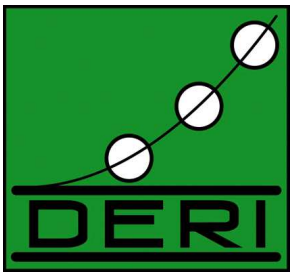

DERI – DIGITAL ENTERPRISE RESEARCH INSTITUTE

University of Innsbruck
Digital Enterprise Research Institute

Unit Testing for Ontologies

Bachelor Thesis

Martin Tanler
Kreuzbichlstr.32
A-6112 Wattens
Matrikelnr.: 0116226

Supervised by Univ. Prof. Dr. Dieter Fensel
and co-supervised by Holger Lausen

Innsbruck, July 26, 2007

Abstract

This thesis explains what unit testing is and why unit testing for ontologies is a helpful process for an ontological engineer in order to verify an ontology. In the course of that we clarify what an ontology is and present some common methodologies for ontological engineering. In a first step we define exactly in which state of the development process unit testing can be helpful. Unit testing is highly dependent on the used logical language, which is underlying the used ontology language. Therefore, in a second step, we examine some logical languages and detail those cases where the use of unit testing makes sense in order to avoid engineering mistakes. The theoretical results of this thesis are used to implement a unit testing prototype for WSML-Flight and WSML-Rule ontologies, based on Eclipse.

Acknowledgements

First, I thank my advisor Holger Lausen. He tought me how to write a thesis and guided me into the right direction. Furthermore, I would like to thank Nathalie Steinmetz for proofreading. Last but not least, a special thanks to my girlfriend Anja Ziller. She is always there for me, whenever I need her. It was her inspiration and patience that enabled me to write this thesis. THANK YOU ALL!

Martin Tanler, Wattens, July 2007

Contents

Chapter 1

Preliminaries

Before going into detail about unit testing for ontologies, it is necessary to understand what an ontology exactly is, how ontologies are being engineered and what the logical foundations behind ontologies are. This chapter focuses on these topics and tries to introduce the basic knowledge to understand the contents of the subsequent chapters.

1.1 Ontologies

Within this section we explain how the term 'ontology' arised. We present the basic philosophical idea behind ontologies and go further to modern definitions in computer science. Furthermore we provide an overview of the main components of ontologies.

1.1.1 What is an Ontology?

The term ontology was used first in philosophy. Ontology is a synonym for the philosophy of being. This branch of metaphysics is concerned with the existence and essence of things. The essence of something is that, what this thing is. This sounds simple. But the question, what the essence of a thing is, concerns mankind already for thousands of years. The ancient greeks described the essence of a thing as that, what remains when this thing changes. This approach, however, encounters some difficulties. As an example one can imagine a seed. A seed first gets rotten before it becomes a tree. There is nothing remaining. This problem was finally solved by Parmenides of Elena. He assumed that something that is, can not begin to be, nor end to be. This means for the seed, that it is nothing else than a very young tree. Consequently a tree has many modes of being. The fact that something has an essence has nothing to do with its existence. For example the comic figure 'Donald Duck' has essence. Donald is a duck that is able to speak. He *is*, although he does not exist. Later, in the Middle Ages, the focus moved from the essence to universals. Universals are some kind of symbols. They can be considered as the origin of todays concepts (or classes) in knowledge management. Thus their introduction was an important step

towards todays understanding of ontologies in computer science. In the modern ages the definition of essence was refined by Emanuel Kant. He stated that the essence of things is not only determined by themselves. According to him also the one who perceives a thing contributes to its essence. Jos Ortega got to the heart of it when he claimed that reality strongly depends on its observer. Reality is something individual. Two persons can have a different idea of the world they live in. However, both of these ideas can be valid. Persons are only able to recognize parts of the whole reality. This is in the figurative sense also true for knowledge systems.

As a simple consequence of the different realities persons live in, many experts in the area of knowledge management created many different definitions of the term ontology. One of these definitions was stated by Robert Neches and colleagues:

"An ontology defines the basic terms and relations comprising the vocabulary of a topic area as well as the rules for combining terms and relations to define extensions to the vocabulary" [5, page 6].

Although this is one of the first definitions, it describes an ontology in a rather illustrative way. This definition does not only refer to explicitly defined terms but also to implicit knowledge. Later Rudi Studer and colleagues defined an ontology in a different way:

"An ontology is a formal, explicit specification of a shared conceptualization. Conceptualization refers to an abstract model of some phenomenon. Explicit means that the type of concepts used, and the constraints on their use are explicitly defined. Formal refers to the fact that the ontology should be machine-readable. Shared reflects the notion that an ontology captures consensual knowledge. This knowledge is not private of some individual, but accepted by a group" [5, page 6].

One of the main issues this definition covers, is (except of the machine readability) that ontologies contain shared and consensual knowledge. Since an ontology can be related to another ontology, engineering mistakes may lead to problems not only in a single ontology, but in a set of ontologies. Unfortunately, some errors are difficult to discover, since they are related to knowledge, which is implicitly stated within an ontology. Thus, the implicit knowledge within ontologies and the fact that ontologies contain shared consensual knowledge underline the interest of an approach, considering the automatization of error discovery in the field of ontology engineering.

1.1.2 Ontology Languages in Computer Science

An ontology language is a formal language, which is used to encode an ontology in a machine readable way. Ontology languages differ in their expressiveness and their intended purpose. Examples for classical ontology languages are Ontolingua, KIF, OCML and FLogic [5]. Ontology languages represent the basic technology behind the semantic web. Thus there was developed a bunch of web-based languages (e. g. SHOE, XOL, RDF(S), OIL, DAML+OIL and OWL). In addition to the languages providing the semantic annotation of classical web resources, there was developed a language called 'Web Service Modeling Language' (WSML). WSML allows the formal notation of ontologies, semantic web services, goals and mediators.

Some ontology languages (like OWL or WSML) provide multiple variants. These variants offer a certain expressiveness and decidability, according to their underlying logical formalism. Particular information about ontology languages and their use can be found in [5, Chapter 4]. Specific information about WSML can be found in [1] and [2].

1.1.3 Ontology Components

Ontological engineers have to choose the ontology language in which they want to encode their ontologies. Ontology languages may differ in their expressiveness, though they may share structural similarities. This section is supposed to give a general overview of these similarities. As even these similarities may behave differently in some cases, one should note that the following descriptions are very general and may differ in detail depending on the underlying ontology language. The term 'structural similarities' is denoted further as 'ontology components'.

Concepts and Instances. Concepts are groups of individuals. Some of their features are similar to the features of classes in UML [11]. Concepts were already mentioned in Section 1.1.1. They can be defined explicitly as well as implicitly within an ontology. A concept may inherit the features of another concept. The inheriting concept is then called subconcept, the inherited concept is called superconcept. This allows one to organize the concepts of an ontology as a taxonomy (i.e. in a hierarchical structure).

Instances represent individuals. An instance can be member of several concepts. If an instance is member of a subconcept, then it is also member of the specified superconcepts.

Relations and Functions. Relations are used to describe the associations between instances or concepts. A classical example of a relationship between two concepts is the 'subconcept-of' relationship, which enables inheritance between them. This relationship is a so called binary relationship, because it contains two arguments, namely the subconcept and the superconcept. This, however, is not the only arity of relationships, which is supported by ontology languages. Some formal languages do not make any restrictions regarding the upper bound of their relationships arity: they support the use of binary up to n-ary relationships (the relation contains n arguments).

Functions are a special kind of relation, where the last argument is determined by the product of its predecessors. Further information about functions can be found in [5, page 13].

Attributes. There are two kind of attributes. Attributes, which are part of instances (instance attribute), and attributes, which are part of concepts (concept attribute). These kind of attributes are similar to attributes of classes and instances in UML respectively. A concept attribute contains usually a type definition. This type definition determines which type(s) the corresponding attribute value matches. The value(s) of an attribute is/are defined as part of an instance attribute.

Axioms. Axioms provide the possibility to represent knowledge that can not be represented by other ontology components. They can be used to verify the consistency of an ontology or to infer implicit knowledge. The expressiveness of the statements used within axioms depends on the corresponding ontology language, especially the kind of logic this language is based on. *"According to Gruber, formal axioms serve to model sentences that are always true"* [5, page 14].

1.1.4 Summary

The term ontology is originated in philosophy. As part of metaphysics an ontology is the philosophy of beings. In context of computer science, ontologies are datamodels that contain shared consensual as well as inferred knowledge. In order to represent ontologies in a machine readable way there were developed different formal languages (e.g. WSML and OWL) called ontology languages. These languages (and their variants - if existent) vary in their expressiveness and decidability. Nevertheless, most of them provide similar components like concepts, instances, relations, axioms and functions.

1.2 Logic

The expressiveness of an ontological language depends on its underlying logical formalism. This formalism is used in order to define axioms as well as for reasoning over an ontology. Therefore we present here a very basic description of the most common formalisms, such as first order logic and description logic. Additionally we describe what logic programming and datalog are and how they are related.

1.2.1 First Order Logic

First order logic (FOL) is an extension of propositional logic. A language of propositional logic contains infinitely many propositional letters[1] and a set of zero-place up to n-place connectives. However, it rarely occurs in practice, that a language contains three-or-more-place connectives [4, page 10]. As examples for a one-place connective and a two-place connective we consider $\neg$ (negation) and $\wedge$ (conjunction). These two connectives already represent the whole expressiveness of all other possible connectives. That is, all formulas which contain various connectives in addition to or instead of $\neg$ and $\wedge$ can be reduced to formulas containing only $\neg$ and $\wedge$ connectives.

The expressiveness of propositional logic is quite limited, as it is only possible to make very specific statements about something. Thus propositional logic was extended to first order logic. A language of first order logic provides at least the quantifiers $\exists$ and $\forall$ (in words 'exists' and 'for all'), variables and propositional connectives [4, page 110]. Depending on the language specification, a first order language may also provide predicate (relation) symbols, function symbols and constant symbols. Predicate symbols and function symbols have an arity greater than zero. The arity of a predicate or function symbol is defined by the amount of variables it takes as parameters. A constant is equal to a function symbol with an arity of zero. Predicate

[1] A propositional letter may be evaluated to either $true$ or $false$.

symbols are evaluated to either $true$ or $false$, depending on their own and their parameters interpretation. As an example of a statement in first order logic we consider $(\forall child)(\forall father)FatherOf(father, child) \wedge \neg IsMarriedTo(father, motherOf(child)) \implies Illegitimate(child)$. One possible interpretation for this statement is: 'If a father is not married to the mother of a child, then the child is an illegitimate child'.

1.2.2 Description Logics

Description logics can be seen as a subset of first order logic. They denote a family of various knowledge representation languages, which all share the same core features. The description logic (DL) family is originated from semantic networks and frame systems, which are both non-logical but graphical approaches (see [10, Section 2.1.1]). DLs distinguish between concepts (also called classes), roles, and instances, where each of them is similar to concepts, attributes of concepts or binary relations, and instances as defined in Section 1.1.3 respectively. Furthermore DLs differentiate the TBox (terminological box) and the ABox (assertional box). While the TBox contains all declarations of concepts and their relations to each other, the ABox contains all instances. A member of the description logic family is classified by the constructors it provides (e. g. conjunction, disjunction, value restriction, etc.). There exists a certain naming convention, that makes this classification visible. The name $\mathcal{SHOIN}(\mathcal{D})$ for example, describes a decidable group of description languages. It extends the description logic $\mathcal{S}$ (providing transitive roles, all the boolean operators on concepts, and existential, as well as universal restrictions), whereas $\mathcal{N}$ stands for unqualified number restrictions, $\mathcal{O}$ for nominals, $\mathcal{I}$ for inverses on roles, $\mathcal{H}$ for role hierarchies and $\mathcal{D}$ for datatypes [9]. This allows to recognize immediately the expressiveness of the according language.

Reasoning with DL. Reasoning is a procedure that allows to infer implicit knowledge from a given set of statements. DL-Reasoners support consistency checking on the knowledge base level, concept subsumption and concept satisfiability within the TBox as well as instance checking within the ABox [10]. A knowledge base is consistent, if it does not contain any contradicting data. Concept subsumption is similar to concept inheritance as described in 1.1.3. A concept is satisfiable if and only if there can exist at least one instance of it. Instance checking clarifies membership relations. That is, it is checked, whether a given individual is an instance of a certain concept, or not.

1.2.3 Logic Programming

Logic programming (LP), in its broadest sense, is the use of mathematical logic for programming a computer (see http://en.wikipedia.org/wiki/Logic_programming). Thereby logic is used as a declarative representation language. This language is processed by a theorem-prover or model generator that solves the actual problem. Logic programming in the narrower sense is to be understood as the use of a declarative and procedural representation language in order to solve a certain problem. This means, that a developer is not only responsible for the correctness of a program, but also for its efficient implementation. Subsequently, this requires a certain knowledge

about the behavior of the according theorem-prover. There exist several theorem-proving strategies, such as sequential first-in-first-out backtracking, parallel search, intelligent backtracking or best-first search. It is out of the scope of this thesis to explain all of them. Nevertheless it is important to recognize, that different strategies have a different performance and may lead to different results. The fact, that there exist multiple ways to process a logic program, is characterized by the equation $Algorithm = Logic + Control$, whereas *Logic* stands for a logic program and *Control* for the underlying theorem-proving strategy.

Datalog. Datalog is a query and rule language for deductive databases[2], which is syntactically a subset of Prolog, and therefore heavily related to logic programming (see http://en.wikipedia.org/wiki/Datalog). Prolog uses a sequential first-in-first-out backtracking strategy. The most important construct of Prolog is the dual declarative/procedural interpretation of implications, written in the form $A :- B_1, ..., B_n$. Such clauses can be restricted to definite clauses or Horn clauses, where A as well as $B_1, ..., B_n$ represent atomic predicate formulae in first order logic. The satisfiability of Horn clauses is decidable. In addition to that, Prolog is a language, which is turing complete. That is, Prolog has the power to simulate a universal turing machine (i.e. a machine which is essentially able to perform any computational task, disregarding efficiency and performance problems). Similar to Prolog, datalog is based on Horn logic and can be interpreted in a declarative and procedural way. Consequently the satisfiability of datalog programs is also fully decidable. However, there seem to be less further similarities. In contrast to prolog, datalog is not turing complete. The execution of a datalog program is independent of the order of the clauses within its source code. Furthermore query answering in datalog is sound and complete. There exist implementations that allow efficient query evaluation even for large datalog databases.

1.2.4 Summary

First order logic is an extension of propositional logic, which enables the use of quantifiers. The main components of a formula in FOL are predicates, that are evaluated to either $true$ or $false$. Description logics is as a subset of FOL. A language out of the DL family always distinguishes between the TBox, containing concepts, and the ABox, containing instances. There is only a limited amount of operations that can be processed by a DL-reasoner (e.g. consistency checking). Logic programming is characterized by the equation $Algorithm = Logic + Control$. Datalog is a deductive database related to Prolog (a logic programming language). Similar to Prolog, Datalog is based on Horn logic. However Datalog has the advantage, that the query results are independent of the order of clauses within the source code.

1.3 Ontological Engineering

It is out of the scope of this thesis to cover the whole area of ontological engineering. Therefore we only describe which parts of ontological engineering are relevant for the

[2] A deductive database contains rules and facts. It uses an inference machine to obtain implicit facts and rules from those already given.

further understanding. In the following we describe what a methodology is. Based on that we present the ontology development process, which is part of the METHONTOLOGY methodology. In addition to that, we give an insight into the methodology, that is used by the Enterprise Integration Laboratories for the design and evaluation of integrated ontologies. This methodology was described by Uschold and Grüninger [12].

1.3.1 Methodologies

In the literature, the terms methodology, method, technique, process, activity, etc. are mentioned in an interchangeable manner [5]. In order to clarify how to interpret the said terms within this thesis, we use the definitions given by the Institute of Electrical and Electronics Engineers (IEEE).

The term 'methodology' was standardized in 1990 by the IEEE. According to the IEEE a methodology is *"a comprehensive, integrated series of techniques or methods creating a general systems theory of how a class of thoughtintensive work ought be performed."*. That is, a methodology is composed of methods and techniques. A method is understood as a set of *"orderly process or procedure used in the engineering of a product or performing a service"*. A technique is defined as *"a technical and managerial procedure used to achieve a given objective"* [6].

These definitions leave a gap, as there is the unexplained term 'process' remaining. To close this gap the IEEE (in 1996) defined a process as a *"function that must be performed in the software life cycle. A process is composed of activities"*, where each activity is *"a constituent task of a process. A task is a well defined work assignment for one or more project members. Related tasks are usually grouped to form activities"* [7].

The difference between a task and an activity is not clearly distinguished by the IEEE. Accordingly an activity is a task. However a task has not to be an activity. Tasks may be part of activities.

To illustrate the difference between a method and a technique, we assume a method to build an ascending sorted list of integers out of an unsorted list of integers. The method then consists in putting the numbers of the unsorted list into an ascending order. There exist several techniques in order to achieve the resulting sorted list. For example, one could use the bubble sort or the merge sort algorithms[3].

Summarized a methodology can be seen as a series of methods and techniques. Methods prescribe what should be done. A technique describes in a detailed manner, how a method is performed. Thus a technique is a kind of a process with activities.

1.3.2 The Ontology Development Process

The ontology development process is defined as part of the METHONTOLOGY methodology [5]. This process identifies three kind of activities. Namely ontology management activities, ontology development oriented activities and ontology support activities.

[3]Particular information about sorting algorithms can be found at http://en.wikipedia.org/wiki/Sorting_algorithm.

Ontology Management Activities. These kind of activities include the quality assurance activity as well as the scheduling and the control activity. The quality assurance activity aims to achieve a satisfiable quality for each and every product output (e.g. an ontology). The scheduling and the control activity are related to each other. The scheduling activity identifies the tasks to be performed, their ordered execution, and their time and resources to be completed, while the control activity assures, that all scheduled tasks are completed and performed in the intended manner.

Ontology Development Oriented Activities. Ontology development oriented activities are grouped into pre-development, development and post-development activities.

Pre-development activities include the conducting environment study [5] in order to determine the platforms the ontology will be used on or the application where the ontology will be integrated. Furthermore, as part of the pre-development activities, the feasibility study [5] gives answers to question like: is it possible and suitable to build the ontology?

The development activities are carried out after having completed all pre-development activities. There are four activities, that belong to the development activities group: the specification, the conceptualization, the formalization and the implementation activity. The specification activity covers the identification of the reason why the ontology is being built, the specification of the ontology's intended use and the specification of the target group (end-users). *The conceptualization activity structures the domain knowledge as meaningful models at the knowledge level* [5, page 110]. That is, a conceptual model [4] is being built. This conceptual model is being converted into a formal model during the formalization activity. Finally, during the implementation activity, that formalized model is implemented in an ontology language.

The post-development activities cover the whole area of maintenance. Therefore the maintenance activity is performed, whenever it is necessary to update or correct an ontology. In addition to the maintenance activity, the post-development activities include the use activity. This activity represents the use and reuse of the ontology (by other ontologies, application, ...).

Ontology Support Activities. Ontology support activities are performed (simultaneously to the development activities) to support the development activities. This group of activities contains activities for knowledge acquisition, ontology and software evaluation, ontology integration, ontology merging, ontology alignment, ontology documentation and configuration management. The knowledge acquisition activity focuses on collecting the sufficient domain knowledge (e.g. from experts or in a semi-automatic way) in order to build the ontology. The evaluation activity (technically) judges the developed ontologies, their associated software environments as well as their documentations against a frame of references at each stage and between stages of the ontology's life cycle. But not every ontology has to be built from scratch. It is a common procedure to reuse already existing ontologies if possible. In that case, the integration activity is performed. The merge activity in contrast to the integration

[4] Further information about conceptual models can be found at http://en.wikipedia.org/wiki/Model_(abstract)#Structure_of_models

activity, merges several ontologies to obtain one ontology that unifies similar ontology elements of the different source ontologies. That is, the merging process refers to ontologies containing overlapping domain knowledge, while the integration activity refers to ontologies containing non-overlapping domain knowledge. The alignment activity establishes links between similar ontology elements of different ontologies. The advantage of the alignment activity compared to the merge activity is, that the tasks of the alignment activity preserve the original ontologies. The configuration management activity handles the versioning of the ontology code and the ontology's documentation. This activity enables among others the development of ontologies by teams.

1.3.3 A Formal Methodology

Based on the experience of the Enterprize Integration Laboratory for design and evaluation of integrated ontologies, Uschold and Grüninger [12] published a formal approach to build, develop and evaluate ontologies. This approach includes capturing motivation scenarios, the elaboration of informal competency questions, the specification of the terminology within a formal language (e.g. first order logic), the formalization of the informal competency questions using the specified terminology, the specification of axioms and definitions for the terms in the ontology within the formal language as well as the justification of the axioms and definitions by proving completeness theorems.

Identification of Motivating Scenarios. This process identifies the reason why an ontology should be developed, by gathering motivation scenarios. Such scenarios are related to the applications that will use the ontology. In particular, a motivating scenario is an unsolved problem within an enterprize, that can be potentially solved by an ontology. As part of a motivating scenario, there are always possible solutions provided. These solutions already prescribe some informal intended semantics for the elements of the finished ontology. Thus, during the development of an ontology, there should always be created more than one motivating scenario.

Elaboration of Informal Competency Questions. Informal competency questions are posed in a natural language. Some of the competency questions may already occur during the identification of motivating scenarios. The competency questions determine the requirements of the ontology. That is, by reasoning over the ontology the competency questions need to be answered correctly.

The competency questions are also useful for evaluation purposes. Before creating a new ontology from scratch, one should evaluate whether there is an existing one, which meets the requirements. In order to evaluate, whether an ontology is applicable for a given problem, one can use the competency questions. That is, if all competency questions can be answered by an existing ontology, then there is no need for a new one. If it is the case, that only some competency questions can be answered, then the existing ontology is at least powerful enough to be extended.

According to [12] the competency questions should be defined in a stratified manner. That is, more complex questions (higher level questions) are supported by the result of simpler questions (lower level questions). If there are only defined questions for an ontology, such that no question relies on the result of another question, then we define

the concerning ontology as not well-designed.

Specification of the Informal and Formal Terminology. The informal terminology of an ontology is extracted from the informal competency questions. In addition to that, further terminology of an ontology can be gathered by ontology capturing techniques as described in [12, page 18 - 24]. Since an ontology does not only contain a set of terms but also the definitions of terms, it is useful to extract the informal definitions of terms from the natural language answers of the competency questions [5, page 121]. *"The informal dictionaries and glossaries defined using this methodology provide the intended semantics of the terminology and lay the foundations for the specification of axioms in the formal language"* [12, page 31]. At the same time or after the identification of the informal terminology, an ontologist formalizes this terminology using an ontology language.

Formalization of Competency Questions. After identifying the informal competency questions and the terminology of the ontology, the competency questions are being formalized using the same formal language that is used for encoding the terminology. According to Uschold and Grüninger (they assume first order logic to be used), *"the competency questions are defined formally as an entailment or as a consistency problem with respect to the axioms in the ontology"* [12].

Specification of Formal Axioms. Axioms are used to specify the definitions of terms as well as the constraints of the ontology. They must be minimally sufficient in order to ensure that reasoning over the ontology, using the formalized competency questions, results in the expected solutions.

Completeness Theorems. The completeness theorems define the conditions under which the solutions of the competency questions are complete. This allows one to infer, that the specified formal axioms are sufficient or not. *"Completeness theorems can also provide a means of determining the extendibility of an ontology, by making explicit the role that each axiom plays in proving the theorem. Any extension to the ontology must be able to preserve the completeness theorems"* [12]. Furthermore, the maintenance of ontologies may involve unwanted solutions of the competency questions. Thus, the competency question also determine the maintenance of an ontology in the same way as the extendibility of an ontology.

1.3.4 Summary

The term 'methodology' was standardized by the IEEE. A methodology is a series of methods and techniques. While a method prescribes what should be done, a technique is composed of activities, which are executed in a certain order, to perform a method. A technique can therefore be seen as a process with activities. An example of such a process is the ontology development process, which was defined as part of the METHONTOLOGY methodology. This process is composed of three kind of activities, ontology management activities, ontology development oriented activities, and ontology support activities.

Uschold and Grüningers approach for engineering ontologies is an example of a field-tested methodology. This approach contains activities in order to capture motivation scenarios, elaborate informal competency questions, specify the according terminology within a formal language, formalize informal competency questions, specify axioms and definitions for the terms in the ontology within a formal language as well as to justify the axioms and definitions by proving completeness theorems.

1.4 Unit Testing in Software Engineering

Unit testing was first defined as part of extreme programming[5]. A unit is the smallest testable part of a program. For example, in object oriented programming languages, the smallest testable part is always a class. In extreme programming a unit test is created first, before the actual program is encoded. This helps the developer to consider what a programm needs to provide, and allows one to test the implemented units immediately. That is, unit testing does not only guarantee high quality code, but also defines the boundaries of the functionality of an application in a very strict manner. In particular, developing an application using unit tests is an alternating procedure. At first a unit test has to be created. This unit test addresses a small part of the problem. Then the simplest possible code is created, such that the unit test will pass. Now a second test is created, and the current program code is extended, that also the second test will pass. This is to be continued until there is nothing left to be tested. Before an application can be released, it has of course to pass all created unit tests.

Another aspect of unit testing is, that it enables harmless code contributions of different developers to one and the same project (collecting code ownership). That is, developers are encouraged to contribute additional code, change existing code fragments, fix bugs or even refactor existing code, which was not originally written by their own. Of course, before coding additional functionalities, there have first to be written the according unit tests in the manner we described before. After changing an existing piece of code, the according unit tests still have to pass, if the programmes intension stays unchanged. Consequently code changes are constantly guarded by unit tests. This converts code ownership to an old fashioned artefact. As unit testing is quite popular in the developers community, there exist several frameworks for unit testing, supporting different programming languages. Particular information related to extreme programming and unit testing can be found at http://www.extremeprogramming.org/rules.html. Further information about unit testing tools is available at http://www.xprogramming.com/software.htm.

[5]Extreme programming is a software development methodology.

Chapter 2

Unit Testing for Ontologies

Ontology engineering is a quite young discipline. Thus we can not refer to the amount of experiences, as available in the area of software engineering. What we can do however, is to invest some time to figure out the similarities between software development and ontology development, in order to avoid problems in ontology engineering, which are already known in software engineering. This includes in a subsequent manner the evaluation of software engineering techniques, which are determined either to be applicable for an ontology engineer, or not. Within this section, we pose the question, if unit testing in a modified form is applicable in the field of ontology engineering. In the course of that we argue why unit testing for ontologies has the power to improve the ontology development process (see Section 1.3.2). Furthermore we present the idea of unit testing for ontologies based on a variant of description logic, which was stated by Denny Vrandečić and Aldo Gangemi. Based on their approaches we introduce a technique to apply unit testing for logic programming ontologies.

2.1 The Power of Unit Testing for Ontologies

Unit testing in software engineering was already described in Section 1.4. There we stated the most important features of unit testing. Unfortunately the power of unit testing in software engineering underlies some limitations (e. g. it is not possible to apply performance tests). Nevertheless it has the power to save time and money within a project. Similar to unit testing in software engineering we expect some limitations for unit testing in ontology engineering. Although we are convinced that the effort will be worth. Within this section we want to show, why the ontology development activities (see Section 1.3.2) should be guided by unit testing. Unit testing itself is assumed as a kind of technical ontology evaluation[1] and belongs therefore to the ontology support activities.

[1]The technical ontology evaluation is performed by a developer. Particular information about ontology evaluation methods can be found at [5, page 178 - 195].

2.1.1 Unit Testing: An Ontology Evaluation Technique

We already stated that the ontology development activities are supported by the ontology support activities (see Section 1.3.2). In this context, the evaluation activity is of special interest for our purpose, since this activity is addressed to observe the implementation of all and ontology components (including imported components). The main criteria in order to evaluate an ontology are consistency, completeness and conciseness [5, page 178 - 195].

Furthermore, ontology evaluation combines ontology verification, ontology validation and ontology assessment. Ontology verification determines if the ontology is implemented correctly, and if the ontology meets its requirements (probably determined by competency questions). Validating an ontology means to check, whether the ontology is compliant with the real world. That is, a verified ontology has to satisfy the competency questions. However, additional ontology elements, that are not specified within the requirements, are out of the scope of ontology verification. At this point ontology validation takes over and ensures, that even putative unneeded elements are implemented correctly (in line with the real world). As an example for such an unneeded element we assume a relation between a concept *Man* and a concept *Person*, stating that every man is a person. This relation is not really required by an ontology that is only intended to find out if a man is a brother of another man. Nevertheless, it makes sense to include such a relation (even more if *Person* is an imported concept) in order to improve the ontology's usability.

Ontology assessment considers the usability of an ontology from the users point of view. Unit testing compares, if an expected result matches the result of a tested unit. Consequently, unit testing is only addressed to ontology verification and validation, as ontology assessment underlies individual criteria which can not be formalized. In addition to verification and validation, unit testing for ontologies is intended to discover consistency problems as well as to evaluate, whether an ontology is complete or incomplete (under the condition, that an ontology is assumed to be complete, if *"all that is supposed to be in the ontology is explicitly stated in it, or can be inferred"* [5, page 178 - 195]).

2.1.2 Advantages of Unit Tests for Ontologies

Unit testing for ontologies is a kind of semi-automated technical evaluation of ontologies. That is, unit testing is addressed to verify and validate an ontology as well as to check its consistency and completeness. We denote unit testing as semi-automated, since it is at least necessary to write a unit test and to specify the expected results. Once a unit test is written, it can be executed again and again. The requirements of an ontology are often specified by competency questions. Thus there is a certain similarity to unit tests in software engineering, as these unit tests set the boundaries of the functionality of a programm. Consequently competency questions and their answers are potential candidates, to be used as part of unit testing for ontologies. The formalization of competency questions allows an automated evaluation, stating whether an ontology meets its requirements, or not. Furthermore unit tests ensure that code manipulations do not result in additional problems. It is expected that the initial development of an ontology is done by a high skilled developer, while the main-

tenance is performed by a less qualified group of developers [3, page 4]. On the one hand, by creating unit tests, the high skilled developers are able to anticipate latter manipulation failures. On the other hand, developers who have fixed problems within an ontology can update the unit tests, in order to be sure that the problems will never occur again. In addition to that, unit testing protects one from undesired side effects, that may arise by adding, removing or exchanging imported ontologies. Without unit testing it is rather difficult to ensure whether the imported ontologies are compliant with the requirements of the actual ontology. Thus the import of ontologies may involve a certain set of problems. By means of unit tests these problems can be detected immediately.

2.1.3 Summary

Unit testing for ontologies is a semi-automated evaluation technique. This technique can be used to verify and validate an ontology as well as to test it for consistency and completeness. The application of unit testing in the field of ontology engineering grants secure code manipulations and protects the developer from making the same failure twice. Furthermore unit testing is an effective technique in order to support the secure import of ontologies.

2.2 Related Work: Unit Testing for Description Logic Ontologies

Unit testing for ontologies was already proposed by Denny Vrandečić and Aldo Gangemi (see [13]). In order to apply the idea of unit testing for ontologies, they suggested different approaches, all of them inspired by unit testing frameworks in software engineering. Within this section we try to capture some of their ideas, which assume ontologies to be based on description logics $\mathcal{SHOIN}(D)$.In the following, we denote the ontology that is to be tested, as the ontology O.

2.2.1 Unit Testing by Entailed Axioms

This approach is based on two test ontologies, a positive test ontology T^+ and a negative test ontology T^-. Both test ontologies contain constraints, related to the content of O, such that the correctness of O, implies that each axiom $A^+_{i|1\leq i\leq n} \in T^+$ and each axiom $A^-_{i|1\leq i\leq n} \in T^-$ is derivable and underivable from O respectively. This can be expressed more formally by two conditions: $O \models A^+_i \forall A^+_i \in T^+$ and $O \not\models A^-_i \forall A^-_i \in T^-$.

The term correctness, therefore, can be reduced to the correctness which can be granted by T^+ and T^-. That is, similar to unit tests in software engineering, unit tests in ontology engineering cannot promise that a developed ontology is totally correct. For example, for an unsatisfiable ontology O, T^+ may trivially fulfil the first condition. Furthermore an empty ontology O implies that the second condition is fulfilled.

This shows that unit testing by entailed axioms encounters some limitations regarding

the correctness of O (i.e. it is not possible to formalize all requirements of O by the two test ontologies). Nevertheless, the application of this method has the power to save time and costs during the ontology lifecycle. *"The test ontologies are meant to be created and grown during the maintenance of the ontology. Every time an error is encountered in the usage of the ontology, the error is formalized and added to the appropriate ontology. Experienced ontology engineers may add appropriate axioms in order to anticipate and counter possible errors in maintenance"* [13, page 4]. This ensures an error to be protected of occurring twice.

Denny Vrandečić and Aldo Gangemi expect an evolution of ontology engineering similar to software engineering. Thus they stressed the fact, that developers creating a program, are in most cases more skilled than developers maintaining programs. If this becomes true for ontology engineers, then it will be even more important to apply unit tests for an ontology from the first moment on. Unit testing by entailed axioms offers in general the same advantages to ontology engineers, as unit testing in software engineering to software developers (see 1.4).

Denny Vrandečić and Aldo Gangemi posed the question, why an ontology should not include both, the axioms of T^+ and the negated axioms of T^- to O, instead of creating two separate test ontologies. In order to answer this question, they stated five arguments for the explicit use of two test ontologies.

Negated Axioms. Unfortunately, there may exists some axioms in T^- that cannot be negated. Thus, in some cases, it is impossible to integrate the negated form of all axioms of T^- within O. As an example we consider the statement R(a, b), where a and b are individuals. According to Vrandečić and Gangemi, this simple relation cannot be negated in OWL DL[2].

Redundancy. Redundant axioms decrease the readability of an ontology. Consequently, it is more difficult to edit such an ontology. This increases the costs of ontology engineering (at least in ontology maintenance), since a developer may need additional time to understand an encoded ontology.

Complexity. The complexity of reasoning is dependent on the kind and amount of axioms stated within an ontology. Reasoning is usually a complicated and time consuming process. It is therefore not advisable to add the axioms of T^- and T^+ to O, as the reasoning performance would be heavily decreased. This effect is even stronger, if O imports other ontologies with similar axioms.

Contradictory Axioms. Since the ontology T^- may contain contradicting axiom sets [13], an integration of T^- within O could result in an unsatisfiable ontology O. Contradicting axioms within the ontology T^- are totally reasonable, as this means that O is unaware of the truth of these axioms.

[2]OWL DL is a variant of the ontology language OWL, based on the description logics $\mathcal{SHOIN}(D)$.

Open World Assumption. Assuming an open world means, that one is aware of the fact that knowledge is always incomplete. That is, if some fact is not known to be true, it is therefore not necessarily false. As some ontology languages use the open world assumption (e.g. OWL DL), the ontology T^- cannot easily be negated without changing its semantics.

2.2.2 Competency Questions

We already described competency questions within Section 1.3.3 as part of Uschold and Grüninger methodology. Furthermore, we explained their role for unit testing within Section 2.1.2. We discovered a certain similarity between unit tests in software engineering and competency questions in ontology engineering. This similarity has also been recognized by Vrandečić and Gangemi. According to [13], the use of competency questions for unit tests is especially useful for the initial build of an ontology, rather than for its maintenance. Applying this approach, one should be aware of the fact, that answering all competency question correctly does not necessarily require a complete ontology.

This kind of unit testing cannot only be used for ontologies containing static knowledge, but also for ontologies stating dynamic knowledge (frequently changing facts). That is, ontologies with dynamic knowledge can be characterized by competency question, but the answers of these questions are either unknown, or if known, only valid for a certain period of time. To avoid the problem of missing or frequently changing answers, Denny Vrandečić and Aldo Gangemi suggested to *"define some checks if the answer is sensible or even possible [...]"* [5, page 5] (e.g. the set of answers may only contain instantiations of a class $Human$).

2.2.3 Consistency Checks

Consistency checks are useful in order to ensure that an ontology fulfils a set of specific rules, e.g. the disjointness of classes. According to Vrandečić and Gangemi, a consistency check should be applied to an ontology at the beginning of its usage, rather then each and every time at reasoning. Therefore Vrandečić and Gangemi proposed to build two ontologies, one lightweight ontology without consistency constraints, and one heavyweight ontology including consistency constraints. Reasoning over the light weight ontology thereby results in a better response time than reasoning over the heavyweight ontology. Furthermore, the translation of O into a logic programming language (e.g. datalog) would allow the use of more expressive constraints.

2.2.4 Domain and Ranges

Denny Vrandečić and Aldo Gangemi assume the specification of constraining domains and ranges as part of unit testing for ontologies. Such specifications are beyond the abilities of description logic ontologies. Therefore, they can be seen as a semantical extension, which increases the expressivity of an ontology language.

In particular, Vrandečić and Gangemi propose to build a second ontology, similar to O, but without any domain or range definitions. By checking the type of all those

instances of the second ontology, which were affected by the removal of domains and ranges, undesired instantiations within O can be discovered. As an example we consider the relation $fatherOf$ with the domain Man and the range $\{Man, Woman\}$. Since many ontology languages do not allow domain and range constraints, encoding the statement *'Andrea is the father of Harry'* using the relation above, would infer that Andrea is a man. In order to be sure about the gender of Andrea, it is necessary to remove the domain and the ranges of $fatherOf$, and to test, if Andrea is still defined as a man within the ontology. Although, as inferring type definitions are not always undesired, this approach does mark places of potential errors, rather than consistency violations.

This procedure is rather complex. Therefore Vrandečić and Gangemi considered the introduction of new relations that enable the explicit definition of constraining domains and ranges. By the means of such relations, the creation of a second ontology becomes unnecessary, as the differentiation of inferring and constraining domains and ranges can be explicitly stated within O. An ontology language that already realizes this idea is the Web Service Modeling Language (WSML). WSML provides the $ofType$ and the $impliesType$ keywords. By the means of $ofType$ one is able to define the range of an entity in a constraining way, while the $impliesType$ keyword allows inferring range definitions.

2.2.5 Summary

Denny Vrandečić and Aldo Gangemi suggested different approaches in order to apply unit testing on $\mathcal{SHOIN}(D)$ ontologies. These approaches include ideas like unit testing by entailed axioms, using competency questions for unit testing, consistency checking as part of unit testing and unit testing against constraining domains and ranges.

Unit testing by entailed axioms is based on a positive and a negative test ontology. In case of a correct ontology O, the positive ontology is entailed by O, while the negative ontology is not entailed by O. The use of competency question for unit testing allows to check, whether O meets the specified requirements. Although, this approach does not guarantee the completeness of the ontology O.

Consistency checking as part of unit testing implies one lightweight ontology, without constraints, and one heavyweight ontology, including constraints. Consequently, reasoning over the lightweight ontology results in a better performance than reasoning over the heavyweight ontology. Thus Vrandečić and Gangemi proposed to perform a consistency test only once, before an ontology is used, rather than testing the ontology each and every time at reasoning.

Using competency questions for unit tests is especially useful for the initial build of an ontology. In order to use this approach for ontologies with dynamic knowledge, Denny Vrandečić and Aldo Gangemi proposed to define some tests in order to evaluate, whether an answer is sensible or even possible.

The specification of constraining domains and ranges can be realized by two different approaches. The first approach assumes the creation of an ontology similar to O, but without any domain or range definitions. By comparing the set of instantiations of the new ontology and O, it is possible to check, if O contains undesired inferred

instantiations. The second approach suggests the introduction of new relations, such that constraining domain and range definitions can be stated explicitly. WSML is an ontology language, already realizing the second approach. Therefore the keyword $ofType$ can be used in order to specify constraining range definitions.

2.3 Functionality of Unit Testing for Logic Programming Ontologies

Logic programming ontologies have, due to their nature, different features than DL ontologies. We assume logic programming ontologies to be based on Horn logic. Inspired by the work of Denny Vrandečić and Aldo Gangemi, we present an approach that enables unit testing for logic programming ontologies. Within this section we define four different kind of tests, namely complex tests, boolean tests, constraint tests and test suites. Complex tests and boolean tests are based on the formalization of competency questions and their answer sets. However, in contrast to the approach of Vrandečić and Gangemi (see Section 2.2.2), we consider the extension of the set of formalized competency questions and their answers during the maintenance of an ontology. This is compliant with unit testing for software engineering, as described in Section 1.4. There we stated, that for each added functionality of a programm, an according unit test has to be added. Furthermore, in case of a changing peace of code, the previous unit tests have still to pass. As we do the same, in a figurative way, within our approach, we grant that an ontology is guarded during the whole ontology life cycle. Finally, within this section, we present a framework for unit testing that can be implemented for every ontology language based on LP. The ontology to be tested, is further denoted as ontology O within this section.

2.3.1 Complex Tests

A complex test is composed of a formalized query, a set of expected answers and a match-value. The query of a complex test must return a set of answers, where each answer consists of a set of variable bindings. For example, the question whether a person called Nathalie has black hair or not, cannot be formalized to a query for complex tests, as it can only be evaluated to either $true$ or $false$. In order to create a complex test, the question has to be reformulated. Therefore, the question 'who has black hair?' represents a reasonable modification of the question before. Whether a complex test will pass or not, depends furthermore on the kind of match between the actual answer set of the query and the answer set, which was expected (expected answer set).

The match-value of a complex test is intended to specify the expected kind of match between the set of expected answers (specified by the user) and the actual answer set of the formalized query. In particular we distinguish five kind of matches: a full match, no match, an intersected match, a subset match and a partial match.

Full Match. A full match assumes the expected answer set and the actual answer set to be equal. The application of a full match (as match-value) guarantees a high

level of control over the content of O, since for each element of the actual answer set, there has to appear an equal element within the expected set of answers and vice versa. That is, if the content of O changes in a way, such that the result set of the formalized query changes, this can always be recognized by means of an appropriate complex test. Nevertheless, one should be aware of the fact, that a full match is only applicable if all expected answers of the formalized query are known at the time of testing (see Section 2.2.2).

No Match. A match value intending no match, indicates that the expected answer set and the actual answer set must be disjoint in order to pass a test. This kind of match is intended to state, that a set of query answers, must not be derivable by O. A match-value intending no match does not assume all items of an answer set to be known at the time of reasoning. Hence, under a certain circumstances it can be reasonable to apply a test determining no match, rather than a full match. For example in order to state that two concepts $Dolphin$ and $Whale$ are never subconcepts of $Fish$, it is rather inefficient to enumerate all fishes, even if all subconcepts of $Fish$ are known at the time of testing.

Intersected Match. An intersected match assumes the expected answer set and the actual result set to share at least one element. For example, we assume O to specify the structure of an enterprize. This enterprise consists of several employees, and one of them has to be the CEO. In order to ensure, that the CEO actually is one of the employees, one can apply a complex test. This test assumes an intersected match: Its expected answer set is composed of all employees and the query is defined in a way, such that the actual result set is supposed to contain the CEO of the enterprize. Consequently, this test will only pass, if the CEO is indeed an employee of the enterprize.

Plug-in Match and Subsume Match. A match is meant to be a plug-in match, if the set of expected answers is a non-empty subset of the set of actual answers. This kind of match is especially useful when the expected answer set of query is partially unknown.

A subsume match is the counterpart of a plug-in match. In particular a match is denoted as partial, if the answer set of a query is a non-empty subset of the expected answer set. For example, in order to ensure that the units of an enterprize consist only of employees of the enterprize, an according complex test can be applied. This test considers a subsume match: The query is defined in a manner, such that the expected result set is be composed of the employees of a certain unit, and the expected answer set contains all employees of the enterprize. This complex test will only pass if the unit constitutes a subset of all employees (of the enterprize).

Since both, a plug-in match and a subsume match, assume the actual answer set to share at least one element with the expected result set, these kind of matches are always implying an intersected match as well.

2.3.2 Boolean Tests

A boolean test is composed of a query, that can only be evaluated to either true or false, as well as a boolean value that represents the expected answer of the query. Thus, a query which is used to serve a boolean test, does not contain any variables. Subsequent we denote such a query as 'ground query'. Due to the nature of a ground query, its result constitutes a single boolean value. In Section 2.3.1, we stated that the question, whether Nathalie has black hair or not, cannot be formalized in order to be used for a complex test. Instead of reformulating the query to work with a complex test (as suggested in Section 2.3.1), one can simply use a boolean test. If the result and the expected answer are equal, then the test will pass.

2.3.3 Consistency Checks, Constraints and Constraint Tests

Some ontology languages allow the creation of constraints. Constraints are used to make restrictions within an ontology. Since logic programming ontologies are based on Horn logic, the constraints encoded within an ontology, have to be translated into Horn logic. That is, a constraint within an ontology, has to be transformed into a Horn formula with an empty head (i.e. an integrity constraint). Consequently, a model that satisfies the body of such a Horn formula implies a constraint violation. By means of an integrity constraint it is possible to state, that two classes, Man and $Woman$, are disjoint (see Section 2.2.3). If an ontology imports another ontology (including some constraints), then the importing ontology has to be in line with the constraints of the imported ontology. Otherwise, the importing ontology passes for inconsistent.

In Section 2.2.3 we presented an approach that considered one lightweight and one heavyweight ontology instead of a single ontology. This approach can be further extended, with regard to imported ontologies into account. That is, the heavyweight version of an ontology imports the lightweight version of an ontology. Apart from that, heavyweight ontologies are only intended to import other heavyweight ontologies. Accordingly, lightweight ontologies do only import other lightweight ontologies. While the heavyweight ontologies are intended to be used for evaluation purposes, their lightweight counterparts can be used in order to perform fast query answering.

Constraint Tests. An ontology (its heavyweight version) should be tested at least once before being released. But still, there is no mechanism available that ensures that the constraints of an ontology are encoded correctly. In order to provide such a mechanism, we introduce constraint tests. A constraint test is related to an auxiliary test ontology T^C. This ontology imports a heavyweight ontology O. In order to test the constraints of the ontology O, T^C specifies a set of ontology elements that violate these constraints. The behavior of a constraint test depends on the ontology language of O, as well as on the implementation of the particular reasoner which is used to check the consistency of O. In particular, all reasoners are intended to evaluate wether an ontology is consistent or not. Nevertheless, some reasoners provide detailed information about the discovered inconsistencies (e.g. the id of violated constraints).

The simplest approach in order to test a constraint, assumes the usage of a reasoner to determine whether an ontology is consistent or not. That is, a constraint test passes, if the reasoner evaluates the associated ontology T^C to be inconsistent. Considering

this approach, T^C is at best containing only one ontology element, which is supposed to violate one constraint. This guarantees, that a constraint test only passes, if each element of T^C violates at least one constraint. Hence, the evaluation of n constraints implies the creation of n test ontologies T^C.

Nevertheless, one should be aware that in applying this approach there is no guarantee, that the violated constraint is the one which was aimed to be violated by an ontology T^C. That is, a constraint test can also pass, if another constraint than the intended one is violated.

The potential power of constraint tests is directly related to the features of the used reasoner. That is, if a reasoner returns particular information about a constraint violation, constraint tests can assimilate this information and apply it for testing purposes. Hence, more powerful reasoners allow the application of more powerful constraint tests.

For example, we consider a reasoner that detects some violated integrity constraint in O. This reasoner provides a function, in order to return the answer sets of the queries, which are related to the detected violated constraints (see Section 2.3.3).

Such a reasoner allows the application of a constraint test that constitutes an expected answer set, stated by the ontologist and similar to query based tests. The constraint test passes, if the answer set, returned by the reasoner, is equal to the expected answer set. Such a constraint test is very powerful, since there is no chance that it passes unintentionally.

The behavior of a constraint test is not exactly defined, since it is heavily related to the used reasoner. This means, there can be multiple different implementations of such constraint tests, based on different reasoning engines. Nevertheless, each of these test settings must at least be applicable in order to evaluate, wether an element of T^C violates some constraint of O, or not.

Applying a constraint test, one should be aware of the fact, that in case of an inconsistent ontology O (due to constraint violations) a passed constraint test does not necessarily imply, that all constraints of O are encoded correctly. For example, some constraint test may pass, whenever O is inconsistent. Thus, before applying a such a constraint test, the consistency of O has to be proven, as in case of an inconsistent ontology O, the application of the constraint test is inefficient.

Query Based Tests Simulating Constraints. The body of an integrity constraint can simply be translated to a query. For example, we consider an ontology that contains the constraint $\texttt{:-}Man(X), Woman(X)$. This constraint states that a man can never be a woman, and vice versa. An according query $Man(X), Woman(X)$ can be applied on a similar ontology that does not specify the constraint above. The result is of this query is a set of answers, where each answer contains a single value that is bound to X. If the answer set of the query is empty, then the constraint is fulfilled.

The fact that the body of an integrity constraint is a query, allows the application of query based tests instead of constraints. That is, in order to simulate the behavior of an integrity constraint, a complex test is composed of an adequate query and an empty expected result set. In a similar manner, if the body of the integrity constraint does not contain any variables, a boolean test can be applied. The expected result of

the boolean test constitutes therefore the boolean value $false$.

2.3.4 Test Suites

A test suite is a special kind of unit test. It constitutes a collection of unit tests, which are intended to be performed in a successive manner. That is, a test suit provides the opportunity to perform a certain combination of unit tests by simply executing the test suite itself. Furthermore, a test suite specifies, which ontology is to be tested.

Since a test suite is composed of other tests, it also may contain another test suite. In such a case, the included test suite does not refer to the ontology O, which is referenced by the originating test suite, but to an ontology O, which is referenced by the included test suite itself. The execution of a test suite does always imply a consistency check of O.

2.3.5 Comparison with Unit Testing by Entailed Axioms

In Section 2.2.1 we presented an approach in order to apply unit testing for DL ontologies. This approach assumes two test ontologies, T^+ and T^- as well as two conditions $O \models A_i^+ \forall A_i^+ \in T^+$ and $O \not\models A_i^- \forall A_i^- \in T^-$. Logic programming provides a different kind of expressivity than description logic. But apart from that, unit testing for logic programming ontologies offers similar features than unit testing by entailed axioms. For example a single axiom A_0^+, constituting an expression $Dolphin \sqsubseteq Mammal$, states that a concept $Dolphin$ has to be a subconcept of $Mammal$. The easiest way to build an according test using our approach, is the creation of a boolean test. This boolean test contains a query, to determine whether a dolphin is a mammal or not, and an expected result value $true$. Since both, the boolean test and the test considering the axiom A_0^+, assume the same configuration of the ontology in order to pass or not, we denote them as semantically equal. Furthermore, one might pose the question, whether there exists a test which is semantically equal to an axiom A_0^-, stating that $Dolphin$ is no subconcept of $Fish$. In order to create such a test, one has to formalize the query: 'Is a dolphin a kind of fish?'. The resulting formalized query can then be used to create a boolean test with an expected result value $false$.

2.3.6 Ontology Language as Test Language

The different kind of unit tests, presented in Section 2.3, form a unit testing framework for LP ontologies. One aspect of the implementation of this framework is to define a syntax to persistently save test definitions. This can be approached in two different ways. The first approach is based on an ontology language, which is used to implement the framework. The second approach considers the development of an abstract test language, that is intended for the creation of unit tests.

Ontology Language. Serializing test definition of the unit testing framework, using an ontology language, involves several advantages. One of them is, that the ontology language can be applied in order to encode unit tests. This saves time and

money, as an ontology engineer, which is already familiar with this language, has an ease in writing tests. Furthermore, existing ontology editors can be used in order to create unit tests.

What is still necessary now, is development of a testing environment. In the best case, if already existing software components (e.g. a parser) are available, the development can be limited to the creation of a graphical user interface.

But the use of an ontology language does not only involve more advantages. This kind of implementation enables additional opportunities for writing tests: e.g., a smart implementation can enable the semi-automated creation of expected answer sets. In Section 2.3.1 we showed an example for the application of an intersected match. This example considers an ontology O, which describes the structure of an enterprize. In order to evaluate whether an employee of the enterprize occupies the position of the CEO, we considered the creation of a complex test. The expected answer set of such a test has to include all employees of the enterprize, and there exist two approaches to achieve that. The first approach assumes the manual enumeration of all employees, each time before running the test. Since the staff of an enterprize is floating, this procedure is necessary in order to guarantee that every employee is captured. The other approach assumes an axiom to be created. This axiom states that the expected answer set contains all members of the enterprize. In contrast to the first approach, this approach opportunity assumes the implementation of the framework within an ontology language, that enables the writing of axioms.

Abstract Test Language. The development of an abstract test language implies some effort. But, once created, an abstract test language can be applied to test every ontology, no matter in which ontology language the ontology is encoded. This is a major advantage, as it enables the creation of cross-language tests. A disadvantage though is, that ontology elements (e.g. axioms), are not a priori available. Hence, the opportunity to create expected answer sets in a semi-automatic way (by means of additional axioms, as described before), is not available. Nevertheless, there exists an alternative approach to achieve an automatization. For this, the framework has to be extended: we consider the expected answer set of a complex test to be occasionally determined by a query Q. That is, the result set of Q represents the expected answer set of the test.

Additional problems do arise, if one tries to encode unit tests, that are based on the creation of auxiliary ontologies. In particular, we refer to constraint tests, as described in Section 2.3.3. Constraint tests assume an ontology T^C, which constitutes a set of ontology elements. These elements are intended to violate some constraints of O.

In order to build a test, similar to such a constraint test, the framework has to be slightly modified. We consider a test, which is further denoted as test language constraint test (TLCT). The behavior of this new test is similar to the behavior of a constraint test. That is, analogue to the ontology elements of constraint tests, the ontology elements of TLCTs are violating some constraints of O. The difference, between a constraint test and a TLCT, is, that the ontology elements of a TLCT are encoded within the abstract test language. Consequently, these terms have to be translated into an ontology language (on the fly), before the actual test is performed. Additionally, it is necessary to add the violating elements temporarily to O in order to achieve the desired effect.

2.3.7 Summary and Conclusion

Unit testing for logic programming ontologies assumes four different kind of tests: complex tests, boolean tests, constraint tests and test suites. These tests constitute a framework, which can be applied in order to evaluate every LP ontology.

Complex tests and boolean tests are based on the formulation of a query. Whether a test passes or not, depends on the actual and the expected result of that query. Complex tests and boolean tests are distinguished by the kind of query they support. That is, a complex test constitutes a query with variables, while a boolean tests assumes a ground query. The expected answer set of a complex test or a boolean test is specified accordingly. Complex tests and boolean tests can be applied, in order to simulate integrity constraints. Therefore the answer set of an according complex test is expected to be empty. In case of boolean test, that simulates a constraint, the expected answer is always $false$.

Whether a complex test passes or not, depends further on its match-value. The match-value determines the expected kind of matching between the answer set of the query and the expected answer set. There are five different kind of matches available. If no match is intended, the answer set and the expected answer set have to be disjoint. In contrast to that, a full match intends the answer set and the expected answer set to be equal. An intersected match implies, that the actual answer set and the expected answer set share at least one element. Plug-in matches and subsume matches are defined contrary: while a plug-in match indicates that the expected answer set is a non-empty subset of the actual answer set, a partial match assumes the actual answer set to be a non-empty subset of the expected answer set.

An ontology O should be tested by means of its heavyweight version (including all constraints) before being released. There have to be applied tests, which ensure O to be consistent, as well as tests, that evaluate whether the effect of the specified constraints meets the engineers expectations. The latter ones are supported by constraint tests. A constraint test considers an auxiliary ontology T^C, that imports the heavyweight version of O and contains a set of elements, which is intended to violate the constraints of O. The behavior of a constraint test is heavily related to the features of the used reasoner. That is, the conditions, which are applied in order to evaluate whether a constraint test passes, are based on the particular information (e.g. the amount of violated constraints) provided by the reasoner after checking O for consistency.

A test suite is a test, that constitutes a set of tests. These test are performed in a successive manner, whenever the test suite is executed. Furthermore a test suite specifies which ontology is supposed to be tested. The execution of a test suite (a test suite can be executed since it is a kind of test itself) implies always a consistency check of the ontology O.

Unit testing for LP and unit testing by entailed axioms have similar features. The latter approach assumes the entailment of ontologies, while unit testing for LP ontologies is heavily based on queries and their result.

In order to implement the unit testing framework for LP ontologies, an appropriate language has to be chosen. There are two approaches available: the first approach assumes the implementation of the framework by means of an ontology language. The second approach considers the development of an abstract test language in order to

implement the framework. Both approaches have advantages and disadvantages. An implementation, considering an ontology language, is easy to realize, if one knows the specific ontology language. Cross language tests, however, can only be supported by means of an abstract test language. The full implementation of the framework, as described in Section 2.3, can only be achieved, assuming an ontology language as implementation language. However, by applying minor changes to the framework, it can also be fully implemented within a test language. Consequently, the creation of an abstract test langauge that implements the unit testing framework, is at least as powerful as the implementation by means of an ontology language.

Chapter 3

Unit Testing for WSML

Within Section 2.3.6 we enumerated some advantages of implementing the unit testing framework by means of an ontology language. The aim of this chapter is to present an exemplary implementation of the unit testing framework, considering the Web Service Modeling Language[1] (WSML) as implementation language. In the course of that, an example use case is provided. The subsequent sections assume an advanced comprehension of WSML. We do not describe WSML in a detailed manner. Instead we refer to [1, Chapter 2, 3, 5, 6 and 8], which provides sufficient information, in order to understand the subsequent sections. For the specification of queries, the logical expression syntax of WSML is applied. Furthermore we present a graphical user interface (testing environment), which is intended to support the execution of unit tests. Subsequent, the ontology which is to be tested, is denoted as ontology O.

3.1 Supported WSML Variants

The Web Service Modeling Language comes in five variants: WSML-Core, WSML-DL, WSML-Flight, WSML-Rule and WSML-Full. Each of them provides a different level of expressivity. WSML-Core is the least expressive variant. This variant *"corresponds with the intersection of Description Logic and Horn Logic"* [1]. In particular, there are two branches, which are both based on WSML-Core. One branch extends WSML-Core in the direction of logic programming. This branch constitutes the variants WSML-Flight and WSML-Rule, which are defined in a stratified manner. That is, *"WSML-Rule extends WSML-Flight to a fully-fledged Logic Programming language"* [1]. The other branch extends WSML-Core in the direction of Description Logics, and contains therefore the variant WSML-DL, which is logically equivalent to $SHIQ$. WSML-Full reunifies the two branches. That is, WSML-Full is *"a superset of both WSML-Rule and WSML-DL. WSML-Full can be seen as a notational variant of First-Order Logic with nonmonotonic extensions"* [1]. It is out of the scope of this work,

[1] The Web Service Modeling Language is an ontology language, which is intended to support the semantic annotation of webservices. Particular information about WSML can be found in [1].

to implement unit testing frameworks for ontologies, which are based on other logical formalisms than logic programming. Thus, we do neither support unit testing for WSML-DL ontologies nor for WSML-Full ontologies.

3.2 An Ontology for Creating Unit Tests

The unit testing framework for LP ontologies (see Section 2.3) assumes four kinds of tests, namely complex tests, boolean tests, constraint tests and test suites. Within this section, we provide an exemplary implementation of this framework, considering WSML as implementation language. In particular, wa assume an ontology $OUnit$, which is supposed to specify appropriate elements for all kind of tests. Furthermore, this ontology specifies an abstract concept[2] $Test$, which is intended to represent the composition of all tests. Hence, the extraction of all tests can be performed by means of a simple query *'?test memberOf Test'*. The full specification of the ontology $OUnit$ can be found in Chapter 4. Figure 4.1 in Chapter 4 shows an UML representation of the ontology $OUnit$.

All concepts which are provided within this section, are specified within the ontology $OUnit$, under the default namespace $http://org.deri.wsml.unittesting\#$.

3.2.1 Complex Tests

A complex test is composed of a formalized query, a set of expected answers and a match-value (see Section 2.3.1). An adequate representation of a complex test considering WSML, can be achieved by means of a concept $ComplexTest$. This concept defines three attributes: $query$, $result$ and $matchValue$.

```
concept ComplexTest subConceptOf Test
    query ofType (1 1) _string
    result ofType (1 1)_string
    matchValue ofType (1 1) _string
```

Listing 3.1: The concept capturing complex tests

The attribute $query$ contains the string representation of the formalized query of a complex test. Moreover, the value of $query$ has to be compliant with the logical expression syntax as defined in [1, Section 2.8].

The value of the attribute $result$ states the query, which has to be executed in order to obtain the expected answer set. In particular, we recommend the specification of an additional relation, which is intended to represent the expected answer set. Therefore, by means of a query considering this relation, the expected answer set can be obtained. One should be aware of the fact that both, the value of $query$ and the value of $result$, have to consider the same names for variables. Otherwise the expected result set and the actual result set can never match, as they both contain different variable bindings.

The matching strategy between the result and the expected result of the query is specified by the attribute $matchValue$. That is, the attribute $matchValue$ represents the match-value of a complex test. There are five choices for a match-value: a full match,

[2] An abstract concept must not be directly instantiated.

no match, an intersected match, a plug-in match or a subsume match. Accordingly the value of $matchValue$ is either *full match*, *no match*, *intersected match*, *plug-in match* or *subsume match*. As there is no further choice for a match-value, the value of of $matchValue$ is restricted by a constraint.

```
axiom matchValueRestriction
    definedBy
    !−(?test memberOf ComplexTest and
    ?test[matchValue hasValue ?matchValue] and
    ?matchValue != "full match" and
    ?matchValue != "no match" and
    ?matchValue != "intersected match" and
    ?matchValue != "plug−in match" and
    ?matchValue != "subsume match").
```

Listing 3.2: The axiom restricting the attribute $matchValue$

3.2.2 Boolean Tests

A boolean test constitutes a ground query and a boolean value, which indicates whether the expected result of the ground query is either $true$ or $false$. In order to represent a boolean test, a concept $BooleanTest$ is considered. This concept is compliant with the description of a boolean test in Section 2.3.2. That is, it is composed of an attribute *query*, representing the ground query of a boolean test, as well as an attribute *result*, intending the expected result of the applied query.

```
concept BooleanTest subConceptOf Test
  query ofType (1 1) _string
  result ofType (1 1) _boolean
```

Listing 3.3: The concept representing boolean tests

3.2.3 Constraint Tests

Constraint tests, as defined in Section 2.3.3, assume an auxiliary ontology T^C. This ontology provides a composition of ontology elements, that are intended to violate a set of selected constraints of O.

The concept $ConstraintTest$ represents the composition of all constraint tests. That is, every concept representing a kind of constraint test, is assumed to be a subconcept of $ConstraintTest$. The concept $ConstraintTest$ provides an attribute *violatingOntology* that references T^C. According to [1] *"the sets of identifiers for the top-level elements, namely ontology, goal, webService, ooMediator, ggMediator, wgMediator and wwMediator, are pairwise disjoint and also disjoint from all other identifiers"*. Hence, there is no way to reference an ontology by means of an IRI. Therefore, we assume the value of *violatingOntology* to be a simple string. The concept $ConstraintTest$ is not intended to be instantiated in a direct manner.

```
concept ConstraintTest subConceptOf Test
    violatingOntology ofType (1 1) _string
```

Listing 3.4: The abstract concept capturing constraint tests

In Section 2.3.3 we stated that the behavior of a constraint test is heavily related to the applied reasoner. In order to provide an implementation for constraint tests, we assume the WSML2Reasoner framework[3], or a similar framework, to be applied. According to the WSML2Reasoner framework, we distinguish three kind of constraint violations. Namely attribute type violations, cardinality violations and user constraint violations.

Attribute Type Violations. Attribute type violations imply a constraining attribute type definition (see Section 2.2.4). In particular, the range of attributes can be specified in a constraining manner by means of the keyword $ofType$. Subsequent, we denote such attributes as 'constraining attributes'. If the assigned value of a constraining attribute is not known to be an instance of the types which are specified within the attributes range, then the constraining attribute is assumed as violated. That is, in case of an instance, assigning a value to such a constraining attribute, this value has to be in line with each and every type of the attributes range. An attribute type violation is represented by the class $AttributeTypeViolation$ within the WSML2Reasoner framework. This class provides access to detailed information about the violation. In particular, there are defined methods returning

- the attribute which is actually related to the violation,
- the type which would have been expected,
- the instance which assigns the violating value and
- the violating value itself.

In order to provide unit testing for constraining attributes, we introduce the concept $AttributeTypeTest$. Each instance of this concept represents an attribute type test. Attribute type tests are intended to check, whether an attribute type definition is specified in a constraining manner and if the range of the attribute constitutes the expected types. In order to provide sufficient information to run an attribute type test, the concept $AttributeTypeTest$ constitutes three attributes: $attribute$, $inst$ and $expectedType$.

The attribute $attribute$ references some attribute that is specified within the ontology O and intended to be violated by an element of the ontology T^C. Therefore, the ontology T^C is supposed to specify an instance that assigns a violating value the referenced attribute. This (violating) instance is then referenced by the attribute $inst$ of the concerning attribute type test. Optionally, the violated types of the attribute can be stated by means of the attribute $expectedType$. Therefore, one can choose, whether an attribute type test is supposed to be aware (aware attribute type test) of the violated types of an attribute, or not (unaware attribute type test). Attribute type tests, which are aware of violated types have to assign according values to the attribute $expectedType$. Such tests are assumed to pass only, if the referenced instance ($inst$) violates each and every expected type ($expectedType$) of the referenced attribute ($attribute$). Unaware attribute type tests though, do not care about a specific violated type. That is, an unaware test even passes, if any type within the range of the

[3]The WSML2Reasoner supports the application of different reasoners, such as KAON2 or MINS. Particular information about the WSML2Reasoner framework and the applicable reasoners can be found at http://tools.deri.org/wsml2reasoner.

referenced attribute is violated by the referenced instance. Moreover, only if the actual type of an assigned (potentially violating) attribute value is compliant with the range of the according attribute, then an unaware attribute type test will fail. Thus, one should be aware of the fact, that unaware attribute type tests, imply a certain margin for the developer. In order to show this, we consider an unaware attribute type test that passes. This test is assumed to test an attribute, which specifies the concept *Parent* within its range. If one replaces the concept *Parent* by one of its subconcepts (e.g. *Mother*) within the attributes range, then the unit test still passes, since it is unaware of the violated types and does not recognize the change therefore.

In the case, that an attribute which is to be tested, is specified in an inferring manner, then neither of the tests (i.e. aware or unaware attribute type test) passes.

```
concept AttributeTypeTest subConceptOf ConstraintTest
    attribute ofType (1 1) _iri
    inst ofType (1 1)_iri
    expectedType ofType (0 *) _iri
```

Listing 3.5: The concept capturing constraining attribute type tests

Instances of T^C, that indicate type violations but are not referenced by any member of *AttributeTypeTest*, have no effect to the test result. However, a graphical user interface supporting this framework is recommended to show an adequate warning to indicate unreferenced instances of T^C. Superfluous instances decrease the performance of reasoning. Therefore it is advisable to remove them immediately. Moreover, a warning turns out as helpful, if one has simply forgotten to write a test for a violating instance.

Cardinality Violations. The variants WSML-Flight, WSML-Rule and WSML-Full support the specification of cardinality constraints. Cardinality constraints can be divided into minimum cardinality constraints and maximum cardinality constraints. While minimum cardinality constraints restrict the minimum number of assigned attribute values, maximum constraints restrict the maximum number of value assignments. Consequently, the number of assigned values is expected to be within the resulting range. Otherwise, the cardinality of an according attribute is assumed as violated.

The WSML2Reasoner framework defines two classes $MaxCardinalityViolation$ and $MinimumCardinalityViolation$, indicating a maximum constraint violation and a minimum constraint violation respectively. Both classes provide methods for accessing information about the involved violating instance[4] as well as the involved restricted attribute[5].

In order to test cardinality constraints, we introduce the concepts $MinMaxCardinalityTest$, $MinCardinaltiyTest$ and $MaxCardinalityTest$. Each of these concepts specifies two attributes *inst* and *attribute*. The attribute *inst* is related to the violating instance(s), while the restricted attributes (which are intended to be violated) are represented by the values of *attribute*.

An instance of $MinMaxCardinalityTest$ is intended to test both, the minimum cardinality constraints and the maximum cardinality constraints of an attribute. Accord-

[4] An instance assigning an inappropriate amount of values to an attribute.
[5] An attribute, which is restricted by cardinality constraints.

ingly, such an instance must assign two values to the attribute *inst*. Each cardinality constraint of an attribute is violated once or never. Consequently, if both violating instances assign some values to a certain attribute, then one instance is assumed to violate the minimum cardinality constraint of the attribute, while the other one is intended to violate the maximum cardinality constraint of the attribute. This kind of constraint test passes, if both cardinality constraints of each referenced attribute are violated by all referenced instances. If some attributes are violated unintentionally by the violating instances, then the test will fail. Since the set of expected violated attributes can be empty, one can even state that an instances must not violate any cardinality constraints. This is useful, if a cardinality constraint of an attribute must not be changed in a manner, that the range between the minimum cardinality constraint and the maximum cardinality constraint of the attribute becomes smaller. Cardinality tests which are represented by instances of $MinMaxCardinalityTest$, are only applicable on attributes that specify a minimum cardinality constraint (greater than zero) and a maximum cardinality constraint (less then infinite).

```
concept MinMaxCardinalityTest subConceptOf ConstraintTest
    attribute ofType (0 *) _iri
    inst ofType (2 2) _iri
```

Listing 3.6: The concept capturing cardinality tests that target minimum and maximum constraint violations

Alternatively, the concepts $MinCardinaltiyTest$ and $MaxCardinalityTest$ can be applied. Both concepts are strictly related to one violating instance. This is reasonable, as two violating instances (intending the same membership) do not change the result of the test. Whether this test passes or not, depends on the cardinality violations which are encountered by testing the consistency of T^C. In particular, a maximum or minimum constraint test passes, if the according cardinality constraint of each referenced attribute is violated by the intended instance. If it is the case, that the cardinality constraint of an attribute is violated unintentionally by the referenced instance, then the test will fail.

```
concept MaxCardinalityTest subConceptOf ConstraintTest
    attribute ofType (0 *) _iri
    inst ofType (1 1) _iri

concept MinCardinalityTest subConceptOf ConstraintTest
    attribute ofType (0 *) _iri
    inst ofType (1 1) _iri
```

Listing 3.7: The concepts capturing minimum and maximum cardinality tests respectively

User Constraint Violations. A user constraint is an integrity constraint, which was specified by means of an axiom. As an example we consider the axiom $matchValueRestriction$ which was intended to restrict the match-value of complex tests.

The WSML2Reasoner framework specifies two kind of user constraint violations. Namely named user constraint violations, which are represented by instances of the class $NamedUserConstraintViolation$ and anonymous user constraint violations, which are represented by instances of the class $UnNamedUserConstraintViolation$. The

major difference between those kind of user constraints is, that named user constraints refer to axioms which are identified by an IRI while anonymous user constraint violations assume an anonymous identifier.

Accordingly, only the class $NamedUserConstraintViolation$ provides a method which returns the related axiom. Instances of $UnNamedUserConstraintViolation$ do not provide any details on the violation. Each and every user constraint can be violated only once. That is, even if two entities violate a specific user constraint, only one constraint violation is triggered. In order to provide unit testing for user constraints, we introduce the concept $UserConstraintTest$. This concept specifies two attributes $axioms$ and $numberOfViolations$.

The attribute $axioms$ represents the set of all named user constraints which are expected to be violated. That is, each value of $axiom$ constitutes the IRI of an axiom. The use of this attribute is not mandatory. Although, a test considering values for that attribute, only passes, if the expected set of violated named user constraints matches the evaluated set of violated named user constraints.

Furthermore, an instance of $UserConstraintTest$ assumes the assignment of exactly one value for the attribute $numberOfViolations$. This value indicates the expected number of violated user constraints (axioms). The attribute $numberOfViolations$ was added to enable the evaluation of anonymous user constraints. Therefore it includes the number of all anonymous as well as the number of all named user constraints, that are supposed to be violated some elements of the referenced ontology T^C. A user constraint test passes only, if the expected number of violated user constraints matches the evaluated number of violated user constraint. Nevertheless, one should be aware of the fact, that this method underlies some restrictions. That is, it can not be granted that the set of intended violated anonymous constraints corresponds to the evaluated set of violated anonymous constraints. Considering a test where n constraints are expected to be violated and indeed n violations were counted, the test still pass even if one of the counted violations was triggered by an unintended axioms.

```
concept UserConstraintTest subConceptOf ConstraintTest
    axioms ofType (0 *) _iri
    numberOfViolations ofType (1 1) _integer
```

Listing 3.8: The concept capturing user constraint tests

3.2.4 Test Suites

We consider a test suite to be an ontology, that is referenced by an instance of $TestSuite$. Thus, instances of $TestSuite$ provide a container for test suites. In effect, they allow the successive execution of multiple test suites. Therefore, the concept $TestSuite$ defines an attribute $testSuite$. The actual ontologies, which represent the test suites, are referenced by this attribute. The values of this attribute are considered to be string representation of IRIs (for the same reasons as the ontology T^C is referenced by means of a string representation of its IRI - see Section 3.2.3). Each ontology which is a test suite, has to import the ontology O. A test suite only passes, if each and every contained test passes and the ontology O is evaluated to be consistent.

```
concept TestSuite subConceptOf Test
```

```
testSuite ofType(1 *) _string
```

Listing 3.9: The concept capturing test suites

3.2.5 Summary and Conclusion

The abstract framework for unit testing, as described in Section 2.3, was implemented by means of the WSML. Therefore, the ontology *OUnit* constitutes a metamodel for unit tests, which can be used to evaluate WSML ontologies, as long as these ontologies are based on logic programming. Thus the concepts, that are defined by the WSML implementation of the unit testing framework, aim the WSML variants WSML-Core, WSML-Flight and WSML-RULE, but not WSML-DL or WMSL-Full.

Complex tests and boolean tests are implemented according to the framework, which is described in Section 2.3. The expected answer set of a complex test can be obtained by performing the query, which is stated by means of the attribute *result*.

The implementation of the unit testing framework assumes, that the WSML2Reasoner is used in order to reason over an ontology. Accordingly, there are distinguished five major concepts in order to test constraining attribute types (*AttributeTypeTest*), cardinality constraints ($MinMaxCardinalityTest$, $MinCardinalityTest$, $MaxCardinalityTest$) and user constraints ($UserConstraintTest$).

An instance of *AttributeTypeTest* can be applied in order to test, if the range of an attribute is specified in a constraining manner ($ofType$) and constitutes the expected types. Instances of $MinMaxCardinalityTest$ aim the evaluation of minimum and maximum cardinality constraints. Instances of $MinCardinalityTest$ and instances $MaxCardinalityTest$ are applicable in order to evaluate the minimum and maximum constraint of an attribute respectively. Constraints, that are defined by means of an axiom, can be tested by means of instances of the concept $UserConstraintTest$. All constraint tests reference an auxiliary ontology T^C, which is assumed to specify a set of constraint violating ontology elements.

We do not expect an ontologist to add a constraint test for each cardinality constraint or constraining attribute type definition. Instead, these kind of constraint tests are intended to be applied on certain strategic points to emphasize the fact, that the type definition of an attribute must be constraining for some reason, or that the cardinality of an attribute has to be constraint, whatever happens. For example, we consider an ontology that represents the domain of families. This ontology may specify a concept *Child* with an attribute *hasParent*. If the ontology engineer initially distinguishes between biological and social parents, a maximum amount of four parents can be assumed (i.e. two biological parents and two social parents). During maintenances, an ontology developer may see the according maximum constraint and simply guess, that it was specified unintentionally. Hence, the value of the maximum constraint gets changed, such that every child is restricted to have two parents only. To avoid such mistakes, a simple unit test (in addition to a well written documentation) turns out sufficient. That is, if an initial ontologist has defined unit tests, then a maintaining developer can test, whether his manipulations changes the semantics of an ontology in an undesired manner (see 2.1.2).

Each instance of *TestSuite* is defined as an instance of *Test*. Though, one should be aware of the fact, that instances of *TestSuite* are containers for test suites, rather than test suites themselves, since they do not specify which ontology is to be tested. Instead, they reference some ontologies, which are test suites according to the definition in Section 2.3.4. An ontology that represents a test suite, must import the ontology which is to be tested.

3.3 An Example Use Case in WSML

This section aims the application of the methodology of Section 1.3.3 in combination with unit testing for LP ontologies in order to create a new WSML ontology. That is, some processes of the methodology are used in a modified manner in order to serve unit testing. In the subsequent sections, we consider the situation of a hospital serving kidney transplantations, as well as tests, in order to determine ones fatherhood.

3.3.1 Identification of Motivating Scenarios

The success of a transplantation is heavily based on the compatibility between the donor and the receiver of a kidney. Since the probability of a potential organ repulsion is less, if the organ donor is directly related to the receiver (i.e. one of them is a parent) or at least a sibling, the ontology is supposed to serve according relationships. Although, not the social relationship of the involved persons is decisive, but the biological one. The same procedure, which is used to determine biological relationships to serve transplantations, can even be applied to support tests that evaluate ones fatherhood. Furthermore, it can be excluded that two persons are siblings, if they have not the same parents.

It is out of the scope of this work to create an ontology, supporting the complexity of gene tests. Instead, the direct biological relation between two persons, can be excluded by comparing their blood groups. Tests, determining whether two persons are biological related, work always with exclusion procedures. Hence, in the case, that the result of such a test affirms two persons to be not directly related, then they are indeed not directly related. Unfortunately, this does not work vice versa. That is, if a biological relationship can not be excluded, then there is not necessarily one. Since blood group tests evaluate less people to be not related than gene tests, the application of gene tests is more common in practice. Although for the purpose of this work, an ontology for blood group tests is totally sufficient.

The blood group 0 with rhesus factor negative is very rare. Unfortunately, this special configuration restricts one to receive only blood from a person with 0-. Therefore, the ontology is also supposed to serve constructs in order to evaluate the comparability between the blood groups of two persons.

3.3.2 Elaboration of Informal Competency Questions

Keeping the scope of the ontology in mind, we pose a small set of competency questions. These questions already determine the structure of the arising ontology. That is, after

the ontology is finished, it must be possible to answer the following questions by means of a reasoner.

- Is it possible that one person is the parent of another one?
- Who are the social parents of a person?
- Who is a sibling (as defined by law) of which person?
- Who is a social parent of a person, but not a biological one?
- Who is a biological half-sister or half-brother of whom?
- Which person is compatible to donate blood for another one?

3.3.3 Specification of the Informal and the Formal Terminology

The terminology of the ontology is specified, considering the competency questions as a source of inspiration. In practice, the process of gathering an ontology's terminology is mostly supported by a set of ontology capturing techniques (see Section 1.3.3). In order to keep the example ontology simple, we omitted the application of such techniques.

The identification of the terminology resulted in five concepts. In particular, we assume the specification of the concepts $Person$, $Father$, $Mother$, Man, $Woman$, $BloodGroup$ and $RhesusFactor$. As each person has a name, an age, a blood group, a rhesus factor and a social parent, we considered the according attributes $hasName$, $hasAge$, $hasBloodgroup$, $hasRhesusFactor$ and $hasParent$ respectively. The attribute $hasChildren$ is intended to state the (social) relationship between a parent and its children.

Additionally, the relations $mayBeParentOf$, $onlySocialParent$, $socialSiblings$, $halfSiblings$ and $mayGiveBloodTo$ are considered to be part of the resulting ontology. In particular, the relation $mayBeParentOf$ states, that due to their blood groups, two persons can be the parents of a third person. Furthermore, the relation $onlySocialParent$ is assumed to indicate, that a person is a social parent of another one, but not a biological parent. The relation $socialSiblings$ states that two persons are siblings as defined by law. That is, they do not necessarily have the same biological parents. Furthermore, the relation $halfSiblings$ supports the existence of siblings which are biologically related due to one parent, rather than both. Therefore, we assume that the social mother is always the biological mother of a child, if not explicitly stated different. Whether one can be a blood donor for a person or not, is assumed to be expressed by the relation $mayGiveBloodTo$.

The blood groups are represented by the instances A, B, AB and $Null$. The rhesus factor can either be positive or negative. Thus, according instances of $RhesusFactor$ are provided: $RhesusPositive$ and $RhesusNegative$.

3.3.4 Formalization of Competency Questions

In Section 1.3.3 we stated, that according to Uschold and Grüninger, *"the competency questions are defined formally as an entailment or as a consistency problem with respect*

to the axioms in the ontology" [12]. In particular, they expressed more formally:

- Determine $T_{ontology} \cup T_{ground} \models Q$
- Determine whether $T_{ontology} \cup T_{ground} \not\models \neg Q$

Considering these sentences, Q represents some first-order sentences using only predicates, $T_{ontology}$ is the set of axioms[6] in the proposed ontology, and T_{ground} specifies a set of instances [12].

We adapt the process of formalizing the competency questions in order to support unit testing. That is, the informal competency questions are formalized as queries by means of the logical expression syntax of WSML. Then, this formalized competency questions are used in order to create unit tests. Therefore, the ontology *AllTests* is assumed to specify an instance of *TestSuite* (see Section 3.2.4). This instance references the ontology $http://biology\#DonorCompetencyQuestionTests$, which is further used as a test suite. The ontology, which is to be tested, is identified by the IRI $http://biology\#DonorOntology$.

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml−syntax/wsml−flight"
namespace { _"http://biology#",
    unit _"http://org.deri.wsml.unittesting#"
 }

ontology AllTests
importsOntology unit#OUnit

instance donorTestSuites memberOf unit#TestSuite
    unit#testSuite hasValue "http://biology#DonorCompetencyQuestionTests"
```

Listing 3.10: The ontology, specifying the test suite.

According to the six competency questions, the ontology *DonorCompetencyQuestionTests* specifies a set of unit tests and their expected results. These unit tests are based on the formal terminology, which was defined in Section 3.3.3. Furthermore, the ontology *DonorCompetencyQuestionTests* imports the ontology *DonorTestInstances*, which is intended to serve a set of instances, extending the ontology *DonorOntology*. This set of instances is required for testing purposes, similar to the set of instances that is represented by term T_{ground}. The source code of these ontologies can be found in Chapter 4.

3.3.5 Specification of Formal Axioms

Axioms are used in order to specify the definitions of terms and constraints (see Section 1.3.3) of an ontology. In particular, we assume the axioms of an ontology to be specified in a manner, that they fulfil the unit test and therefore satisfy the competency questions. Before encoding new axioms, we assume an ontologist to specify further unit tests that support the axioms evaluation. This is compliant with the idea of unit testing in software development (see Section 1.4). Nevertheless, we do not expect an ontologist to add a constraint test for each cardinality constraint or constraining attribute type definition (see 3.2.5). Instead, constraint tests are intended to be applied

[6]The term 'axiom' is used according to [12]. That is, an axiom constitutes a set of concepts and core axioms.

on strategic points in order to emphasize, that some ontology elements and features must not be changed. The application of user constraint tests, however, is supposed to be handled contrary. That is, each user constraint is assumed to be guided by a respective unit test.

In order to create further tests, we assume two more test suites. Therefore, the test suite *DonorConstraintTests* is intended as a container for constraint tests, while the test suite *DonorRuleTests* is assumed to specify a set of tests, in order to evaluate axioms that do not constitute constraints. The instance *donorTestSuites* is extended accordingly.

In Section 3.2.3 we stated that an ontology T^C is needed in order to serve constraint tests. This ontology is represented by the ontology *DonorConstraintViolatingOntology*. According to the definition of T^C, the ontology *DonorConstraintViolatingOntology* composes a set of constraint violating ontology elements.

```
instance donorTestSuites memberOf unit#TestSuite
    unit#testSuite hasValue "http://biology#DonorCompetencyQuestionTests"
    unit#testSuite hasValue "http://biology#DonorConstraintTests"
    unit#testSuite hasValue "http://biology#DonorRuleTests"
```

Listing 3.11: The instance *donorTestSuites* specifies two more test suites.

The source code of the test suites as well as the final ontology *DonorOntology* are presented in Chapter 4. A short description of each test can be found on the top of its specification. Therefore, we omit a detailed explanation at this point.

3.3.6 Completeness Theorems.

According to Mike Uschold and Michael Grüninger, completeness theorems define the conditions under which the solutions of the competency questions are considered as complete [12]. More formally, they assume a completeness theorem to have one of the following forms:

- $T_{ontology} \cup T_{ground} \models \Phi$ if and only if $T_{ontology} \cup T_{ground} \models Q$
- $T_{ontology} \cup T_{ground} \models \Phi$ if and only if $T_{ontology} \cup T_{ground} \cup Q$ is consistent.
- $T_{ontology} \cup T_{ground} \cup \Phi \models Q$ or $T_{ontology} \cup T_{ground} \models \neg Q$
- All models of $T_{ontology} \cup T_{ground}$ agree on the extension of some predicate P.

Considering these sentences, Q specifies the query in the competency question as first order sentences, $T_{ontology}$ is the set of axioms in the proposed ontology, T_{ground} specifies a set of instances, and Φ defines the set of conditions under which the solution of a problem (competency question) is assumed to be complete[12].

Accordingly, we define an ontology to be complete, if all unit tests have passed. Therefore, the ontology which is to be tested has to be consistent internally, and with the additional instances, that are specified by the ontology *DonorTestInstances*. Furthermore, the integrity constraints, which are supposed to be tested by the constraint tests, have to be violated by the elements of the ontology *DonorConstraintViolatingOntology*. The formalized competency question are evaluated by means of unit tests. Consequently, if all unit tests have passed, also the competency questions are assumed to be satisfied.

3.3.7 Summary

Within this section a modified form of the formal methodology of Section 1.3.3 was applied in order to create an example ontology (*DonorOntology*). Considering an example scenario, the ontology *DonorOntology* targets a certain set of questions, that may arise in the health sector: (1) whether one person can denote blood to another one, (2) if one person may be the father of another one, (3) or if one person can be the donor of a kidney for another one.

In the course of developing the ontology *DonorOntology*, a set of unit tests was created. In particular, the test suite *DonorCompetencyQuestionTests* contains unit tests in order to evaluate, whether the elaborated competency questions are satisfied. A second bunch of unit tests constitutes the test suite *DonorRuleTests*, which is assumed to evaluate, whether the axioms of the tested ontology behave in the intended manner. Furthermore, the test suite *DonorConstraintTests* is intended for the evaluation of the constraints of the ontology *DonorOntology*. The ontology that is to be tested, is only assumed as complete if each and every constraint test passes.

3.4 A Graphical User Interface for Unit Testing

In Section 3.3.3 we described the creation of the ontology *DonorOntology*. In the course of that, we specified a set of unit tests, which are intended to evaluate this ontology. Each of the applied unit tests is compliant with the metamodel, which is specified by the ontology *OUnit*. In Section 3.2 we explained each kind of unit test and stated under which circumstances it passes. In particular, the execution of a unit test needs more operations than provided by the WSML2Reasoner framework (e.g. the evaluation of a complex test implies always the comparison of two answer sets). Therefore, within this section we describe an Eclipse plug-in (OUnit plug-in), which is intended to support the semi-automatic evaluation of WSML ontologies by means of unit testing.

An Eclipse plug-in is a software component that contributes to the IBM Eclipse framework[7]. The OUnit plug-in is built on the top of some other plug-ins, that constitute the Web Service Modeling Toolkit (WSMT). Therefore, we give a brief introduction to the WSMT. Furthermore, we explain the architecture of the OUnit plug-in and provide a user guide for the integrated graphical user interface (GUI). The implementation of the plug-in is based on the ontology *OUnit*. Therefore, in this section, the term ontology refers to the variants WSML-Core, WSML-Flight and WSML-Rule (see Section 3.1).

3.4.1 The Web Service Modeling Toolkit

The Web Service Modeling Toolkit (WSMT) is an integrated development environment that constitutes a set of software components (eclipse plug-ins), which are intended for the creation and maintenance of semantic Web Service descriptions encoded in the

[7]Particular information about the IBM Eclipse Framework can be found at http://www.eclipse.org.

WSML. The fact, that the WSMT is built on the top of the IBM Eclipse framework (and therefore plug-in based), allows one to extend it in an arbitrary manner.

The editors and views of the WSMT are grouped into three different Eclipse perspectives[8]. These perspectives are the Mapping, SEE and WSML perspective. The Mapping perspective includes a set of tools in order to create mappings between two ontologies and supports therefore the process of ontology mediation. Furthermore, the SEE perspective allows one to integrate Semantic Execution Environments like WSMX and IRSIII, which are applicable in order to perform automatic discovery, composition, selection, mediation, and invocation of semantic Web Services [8, page 7]. These two perspectives are not described in a detailed manner, since they are not directly related to the creation of ontologies and consequently not in the scope of unit testing for ontologies.

The OUnit plug-in provides a view (OUnit View) that contributes to the WSML perspective. This perspective targets the creation of semantic descriptions and therefore the development of ontologies. Currently there are provided three different editors in order to manipulate service descriptions (WSML Text Editor, WSML Visualizer and WSML Form Based Editor) as well as two views (WSML Reasoner View) in order to evaluate service descriptions. In particular, the WSML Reasoner View allows one to test an ontology by means of arbitrary queries. In a similar manner, the Discovery View is intended to evaluate, whether a goal matches a set of Web Services. That is the Reasoner View aids one in the development of ontologies, while the Discovery View supports an engineer in the creation of well defined goals and Web Services. Hence, the OUnit View has actually the same concern than the Reasoner View. Nevertheless, the concept of unit testing enables much more efficient testing scenarios for ontologies, the Reasoner View is still useful in order to debug an ontology, or simply test a query before applying it for a unit testing. Thus, the application of unit testing in combination with the functionalities provided by the other views of the WSML perspective, represents a major step towards quality engineering in the field of semantic Web Service development.

3.4.2 Architecture

In Section 3.4.1 we mentioned that the WSMT is based on the Eclipse framework and relies therefore on a plug-in based architecture. A plug-in based architecture allows one to compose a set of selected software components. That is, by installing new plug-ins or de-installing already integrated plug-ins, one can adapt a software, such that it meets ones requirements. Although, one should be aware of that a plug-in may be dependent on the existence of some other plug-ins. Thus, the installation of a single plug-in often results in the installation of a bunch of plug-ins.

In particular, the OUnit plug-in is dependent on a set of plug-ins of the IBM Eclipse framework and of the WSMT. These plug-ins are the *org.eclipse.ui*, *org.eclipse.ui.ide*, *org.eclipse.debug.ui*, *org.eclipse.core.runtime*, *org.eclipse.core.resources* as well as the *org.deri.wsml.eclipse*, *org.deri.wsml.eclipse.reasoner* and *net.sourceforge.wsmo4j* plug-in. The most important ones for unit testing are the *org.deri.wsml.eclipse.reasoner* plug-in and the *net.sourceforge.wsmo4j* plug-in.

[8] An Eclipse perspective constitutes a set of editors and views, that are used to achieve a certain goal.

While the first one includes the WSML2Reaoner framework, the latter one provides the libraries of WSMO4J. The plug-in internal architecture is compliant with the Model-View-Controller (MVC) paradigm[9] and therefore separates the model from the user interface.

The WSMO4J API. The WSMO4J API provides an object model that is compliant with the Web Service Modeling Ontology[10] [8, 5]. This object model is used by the WSMT in order to parse and serialize WSML documents. That is, on eclipse startup each WSML ontology is cached by means of this object model and therefore globally accessible for views an editors.

The WSML2Reasoner Framework The WSML2Reasoner framework was already mentioned in Section 3.2.3. There we stated, that this framework supports the application of different reasoners engines like KAON and MINS. The KAON reasoner is currently available for WSML-Core and WSML-Flight, but not for WSML-Rule. Nevertheless, this gap is closed by the MINS reasoner, which supports WSML-Core, WSML-Flight and WSML-Rule. By means of the preferences pages of the WSMT one can select the desired reasoner for each variant. The OUnit plug-in is aware of the (user specified) variant of a WSML document and applies the according reasoner. In the case, that the variant specification of a WSML document is missing, then the variant WSML-Rule (i.e. the most expressive testable variant) is assumed.

3.4.3 User Guide

The OUnit View supports all kind of unit tests, that are explained in Section 3.2. There is only one restriction concerning test suites, specifying that a test suite must not contain any other test suites. This restriction was established in order to avoid recursive configurations, where two test suites are referring each other directly or indirectly.

Running and Stopping a Test. There are multiple ways in order to run a test. One way considers a test to be performed by opening the context menu of a file in the package explorer and selecting the function *Run As - OUnit Test* (see Figure 3.1). In the case that the selected file contains an instance of $TestSuite$, all according test suites are performed immediately. The same can be achieved by means of the context menu of an editor. Therefore, the file which is currently edited must contain an instance of $TestSuite$.

Once a unit test was performed, it can be repeated by pressing the green button on the toolbar of the OUnit View. Furthermore, the context menu of the *Failure Tree* and the *Hierarchy Tree* (see Figure 3.2), allows one to perform even a subset of the displayed tests. The difference between the *Failure Tree* and the *Hierarchy Tree* is,

[9] Particular information about the Model-View-Controller paradgim can be found at http://en.wikipedia.org/wiki/Model_View_Controller.

[10] The Web Service Modeling Ontology (WSMO) provides a metamodel for languages, which are intended for the semantic annotation of Web Services (e.g. WSML).

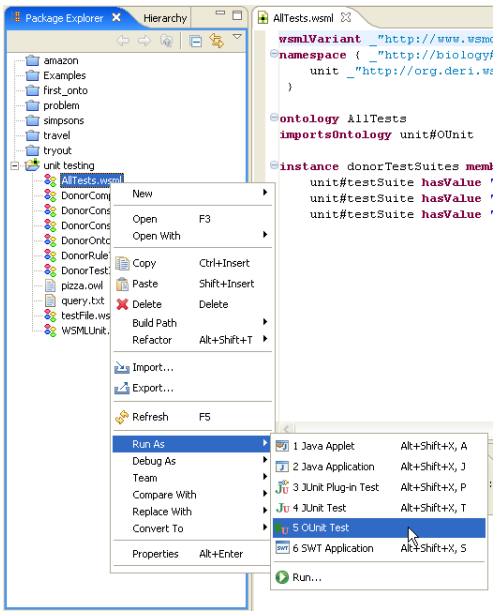

Figure 3.1: The context menu of a file in the package explorer showing the *Run As - OUnit Test* function.

that the Hierarchy Tree shows all tests, while the Failure Tree shows only tests which did not pass.

The performance of tests does not block the user interface. Since reasoning is a complex task, the execution of a set of tests may take a while. Therefore, the red button on the top of the OUnit View can be pressed in order to stop a test run.

Runs, Errors and Failures. The string on the top of the OUnit View which is labeled as 'Runs:', includes two integer values that are separated by a slash ('/'). The right value indicates the number of tests which are supposed to be performed, while the right number represents the number of tests that have already been finished. The left value is update simultaneously while running a set of tests. In particular, both integer values are aware of every kind of test, therefore also of ontologies that represent test suites.

An error indicates a problem with a test specification, while a failure implies, that an executed test failed due a problem in the ontology which is to be tested. In order to discover inappropriate test specification, a test suite is evaluated for consistency by means of the ontology *OUnit* (see Section 4).

Each test causing a problem is marked by a red crossed icon (in case of an error) or a blue crossed icon (in case of a failure) within the *Hierarchy Tree* as well as the *Failure Tree*. Moreover, the parents of such a test are marked by the same icon. If some test suite contains both, a set of error indicating tests as well as a set of failure indicating

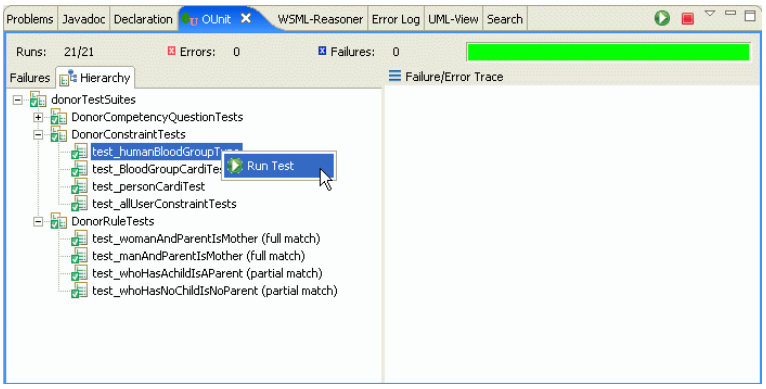

Figure 3.2: The context menu of the *Hierarchy Tree* showing the context menu in order to run a single unit test.

tests, it is marked as an error indicating test. A green icon indicates, that test has passed without any problems.

By selecting a problem causing test, an according error trace is shown on the right of the *Hierarchy Tree*. The selection between the *Hierarchy Tree* and the *Failure Tree* is synchronized. That is, if one selects an item of the *Failure Tree*, also the according element of the *Hierarchy Tree* gets selected. This does also work vice versa, as long as not a passed test is selected. By double-clicking on an item within the error trace, the problem causing element gets selected within an opened editor. In a similar manner, double-clicking on an item of the *Hierarchy Tree* or the *Failure Tree*, triggers the selection of the associated test within an opened editor.

The amount of detected errors and failures can be seen on the top of the OUnit View next the the according icons. Furthermore, the progress-bar of the OUnit View indicates the progress of a run. In particular, the progress-bar is colored green, as long as no problem occurs. However, as soon as problem is detected its color changes to red (see Figure 3.3).

Figure 3.3 shows the OUnit View after having detected a set of failures and errors. The selected test in the *Hierarchy Tree* is selected on the opened editor.

3.4.4 Summary

The WSMT allows one to edit WSML documents and therefore WSML ontologies. In order to provide a tool that supports unit testing for WMSL, the WSMT was extended in an appropriate manner. The WSMT already provided a set of plug-ins in order to evaluate semantic descriptions (e.g. WSML Reasoner View). Nevertheless, none of those plug-ins supports testing scenarios that are similar effective than the ones provided by the OUnit plug-in.

In particular, the OUnit plug-in is depended on a set of other plug-ins of the Eclipse IBM Framework and the WSMT. The most important of them are the

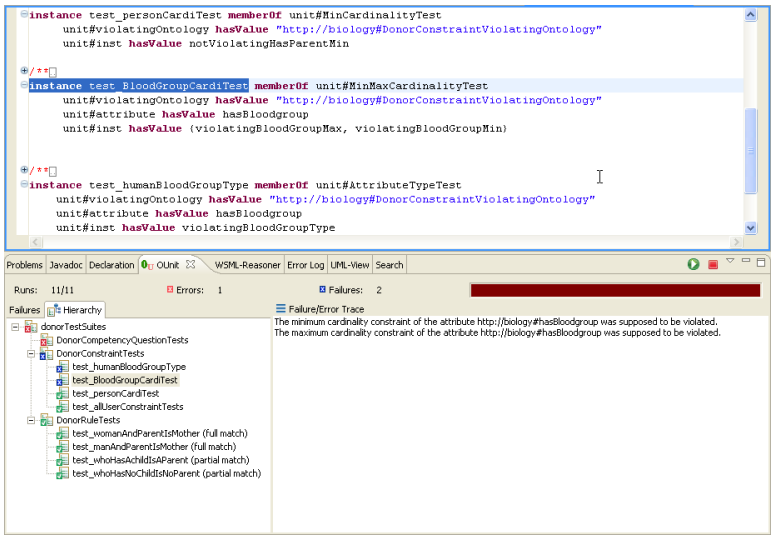

Figure 3.3: The OUnit View after a test run.

org.deri.wsml.eclipse.reaonser plug-in and the *net.sourceforge.wsmo4j* plug-in. While the first of those plug-ins works as wrapper for the WSML2Reasoner framework (which is needed for reasoning), the latter one contains the libraries of the WSMO4J API (which provides an object model for semantic descriptions).

The OUnit View provides a set of features in order to run a set of test suites or even a single test. Each test run can be interrupted by means of the red button on the top. Problems that occur while testing are grouped in errors and failures. While an error indicates, that the specification of a test is corrupt, a failure implies a problem within the ontology that is to be tested. The double-click on an item in the $FailureTree$ or the $HierarchyTree$, triggers the selection of the according test in an opened editor. The difference between the those two trees of the OUnit View is, that the $FailureTree$ only contains test, that failed (due to an error or a failure), while the $HierarchyTree$ shows the full test hierarchy.

Chapter 4

Related Work and Conclusion

The abstract unit testing framework that is explained in Section 2.3, is applicable to test every ontology that is based on logic programming. However, the idea of unit testing for ontologies is not new. Tools like Protégé [1] and OntoStudio [2] already support the evaluation of ontologies by means of unit testing. While unit testing in Protégé is based on the approach of Mike Uschold and Michael Grüninger [12], OntoStudio (version 2.0) provides so-called regression tests, which are similar to complex tests that consider a full-match strategy in order to compare the answer set and the expected answer set of an according query. Thus, OntoStudio partially implements the ideas of this work. Regression tests are encoded by means of an abstract test language based on the Extended Markup Language (XML). OntoStudio provides a user interface for the creation of queries and regression tests as well as a testing environment in order to run regression tests.

Unit testing is a powerful technique in order to evaluate ontologies. In particular, while creating the ontology *DonorOntology*, unit testing turned out as helpful in order to detect modeling faults, consistency problems and even simple spelling mistakes. Especially the maintenance of ontologies profits of the initial development of unit tests, since common and previous mistakes are already anticipated. Wether unit testing will establish as a common technique for the evaluation of ontologies, depends on the creators of ontology development tools. Therefore, tools like the WSMT, Protégé and OntoStudio are necessary and important in order to forward the progress of ontology engineering techniques and to improve the acceptance of ontologies in the community of entrepreneurs. That is, wether ontologies will be used for future industrial applications, depends not least on the existing development environments and their features.

[1] http://protege.stanford.edu/
[2] http://www.ontoprise.de

Appendix A

This chapter contains the ontologies which were developed while creating this document. These ontologies are the *OUnit*, *DonorOntology*, *DonorCompetencyQuestionTests*, *DonorConstraintTests*, *DonorRuleTests*, *DonorTestInstances* and *DonorConstraintViolatingOntology*.

The ontology *OUnit* defines a set of unit tests, that can be used to evaluate WSML ontologies which are compliant with the variants WSML-Core, WSML-Flight or WSML-Rule.

The ontology *DonorOntology* was created by means of the methodology, which was described in Section 1.3.3.

The three ontologies *DonorCompetencyQuestionTests*, *DonorConstraintTests* and *DonorRuleTests* constitute a set of unit tests in order to evaluate the ontology *DonorOntology*.

The ontology *DonorTestInstances* serves the necessary instances, which are necessary to perform unit tests. This ontology is imported by all test suites.

In Section 3.2.3 we stated, that an auxiliary ontology T^C specifies a set of ontology elements, in order to serve a set of constraint tests. In a similar manner, the ontology *DonorConstraintViolatingOntology* serves the constraint tests of the ontology *DonorConstraintTests*.

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-flight"
namespace { _"http://org.deri.wsml.unittesting#"
 }

ontology OUnit

concept TestSuite subConceptOf Test
      testSuite ofType(1 *) _string

concept Test

concept BooleanTest subConceptOf Test
      query ofType (1 1) _string
      result ofType (1 1) _boolean

concept ComplexTest subConceptOf Test
      query ofType (1 1) _string
      result ofType (1 1) _string
      matchValue ofType (1 1) _string

concept ConstraintTest subConceptOf Test
     violatingOntology ofType (1 1) _string

concept UserConstraintTest subConceptOf ConstraintTest
      axioms ofType (0 *) _iri
      numberOfViolations ofType (1 1) _integer


concept AttributeTypeTest subConceptOf ConstraintTest
     attribute ofType (1 1) _iri
     inst ofType (1 1)_iri
     expectedType ofType (0 *) _iri

concept CardinalityTest subConceptOf ConstraintTest
      attribute ofType (0 *) _iri
      inst ofType (2 2) _iri

concept MaxCardinalityTest subConceptOf ConstraintTest
     attribute ofType (0 *) _iri
     inst ofType (1 1) _iri


concept MinCardinalityTest subConceptOf ConstraintTest
     attribute ofType (0 *) _iri
     inst ofType (1 1) _iri

axiom matchValueRestriction
      definedBy
            !- ?test memberOf ComplexTest
  and ?test[matchValue hasValue ?matchValue]
  and ?matchValue != "full match"
  and ?matchValue != "no match"
  and ?matchValue != "intersected match"
  and ?matchValue != "plug-in match"
  and ?matchValue != "subsume match".
```

Listing 4.1: The ontology for creating unit tests.

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-rule"
namespace { _"http://biology#"
 }

ontology DonorOntology


concept Parent subConceptOf Person
     hasChildren ofType Person
```

```
concept Person
    hasName ofType (1 1) _string
    hasAge ofType (1 1) _integer
    hasParent inverseOf (hasChildren) ofType (0 2) Parent
    hasBloodgroup ofType (1 1) BloodGroup
    hasRhesusFactor ofType (1 1) RhesusFactor


concept Man subConceptOf Person

concept Woman subConceptOf Person

concept Father subConceptOf {Parent, Man}

concept Mother subConceptOf {Parent, Woman}

concept BloodGroup

concept RhesusFactor

instance A memberOf BloodGroup

instance B memberOf BloodGroup

instance AB memberOf BloodGroup

instance Null memberOf BloodGroup

instance RhesusPositive memberOf RhesusFactor

instance RhesusNegative memberOf RhesusFactor

relation mayBeParentOf(ofType Person, ofType Person, ofType Person)

relation onlySocialParent(ofType Parent, ofType Person)

relation socialSiblings(ofType Person, ofType Person)

relation halfSiblings(ofType Person, ofType Person)

relation mayGiveBloodTo(ofType Person, ofType Person)

relation matchingBloodGroups(ofType BloodGroup, ofType BloodGroup,
ofType BloodGroup)

relation matchingRhesusFactor(ofType RhesusFactor, ofType
RhesusFactor, ofType RhesusFactor)

relation dontateAbleBlood(ofType BloodGroup, ofType RhesusFactor,
ofType BloodGroup, ofType RhesusFactor)

/**
* This axiom states that everybody who has children is a parent.
*/
axiom isParentAxiom
    definedBy
    ?x memberOf Parent :-
    ?x[hasChildren hasValue ?child] and ?child memberOf Person.


axiom isMotherAxiom
    definedBy
    ?mother memberOf Mother :- ?mother memberOf {Woman, Parent}.


axiom isFatherAxiom
    definedBy
```

```
    ?father memberOf Father :- ?father memberOf {Man, Parent}.


axiom mayBeParentOfAxiom
    definedBy
    mayBeParentOf(?father, ?mother, ?child):-
    ?father memberOf Man and
    ?mother memberOf Woman and
    ?father != ?mother and
    ?mother != ?child and
    ?father != ?child and
    ?father[hasBloodgroup hasValue ?fatherBlodd] and
```

Listing 4.2: The ontology which was evaluated by means of unit tests.

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-rule"
namespace { _"http://biology#"
 }

ontology DonorTestInstances

instance abraham_simpson memberOf Man
    hasName hasValue "Abraham Simpson"
    hasAge hasValue 68
    hasChildren hasValue homer_simpson
    hasBloodgroup hasValue A
    hasRhesusFactor hasValue RhesusNegative

instance mona_simpson memberOf Woman
    hasName hasValue "Marge Simpson"
    hasAge hasValue 71
    hasChildren hasValue homer_simpson
    hasBloodgroup hasValue B
    hasRhesusFactor hasValue RhesusNegative

instance homer_simpson memberOf Man
    hasName hasValue "Homer Simpson"
    hasAge hasValue 40
    hasChildren hasValue {bart_simpson, lisa_simpson, maggy_simpson}
    hasBloodgroup hasValue AB
    hasRhesusFactor hasValue RhesusNegative

instance marge_simpson memberOf Woman
    hasName hasValue "Marge Simpson"
    hasAge hasValue 38
    hasChildren hasValue {bart_simpson, lisa_simpson, maggy_simpson}
    hasBloodgroup hasValue Null
    hasRhesusFactor hasValue RhesusPositive

instance bart_simpson memberOf Man
    hasName hasValue "Bart Simpson"
    hasAge hasValue 12
    hasBloodgroup hasValue B
    hasRhesusFactor hasValue RhesusPositive

instance lisa_simpson memberOf Woman
    hasName hasValue "Lisa Simpson"
    hasAge hasValue 9
    hasBloodgroup hasValue A
    hasRhesusFactor hasValue RhesusPositive

instance maggy_simpson memberOf Woman
    hasName hasValue "Margret Simpson"
    hasAge hasValue 1
    hasBloodgroup hasValue Null
    hasRhesusFactor hasValue RhesusPositive
```

Listing 4.3: The ontology which is supposed to provide additional instances for

testing purposes.

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-rule"
namespace { _"http://biology#",
    unit _"http://org.deri.wsml.unittesting#"
 }

ontology DonorCompetencyQuestionTests
/**
* - The ontology 'DonorOntology' is the ontology which is to be tested.
* - The ontology 'unit#OUnit' implements the metamodel (framework) for
* unit testing and is needed to check, wether the tests are specified
* in an appropriate manner.
* - The ontology 'DonorTestInstances' contains additional
* instances, which are intended for test purposes only.
*/
importsOntology {DonorOntology, DonorTestInstances, unit#OUnit}

/**
* The following test is created according to the competency question:
* 'Is it possible, that one person is the parent of another one?'
* Therefore, it is expected that the actual ontology, which is to be
* tested, specifies a relation 'mayBeParentOf'. This relation is
* intended to state, that a father and a mother may be the
* parents of a certain child, according to their blood groups.
*
* This test applies a full-match strategy in order to evaluate,
* wether the test passes or not. The expected result can be
* obtained by means of the query, stated by the attribute
* 'unit#result'. The relation, which is used by this query can
* further be found within this ontology.
*/
instance test_Query1 memberOf unit#ComplexTest
        unit#query hasValue "mayBeParentOf(?father, ?mother, ?child)"
        unit#result hasValue "testResult_Query1(?father, ?mother, ?child)"
        unit#matchValue hasValue "full match"

/**
* The following test is related to the same competency question than the
* test above ('test_Query1'). Although, by means of this test,
* it is explicitly stated, that Homer Simpson and Marge Simpson
* can not the parents of Maggy Simpson. This is simply a consequence
* of their blood groups (which are defined as part of the ontology
* 'DonorTestInstances').
*/
instance test_Query1_bool memberOf unit#BooleanTest
        unit#query hasValue "mayBeParentOf(homer_simpson, marge_simpson,maggy_simpson)"
        unit#result hasValue _boolean("false")

/**
* The following test is related to the competency question:
* 'Who are the social parents of a person?'
* Thus, the expected result constitutes a set of tuples, stating that
* a person (?parent) is the social parent of another one (?child).
*/
instance test_Query2 memberOf unit#ComplexTest
        unit#query hasValue "?parent memberOf Parent[hasChildren hasValue ?child]"
        unit#result hasValue "testResult_Query2(?parent, ?child)"
        unit#matchValue hasValue "full match"


/**
    The following test is related to the competency question:
    'Who is a sibling (as defined by law) of which person?'
*/
instance test_Query3 memberOf unit#ComplexTest
        unit#query hasValue "socialSiblings(?sibling1, ?sibling2)"
        unit#result hasValue "testResult_Query3(?sibling1, ?sibling2)"
```

```
        unit#matchValue hasValue "full match"

/**
* The following test is actually not necessary, as it is already
* covered by the test 'test_Query3'. Although, it was added in
* order to show, how to encode a unit test, which expresses
* that neither Abraham Simpson, Homer Simpson, Marge Simpson or
* Mona Simpson have siblings. The expected result set of this
* test is assumed to be empty.
*/
instance test_Query3_1 memberOf unit#ComplexTest
        unit#query hasValue "socialSiblings(abraham_simpson, ?s1) or
                             socialSiblings(homer_simpson, ?s1) or
                             socialSiblings(marge_simpson, ?s1) or
                             socialSiblings(mona_simpson, ?s1)"
        unit#result hasValue "testResult_Query3_1(?s1)"
        unit#matchValue hasValue "full match"

/**
* The following test is related to the competency question:
* 'Who is a social parent of a person, but not a biological one?'
*/
instance test_Query4 memberOf unit#ComplexTest
        unit#query hasValue "onlySocialParent(?parent, ?person)"
        unit#result hasValue "testResult_Query4(?parent, ?person)"
        unit#matchValue hasValue "full match"

/**
* The following test is related to the competency question:
* 'Who is a biological half-brother or half-sister of whom?'
*/
instance test_Query5 memberOf unit#ComplexTest
        unit#query hasValue "halfSiblings(?halfSibling, ?sibling)"
        unit#result hasValue "testResult_Query5(?halfSibling, ?sibling)"
        unit#matchValue hasValue "full match"

/**
* The following test is related to the competency question:
* 'Which person is compatible to donate blood for another one?'
* This test expects a partial match between the result and the
* expected result. Therefore, this test is very close to a real
* world scenario, where the enumeration of all opportunities is no
* option, due to the huge amount of data. Instead, some exemplary
* items are picked out, which constitute the expected
* result-set.
*/
instance test_Query6 memberOf unit#ComplexTest
        unit#query hasValue "mayGiveBloodTo(?person1, ?person2)"
        unit#result hasValue "testResult_Query6(?person1, ?person2)"
        unit#matchValue hasValue "subsume match"

/**
* This tests reflects the fact, that everybody can denote blood to
* himself. Hence, this test supports the test 'test_Query6'.
*/
instance test_Query6_1 memberOf unit#ComplexTest
        unit#query hasValue "mayGiveBloodTo(?person1, ?person2)"
        unit#result hasValue "?person1 memberOf Person and
                              ?person2 memberOf Person and
                              ?person1 = ?person2"
        unit#matchValue hasValue "subsume match"

/**
* This tests states, that nobody who has the blood group AB and
* rhesus factor positive can donate blood to anyone else, having a
* different blood group or rhesus factor. Similar to 'test_Query6_1',
* this test supports 'test-Query6'.
*/
```

```
instance test_Query6_3 memberOf unit#ComplexTest
        unit#query hasValue "mayGiveBloodTo(?person1, ?person2)"
        unit#result hasValue "?person1[hasRhesusFactor hasValue RhesusPositiv,
                                    hasBloodGroup hasValue AB] and
                              ?person2[hasRhesusFactor hasValue ?rf2,
                                    hasBloodGroup hasValue ?bg2] and
                              ?bg2 != AB and
                              ?rf2 != RhesusPositive"
        unit#matchValue hasValue "no match"


/**
* The following relations are intended to serve the expected results for some
* tests.
*/
relation testResult_Query1(ofType Person, ofType Person, ofType Person)
relation testResult_Query2(ofType Parent, ofType Person)
relation testResult_Query3(ofType Person, ofType Person)
relation testResult_Query4(ofType Parent, ofType Person)
relation testResult_Query5(ofType Person, ofType Person)
relation testResult_Query6(ofType Person, ofType Person)

/**
* The following axiom states that the relation between siblings is expected
* to be symmetric.
*/
axiom testResult_Query3_axiom
    definedBy
    testResult_Query3(?sibling1, ?sibling2) :-
    testResult_Query3(?sibling2, ?sibling1).

/*
*This axiom is used in order to create the respective relation
*instances for some complex tests.
*/
axiom results_axiom
    definedBy
    testResult_Query1(abraham_simpson, lisa_simpson, maggy_simpson).
    testResult_Query1(abraham_simpson, marge_simpson, maggy_simpson).
    testResult_Query1(abraham_simpson, marge_simpson, lisa_simpson).
    testResult_Query1(abraham_simpson, mona_simpson, homer_simpson).
    testResult_Query1(homer_simpson, marge_simpson, bart_simpson).
    testResult_Query1(homer_simpson, marge_simpson, lisa_simpson).
    testResult_Query1(bart_simpson, marge_simpson, maggy_simpson).
    testResult_Query1(bart_simpson, lisa_simpson, maggy_simpson).
    testResult_Query1(bart_simpson, mona_simpson, maggy_simpson).

    testResult_Query2(abraham_simpson, homer_simpson).
    testResult_Query2(mona_simpson, homer_simpson).
    testResult_Query2(homer_simpson, bart_simpson).
    testResult_Query2(homer_simpson, lisa_simpson).
    testResult_Query2(homer_simpson, maggy_simpson).
    testResult_Query2(marge_simpson, bart_simpson).
    testResult_Query2(marge_simpson, lisa_simpson).
    testResult_Query2(marge_simpson, maggy_simpson).

    testResult_Query3(bart_simpson, lisa_simpson).
    testResult_Query3(bart_simpson, maggy_simpson).
    testResult_Query3(lisa_simpson, maggy_simpson).

    testResult_Query3(bart_simpson, lisa_simpson).
    testResult_Query3(bart_simpson, maggy_simpson).
    testResult_Query3(lisa_simpson, maggy_simpson).
    testResult_Query4(homer_simpson, maggy_simpson).

    testResult_Query5(maggy_simpson, bart_simpson).
```

Listing 4.4: The ontology, which contains all tests, representing competency

question tests.

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-rule"
namespace { _"http://biology#" }

ontology DonorConstraintViolatingOntology
    importsOntology {DonorTestInstances, DonorOntology}

    /**
    * The ontology 'DonorOntology' is supposed to deny
    * the existence of a person, who is male as well as
    * female. Thus, the instance 'man_and_woman' is intended to
    * violate an according user constraint, in order to serve
    * a constraint test.
    */
    instance man_and_woman memberOf {Man, Woman}
        hasBloodgroup hasValue A

    /**
    * The instances 'frank' and 'enola' are supposed to violate
    * a constraint which states, that no child can be older than its
    * parent.
    */
    instance frank memberOf Man
        hasAge hasValue 68

    instance enola memberOf Woman
        hasAge hasValue 2
        hasChildren hasValue frank

    /**
    * The cardinality of 'hasParent' is set to a minim value of '0',
    * because it might happen, that ones parents are unknown and the
    * person is an orphan. This instances serves a cardinality test,
    * which emphasizes the fact, that the minimum cardinality constraint
    * of 'hasParents' must not be changed.
    */
    instance notViolatingHasParentMin memberOf Parent
        hasName hasValue "unknown"
        hasAge hasValue 3
        hasBloodgroup hasValue A
        hasRhesusFactor hasValue RhesusPositive

    /**
    * This instance is intended to violate the minimum constraint
    * of 'hasBloodGroup'.
    */
    instance violatingBloodGroupMin memberOf Parent
        hasName hasValue "unknown"
        hasAge hasValue 3
        hasRhesusFactor hasValue RhesusPositive

    /**
    * This instance is intended to violate the maximum constraint
    * of 'hasBloodGroup'.
    */
    instance violatingBloodGroupMax memberOf Parent
        hasName hasValue "unknown"
        hasAge hasValue 3
        hasBloodgroup hasValue {A, B}
        hasRhesusFactor hasValue RhesusPositive


    /**
    * This instance was intended to serve a test in oder to
    * evaluate, wether the type definition for the attribute
    * hasBloodGroup is constraining and the range of the
    * attribute meets ones expectations.
```

```
    */
    instance violatingBloodGroupType memberOf Human
        hasBloodgroup hasValue {"unsupported type"}
```

Listing 4.5: The ontology which contains a set of constraint violating elements. purposes.

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-rule"
namespace { _"http://biology#",
    unit _"http://org.deri.wsml.unittesting#"
 }


ontology DonorConstraintTests importsOntology {unit#OUnit,
DonorOntology, DonorTestInstances}

/**
* The instance 'allUserConstraintTests' is intended for testing the
* axioms of the ontology 'DonorOntology', that represent constraints.
*/
instance test_allUserConstraintTests memberOf
unit#UserConstraintTest
    unit#axioms hasValue {manDisjointWoman, parentHasChildConstraint}
    unit#numberOfViolations hasValue 2
    unit#violatingOntology hasValue "http://biology#DonorConstraintViolatingOntology"

/**
* This test fails, if one forces a person to have parents
* by means of a cardinality constraint.
*/
instance test_personCardiTest memberOf unit#MinCardinalityTest
     unit#violatingOntology hasValue "http://biology#DonorConstraintViolatingOntology"
     unit#inst hasValue notViolatingHasParentMin

/**
* Nobody can have more or less than one blood group. An accroding
* cardinality constraint is expected to be applied.
*/
instance test_personCardiTest memberOf unit#MinMaxCardinalityTest
     unit#violatingOntology hasValue "http://biology#DonorConstraintViolatingOntology"
     unit#attribute hasValue hasBloodgroup
     unit#inst hasValue {violatingBloodGroupMax, violatingBloodGroupMin}


/**
* This test is an example for an attribute type test.
* In particular, the attribute 'hasBloodGroup' of the concept
* 'Human' is tested in order to define its range in a constraining
* manner. The expected type (which constitutes the range)
* therefore is 'BloodGroup'.
*/
instance test_humanBloodGroupType memberOf unit#AttributeTypeTest
    unit#violatingOntology hasValue "http://biology#DonorConstraintViolatingOntology"
    attribute hasValue hasBloodGroup
    inst hasValue violatingBloodGroupType
    expectedType hasValue BloodGroup
```

Listing 4.6: The ontology which contains a set of constraint tests. purposes.

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-rule"
namespace { _"http://biology#",
    unit _"http://org.deri.wsml.unittesting#"
 }

ontology DonorRuleTests importsOntology {DonorOntology,
DonorTestInstances}
```

```
/**
* The ontology 'DonorOntology' includes a rule in order to
* state, that each person, who has at least one child is said
* to be a parent. The instance 'test_whoHasAchildIsAParent'
* is intended to test the rule.
*/
instance test_whoHasAchildIsAParent memberOf unit#ComplexTest
      unit#query hasValue "?parent memberOf Parent"
      unit#result hasValue "?parent = marge_simpson"
      unit#matchValue hasValue "subsume match"

/**
* This test passes, if Lisa Simpson, Bart Simpson and Maggy Simpson are
* not defined as parents, due to an axiom (rule) of the ontology
* 'DonorOntology'.
*/
instance test_whoHasNoChildIsNoParent memberOf unit#ComplexTest
      unit#query hasValue "?person memberOf Person and naf(?person memberOf Parent)"
      unit#result hasValue "?person = lisa_simpson or
                            ?person = bart_simpson or
                            ?person = maggy_simpson"
      unit#matchValue hasValue "subsume match"

/**
* The ontology 'DonorOntology' is supposed to define a rule,
* such that each woman, which is a parent, is also a mother.
* The instances 'marge_simpson' and 'mona_simpson' are
* expected to satisfy this rule.
*/
instance test_womanAndParentIsMother memberOf unit#ComplexTest
      unit#query hasValue "?person memberOf Mother"
      unit#result hasValue "?person = marge_simpson or
                            ?person = mona_simpson"
      unit#matchValue hasValue "full match"


/*
* The ontology 'DonorOntology' is supposed to define a rule,
* such that each man, which is a parent, is also a father.
* The instances 'homer_simpson' and 'abraham_simpson' are
* expected to stisfy this rule.
*/
instance test_manAndParentIsMother memberOf unit#ComplexTest
      unit#query hasValue "?person memberOf Father"
      unit#result hasValue "?person = homer_simpson or
                            ?person = abraham_simpson"
      unit#matchValue hasValue "full match"
```

Listing 4.7: The ontology which is supposed to provide tests for user defined axioms.

Appendix B

Figure 4.1: An UML representation of the ontology *OUnit*. The classes of the diagram are related to concepts. Accordingly, the attributes of the classes are related to attributes of the concepts. A 'subconcept-of' relationship between two concepts is shown by means of a generalization. The UML diagram is compliant with the mapping between WSML and UML as suggested in [11].

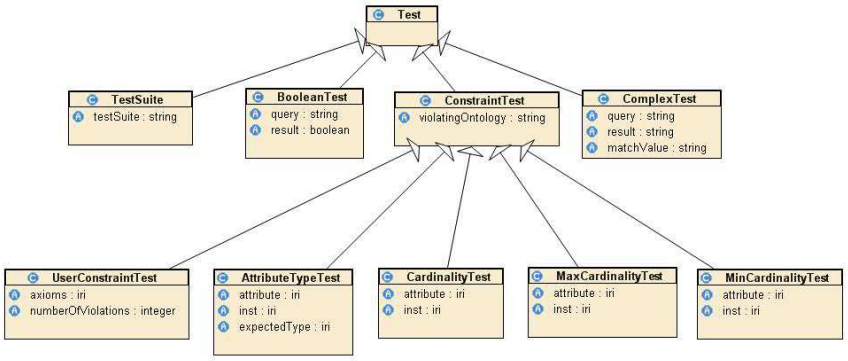

Figure 4.1: The UML representation of the ontology *OUnit*.

Bibliography

[1] Jos de Bruijn, Holger Lausen, Reto Krummenacher, Axel Polleres, Livia Predoiu, Michael Kifer, and Dieter Fenlser. The Web Service Modeling Language WSML, 2005.

[2] Jos de Bruijn, Holger Lausen, Axel Polleres, and Dieter Fensel. The Web Service Modeling Language WSML: An Overview. In *ESWC*, pages 590–604, 2006.

[3] Michael Felderer. Combining First-Order Logic Knowledge Bases and Logic Programming using fol-programs, 2006.

[4] Melvin Fitting. *First-order logic and automated theorem proving (2nd ed.)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[5] Asuncion Gomez-Perez, Oscar Corcho-Garcia, and Mariano Fernandez-Lopez. *Ontological Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[6] IEEE. IEEE Standard Glossary of Software Engineering Terminology, 1990. IEEE Standard 610.12.

[7] IEEE. IEEE Standard for Developing Software Life Cycle Process, 1996. IEEE Standard 1074-1995.

[8] Mick Kerrigan, Adrian Mocan, Martin Tanler, and Dieter Fensel. The Web Service Modeling Toolkit - An Integrated Development Environment for Semantic Web Services. In *ESWC*, pages 789–798, 2007.

[9] Evren Sirin, Bernardo Cuenca Grau, and Bijan Parsia. Optimizing Description Logic Reasoning with Nominals: First Results. Technical report, Maryland Information and Network Dynamics Lab., 2004.

[10] Nathalie Steinmetz. WSML-DL reasoner, 2006.

[11] Martin Tanler. Visualizing WSML Using an UML Profile, 2007.

[12] Mike Uschold and Michael Grüninger. Ontologies: principles, methods, and applications. *Knowledge Engineering Review*, 11(2):93–155, 1996.

[13] Denny Vrandecic and Aldo Gangemi. Unit Tests for Ontologies. In *OTM Workshops (2)*, pages 1012–1020, 2006.