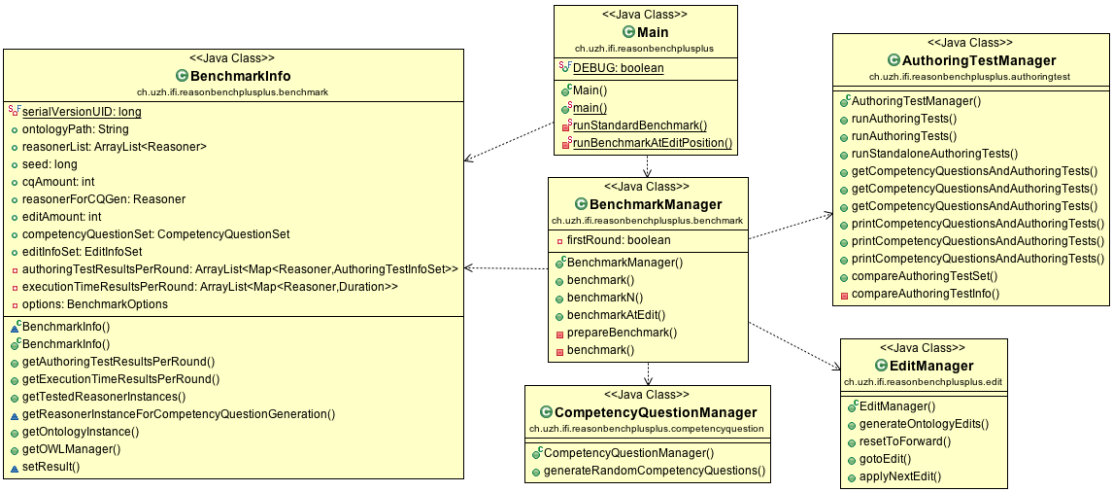

Figure 3.5: Class Diagram of `BenchmarkManager` and its surrounding classes.

Input-parameters directly influencing the `benchmark` package are:

- `USE_BUFFERED_REASONER`: Whether the tested reasoner created should run in buffered- or non-buffered mode. A reasoner running in buffered mode is allowed to cache changes applied to the ontology before applying them. Compared to a standard setting, where a change is applied and executed immediately, it can be expected that the buffered reasoner will suffer less concerning response times, but could - depending on the exact implementation of the buffer - return wrong results to an user until the method `flush()` is called that forces the reasoner to apply the changes. *Default value:* `TRUE`.

- `SHUFFLE_REASONERS_DURING_BENCHMARK`: Flag that sets whether the benchmarked reasoners should be shuffled after every run in their benchmark execution order or not. This function is used to prevent a potential speed up of the reasoners - or that some reasoner could have an advantage over another due to their execution order. *Default value:* `TRUE`.

- `INCLUDING_FLUSHING_IN_RUNTIMES`: Flag that sets whether the flushing call should be included in the timing measurements or not. The method is then called either before the timer is started or after. `TRUE` includes flushing time. *Default value:* `TRUE`.

- `n`: The number of consecutive benchmark runs (iterations). If four reasoners are provided, every of the four reasoners has to pass the benchmark run `n` times. The higher `n`, the more robust the results are going to be.
- `ontologyPath`: The local file path or URL to load the ontology. RB++ is capable to detect whether the input is an URL or a path to local storage.

3.3.4 Competency Question Package

The `competencyquestion` package contains its main logic in `CompetencyQuestionManager`, the implementation of the CQ Archetypes, as described in the Problem Analysis (see Table 3.2), the CQ-Generator `CompetencyQuestionGenerator` and a class to define the distribution of the CQs, `CompetencyQuestionProbabilityDistribution`. An overview of the contents are displayed in Figure 3.6.

The manager class only contains one method to call, being `generateRandomCompetencyQuestions()`. After the generation of the CQs, no more work on them is required. ATs are run from the package `authoringtest`.

The logic contained in this package is called from class `BenachmarkManager`. The CQs are reused in every benchmark run, once for every reasoner. There are usually many more CQs generated than required, therefore `CompetencyQuestionManager` selects - based on the class `CompetencyQuestionProbabilityDistribution`, as many CQs as required.

The CQ Archetypes inherit from their base class `CompetencyQuestionArchetype`, and contain additional fields for all their parameters, as well as the method `runAuthoringTests()` that executes all ATs defined in the mapping in Table 3.2.

The distribution of CQ Archetypes is realized in class `CompetencyQuestionProbabilityDistribution`. It is based on Table 2.3 and is used by the generators to return a real-life arrangement of CQs. The distribution is static and allows immediate changes and automatically adjusts the proportions of all archetypes if one of them is changed - for example in the case that not enough CQs were generated of a certain Archetype.

The CQ generators are extending class `CompetencyQuestionGenerator`. There exist two versions:

- `RandomCompetencyQuestionGenerator`: All entities available in the signature of the ontology are iterated over and mapped against each other. This results in many unauthentic CQs, and most of their ATs are not satisfiable.
- `ExpectedTrueCompetencyQuestionGenerator`: For every entity, one iterates over all entities in its signature. An exact explanation of its inner workings can be found below. This way, only related entities are matched with each other. The resulting ATs of the CQs are a lot more reasonable and are satisfiable in about 50% of the cases. It is used as default, as it returns the better results of the two.

To be more specific, the different CQs require different approaches to generate fitting parameter sets. The Archetypes and their approaches in the more elaborate version, `ExpectedTrueCompetencyQuestionGenerator`, are listed below.

Figure 3.6: Class Diagram of `CompetencyQuestionManager` and its surrounding classes.

- Archetypes 1, 2, 5, 11 and 12 containing an Object Property or Data Property Relation can be queried for their respective domain and range. From the returned classes, all subclasses can be matched against each other to get parameter pairs. For Archetype 1: If domains and ranges are not used by the authors (which would return an empty set for every possible parameter set), a fall-back method is implemented that uses subsumption (a subclass of a class) to determine object property relations.

- Archetype 4 is referring to subsumption as well. Therefore, for every class, the ontology is queried for all of its subclasses. The returned set and the class in the query form one parameter set.

- For Archetypes concerned with cardinalities, such as 11 and 12, the ontology is additionally queried for existing axioms defining some cardinality restrictions.

- For Archetypes 3 and 8, which both are concerned with individuals, the ontology is queried for instances fulfilling the given parameter set.

The input-parameters directly influencing this package are:

- `CQ_GEN_TYPE`: Which of the above CQ-generators should be used for the CQ-generation. There exist two options: `RandomCompetencyQuestionGenerator` and `ExpectedTrueCompetencyQuestionGenerator`. *Default value:* `EXPECTED_TRUE`.

- `CQ_GEN_REMOVE_OWL_THING`: When generating CQs, `OWLThing` is also taken into consideration. By setting this flag to false, `OWLThing` is not allowed for any generated question. *Default value:* `TRUE`.

- `CQ_GEN_ONLY_SELECT_SATISFIABLE_CQS`: Whether the generator is allowed to select CQs that have unsatisfiable ATs. If set to `TRUE`, the ATs of the generated CQs are executed before selecting among all CQs. *Default value:* `TRUE`.

- `reasonerForCQGen`: The reasoner that is used to generate the CQs. This should be the most stable of all reasoners, as the result of this CQ-Generation will be used as baseline dataset for all other benchmark runs.

- `cqGenAmount`: The amount of CQs that should be generated (= "selected"). During generation, all possible combinations of entities among each other are elaborated. The generator then randomly selects CQs, based on the given distribution of the CQ Archetypes.

3.3.5 Authoring Test Package

The `authoringtest` package includes logic for execution of ATs and is used in conjunction with package `competencyquestion`. The implementation of all ATs (see Table 2.4 and 3.2) can be found in class `AuthoringTest`. To test these ATs, there exists a test class called `AuthoringTestDynamic` in the "Test" subdirectory of the project using junit-tests. As every CQ comprises multiple ATs, the results of these tests is stored in the container class `AuthoringTestInfo`. This container is used in conjunction with every type of CQ, therefore it contains information on whether a certain AT was applicable for this CQ, if the test was executed successfully and the result of the test. The results of all CQs tested is aggregated inside the container class `AuthoringTestSet`.

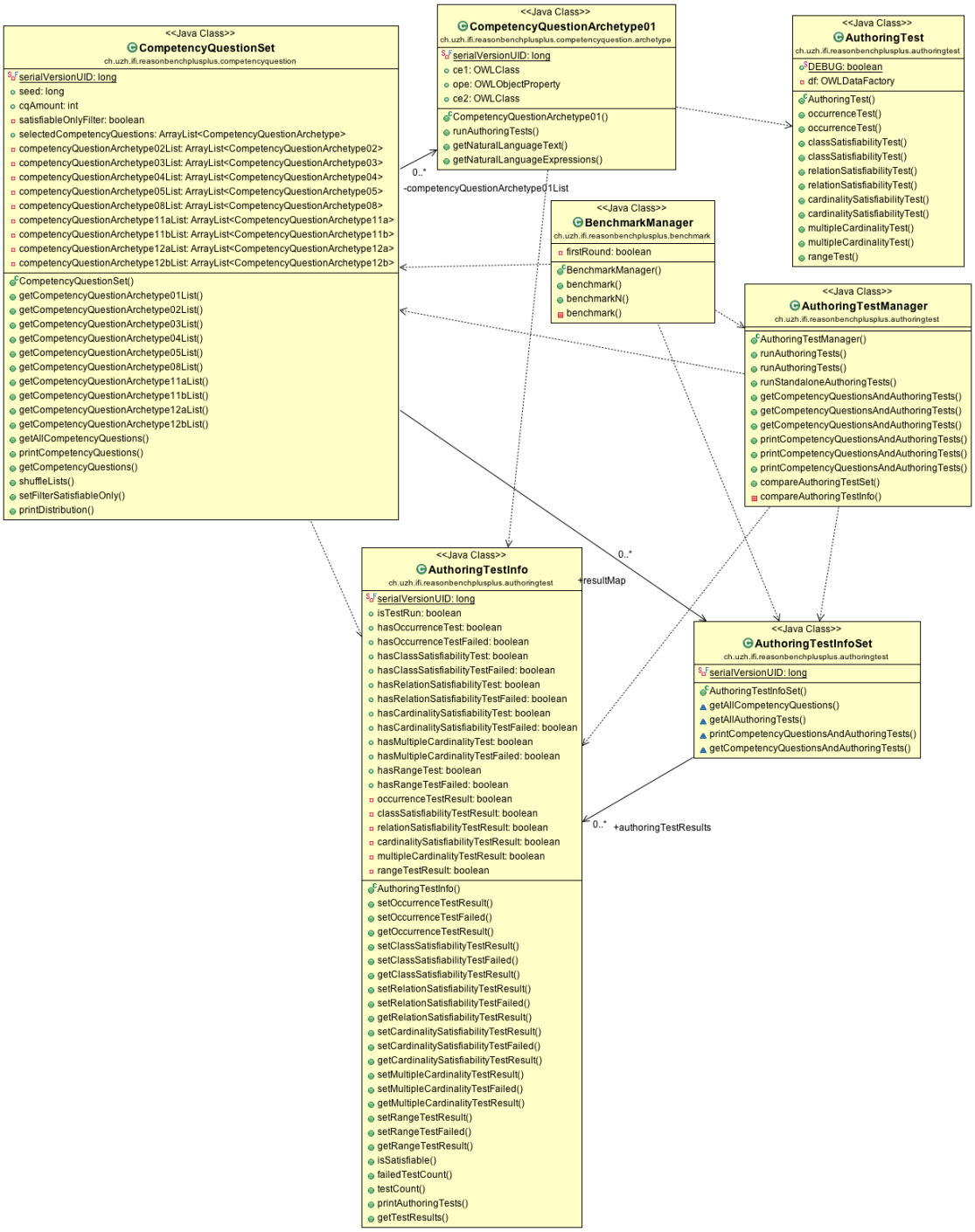

Figure 3.7: Class Diagram of `AuthoringTestManager` and its surrounding classes.

Every instance of `CompetencyQuestionArchetype` has knowledge of its applicable ATs and will call them itself if required. That way, from the generated CQs that are returned

by CompetencyQuestionManager, the ATs of every CQ can be called. AuthoringTestManager will return a map of reasoners to AuthoringTestInfoSet, in which the results of the run ATs are stored.

Figure 3.7 depicts all dependencies among AuthoringTestManager and its surrounding classes. It contains the class CompetencyQuestionArchetype01 as place-holder for all other Archetypes.

The AuthoringTestInfo class, used as data container, comprises fields for applicability of a specific AT, whether this AT has failed and a result column. All of these fields are booleans. Whether a test is applicable or not is only defined as soon as ATs are run - the instance of CompetencyQuestionArchetype will set the applicability of all tests. This allows us to have a more narrow implementation of the authoringtest package. It additionally contains convenience methods, such as isSatisfiable() or failedTestCount() that both evaluate their own results.

The authoringtest package is only influenced by one input-parameter:

- TEST_SELECTED_COMPETENCY_QUESTIONS_ONLY: This flag sets whether every time ATs should be run for all created CQs or only the ones selected to be used for the benchmark run. If set to FALSE, a big rise in computing time can be expected. *Default value:* TRUE.

3.3.6 Edit Package

The edit package includes all logic required to generate edits based on any given ontology. Its main components are the EditManager class that contains most logic, the EditInfo and EditInfoSet data containers that persist all generated edits, the changeoperation subpackage that consists of all implementations of the different Change Operations, the graph subpackage that incorporates a node-based representation of an ontolgoy, as well as two distributions, the ChangeOperationProbabilityDistribution and the EditSizeProbabilityDistribution. It is based on Ontology Evolution Mapping, as discussed in Sections 2.4 and 3.1.3.

EditManager provides methods to generate new edits (generateOntologyEdits(), using ChangeOperationEditGenerator), and multiple methods to iterate over these edits. One edit is defined as m Change Operations, where m is set by the EditSizeProbabilityDistribution. As the edits cannot be applied in any order but must be used in the exact order they were generated in, this manager class also provides methods to iterate over the edits. There exists a "forward-" and "backward"-mode, representing the direction the edits are iterated over. "Forward"-mode serves as the natural way edits would be applied to an ontology, for example by an ontology author. In the most extreme case, before applying the first edit, the ontology would be empty. After applying the last edit, the ontologies contents would be equal to its original state. "Backward"-mode is used to generate the edits, as well as to reset the ontology to its starting state as preparation for the next benchmark iteration.

As [Hartung et al., 2013] rely on graphs to detect their edits, there exists the subpackage graph that contains logic to generate a partial graph representation of any ontology.

`GraphGenerator` contains the logic to generate such a graph, and every class of an ontology exists as an instance of `GraphNode`. Nodes contain many additional properties, such as information on being a leaf, being an inner node, the children and parents of this node, level information and many more. Most data is not persisted but calculated if needed. These properties are used during the generation of any edit. Representing data- and object property relations among the nodes as well as complex class expressions is in the current implementation not possible.

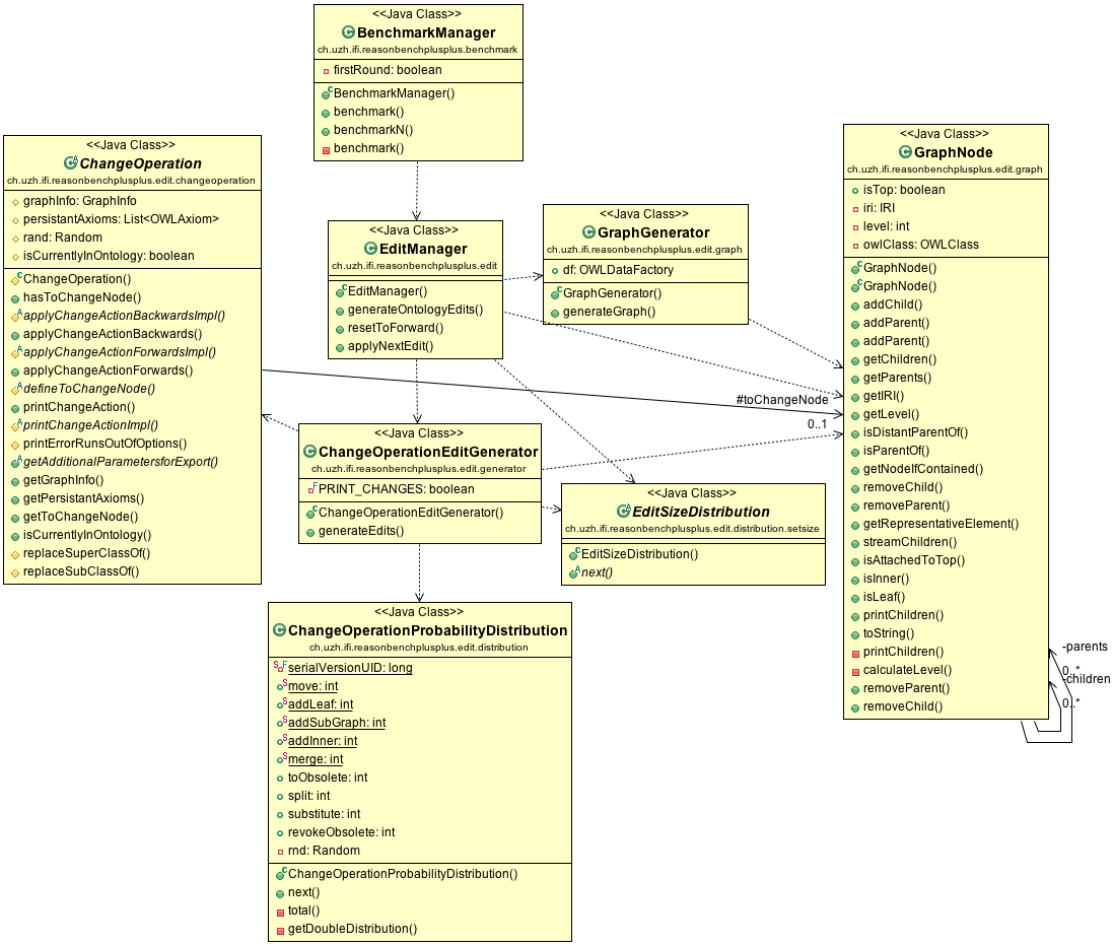

Figure 3.8: Class Diagram of `EditManager` and its surrounding classes.

During the generation of edits, `ChangeOperationEditGenerator` traverses the graph, looking for possible candidates for an edit. Among all candidates, a random instance is chosen. The Change Operation is then applied to that candidate, and the graph as well as the ontology are updated. Axioms residing in the ontology that are no longer required are persisted and stored in the `ChangeOperation` instance, such that they can be recalled.

The `ChangeOperationProbabilityDistribution` is based on Table 3.3 and returns

a set of Change Operations mimicking said distribution. If a certain Change Operation cannot be applied, i.e. due to lacking candidates, the distribution is queried again until a fitting Change Operation is found. The `EditSizeProbabilityDistribution` defines the amount of Change Operations included into an edit. It uses a negatively skewed exponential function, returning many small edit amounts but few big edits.

The data container classes `EditInfo` and `EditInfoSet` are returned by `EditManager` and contain all information on the generated edits. An instance of `EditInfo` is a set of Change Operations to be applied, while `EditInfoSet` contains all instances of `EditInfo`. The Change Operations included are always linked to the generated graph.

The following parameters directly influence this package:

- `CHANGE_OPERATION_SUB_GRAPH_MAX_DIFFERENCE_OUTERMOST_LEVEL`: When generating a Change Operation of type "subgraph", how high the difference in level of the candidate node inside the graph to the outermost leaf is allowed to be. 0 = all nodes allowed. Minimum level = 2. *Default value:* `2`.
- `CHANGE_OPERATION_MOVE_MAX_DIFFERENCE_OUTERMOST_LEVEL`: When generating a Change Operation of type "move", how high the difference in level of the candidate node inside the graph to the outermost leaf is allowed to be. 0 = all nodes allowed, minimum level = 1 (will just move leaves). *Default value:* `2`.
- `CHANGE_OPERATION_MOVE_MAX_LEVEL_UP`: When generating a Change Operation of type "move", how far up the algorithm is allowed to push the the node from its previous location. Minimum level = 2. *Default value:* `2`.
- `editAmount`: Sets the amount of edits that should be generated.

3.4 Benchmarking Process

The final implementation incorporates the following benchmarking process (see Figure 3.3):

1. The ontology is loaded. Required resources are allocated, being the `CompetencyQuestionManager`, the `OntologyEditManager` and the `AuthoringTestManager`.
2. The n CQs are generated. If required, the CQs have to pass their respective ATs. These results are used as baseline answers to compare the results of the reasoners.
3. Among all generated CQs, a specified amount is "selected" to be used in every benchmark run.
4. The m ontology edits are generated.
5. The reasoner instances are disposed and a new set of reasoners is loaded.

6. Repeated k times, the benchmarks are executed. Every benchmark consists of $m+1$ rounds, where m are the ontology edits and $+1$ is the ontology in its "empty" state. Before every round - in case the reasoners are using buffering - the buffer is flushed (selectibly this time is included in the time measurements or not). The edits are all applied to the ontology, yielding version $t_0 - m$, where t_0 is the ontologies original state. The reasoners individually run through all rounds of the benchmark after each other. After every benchmark iteration, the baseline results gathered during CQ generation are compared to the state of round m and differences are printed to the console. In addition, all reasoners are disposed and the Java Virtual Machine is tasked to run garbage collection.
   a) Round 1: The ontology is at state $t_0 - m$. In the first round, no edits get applied. The reasoner gets tasked to answer all ATs of the given CQs. This round is used as second baseline measurement. The measurements are stored.
   b) Round 2: The first edit is applied, we are at state $t_0 - (m + 1)$. The reasoner is again tasked to answer all ATs of the given CQs. The measurements are stored.
   c) Round $m - 1$: The second-to-last edit is applied, we are at state $t_0 - (m + (m - 1))$. The reasoner is again tasked to answer all ATs of the given CQs. The measurements are stored.
   d) Round m: The last edit is applied, we are now at state t_0. The reasoner is tasked one last time to answer all ATs of the given CQs. The measurements are stored.
7. Finally, the statistics package is used to display general metrics as means and standard deviation over the multiple iterations of the m rounds.

4

Results

This chapter discusses the results of the benchmarks of the multiple reasoners on different ontologies and parameter sets. It is split into the following sections: Section 4.1 contains information on the infrastructure, methodology, data and limitations of specific reasoners. Section 4.2 depicts the results of multiple benchmark runs.

4.1 Methodology and Infrastructure

This section discusses the setup of the benchmarks, the infrastructure used as well as different limitations of the reasoners. Section 4.1.1 presents the different hardware setups to run the benchmarks. Section 4.1.2 features the different ways RB++ can be parametrized to run a tailored benchmark. Section 4.1.3 depicts the different ontologies used for the benchmarks. Finally, Section 4.1.4 discusses limitations of some of the reasoners used and explains adjustments to the workflow required to circumvent these limitations.

4.1.1 Hardware

The benchmark is mostly CPU-frequency limited. Therefore, a high CPU clock speed accelerates the execution of the benchmark. All machines run Java 8 or higher, RB++ is started from the command line. System memory is no limiting factor (especially with state of the art machines), although RB++ requires a minimum of 4GB of RAM for JVM. Therefore the `-Xmx4g` flag is used at startup, as RB++ could run out of heap space if the amount of benchmark iterations is greater than approximately 45.

The benchmarks were run on two machines, depending on the workload and size of the used ontologies. The first machine, used for small ontologies, is a Macbook Air, 2015, with 8GB of RAM and an Intel Core i5 5250U CPU with 1.60GHz (boost up to 2.70GHz). The second machine is a Lenovo Yoga 2017 notebook with 16GB of RAM, an Intel Core i7 7600U CPU with 2.8GHz (with boost up to 3.9GHz) and was used for the bigger ontologies.

4.1.2 Configurations of RB++

The framework can be built with the command `mvn install` and is then packaged into a `.jar` file. The dependencies required are automatically packaged into the archive (`[...]-with-dependencies.jar`). RB++ can be run with the command depicted in Listing A.1 for a standard benchmark run and Listing A.2 for a benchmark of a certain edit-version.

Some input parameters, with an in-depth explanation of them in Section 3.3.1, can be set directly from the command-line. Others not listed here require adjustments in the code itself. Parameters accessible from the command-line include the following:

1. **arg1:** Path to the ontology file on the host computer.
2. **arg2:** Path to the output folder where RB++ will store the data generated throughout the benchmark.
3. **arg3:** Seed that is used for all activities requiring randomness.
4. **arg4:** Number of CQs to generate.
5. **arg5:** Number of edits to generate.
6. **arg6:** Number of iterations of the benchmark that should be executed.
7. **arg7:** Whether buffered reasoners should be used or not. `TRUE` results in the usage of buffered reasoners.
8. **arg8:** Whether flushing time of the reasoners should be included in the timings or not. `TRUE` includes the flushing time. This argument is only used if arg7 is set to `TRUE`.
9. **arg9:** Whether complete benchmark runs should be executed, or the benchmark should be only run at a certain versioning-point (with a specific amount of edits applied). Possible answers: `[runStandardBenchmark, runBenchmarkAtPosition]`
10. **arg10***: Optional, only required if arg9 is set to `runBenchmarkAtPosition`: Sets the edit position of the ontology.

The `runStandardBenchmark` flag starts a complete benchmark run, as specified in Section 3.4. During these runs, whether buffering is enabled, whether the changes are flushed and whether reinitialization is active all have direct influence on the result. Additionally, the results of the benchmarks vary if flushing is included into the runtimes or not.

The `runBenchmarkAtPosition` flag runs the benchmark at a chosen version of the ontology (with some specified amount of edits applied). As no changes are applied between the runs, flushing and buffering both have no influence. This flag helps to verify results of reasoners of a standard benchmark run.

4.1.3 Ontologies

The ontologies used in the benchmarks and during development are the following:

- **Pizza Ontology:** The pizza ontology `pizza.owl.xml`[1] was developed by the Universities of Manchester and Standford and is mainly used for tutorials and as example. It is however employing most concepts that are available in OWL 2 EL and was used for the development as well as testing of RB++. It contains subsumption, data- and object property relations and features few cardinalities. It is rather small (∼160KB)
- **Gene Ontology:** The gene ontology `gene_ontology_edit_2010-02-01.obo`[2] by [Harris et al., 2008] features a computational model of biological systems. It is available in older versions, which is why is was used to generate the Change Operation Distribution in Section 3.1.3. We are using a rather old version for the benchmarks. It features subsumptions and object property relations and is comparably large (∼17MB).
- **Univ-Bench:** The Univ-Bench ontology `univ-bench.owl.xml`[3] was developed by the Lehigh University for the LUBM benchmark (Lehigh University Benchmark) [Guo et al., 2005]. It contains subsumptions and object property relations. It is very small (∼14KB)

Using these ontologies, the reasoners (as listed in Section 3.2.2) have to answer all ATs entailed by the generated CQs as quickly as possible, following the benchmarking process as described in Section 3.4. The benchmarks are usually executed 50 times or more to ensure consistent results (if not noted differently).

4.1.4 Limitations and Unexpected Behaviour of Reasoners

During development and the final benchmark runs, some shortcomings of the tested reasoners were detected. The reasoners Hermit and Pellet have no difficulties managing all ontologies and parameter setups (with a few exceptions stated below).

First, ELK lacks the implementation of some OWL-API methods, such as `getObject-PropertyRanges()`. It therefore fails all "Range" ATs, as this method is required in that call.

Second, JFact can crash for some setup of ontology, seed and parameters. Table 4.1 shows the setup of the benchmark to break the reasoner. The gathered data suggests that enabling or disabling CQ Archetype 4 has direct influence on the behaviour of JFact. When CQ Archetype 4 is enabled, JFact throws errors during the benchmark iterations. The errors of JFact only appear in the first few rounds of every iteration of the benchmark (see Table A.2). It has to be mentioned that unproblematic seeds and ontologies do exist, where this behaviour of JFact cannot be observed. In addition, JFact

[1] cf. https://protege.stanford.edu/ontologies/pizza/pizza.owl
[2] cf. http://www.geneontology.org/
[3] cf. http://swat.cse.lehigh.edu/projects/lubm/

Table 4.1: Setup of RB++ used to review error sources of reasoners ELK and JFact. Ontology: Pizza Ontology. Seed: 1234567

Buffering enabled	CQ Archetype 4 enabled	Result JFact
Yes	No	Freezes
No	No	Freezes
Yes	Yes	Throws Errors
No	Yes	Throws Errors

sometimes freezes when buffering is disabled. We did not find a reason for this, however the error messages are connected to `NullpointerExceptions` from within the reasoners package. The reasoner usually freezes at code locations where it should answer ATs for a specific CQ, and this CQ is usually concerned with in the ontology higher leveled entities.

With these flaws in mind, the parameters of the following benchmarks are set such that these errors do not occur. The adaptations are always mentioned in their corresponding sections. The above stated conditions of JFact, as well as the failing Range ATs of ELK, are in the following subsections accepted as "Expected behaviour" of the reasoners. The expected behaviour of Hermit and Pellet and is errorless.

Third, when running benchmarks, it occurred that the reasoners would speed up over the duration of the benchmark - for the exactly same tasks. In an attempt to counteract this behaviour, we took the following counter-measures:

- **Reasoner-Rotation:** RB++ rotates the execution order of the reasoners at every new round of the benchmark.
- **Reinitialization:** RB++ disposes the reasoners and sets their variables to `NULL` before every benchmark iteration.
- **Garbage Collector:** RB++ calls the garbage collector after every reinitialization to ensure that the reasoners as disposed.
- **Flushing:** If one or many Change Operation(s) have been applied and buffering is enabled, RB++ forces the reasoner to flush this change before answering the Authoring Tests (and therefore it has to empty its buffer).

Figure 4.1 shows the results of running the `runBenchmarkAtEditPosition` benchmark with the "univ-bench" ontology, seed 1234567 for 400 iterations using reasoner Hermit. During the first 150 iterations, Hermit continuously speeds up in answering the same ATs over and over again. After these 150 iterations, the runtime stabilizes.

The differently coloured lines plotted display two parameters of the reasoner: the usage of buffering and re-initialization of a reasoner before a new run. The garbage collector and flushing (if applicable, only required if the reasoner is set to mode buffering) are always active.

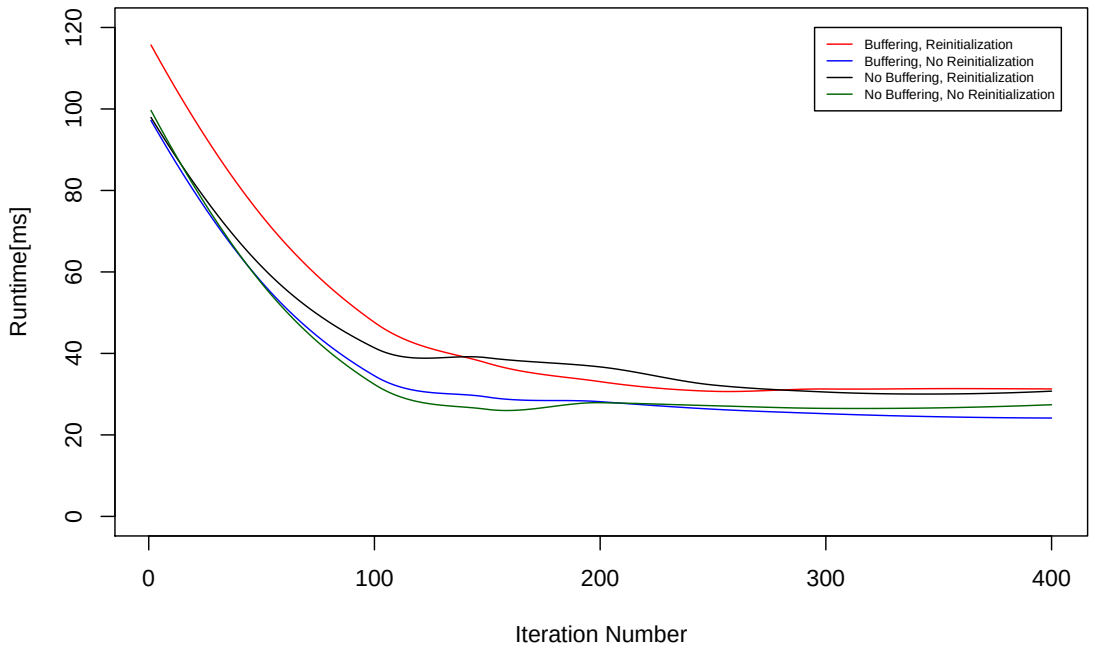

Figure 4.1: Influence of Reinitialization of a Reasoner on Runtimes. Ontology: univ-bench, Seed: 1234567, Reasoner: Hermit

The figure shows that re-initialization does effectively slow down the reasoners a little bit. However, over time and in general, the reasoners still speed up to the same speed as if they were not reinitialized, meaning that the reasoner cannot be effectively reset. This can be observed for all reasoners used in the benchmark (see Section A.2.1 for Figures of the same test of the other reasoners).

Re-initialization therefore is assumed to be having no influence onto runtimes after a certain point in any benchmark. Accordingly, the measurements taken at specific edit-versions of the ontology have their first 150 observations removed to ensure a fair comparison between benchmark data and verification data.

4.2 Benchmark Results

In this section, the results of the benchmarks are discussed. In a first endeavour, we execute a benchmark without any re-materialization times such that we are able to compare these results with point measurements at specific edit-versions of the ontology. This allows us to verify that the results RB++ produces are meaningful. In a second step, the benchmarking-measurements include the actual rebuild times. The difference between results with and without flushing show the actual time the reasoners require to re-materialize. Steps one and two are discussed in Section 4.2.1. In a third step, the consistency of the reasoners is analysed. One reasoner should return the same answer

concerning the satisfiability of a CQ as any other. Further, the RB++ environment is analysed to guarantee equivalent test conditions for all reasoners. Finally, the above results are compared to results of the benchmark runs involving the "gene ontology" (see Section 4.2.3).

4.2.1 Verification of Runtime Results

The first benchmark considers all four reasoners: Pellet, Hermit, ELK and JFact. The univ-bench ontology was used, and the benchmark was run on the Macbook Air machine (see hardware specifications in Section 4.1). Table 4.2 depicts the parametrization of this benchmark. Competency Question Archetype 4 is disabled, see reasons in Section 4.1.4. The goal of this benchmark is to compare runtimes of reasoners in a normal benchmark situation, as well as runtimes taken at specific edit-versions repeatedly to show that the measurements of RB++ are meaningful. The two benchmarks, the standard benchmark and the one at specific versions of the ontology should be congruent.

Table 4.2: Setup of univ-bench benchmark for all reasoners.

Paramter	Value
Benchmarked reasoners	Hermit, Pellet, JFact, ELK
Baseline reasoner	Pellet
Seed	1234567
Number of CQs to generate	100
Number of edits to generate	100
Number of benchmark iterations	50
Use buffered reasoner	`TRUE`
Include flushing time	`TRUE`

Table 4.3: Generated Change Operations for univ-bench benchmark.

Kind	Amount
AddLeaf	28
Move	20
AddSubGraph	3
AddInner	2
Merge	1
Total	54

The benchmark created 44 distinctive edits, with one additional edit being the empty ontology (with all classes removed). The 44 edits contained a total of 54 Change Operations. Table 4.3 shows the amount of types of Change Operations. The chosen distribution of Change Operations by RB++ tries to approximate the distribution of

Change Operations in the gene ontology (see Section 3.3). Comparing both distributions shows that there are many differences, mainly caused by the small size of the ontology, as there are not enough candidates for all types of Change Operations available. For example, AddLeaf is always applicable whereas for all other Change Operations some conditions need to be given (see Section 3.3.6).

The distribution of CQ Archetypes is depicted in Table 4.4. The univ-bench ontology does not contain any cardinality restrictions, therefore no CQs of Archetype 5, 11 and 12 can be generated. CQ Archetype 3 requires individuals which were not given, and Archetype 4 was disabled. The distribution of the CQ Archetypes is an approximation of the distribution found by [Ren et al., 2014], see Section 2.2.

Table 4.4: Generated CQ Archetypes for univ-bench benchmark.

Kind	Amount
Archetype 1	70
Archetype 2	4
Archetype 8	26
Total	100

Based on Table 4.4, the amount of AT's that every reasoner has to answer in every iteration of the benchmark is depicted in Table 4.5. As mentioned in Section 4.1.4, ELK is unable to answer any ATs of type Range.

Table 4.5: Amount of ATs to answer per reasoner per iteration in the univ-bench benchmark.

Kind	Amount
Occurrence	100
Class Satisfiability	100
Relation Satisfiability	74
Range	70
Total	344

Figure 4.2 depicts the results of this RB++ benchmark. While ELK failed all Range ATs (70) in every iteration of the benchmark, JFacT had some issues for all kinds of ATs. The other reasoners had no problem in executing all ATs. The results of ELK and JFacT therefore have to be treated with caution. An evaluation of the failed ATs per iteration can be found in Tables A.1 and A.2. Figure 4.2 has two scales, being the runtime (y1, to the left) and the amount of axioms changed per edit (y2, to the right). The amount of axioms changed represents the amount of work the reasoner has to process before answering the ATs. The processing of the changes is not included in the time measurements. A round of the benchmark (x-axis) equals the time the reasoner uses to answer all ATs for this edit-version of the ontology. An iteration of the benchmark

includes all rounds below, the runtimes displayed are the mean for that round over all iterations of the benchmark.

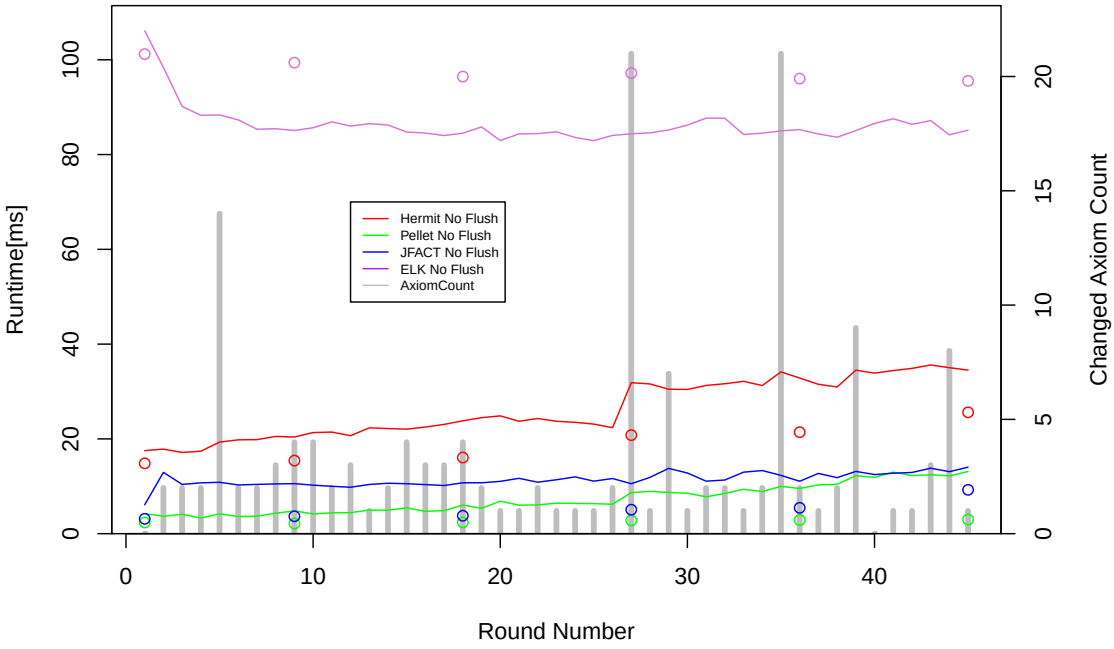

Figure 4.2: Results RB++ benchmark with ontology univ-bench, 45 edits and 100 CQs. Dots: Repeated measurements at one edit version of the ontology. Coloured lines: RB++ standard benchmark. Grey lines: Amount of axioms added / removed from the ontology for this edit-version. Measurements equal reasoning time without flushing.

At round 1, the ontology is empty. It can be assumed that the workload to answer all ATs must be the smallest, as the reasoner has the fewest axioms to check the stated queries against. The results reflect this assumption: Generally, the more the time required to process the ATs rises, the more content is present in the ontology. The amount of axioms that are added or removed from the ontology also have influence on the runtimes, best visible in reasoners Hermit, Pellet and JFact, where Pellet has some outliers that do not reflect the amount of changes. The Reasoner ELK behaves unexpectedly, its runtimes appear to be linear throughout the benchmark. The quickest reasoner is Pellet, followed by JFact and Hermit. ELK is by distance the slowest of the bunch.

The slowness of ELK could be explained by its faulty ATs, where the reasoner throws an error every time it cannot answer a specific AT. We do not know what ELK does internally to recover from that situation, but most possibly the thrown errors lead to extra work it has to go through before answering the next AT. This however does not match the observations on JFact, which also failed some tests but kept performing pretty well.

The plotted points in the figure represent runs where the reasoner did not have to

apply incremental changes to its materialization. What we can see is that these times tend to be lower, drifting slightly apart throughout one iteration of the benchmark, for Hermit at least. For ELK the tend do be a little higher. The coinciding benchmark measurement and point measurements speak for our approach - we are able to see that the actual difference is rather small and follows the standard benchmark runtimes.

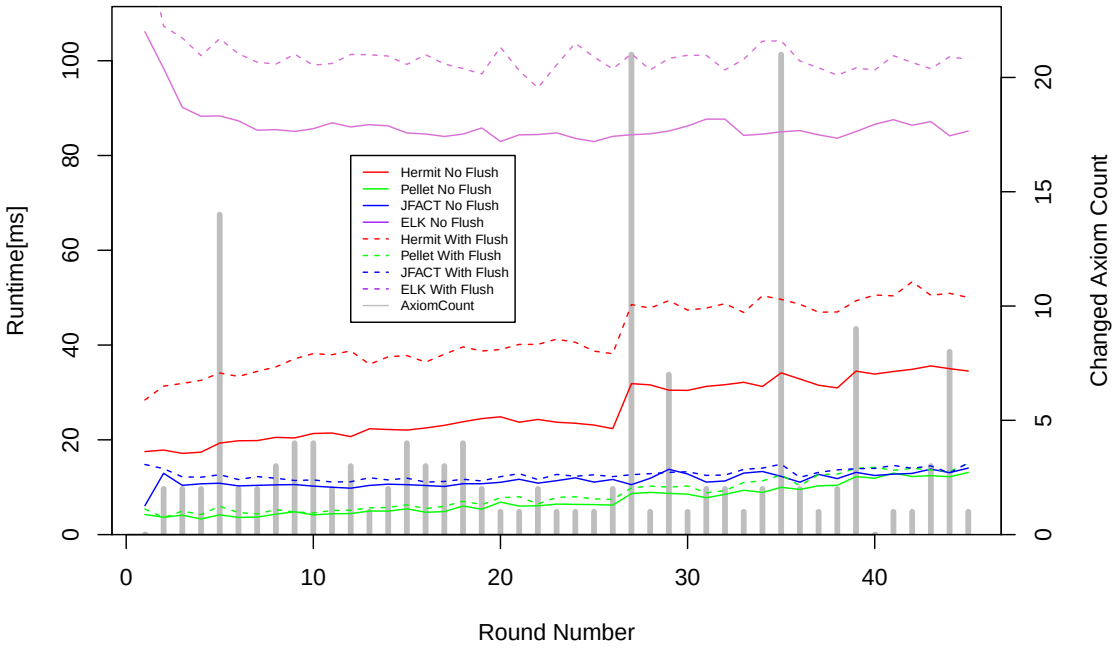

Figure 4.3: Results RB++ benchmark with ontology univ-bench, 45 edits and 100 CQs. Dots: Repeated measurements at one edit version of the ontology. Coloured lines: RB++ standard benchmark. Grey lines: Amount of axioms added / removed from the ontology for this edit-version. Measurements equal both reasoning time without flushing / with flushing.

In a next step, the same benchmark is repeated, but the flushing of the reasoner is included in the measurements. The other parameters of the benchmark stay the same. The results are displayed in Figures 4.3 and 4.4. The former displays the results of the previous benchmark, as well as the results of the second benchmark. The difference between the lines of the same colour families represent the time required to re-materialize the ontology after an edit has been applied.

We can observe that the flushing time of Pellet and JFact is low, while Hermit and ELK require a significant amount of time to process the changes. The absolute difference is depicted in the Figure 4.4. Again, reasoners Hermit and ELK require more time to flush the changes. Based on the amount of axioms changed, there is no clear tendency towards higher flushing time when more axioms are changed in the ontology.

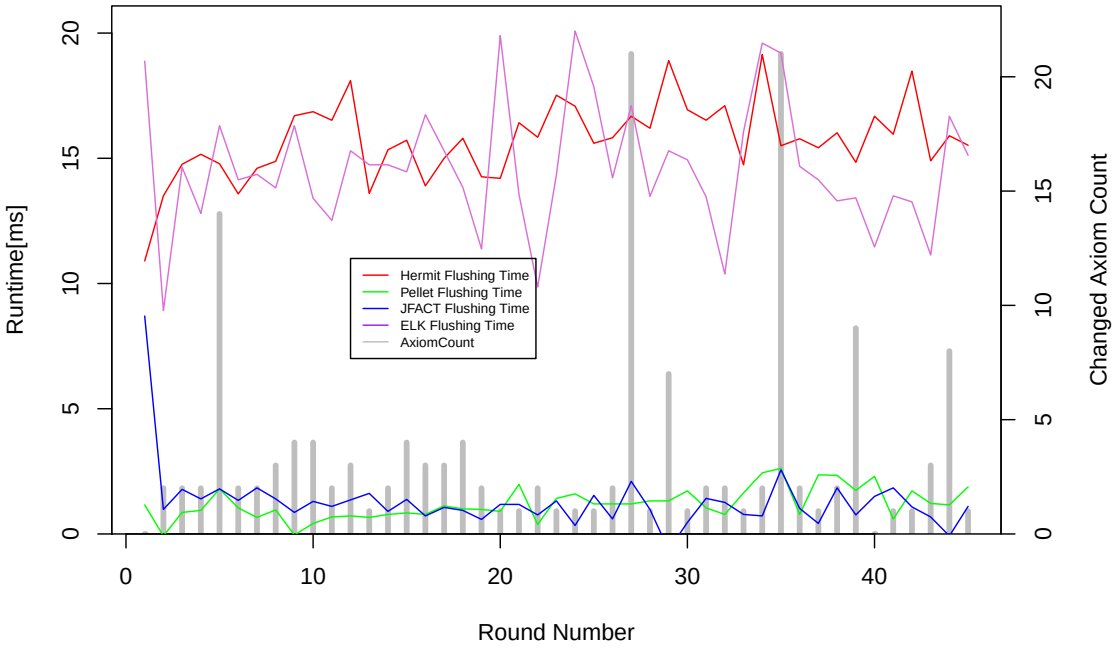

Figure 4.4: Differences between runtimes in 4.3 with flush / without flush on ontology univ-bench, 45 edits and 100 CQs. Coloured lines: RB++ Standard Benchmarks. Grey lines: Amount of Axioms added / removed from the ontology for this edit-version.

4.2.2 Verification of Reasoner Consistency

Additional to the verification of runtimes, it is also necessary to check the consistency of the reasoners throughout the benchmark as well as the environment they are used in. To perform the checks below, we use the results of the previous section. There are multiple aspects to take into account:

- Assessment that the reasoners are all used in a fair, equal environment for the tasks they have to process.
- Assessment that the results of the reasoners among each other are equal or fit the results being expected by their expressivity.
- Assessment that the results of one reasoner compared over multiple iterations are the same.

The environment consists of the tasks, the data and the changes applied to that data. The tasks (CQs) are copied from iteration to iteration and therefore are never subject to any change. This is the same for the changes that are applied between the ontologies. To check the consistency of the data, the ontology was exported from the benchmark after every round and for every reasoner. A comparison of ontologies for the different

reasoners of the same rounds showed that there were no differences among them. This also verifies the check above on the equality of changes for all reasoners - if the changes would be different between the reasoners, the ontologies exported would show differences. We can therefore attest that the environment is the same for all reasoners.

Table 4.6: Comparison of answers to ATs between Hermit and Pellet. (Occ. = Occurrence Test, Cls. Sat. = fiability Test, Rel. Sat. = Relation Satisfiability Test, Rg. = Range, Sat. = Satisfiability)

It#	Rd#	Reasoner	Ar#	Natural Language Text	Occ.	Cls. Sat.	Rel. Sat.	Rg.	Sat.
2	20	Hermit	1	Which SystemsStaff worksFor Program?	FALSE	TRUE	TRUE	TRUE	FALSE
1	20	Pellet	1	Which SystemsStaff worksFor Program?	FALSE	FALSE	FALSE	FALSE	FALSE
2	20	Hermit	1	Which Professor headOf Department?	FALSE	TRUE	TRUE	TRUE	FALSE
1	20	Pellet	1	Which Professor headOf Department?	FALSE	FALSE	FALSE	FALSE	FALSE
2	20	Hermit	8	Which are University?	FALSE	TRUE	FALSE	FALSE	FALSE
1	20	Pellet	8	Which are University?	FALSE	FALSE	FALSE	FALSE	FALSE
2	20	Hermit	1	Which Professor headOf University?	FALSE	TRUE	TRUE	TRUE	FALSE
1	20	Pellet	1	Which Professor headOf University?	FALSE	FALSE	FALSE	FALSE	FALSE
2	20	Hermit	1	Which FullProfessor memberOf University?	FALSE	TRUE	TRUE	TRUE	FALSE
1	20	Pellet	1	Which FullProfessor memberOf University?	FALSE	FALSE	FALSE	FALSE	FALSE
2	20	Hermit	1	Which Department member UndergraduateStudent?	FALSE	TRUE	TRUE	TRUE	FALSE
1	20	Pellet	1	Which Department member UndergraduateStudent?	FALSE	FALSE	FALSE	TRUE	FALSE

To verify that the reasoners return the same results for the same ATs requires us to compare these results among each other. More importantly, the reasoners must return the same results over multiple iterations. To examine this, the results of the above benchmark runs were compared. As already stated, JFact and ELK failed some of the tests (see Tables A.1 and A.2), making a fair comparison between all reasoners and the two infeasible.

To compare the two remaining reasoners, Pellet and Hermit, their results were compared, either with themselves or among each other. Both reasoners return the same results concerning satisfiability of any CQ. Satisfiability of a CQ is defined as being true if all ATs pass, or as false if at least one AT fails. It occurs that if the Occurrence AT fails, Hermit will still allow other tests to pass, while Pellet will return false for all other tests if the entities in question are not part of the ontology. This behaviour is depicted in Table 4.6. The reasoners return the same result concerning satisfiability of the whole CQ and their entailed ATs, but the ATs themselves resulted differently. Cardinality Satisfiability and Multiple Cardinality tests were not added to the table, as the Archetypes shown do not entail such tests.

Whether reasoners themselves stay consistent throughout multiple iterations was veri-

fied by comparing the results of all iterations. All reasoners stayed consistent throughout the benchmark, no changes in answers could be observed.

4.2.3 Results of the Gene Ontology

The second benchmark was executed with the reasoners Pellet, Hermit and ELK. As input, the gene ontology was used. JFact was excluded from this benchmark due to time constraints, as it is comparably slower in handling the gene ontology (∼20 times slower then the other reasoners). The benchmark was executed on the Lenovo Yoga machine (see hardware specifications in Section 4.1). The parametrization of this benchmark is the same as for the univ-bench benchmark and depicted in Table 4.2. Competency Question Archetype 4 is enabled. Again, the goal of this experiment is the comparison of runtimes of reasoners in a normal benchmark situation and the runtimes taken at specific edit-versions repeatedly.

Table 4.7: Generated Change Operations for the gene ontology benchmark.

Kind	Amount
Move	62
AddLeaf	25
AddSubGraph	23
AddInner	13
Merge	3
Total	126

RB++ created 100 distinctive edits with one additional edit being the "empty" ontology (in this case the ontology is far from empty due to its size). In these 100 edits, a total of 126 Change Operations were applied. Table 4.7 shows the amount of types of Change Operations. In this second benchmark, the distribution of Change Operations generated by RB++ is closer to the distribution by [Pernischova, 2018] (see Table 3.3) compared to the previous univ-bench benchmark .

Table 4.8: Generated CQ Archetypes for the gene ontology benchmark.

Kind	Amount
Archetype 1	72
Archetype 4	1
Archetype 8	27
Total	100

Table 4.8 displays the distribution of CQ Archetypes. The gene ontology does not contain any cardinalities, individuals and data property relations. Therefore, CQs of Archetypes 2, 3, 5, 11 and 12 are not generated. Again, the distribution of CQ Archetypes approximates the distribution by [Ren et al., 2014], see Section 2.2.

Table 4.9: Amount of ATs to answer per reasoner per iteration in the univ-bench benchmark.

Kind	Amount
Occurrence	100
Class Satisfiability	100
Relation Satisfiability	72
Range	72
Total	344

Finally, Table 4.9 shows the amount of AT's that every reasoner has to answer in every iteration of the benchmark. Similar to the univ-bench benchmark, ELK was unable to answer the Range ATs.

Figure 4.5 shows the results of this second benchmark run of RB++. The benchmark run consisted out of two sub-runs, where one of them includes the time required to flush the changes, while the other one does not. ELK, as mentioned earlier, failed all Range ATs. Again, the two y scales are used, one being the actual runtime, the other one being the count of the changed axioms in that round.

As the gene ontology is fairly big, the ontology is far from empty at round number 1. The total of axioms is 344262 before applying any Change Operations and reduces to 309770 axioms after applying all Change Operations, a decrease of 10%. Comparing that to the runtimes of Hermit, being 2800ms in round 101 and 2760ms in round 1 (a decrease of 1.5%) shows that the change in axiom count does influence the runtime by the same magnitude.

Overall, Pellet is the slowest of all competing reasoners and ELK is the fastest, which is different in the univ-bench benchmark where ELK was by far the slowest. A reason might be that ELK is the only reasoner of the contestants that makes use of multi-threading.

JFact, not being part of this benchmark, is very slow when using the gene ontology. Compared to Hermit's results, JFact was approximately 20 times slower than Hermit. The difference to the results in the univ-bench benchmark, where JFact was one of the fastest reasoners, cannot be explained by now. Again, as in Section 4.2.1, we have to mention that the results of JFact in the univ-bench benchmark as well as the observations from this run have to be treated with caution.

Over the duration of the benchmark run, the runtimes of the reasoners stayed more consistent compared to the univ-bench runs, only minor speed-ups can be seen. Pellet shows some spikes in the results both with and without flushing, with the spikes being larger for the runs with flushing. The reason for this is not known. Finally, the times taken at specific edit-versions of the ontology show reasonable results for ELK, where they take the same amount of time as in the normal benchmark run. Hermit and Pellet are able to respond within few milliseconds to the queries (Hermit: ∼35ms, Pellet: ∼3ms). We do not know why the reasoners are that quick in answering the same queries over and over again. We hypothesize that some form of caching comes into play here.

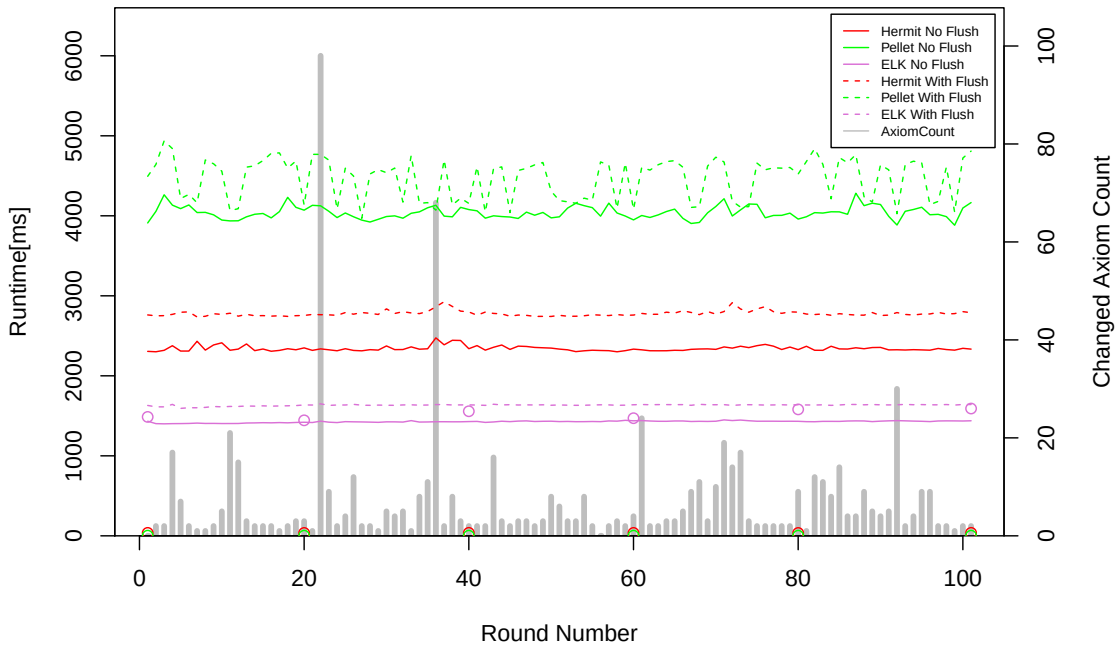

Figure 4.5: Results RB++ benchmark with the gene ontology, 100 edits and 100 CQs. Dots: Repeated measurements at one edit version of the ontology. Coloured lines: RB++ standard benchmark. Grey lines: Amount of axioms added / removed from the ontology for this edit-version. Measurements equal reasoning time without flushing / with flushing.

5

Limitations

This chapter describes limitations of RB++, existing due to time-restrictions and other causes.

Starting of with the theory of CQOA by [Ren et al., 2014]: Some Archetypes were not implemented due to time limitations, low usefulness or technical constraints. Summarizing the content of Section 3.1.1: Archetype 7 and 9 both require some form of logic (temporal or spatial restrictions) that are not included as a standard in every ontology. Archetype 6 would on one hand be an interesting addition to the implementation due to its complication factor and entailed workload, but was omitted due to time constraints and replaced by instances of Archetypes 1 and 5. Finally, all Sub-Archetypes listed in Table 2.2 are minor variations of their parent, Archetype 1. An implementation would not have made sense out of a benchmarking perspective, as the resulting ATs for all CQs would have been the same.

We also found that the Comparative Cardinality AT (see Sections 2.3, 3.1.2) has no application among the CQ Archetypes, therefore it is never used. Both the Meta-Instance and the Range ATs are not applicable to OWL API based frameworks, as they would always result in a passed test due the always-correct mapping of entities during the CQ generation. OWL and the frameworks logic are never going to be in a state where these tests could actually fail.

In addition, the current implementation of Ontology Evolution Mapping by [Hartung et al., 2013] in RB++ does not provide any changes on object- and data-property relations. RB++ therefore only removes class structures from ontologies until they are empty, but all property relations remain. This has especially implications on reasoners that are unable to incrementally add or remove property relations, which would require them to fully re-materialize. However, as there are not many ontologies actually including these relations by today, their usefulness for a real-life application benchmark can at least currently be questioned. The Change Operations are in addition restricted by RB++ lacking domain knowledge of the used ontology. A Change Operation requiring the addition of new, never seen entities into the ontology is currently infeasible, as there are risks that the addition of for example new subclasses could lead to irrational classes inside the ontology. Therefore RB++ is constrained to using a complete ontology and moving / removing parts of that ontology to generate edits. Generating an ontology from nothing is not possible.

On the topic of chosen ontologies: There exist only few publicly available, real-life ontologies that make use of all concepts of OWL 2, as for example the family ontology[1]. Therefore, the generated CQ's, AT's and edits are constrained from the input data on and we are currently not able to use their full potential. The missing concepts include data property relations as well as cardinalities and individuals. Depending on the use-case of the benchmark, be it spotting the best reasoner for a certain application, the availability of the above concepts might not even be a necessity, as their usage in an existing ontology for a certain application defines the parameters for that benchmark.

Depending on the chosen ontologies, especially large ones (such as the newest version of the gene ontology[2]), reasoners had problems loading these ontologies. The overall selection of tested ontologies was constrained by this fact, as for example an older version of the gene ontology was used instead of a new one.

Also, the usage of OWL API as baseline framework can constrain the applications of RB++, but due to OWLs standardization and worldwide acceptance we would interpret this issue as a rather small one.

Finally, RB++ is currently only limited by CPU-speed and RAM. Most of the process runs single threaded, but the reasoners (for example `ELK`) are using multi-threading for higher performance. The biggest bottleneck is the CPU-clock speed, especially for reasoners that are not using multi-threading.

[1] cf. www.cs.man.ac.uk/∼stevensr/ontology/family.rdf.owl
[2] cf. http://www.geneontology.org/page/download-ontology

6

Future Work

In this Section, potential improvements for the RB++ framework are discussed. Most of the pointed out improvements are mentioned in Section 5.

The implementation of the missing Archetypes (see Section 3.1.1 and Chapter 5) would be an addition to RB++ that would further enhance the overall output of queries. We could envision such an extension of the benchmarking framework, making further use of the idea seen in Archetype 6. Encapsulating more than one CQ into a complex CQ would reflect some much more real life questions. As the back-end with the ATs to counter-check on these CQs exists, an implementation would be feasible and straight forward.

The distributions used, as described in Section 3.3.1, could be fine-tuned to match specific ontologies better. The CQ-distribution stems from empirical research of humans editing different ontologies, while the Change Operation distribution was created by using different versions of the gene ontology. In both cases, the data for those distributions could be empirically gathered based on the specific ontology used in the benchmarks. This however requires a versioned ontology, which is a rather rare occurrence.

The range of Change Operations could be extended by introducing differences in relation properties. The current implementation just takes into account the subsumption of classes, whereas this would allow much more complex changes to be applied to the ontology.

The evaluation of the capability of reasoners to change their materializations incrementally needs further investigation. We would propose a new benchmark process that investigates the change in processing time versus an ontology that has changes applied from 0% to 100%. This would allow the comparison of the different mechanisms the reasoners use for their incremental updates.

7

Conclusions

This chapter concludes the findings and implementation of this thesis.

With ReasonBench++, we are able to present a new approach for benchmarking incremental reasoning systems. Current benchmarking approaches contain queries and data as input parameters, often created by hand. ReasonBench++ contains logic that automates the generation of queries based on an input ontology. Further, it is able to change the input ontology, creating a sequence of edits that piece by piece remove or add axioms from / to the ontology. By using leading and steady concepts such as Competency Question-driven Ontology Authoring and Ontology Evolution Mapping, we can assume that our resulting queries and edits are close to real-life and meaningful. While CQOA and its theory of mapping sentence structures (Archetypes) to Authoring Tests (ATs) is used to create queries based on said Archetype-Templates, Ontology Evolution Mapping is used to analyse changes between versions of the same ontologies, resulting in a distribution of Change Operations that reflect the usual work of an ontology author. The theory of CQOA was additionally extended with a mapping of CQ Archetype to Authoring Tests, which did not exist before.

RB++'s biggest advantage compared to other benchmarks is its capability to generate queries and edits for any ontology, allowing it to be used for many different application scenarios. While other approaches usually use ontologies and queries developed and created for that specific purpose, we are able to return a complete benchmark without any labour required but supplying an ontology. With very little work, the distribution of either CQ Archetypes or Change Operations can be adjusted. This allows very granular control of the application case. The OWL API also grants the usage of any reasoner that implements that interface - which is the most commonly used API for such applications today. By providing a documented, clearly structured code-base, built with a possible extension and reuse in mind, the code can in addition be easily split and recycled for any other project that requires a mechanism to create queries or ontology edits.

The results of the benchmarks depict that our implemented benchmarking process works. By showing that RB++ creates stable environments and by comparing the benchmark results to point measurements at specific versions of the ontology, we can assume that our benchmark runs stable and fairly compares a set of reasoners. We ran benchmarks with both the univ-bench ontology from the LUBM-Benchmark as well as the gene ontology, using the four different reasoners - Hermit, Pellet, JFacT and ELK

as test subjects. We were also able to present findings concerning the competing reasoners, where some were more performant than others and some have shown difficulties in applying changes and immediately answering to a set of queries.

In future, we would like to further enhance the benchmarking process, to be able to investigate the capability of the reasoners of actual incremental reasoning. While we could find differences in speed, the different methods used for incremental reasoning are not yet examined sufficiently. It would in addition be worthwhile to run benchmarks with ontologies that use all concepts of OWL 2, as there are no such ontologies available publicly.

References

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

[Bock et al., 2008] Bock, J., Haase, P., Ji, Q., and Volz, R. (2008). Benchmarking owl reasoners.

[Chaussecourte et al., 2013] Chaussecourte, P., Glimm, B., Horrocks, I., Motik, B., and Pierre, L. (2013). The energy management adviser at edf. In *The Semantic Web - ISWC 2013*, volume 12, pages 49–64. 12th International Semantic Web Conference, Alani, H., Kagal, L., Fokoue, A., Groth, P., Biemann, C., Parreira, J.X., Aroyo, L., Noy, N., Welty, C., Janowicz, K. (Eds.).

[Colombetti, 2017] Colombetti, M. (2017). Lecture notes of the knowledge engineering course 2017. Technical report, Politecnico di Milano, Department of Electronics, Information, and Bioengineering.

[Dennis et al., 2017] Dennis, M., van Deemter, K., Dell'Aglio, D., and Pan, J. Z. (2017). Computing authoring tests from competency questions: Experimental validation. In d'Amato, C., Fernandez, M., Tamma, V., Lecue, F., Cudré-Mauroux, P., Sequeda, J., Lange, C., and Heflin, J., editors, *The Semantic Web - ISWC 2017*, pages 243 – 259, Cham. Springer International Publishing.

[Dentler et al., 2011] Dentler, K., Cornet, R., ten Teije, A., and de Keizer, N. (2011). Comparison of reasoners for large ontologies in the owl 2 el profile. *Semant. web*, 2(2):71–87.

[Glimm et al., 2014] Glimm, B., Horrocks, I., Motik, B., Stoilos, G., and Wang, Z. (2014). Hermit: An owl 2 reasoner. *Journal of Automated Reasoning*, 53(3):245–269.

[Guo et al., 2005] Guo, Y., Pan, Z., and Heflin, J. (2005). Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158 – 182. Selcted Papers from the International Semantic Web Conference, 2004.

[Harris et al., 2008] Harris, M., Deegan, J., Lomax, J., Ashburner, M., Tweedie, S., Carbon, S., Lewis, S., Mungall, C., Day-Richter, J., Eilbeck, K., Blake, J., Bult, C., Diehl, A., Dolan, M., Drabkin, H., Eppig, J., Hill, D., Ni, L., Ringwald, M., and Ontology Consortium, G. (2008). The gene ontology project in 2008. 36.

[Hartung et al., 2013] Hartung, M., Groß, A., and Rahm, E. (2013). Conto–diff: generation of complex evolution mappings for life science ontologies. *Journal of Biomedical Informatics*, 46(1):15 – 32.

[Horridge and Bechhofer, 2011] Horridge, M. and Bechhofer, S. (2011). The owl api: A java api for owl ontologies. *Semant. web*, 2(1):11–21.

[Kazakov et al., 2012] Kazakov, Y., Krötzsch, M., and Simančík, F. (2012). Elk: A reasoner for owl el ontologies. Technical report, University of Oxford, Ulm University.

[Motik et al., 2015] Motik, B., Nenov, Y., Piro, R., and Horrocks, I. (2015). Incremental update of datalog materialisation: The backward/forward algorithm. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pages 1560–1568. AAAI Press.

[Palmer and Felsing, 2001] Palmer, S. R. and Felsing, M. (2001). *A Practical Guide to Feature-Driven Development*. Pearson Education, 1st edition.

[Patel-Schneider et al., 2004] Patel-Schneider, P. F., Hayes, P., and Horrocks, I. (2004). Owl web ontology language semantics and abstract syntax. *W3C Recommendation*.

[Pernischova, 2018] Pernischova, R. (2018). Impact of changes on operations over knowledge graphs. Master's thesis, University of Zurich.

[Ren et al., 2016] Ren, Y., Pan, J. Z., Guclu, I., and Kollingbaum, M. (2016). A combined approach to incremental reasoning for el ontologies. In Ortiz, M. and Schlobach, S., editors, *Web Reasoning and Rule Systems*, pages 167–183, Cham. Springer International Publishing.

[Ren et al., 2014] Ren, Y., Parvizi, A., Mellish, C., Pan, J. Z., van Deemter, K., and Stevens, R. (2014). Towards competency question-driven ontology authoring. *European Semantic Web Conference*, pages 752–767.

[Sirin et al., 2007] Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A., and Katz, Y. (2007). Pellet: A practical owl-dl reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):51 – 53. Software Engineering and the Semantic Web.

[Tsarkov, 2014] Tsarkov, D. (2014). Incremental and persistent reasoning in fact++. Technical report, The University of Manchester.

[Tsarkov and Horrocks, 2006] Tsarkov, D. and Horrocks, I. (2006). Fact++ description logic reasoner: System description. In Furbach, U. and Shankar, N., editors, *Automated Reasoning*, pages 292–297, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Uschold and Gruninger, 1996] Uschold, M. and Gruninger, M. (1996). Ontologies: principles, methods and applications. *The Knowledge Engineering Review*, 11(2):93–136.

[Volz et al., 2005] Volz, R., Staab, S., and Motik, B. (2005). *Incrementally Maintaining Materializations of Ontologies Stored in Logic Databases*, pages 1–34. Springer Berlin Heidelberg, Berlin, Heidelberg.

[W3C OWL Working Group, 2009] W3C OWL Working Group (2009). Owl 2 web ontology language document overview.

A

Appendix

A.1 Terminal commands for RB++

```
java -Xmx4g -jar reasonbenchplusplus-0.1.6.1-jar-with-
    dependencies.jar <PATH_TO_ONTOLOGY_FILE> <
    PATH_TO_EXPORT_FOLDER> 1234567 100 100 50 true true
    runStandardBenchmark
```

Listing A.1: Startup command example for a RB++ standard run.

```
java -Xmx4g -jar reasonbenchplusplus-0.1.6.1-jar-with-
    dependencies.jar <PATH_TO_ONTOLOGY_FILE> <
    PATH_TO_EXPORT_FOLDER> 1234567 100 100 50 true true
    runStandardBenchmark
```

Listing A.2: Startup command example for a RB++ run at a specific edit position.

A.2 Results

A.2.1 Additional Figures of Reasoner Behaviour over Time

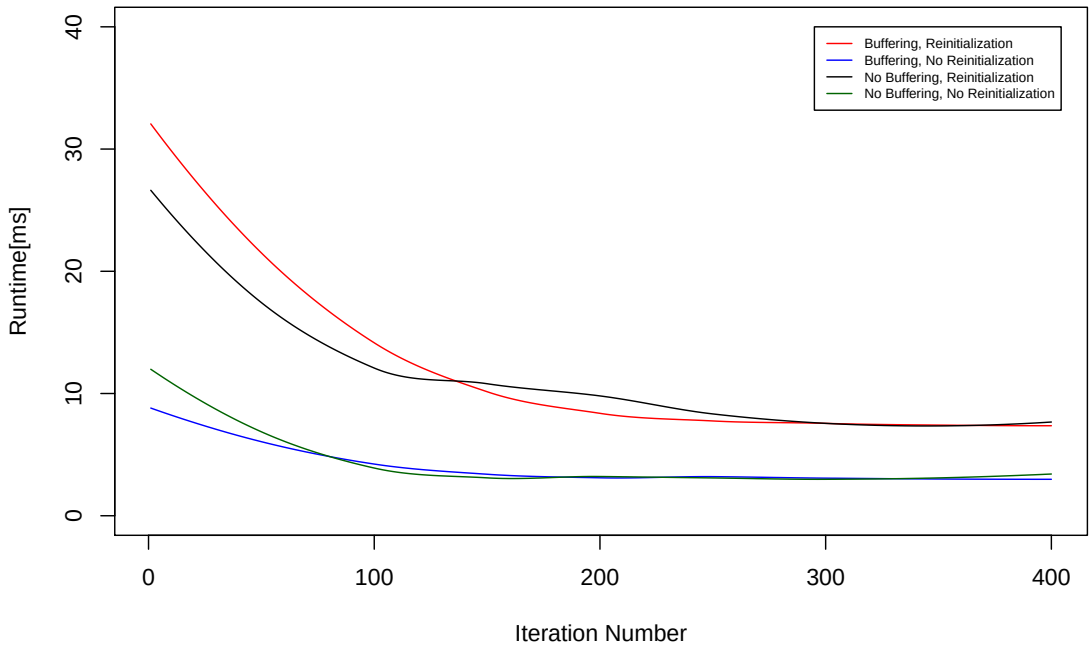

Figure A.1: Influence of Reinitialization of a Reasoner on Runtimes. Ontology: univ-bench, Seed: 1234567, Reasoner: Pellet

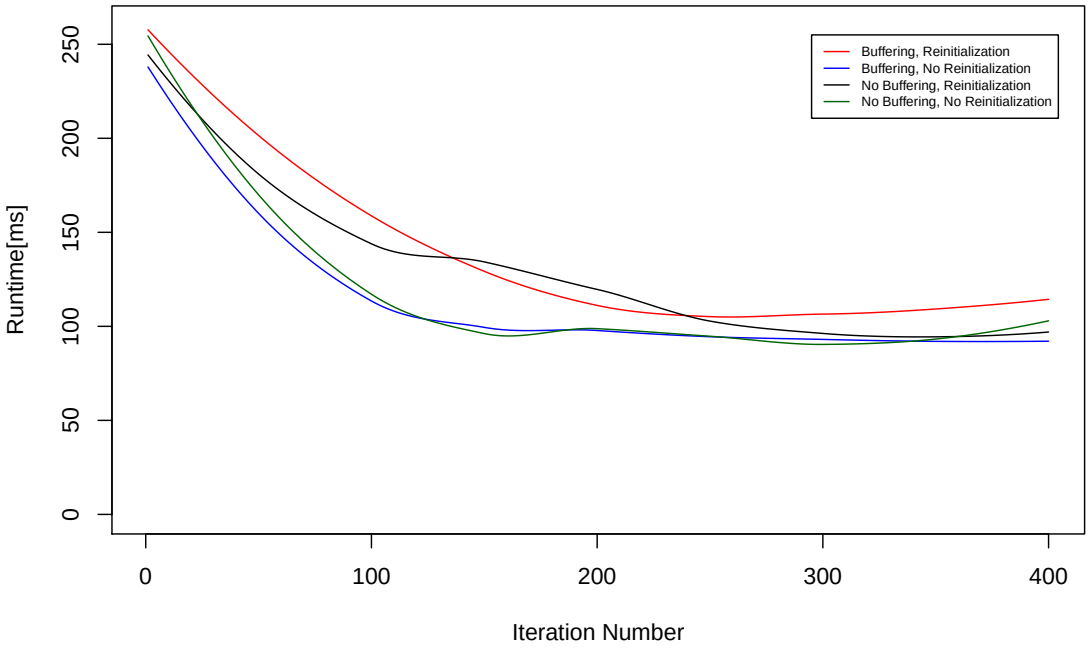

Figure A.2: Influence of Reinitialization of a Reasoner on Runtimes. Ontology: univ-bench, Seed: 1234567, Reasoner: ELK

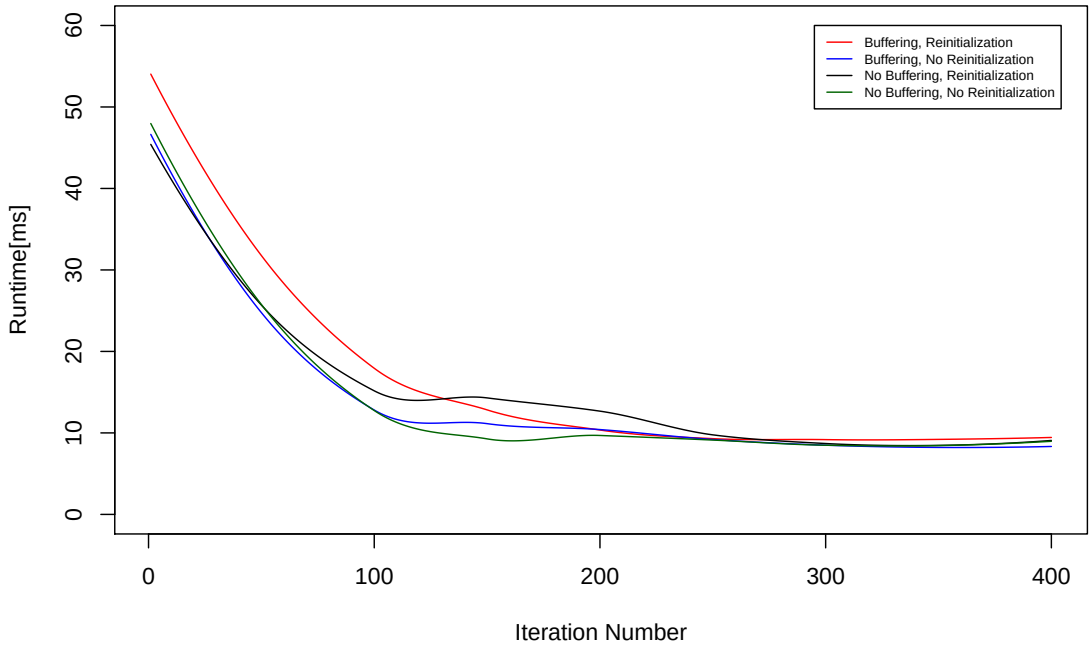

Figure A.3: Influence of Reinitialization of a Reasoner on Runtimes. Ontology: univ-bench, Seed: 1234567, Reasoner: JFact

A.2.2 Failed Authoring Tests per Reasoner

Table A.1: Failed ATs by reasoner ELK in iteration 1 of the univ-bench standard benchmark.

Round	Occurrence	Class Satisfiability	Relation Satisfiability	Cardinality Satisfiabiliy	Multiple Cardinality	Range
1	0	0	0	0	0	70
2	0	0	0	0	0	70
3	0	0	0	0	0	70
4	0	0	0	0	0	70
5	0	0	0	0	0	70
6	0	0	0	0	0	70
7	0	0	0	0	0	70
8	0	0	0	0	0	70
9	0	0	0	0	0	70
10	0	0	0	0	0	70
11	0	0	0	0	0	70
12	0	0	0	0	0	70
13	0	0	0	0	0	70
14	0	0	0	0	0	70
15	0	0	0	0	0	70
16	0	0	0	0	0	70
17	0	0	0	0	0	70
18	0	0	0	0	0	70
19	0	0	0	0	0	70
20	0	0	0	0	0	70
21	0	0	0	0	0	70
22	0	0	0	0	0	70
23	0	0	0	0	0	70
24	0	0	0	0	0	70
25	0	0	0	0	0	70
26	0	0	0	0	0	70
27	0	0	0	0	0	70
28	0	0	0	0	0	70
29	0	0	0	0	0	70
30	0	0	0	0	0	70
31	0	0	0	0	0	70
32	0	0	0	0	0	70
33	0	0	0	0	0	70
34	0	0	0	0	0	70
35	0	0	0	0	0	70
36	0	0	0	0	0	70
37	0	0	0	0	0	70
38	0	0	0	0	0	70
39	0	0	0	0	0	70
40	0	0	0	0	0	70
41	0	0	0	0	0	70
42	0	0	0	0	0	70
43	0	0	0	0	0	70
44	0	0	0	0	0	70
45	0	0	0	0	0	70

Table A.2: Failed ATs by reasoner JFacT in iteration 1 of the univ-bench standard benchmark.

Round	Occurrence	Class Satisfiability	Relation Satisfiability	Cardinality Satisfiabiliy	Multiple Cardinality	Range
1	0	0	0	0	0	0
2	0	90	57	0	0	38
3	0	85	56	0	0	38
4	0	77	50	0	0	38
5	0	76	47	0	0	35
6	0	72	50	0	0	35
7	0	70	48	0	0	34
8	0	57	36	0	0	26
9	0	50	29	0	0	18
10	0	40	20	0	0	15
11	0	35	19	0	0	14
12	0	26	14	0	0	9
13	0	23	15	0	0	8
14	0	23	15	0	0	8
15	0	22	12	0	0	8
16	0	19	8	0	0	8
17	0	11	5	0	0	3
18	0	10	4	0	0	3
19	0	7	0	0	0	3
20	0	7	0	0	0	3
21	0	7	0	0	0	3
22	0	4	0	0	0	3
23	0	4	0	0	0	3
24	0	4	0	0	0	3
25	0	4	0	0	0	3
26	0	4	0	0	0	3
27	0	0	0	0	0	0
28	0	0	0	0	0	0
29	0	0	0	0	0	0
30	0	0	0	0	0	0
31	0	0	0	0	0	0
32	0	0	0	0	0	0
33	0	0	0	0	0	0
34	0	0	0	0	0	0
35	0	0	0	0	0	0
36	0	0	0	0	0	0
37	0	0	0	0	0	0
38	0	0	0	0	0	0
39	0	0	0	0	0	0
40	0	0	0	0	0	0
41	0	0	0	0	0	0
42	0	0	0	0	0	0
43	0	0	0	0	0	0
44	0	0	0	0	0	0
45	0	0	0	0	0	0

List of Figures

3.1 Graphical representation of the CQ generation process of RB++. 16
3.2 Graphical representation of the edit generation process of RB++. 17
3.3 Graphical representation of the benchmarking process of RB++. 18
3.4 Contents of the `BenchmarkInfo` output. 30
3.5 Class Diagram of `BenchmarkManager` and its surrounding classes. 32
3.6 Class Diagram of `CompetencyQuestionManager` and its surrounding classes. 34
3.7 Class Diagram of `AuthoringTestManager` and its surrounding classes. . . 36
3.8 Class Diagram of `EditManager` and its surrounding classes. 38

4.1 Influence of Reinitialization of a Reasoner on Runtimes. Ontology: univ-bench, Seed: 1234567, Reasoner: Hermit 45
4.2 Results RB++ benchmark with ontology univ-bench, 45 edits and 100 CQs. Dots: Repeated measurements at one edit version of the ontology. Coloured lines: RB++ standard benchmark. Grey lines: Amount of axioms added / removed from the ontology for this edit-version. Measurements equal reasoning time without flushing. 48
4.3 Results RB++ benchmark with ontology univ-bench, 45 edits and 100 CQs. Dots: Repeated measurements at one edit version of the ontology. Coloured lines: RB++ standard benchmark. Grey lines: Amount of axioms added / removed from the ontology for this edit-version. Measurements equal both reasoning time without flushing / with flushing. . . 49
4.4 Differences between runtimes in 4.3 with flush / without flush on ontology univ-bench, 45 edits and 100 CQs. Coloured lines: RB++ Standard Benchmarks. Grey lines: Amount of Axioms added / removed from the ontology for this edit-version. 50
4.5 Results RB++ benchmark with the gene ontology, 100 edits and 100 CQs. Dots: Repeated measurements at one edit version of the ontology. Coloured lines: RB++ standard benchmark. Grey lines: Amount of axioms added / removed from the ontology for this edit-version. Measurements equal reasoning time without flushing / with flushing. 54

A.1 Influence of Reinitialization of a Reasoner on Runtimes. Ontology: univ-bench, Seed: 1234567, Reasoner: Pellet 66

A.2 Influence of Reinitialization of a Reasoner on Runtimes. Ontology: univ-bench, Seed: 1234567, Reasoner: ELK 67
A.3 Influence of Reinitialization of a Reasoner on Runtimes. Ontology: univ-bench, Seed: 1234567, Reasoner: JFact 68

List of Tables

2.1 CQ Archetypes by [Ren et al., 2014]. (PA = Predicate Arity, RT = Relation Type, M = Modifier, DE = Domain Independent Element; CE = Class Expression, OPE = Object Property Expression, DP = Datatype Property, I = Individual, NM = Numeric Modifier, PE = Property Expression, QM = Quantity Modifier; obj. = Object Property Relation, data. = Data Property Relation, num = Numeric Modifier, quan. = Quantitative Modifier, tem. = Temporal Element, spa = Spatial Element) 9

2.2 CQ Sub-types of Archetype 1 by [Ren et al., 2014]. (QT = Question Type, V = Visibility, QP = Question Polarity; CE = Class Expression, OPE = Object Property Expression; sel. = Selection Question, bin. = Binary Question, cout. = Counting Question, exp. = Explicit, imp. = Implicit, sub. = Subject, pre. = Predicate, pos. = Positive, neg. = Negative) . . . 10

2.3 CQ Archetype absolute Distribution based on [Ren et al., 2014]. 10

2.4 Authoring Tests by [Ren et al., 2014]. (E: Expression, CE: Class Expression, P: Property, n: Modifier) 12

2.5 Excerpt of COnto-Diff operations with descriptions and their inverses by [Hartung et al., 2013] with additional Change Operations not named in their paper (marked by *). 14

3.1 CQ Archetypes used by RB++. (PA = Predicate Arity, RT = Relation Type, M = Modifier, DE = Domain Independent Element; CE = Class Expression, OPE = Object Property Expression, DP = Datatype Property, I = Individual, NM = Numeric Modifier, PE = Property Expression, QM = Quantity Modifier; obj. = Object Property Relation, data. = Data Property Relation, num = Numeric Modifier, quan. = Quantitative Modifier, abs. = Absolute, sup. = Superlative, tem. = Temporal Element, spa = Spatial Element) 21

3.2 Mapping from Archetype ID to applicable Authoring Tests. (CE = Class expression, OPE = Object Property Expression, DP = Datatype Property, I = Individual, NM = Numeric Modifier, PE = Property Expression, QM = Quantity Modifier, * = currently not implemented.) 23

3.3 Absolute amount of Change Operations detected in the Gene Ontology [Harris et al., 2008], dating between 2010-01-01 and 2018-04-01. * points to Change Operations that are not implemented either due to constraints or sparse usage. 24

4.1 Setup of RB++ used to review error sources of reasoners ELK and JFact. Ontology: Pizza Ontology. Seed: 1234567 44
4.2 Setup of univ-bench benchmark for all reasoners. 46
4.3 Generated Change Operations for univ-bench benchmark. 46
4.4 Generated CQ Archetypes for univ-bench benchmark. 47
4.5 Amount of ATs to answer per reasoner per iteration in the univ-bench benchmark. 47
4.6 Comparison of answers to ATs between Hermit and Pellet. (Occ. = Occurrence Test, Cls. Sat. = fiability Test, Rel. Sat. = Relation Satisfiability Test, Rg. = Range, Sat. = Satisfiability) 51
4.7 Generated Change Operations for the gene ontology benchmark. 52
4.8 Generated CQ Archetypes for the gene ontology benchmark. 52
4.9 Amount of ATs to answer per reasoner per iteration in the univ-bench benchmark. 53

A.1 Failed ATs by reasoner ELK in iteration 1 of the univ-bench standard benchmark. 69
A.2 Failed ATs by reasoner JFacT in iteration 1 of the univ-bench standard benchmark. 70