A FORMAL MODELING APPROACH TO ONTOLOGY ENGINEERING

MODELING, TRANSFORMATION & VERIFICATION

YUAN FANG LI

B.Sc.(Hons). NUS

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE

2006

Acknowledgement

I would like to take this opportunity to express my sincere gratitude to those who assisted me, in one way or another, with my Ph.D. in the last four years.

First and foremost, I would like to thank my Honor's Year Project and Ph.D. advisor Dr. Dong Jin Song for his never-ending enthusiasm, guidance, support, encouragement and insight throughout the course of my post graduate study. His diligent reading and insightful and constructive criticism of early drafts and many other works made this thesis possible.

To my fellow students, Chen Chunqing, Sun Jun and my cousin Feng Yuzhang – your friendship, collaboration and funny chit chat gave me inspiration and helped me go through the long and sometimes not-so-smooth ride of Ph.D. study.

To my former lab mates Dr. Sun Jing and Dr. Wang Hai – for your suggestions on all aspects of research works and generous hospitality.

I am indebt to Dr. Bimlesh Wadha and Dr. Khoo Siau Cheng for the valuable comments on an early draft of this thesis. Dr. Wadha, in particular, carefully reviewed the entire thesis and corrected many language errors. I am sincerely grateful to her for the time and effort put into this.

I am also grateful to the external examiner and many anonymous reviewers who reviewed this thesis and previous publications that are part of this thesis and provided critical comments, which contributed to to the clarification of many of the ideas presented in this thesis.

This thesis was in part funded by the "Defence Innovative Research Project – Formal Design Methods and DAML" by the Defence Science and Technology Agency of Singapore. The Advanced Study Institute of NATO Science Committee sponsored me for attending the 2004 Marktoberdorf Summer School. My gratitude also goes to Singapore Millennium Foundation and National University of Singapore for the generous financial support, in forms of scholarship, the President's Graduate Fellowship and conference travel allowance.

I wish to thank sincerely and deeply my parents who have raised me, taught me and supported me all these years and who always have faith in me.

Finally and most importantly, to my beloved wife Xing Meng Nan. Your ceaseless love, encouragement, patience and wonderful cooking have kept my morale and stamina high.

Contents

Summary

The Semantic Web has been regarded by many as the new generation of the World Wide Web. It enables software agents on the Web to autonomously and collaboratively understand, process and aggregate information by giving Web resources well-defined and machine-interpretable markups, in the form of ontologies.

Ensuring the correctness of ontologies is very important as inconsistent ontologies may lead software agents to reason erroneously. Such tasks are non trivial as the more expressive ontology languages are, the less automated are the reasoners/provers and with the growth of the size of ontologies, locating inconsistencies is also more difficult.

Further, as the expressivity of these languages is also limited in more than one way, certain desirable ontology-related properties cannot be expressed in these languages. The ability to express and check these properties will make ontologies more accurate and more robust. It is therefore highly desirable.

Dynamic Web services help make the Web truly ubiquitous. In the Semantic Web, service ontologies describe the capabilities, requirements, control structures, etc., of Web services. Their consistency must also be guaranteed to ensure the correct functioning of software agents.

Software engineering and in particular formal methods are an active and well-developed research area. We believe that mature formal methods and their tool support can contribute to the development of the Semantic Web. This thesis presents a formal modeling approach for verifying ontologies. By defining semantics of ontology languages in expressive formal languages, their proof tools can be used to ensure the correctness of ontology-related properties.

The validity of the above approach entirely relies on the correctness of the semantics of ontology languages in formal methods. Hence, the other important topic in this thesis is the proof of such correctness. An abstract approach using institutions and institution morphisms is employed to represent and reason about ontology languages and formal languages. An integrated tools environment is also presented to facilitate the application of the verification approach.

Key words: **Semantic Web, DAML+OIL, institutions, ontology, OWL, verification, Z, LSC**

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation and Goals

The World Wide Web (WWW) is a computer network where data is shared mainly for human consumption. Web contents are *visually* marked up by languages such as HTML, CSS, etc. The Web has been tailored for human consumption. The usefulness of the Web is limited by the fact that information cannot be easily understood and processed by machines.

Recent advances of XML [108] technology have separated the markup of contents of information from its layout. XML's characteristics, such as the separation of concerns, strict syntax well-formedness and the ability to allow user-defined tags permit for greater flexibility. However, with no mutually-agreed meaning for tag names, it is hard for information to be shared across organizational boundaries.

Proposed by Tim Berners-Lee *et al*, the Semantic Web [8] is a vision to extend the current World Wide Web so that Web resources are given well-defined, content-related and mutually-agreed meaning. The Semantic Web aims at realizing the full potential

of the Web by enabling software agents (intelligent software on the Web) to understand, process and aggregate information autonomously and collaboratively.

The realization of this vision depends on the ability to semantically markup Web resources, including both static data and dynamic Web services, by ontologies. Ontologies are formal specifications of conceptualizations [34]. Building on mature technologies such as XML, Unicode and URI (Uniform Resource Identifier) [7], the ontology languages are positioned in a layered "cake", as depicted in Fig. 1.1 by Tim Berners-Lee.

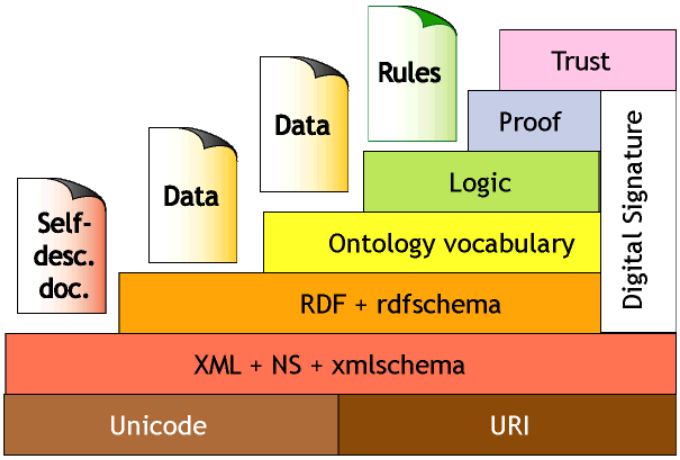

Figure 1.1: Generic architecture of the Semantic Web

Resource Description Framework (RDF) [68] and RDF Schema [17] are the foundation of the Semantic Web stack. They provide the core vocabularies and structure to describe Web resources. Based on RDF Schema and description logics (DLs) [74], the Web Ontology Language (OWL) [49] was developed and it provides more vocabulary for describing resources. Briefly, Web resources are categorized as *classes*, each of which holds a set of *instances*, pairs of which are related by *properties*.

Software agents' ability of autonomously understanding, processing and aggregating information builds on the decidability of the core ontology languages of the Semantic

Web. It is for this reason that DAML+OIL [101] and (a subset of) OWL were designed to be decidable [46, 40]. This is achieved by limiting their expressivity.

This design decision has made possible the construction of fully automated reasoning engines for ontologies written in these languages. However, certain desirable properties of resources cannot be represented by these languages due to the limited expressivity. This is mainly exhibited in the following two areas: expressivity limitation of the DL against first-order logic and the the dynamic nature of Web services.

Description logics are a very important knowledge representation formalism with a formal and rigid logical basis. They are a subset of first-order logic (FOL) [58] by carefully selecting only certain features to include. By limiting their expressivity, DLs are made decidable so that core reasoning services, namely concept subsumption, satisfiability and instantiation, can be solved in full automation. Being based on DL, ontology languages such as DAML+OIL and OWL are not expressive enough for certain complex ontology-related properties to be represented in these languages.

For example, consider the scenario of an ticket booking agent on the Semantic Web. It is very natural to express such a property that it should not book two tickets for any client with the durations of the two tickets overlap. Allowing booking only one ticket for a client is a possible, but overly restrictive solution. It is thus highly desirable that this information can be explicitly stated in the ontology and verified by reasoners.

In the light of this, the OWL Rules Language, (ORL) [47] (and its later version, the Semantic Web Rules Language (SWRL) [48]), a rules extension to OWL, was proposed to add Horn-style rules to OWL. Although SWRL extends the expressiveness of OWL, it is still limited in expressing certain properties. The correctness of these properties may, as we will see later in Chapter 2, have a significant impact on the validity of the ontology. Hence, the expression and verification of these properties are very

important.

Formal methods [16, 12, 11] have made significant development [41, 100, 62] and received much attention in both academia and industries. Z [89, 107] is a formal specification language designed to model system data and states. It is based on ZF set theory and first-order predicate logic. Therefore, Z is more expressive than ontology languages and it allows the specification of complex constraints which is not available in ontology languages. There are tools developed to support it. Z/EVES [84] is one such interactive proof tool for checking and reasoning about Z specifications.

Alloy [54], originally developed as a lightweight modeling language, is essentially aimed at automated analysis. Its design is influenced by Z but is less expressive. Alloy Analyzer [55] is a fully-automated tool for analyzing Alloy specifications with special model checking features, which are helpful to trace the exact source of errors.

Our earlier works [24, 27] showed that data-oriented formal methods and tools, e.g., Z/EVES and Alloy Analyzer, are capable of reasoning about ontologies. We also noticed the complementary reasoning capabilities among Z/EVES, Alloy Analyzer and Semantic Web reasoners such as FaCT++ [98] and RACER [36]. This motivated us to propose a combined approach [23] to using these tools in conjunction so that the synergistic reasoning power of these tools can be harnessed. By applying these tools systematically to an ontology, not only can we uncover more errors than using any one of them alone, inconsistencies can also be corrected more easily and precisely.

The effectiveness of the above combined approach relies on the soundness of the transformation from DAML+OIL/OWL ontologies to Z specifications. As these languages have different semantical bases, a higher-level device that is able to abstract and represent the underlying logics of DAML+OIL/OWL and Z is necessary to prove the soundness of the transformation. The notion of institutions [31] was introduced to formalize the concepts of "logical systems". Institutions provide a means of reasoning

about software specifications regardless of the logical system. We find the concept of institutions suitable for proving the soundness of our approach. It was observed that the underlying logical systems of DAML+OIL (OWL) and Z can be represented as institutions and further, by applying Goguen and Roşu's institution comorphisms [33], the soundness of the transformation from OWL (hence DAML+OIL) to Z can be proved.

Not all Semantic Web practitioners are experts in formal methods and they may find it difficult to interact with tools such as Z/EVES or Alloy Analyzer. An integrated tools environment is also conceived and designed to ease the application of the combined approach. The functionalities of this environment include systematic ontology creation, automatic ontology transformation, ontology querying, invocation of various reasoning tools, etc.

The above text highlights the issues related to the static aspect of the Web. However, the Web is more useful only if online services can be dynamically discovered and invoked to effect changes in the real world. The Semantic Web can also play a role by semantically marking up Web services to facilitate automatic service advertisement, discovery, invocation and composition. The OWL Services ontology (OWL-S) [95] is an OWL ontology that defines a core set of vocabularies to describe the Web services' capabilities, requirements, control constructs, etc. The dynamic nature of services makes the static reasoning techniques such as theorem proving insufficient. Live Sequence Charts (LSCs) [18] are a broad extension of the classic Message Sequence Charts (MSCs [53]). They rigorously capture communicating scenarios between system components. Play-Engine [38] is the tool support to visualize and simulate LSCs. In this thesis, we use LSC to represent OWL-S service process model ontologies and use Play-Engine to visualize and simulate them. This enables us to simulate and inspect the execution of services without actually implementing them.

1.2 Thesis Outline

This section gives an overview of the structure of the thesis.

1.2.1 Chapter 2 – Overview

Chapter 2 introduces background information on technologies, languages, tools and notations used in the presented work.

The Semantic Web languages take the central stage in this thesis. Hence, we first introduce the Semantic Web and the various ontology languages, such as RDF, RDF Schema, DAML+OIL, OWL, OWL$^-$ [56], SWRL, SWRL FOL [9] and WRL [1]. We present the syntax and semantics of the main language constructs, followed by a brief discussion on their tool support, including reasoners and visual editors.

Formal languages Z and Alloy are used extensively in the combined approach briefly introduced in the previous section. These languages together with their proof tools such as Z/EVES and Alloy Analyzer are also discussed and compared.

As a preparation for the discussion of the formal soundness proof of the transformation from ontology language OWL to Z using institutions, we present background information on category theory, institutions and institution morphisms.

Lastly, we introduce the OWL Services (OWL-S) ontology and the visual design language Live Sequence Charts (LSC). The visualization and simulation tool PlayEngine is also discussed to facilitate the presentation of the work later in Chapter 7 on simulating and checking Semantic Web services.

1.2.2 Chapter 3 – Checking DAML+OIL Ontologies using Z/EVES

Software engineering is a broad and well-developed research area over the past decades. We believe that mature software engineering languages and tools can contribute to the development of the Semantic Web vision. In this chapter, we demonstrate the ability of formal language Z in expressing Web ontologies and checking ontology-related properties. Specifically, we define the semantics of ontology language DAML+OIL in Z. By automatically transforming DAML+OIL and RDF ontologies into Z specifications, Core ontology reasoning services, namely concept subsumption, satisfiability and instantiation, checking can be performed in Z/EVES, a powerful theorem prover for Z.

It can be observed in this chapter that the proof process using Z/EVES is very interactive and requires substantial user expertise. This inspired us to propose a combined approach of checking Web ontologies to harness the synergy of Semantic Web and software engineering tools. This work is presented in the following chapter.

1.2.3 Chapter 4 – A Combined Approach to Checking Web Ontologies

As briefly discussed in Section 1.1, the trade-off between decidability and expressivity of ontology languages makes it awkward and difficult to represent certain complex properties in these languages. The newly proposed rules extension SWRL and SWRL FOL provide a partial remedy to this problem but they are still not as expressive as first-order predicate logic. Further, since they are undecidable languages, a reasoning engine to support full automation of all reasoning tasks would be an impossible task.

This shortcoming of DAML+OIL and SWRL led us to and propose to use Z to express complex properties inexpressible in DAML+OIL, OWL or SWRL. This makes it possible for Z proof tool such as Z/EVES to perform formal reasoning on these properties to ensure the correctness of ontologies.

Proof using Z/EVES is highly interactive and requires substantial expertise. The ontology languages were designed so that core reasoning tasks can be performed using Semantic Web reasoning tools fairly automatically. Hence, it is natural to combine Z/EVES and Semantic Web reasoning tools to harness their synergistic proof power. Moreover, the inclusion of Alloy Analyzer adds another useful dimension to the synergy since Alloy Analyzer is able to locate the source of errors in a specification.

In the rest of Chapter4, we present a combined approach to checking DAML+OIL and RDF ontologies by using proof tools RACER, Z/EVES and Alloy Analyzer together. We begin by defining Z and Alloy semantics for DAML+OIL. The Z and Alloy semantics enables Z/EVES and Alloy Analyzer to understand DAML+OIL and RDF ontologies. With this semantics as a basis, we then develop a transformation program to automatically transform an ontology to Z and Alloy specifications, respectively.

The complementary proof power can be exploited through applying these reasoning tools in turn and expressing complex properties in Z and use Z/EVES to prove these properties. Firstly, ontological consistency can be checked by SW reasoning engines such as RACER and FaCT++ with full automation. Secondly, any such inconsistency found can be precisely located by Alloy Analyzer. Thirdly, more complex properties inexpressible in DAML+OIL and OWL can be expressed in Z and checked by Z/EVES. The strength of the combined approach is demonstrated through a real-world military planning case study. It is observed that Alloy Analyzer located the source of ontological inconsistencies found by RACER; and a number of errors undiscovered by RACER were found by Z/EVES.

1.2.4 Chapter 5 – Z Semantics for OWL: Soundness Proof Using Institution Morphisms

Chapter 4 presents on the practical aspects of the combined approach, namely, the transformation from DAML+OIL to Z and Alloy and the actual reasoning approach using the combination of tools. A fundamental issue, the soundness of the Z and Alloy semantics of DAML+OIL, is not addressed there.

Replacing DAML+OIL, the Web Ontology Language (OWL) became the W3C recommendation in February 2004[1]. As OWL is the successor of DAML+OIL, they are very similar in many aspects. Since OWL is also a W3C recommendation as the ontology language designed to replace DAML+OIL, it is natural to shift focus to the support of OWL.

Based on our work in [24], we have developed a Z semantics for OWL. In chapter 5, we attempt to formally prove the soundness of the Z semantics for OWL by using institutions [31] and institution morphisms [33].

Introduced by Goguen and Burstall [31], institutions are used to formalize the notion of "logical systems". They provide a means of reasoning about software specifications regardless of the underlying logical systems.

The basic components of a logical system, an institution, are *models* and *sentences*, related by the *satisfaction relation*. The compatibility between models and sentences is provided by *signatures*, which formalize the notion of vocabulary from which the sentences are constructed. By modeling the signatures of a logical system as a category, we get the possibility to translate sentences and models across signature morphisms. The consistency between the satisfaction relation and the translation is given by the

[1] This is about the time when the work on combined approach [23] was in progress.

satisfaction condition, which intuitively means that *the truth is invariant under the change of notation*.

Institutions are suitable for relating Z and OWL DL (and DAML+OIL) as the logical systems (semantics) of these languages can be represented as institutions. In Chapter 5, we also present the institutions of Z and OWL and by applying Goguen and Roşu's institution comorphisms [33], the soundness of the Z semantics for OWL (and DAML+OIL) can be proved.

1.2.5 Chapter 6 – SESeW - An Integrated Tools Environment for the Semantic Web

Formal methods usually make extensive use of mathematical concepts and symbols, which often prove to be difficult for users without the relevant mathematical background. In order to hide as much underlying formal methods notations as possible and make the combined approach more friendly to users who are not familiar with the various reasoning tools, an easy-to-use visual tool that supports automated creation, transformation and querying of ontologies is much desired and valuable.

In Chapter 6, we present such an integrated tools environment, the SESeW (**S**oftware **E**ngineering for **Se**mantic **W**eb), that serves as a graphical front-end to the various reasoning tools used in the combined approach under one umbrella. Using SESeW, tasks such as ontology transformation, validation, querying, etc. can be visually performed. To make SESeW more more versatile, we also implemented a systematic approach to ontology creation, the Methontology [29]. With these functionalities, SESeW is a prototype of an ontology creation, transformation, validation and querying tool based on sound software engineering methods.

1.2.6 Chapter 7 – Simulating Semantic Web Services with LSC and Play-Engine

The full potential of the Semantic Web can only be realized when dynamic resources such as the Web Services are incorporated. The Semantic Web services ontology OWL-S is an OWL ontology that defines an essential set of vocabularies for describing the capabilities, requirements, effects, output, etc., of a Web service. It is meant to be used together with the Web Services standards such as WSDL [14] and SOAP [110] to enable software agents to automatically advertise, discover and negotiate Web services.

The correctness of Semantic Web services is essential to the functioning of software agents crawling the Semantic Web. We believe that erroneous service descriptions will give rise to invocation of wrong services, with wrong parameters or resulting in undesired outcome.

In Chapter 7, we propose to apply software engineering methods and tools to visualize, simulate and verify OWL-S process models. Live Sequence Charts (LSCs) [18] are a broad extension of the classic Message Sequence Charts (MSCs [53]). They capture communicating scenarios between system components rigorously. LSCs are used to model services, capturing the inner workings of services, and its tool support Play-Engine [38] is used to perform automated visualization, simulation and checking.

1.2.7 Chapter 8 – Conclusion

Chapter 8 concludes the thesis, summarizes the main contributions and discusses future work directions.

1.3 Publications

Most of the work presented in this thesis has been published/accepted in international conferences proceedings.

The work on the Z semantics (Chapter 3) of DAML+OIL and checking DAML+OIL ontologies using Z/EVES has been published in *The Twenty-sixth International Conference on Software Engineering* (ICSE'04, May 2004, Edinburgh, acceptance rate 13%) [24].

The combined approach for checking Web ontologies (Chapter 4) has been published in *The Thirteenth International World Wide Web Conference* (WWW'04, May 2004, New York, acceptance rate 14.6%) [23].

Work on soundness proof of transformation from OWL to Z using institutions [63] in Chapter 5 has been published in *The Seventeenth International Conference on Software Engineering and Knowledge Engineering* (SEKE'05, July 2005, Taipei) [64].

The work on the integrated tools environment was presented at *The Twelfth Asia-Pacific Software Engineering Conference* (APSEC'05, December 2005, Taipei) [22].

The work on simulating and visualizing Semantic Web services using LSC and Play-Engine was published in *Seventh International Conference on Formal Engineering Methods* (ICFEM'05, November 2005, Manchester) [90].

I have also contributed to other published works [25, 26, 21, 91, 104, 103, 105, 61, 65], which are mostly as pre-thesis/follow-up works.

Chapter 2

Background

This chapter presents the background information of the various languages, notations, techniques and tools that are involved in this thesis. It is divided into five parts. In Section 2.1, we give a brief account of Semantic Web languages and tools. Following that, Section 2.2 is devoted to the introduction to the Semantic Web services ontology OWL-S, an OWL ontology that defines a set of core vocabularies for describing Web services. In Section 2.3, we briefly introduce the formal languages Z and Alloy and their tool support Z/EVES and Alloy Analyzer. Finally, institutions and institution morphisms are briefly covered in Section 2.4.

2.1 The Semantic Web – Languages & Tools

Proposed by Tim Berners-Lee et al., the Semantic Web [8] is a vision of next generation of the Web. The current World Wide Web is designed mainly for human consumption. It is believed that in the future, the Web is also ready for intelligent software agents and it will be truly ubiquitous. Software agents will reside in, for

example, household appliances (which can also be part of the Web), and will be able to understand the meaning of information on the Web and undertake tasks without human's supervision. To sum up, in the Semantic Web, software agents will be able to autonomously and cooperatively understand, process and aggregate Web resources, which include not only static data, but also dynamic Web services.

Semantic Web ontologies give precise and non-ambiguous *meaning* to Web resources, enabling software agents to understand them. An ontology is a specification of a conceptualization [34]. It is a description of the concepts and relationships for a particular application domain. Ontologies can be used by software agents to precisely categorize and deduce knowledge.

Languages in the Semantic Web

Ontology languages are the building blocks of the Semantic Web. As briefly mentioned in Chapter 1, the development of ontology languages takes a layered approach. Depicted in Fig. 1.1, the Semantic Web languages are constructed on top of mature languages and standards such as the XML [108], Unicode and Uniform Resource Identifier (URI) [7]. In the rest of this section, we briefly present some important languages in the Semantic Web.

The Resource Description Framework (RDF) [68] is a model of metadata that defines a mechanism for describing resources and makes no assumptions about a particular application domain. RDF allows structured and semi-structured data to be mixed and shared across applications. XML describes documents, whereas RDF is a framework for metadata: it describes actual things. RDF provides a simple *triples* structure to make *statements* about Web resources. Each triple is of the form $\langle subject\ predicate\ object \rangle$, where *subject* is the resource we are interested in, *predicate* specifies the property or characteristic of the subject and *object* states the value of

the property. Besides this basic structure, a set of basic vocabularies are defined to describe RDF ontologies. This set includes vocabularies for defining and referencing RDF resources, declaring containers such as bags, lists, and collections. It also has a formal semantics that defines the interpretation of the vocabularies, the entailment between RDF graphs, etc.

RDF Schema (RDFS) [17] defines additional language constructs for RDF ontologies. It adds considerable expressivity to RDF by enabling one to group Web resources into classes, to denote the domain and range of a property, to state the subsumption relationship between classes and properties, etc.

RDF Schema can be considered as the first ontology language for the Semantic Web. However, RDF and RDFS have a number of disadvantages. For instance, in order for agents to understand Web resources unambiguously, it is necessary that these resources are strictly structured. This requirement is relaxed by RDF to allow for greater flexibility. Also, RDF Schema does not contain all modeling primitives users desired.

In RDF, RDF Schema and subsequent ontology languages, Web resources are referenced using full , URI references. It consists of a URI prefix (a namespace) and the name of the resource, separated by a separator "#". RDF also defines a shorthand form for convenience. In this form, the full URI representing the resource is given an XML qualified name, containing a prefix that is assigned to the namespace URI, the local name (which is the name of the resource), separated by a colon (:). A number of qualified name prefixes have been predefined in the Semantic Web domain. These are summarized in Table 2.1.

With the above mapping between prefixes and full namespace URIs, a long URI reference can be shortened. For example, the full URI reference for RDFS class is `http://www.w3.org/2000/01/rdf-schema#Class`. With the above representation,

Table 2.1: Predefined Qualified Name Prefixes

Prefix	Namespace URI
xsd:	http://www.w3.org/2001/XMLSchema#
rdf:	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs:	http://www.w3.org/2000/01/rdf-schema#
daml:	http://www.w3.org/2001/10/daml+oil#
owl:	http://www.w3.org/2002/07/owl#
swrl:	http://www.w3.org/2003/11/swrl#

it can be shortened to `rdfs:Class`.

The DARPA Agent Markup Language (DAML) is built on top of RDF Schema, but with a much richer set of language constructs to express class and property relationships and more refined support for data types. DAML project combined effort with the Ontology Inference Layer (OIL) [13] project and it is now referred to as DAML+OIL [101]. Being semantically equivalent to the expressive description logic $\mathcal{SHIQ}$ [50], the other major advantage of DAML+OIL over RDFS is the ability to define new classes and properties by defining restrictions on existing classes and properties. This enhances ontology structure and facilitates ontology reuse.

The main ingredients of DAML+OIL can be categorized into three types: objects, classes and properties, with data types supplying concrete values. The Object domain consists of objects (individuals) that are members of DAML+OIL or RDFS classes. Classes are the focus of DAML+OIL and they are elements of `daml:Class`, a sub class of `rdfs:Class`. DAML+OIL defines a number of built-in properties. They serve a number of purposes, which can be briefly summarized below.

- Some of the properties are used to relate two classes to define certain relationship between them. For example, the property `daml:disjointWith` is used to denote the disjointness of two classes.
- Some properties are used to construct classes from a list of classes or individ-

uals. For example, the property `daml:unionOf` relates a `daml:Class` X and a `daml:List` Y of classes such that the instances of X is the union of all the instances of classes in Y. The property `daml:disjointUnionOf` is similar, with an additional constraint that the classes in the list Y are mutually disjoint.

- Some properties are used to define new classes by constructing "restrictions", which are (anonymous) classes that can be linked to other properties or cardinality constraints.

  For example, the built-in property `daml:toClass` can be used to define the class of all objects for whom the values of property all belong to the class expression. It can be used to define, for instance, a restriction whose instances eats only Animals, as shown below.

```
<daml:Restriction>
   <daml:onProperty rdf:resource="#eats"/>
   <daml:toClass rdf:resource="#Animals"/>
</daml:Restriction>
```

  In the above example, the restriction is defined on the property `eats` and class `Animals`. This restriction can be used to define a class `Carnivores` by making it a sub class of this restriction.

  The cardinality properties define restrictions each of whose instances has exactly, at least or at most `n` distinct property values.

  The following DAML+OIL fragment defines a restriction, each of whose instances has *exactly one* nationality.

```
<daml:Restriction>
   <daml:cardinality>1</daml:cardinality>
   <daml:onProperty rdf:resource="#natitonality"/>
</daml:Restriction>
```

- Finally, some built-in properties are used to define or relate other properties. For example, the property `daml:samePropertyAs` asserts that the two properties it

relates are actually equivalent, meaning that their property extensions (the pair of objects they relate) are actually the same.

In 2003, the W3C published a new ontology language, the Web Ontology Language (OWL) [69] to replace DAML+OIL. Based on DAML+OIL, OWL is a suite of languages consisting of three species: Lite, DL and Full, with increasing expressiveness.

The three sublanguages are meant for user groups with different requirements of expressiveness and decidability. OWL Lite is the least expressive sublanguage, obtained by imposing restrictions on the usage of OWL Full language constructs. OWL DL is more expressive than Lite but is also a subset of OWL Full.

OWL Lite and DL are decidable whereas OWL Full is not. Simplistically speaking, an OWL Lite or DL ontology is an OWL Full ontology with some constraints added. These constraints include, for example, in OWL Lite, cardinality constraints can only be 0 or 1; mutual disjointness among individuals, classes, properties, data types, etc., in OWL Lite and DL ontologies. DAML+OIL is most comparable to OWL DL, which is a notational variance of description logic $\mathcal{SHOIN}$(D) [49].

The following OWL DL fragment shows the definition of carnivores in an animal-plant ontology. It defines an OWL class `Carnivores` that is a sub class of `Animals`. It is also a sub class of an anonymous class that only `eats Animals` (the allValuesFrom restriction). Note that the built-in DAML+OIL property `toClass` is renamed in OWL to `allValuesFrom`.

```
<owl:Class rdf:about="#carnivore">
   <rdfs:subClassOf>
      <owl:Class rdf:ID="animal"/>
   </rdfs:subClassOf>
   <rdfs:subClassOf><owl:Restriction><owl:onProperty>
      <owl:ObjectProperty rdf:ID="eats"/></owl:onProperty>
      <owl:allValuesFrom>
         <owl:Class rdf:about="#animal"/>
      </owl:allValuesFrom>
   </owl:Restriction></rdfs:subClassOf>
</owl:Class>
```

For any DAML+OIL or OWL ontology there are three types of core inference problems, namely concept (class) subsumption, concept consistency and instantiation reasoning. Concept subsumption checks if a concept subsumes another concept; concept consistency checks if a concept is meaningful with respect to the ontology, and property instantiation checks whether a given individual is an instance of a class. All the above inference problems can be checked by mature tableau algorithms for description logics in full automation.

The consistency of ontologies is essential to the proper functioning of agents. For example, we can imagine how chaotic it can be if an online marriage registry agent allows a person already married to register for marriage again. This could happen if the marriage ontology does not constrain that a person can only have at most one spouse. A consistent ontology satisfies the following two criteria: *realization*, that every class has at least one instance and *retrieval*, that every individual is an instance of some class [74]. Hence, the ontology consistency problem (and actually all the other types of inference problems) can be reduced to the concept consistency problem above.

Although the design of OWL has taken into consideration the different expressivity needs of various user groups, it is still not powerful enough as only relatively simple relationships can be expressed: such as class and property membership, individual (in)equalities, etc. The main reason for these limitations is that although OWL provides relatively rich language constructs for describing class relationships, it does not provide enough language primitives for describing properties. For example, properties in OWL cannot be composed to construct complex properties.

These limitations have been recognized by a number of researchers and in 2004, Horrocks and Patel-Schneider proposed a rules extension to OWL DL. The new language is called OWL Rules Language (ORL) [47] and it is syntactically and semantically

coherent to OWL. By incorporating Horn clause rules into OWL and making rules part of OWL axioms, which are used to construct classes and properties, ORL can express more complex properties. ORL is now known as SWRL [48], with some sets of built-ins for handling data type, such as numbers, booleans, strings, date & time, etc.

The major extensions of SWRL over OWL DL include Horn style rules and (universally quantified) variable declaration. For presentation and brevity purposes, the rules are in the form of `antecedent` $\rightarrow$ `consequent`, where both antecedent and consequent are conjunctions of the following kinds of atoms: class membership, property membership, individual (in)equalities and built-ins. Informally, a rule means that if the antecedent holds, the consequent must also hold. Moreover, an empty antecedent is treated as trivially true and an empty consequent is treated as trivially false. In SWRL, variables are prefixed with a question mark (?). A simple example rule states that if $?b$ is a parent of $?a$ and $?c$ is a brother of $?b$, then $?c$ is an uncle of $?a$, where $?a$, $?b$ and $?c$ are variable names.

$$\texttt{hasParent}(?a, ?b) \land \texttt{hasBrother}(?b, ?c) \rightarrow \texttt{hasUncle}(?a, ?c)$$

SWRL extends the expressivity of OWL by providing more support for describing and composing properties as shown in the previous example. It has been shown to be non-decidable. However, it is still not as expressive as Z. As one of the main motivations of the rules extension is to infer knowledge not present in the ontology, disjunction and negation are not allowed in SWRL. It also does not support explicit quantification over rules. As we stated above, these design constraints hinder expressing certain properties.

In view of this, Patel-Shneider proposed the language SWRL FOL [9] as a step further towards first-order logic. On top of SWRL, it adds logical connectors such as 'and',

'or', 'negation', 'implication', and 'existential' and 'universal' quantification.

The ontology languages DAML+OIL and OWL are based on description logics, for which highly optimized algorithms for solving concept consistency problems exist. However, OWL has also been criticized for a number of reasons [56], such as the inappropriate layering on top of RDFS; unnaturalness of certain modeling decisions; inefficiency of query answering mechanisms; the lack of distinction between restrictions and constraints, etc. To overcome these disadvantages, the OWL$^-$ [56] suite of languages were proposed. OWL$^-$ also consists of three sublanguages: OWL Lite$^-$, DL$^-$ and Full$^-$, where OWL Lite$^-$ and DL$^-$ are strict subsets of the respective OWL species. OWL DL$^-$ is an extension of OWL Lite$^-$ and OWL Full$^-$ is an extension of OWL DL$^-$ towards OWL Full.

The semantics of OWL$^-$ languages are based on logic programming. OWL Lite$^-$ and DL$^-$ are constructed in such a way that they can be directly translated into Datalog programs. Hence mature techniques in the deductive databases in query answering and rule extensions can be borrowed.

An extension to OWL$^-$, the OWL Flight [20], has also been proposed. It adds a number of features on top of OWL$^-$, such as constraints and local closed-world assumption.

The Web Rule Language (WRL) [1] is a proposal of a rule-based ontology language. Based on deductive databases and logic programming, WRL is designed to be complementary to OWL which is strong at checking subsumption relationships among concepts. WRL focuses on checking instance data, the specification of, and reasoning about arbitrary rules. A new layering of Semantic Web ontology languages is also proposed [19], as shown in Fig. 2.1. Moreover, WRL assumes a "Closed World Assumption", whereas OWL and SWRL assume an "Open World Assumption".

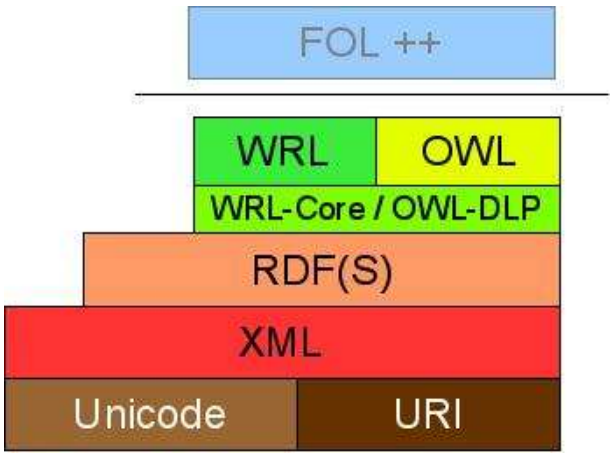

Figure 2.1: A newly proposed layering of the Semantic Web

There also exist other rules extensions besides the ones mentioned above. The Semantic Web Services Language (SWSL) [2] has been developed under the Semantic Web Services Initiative (SWSI)[1] framework. It is a logic-based language for specifying formal characterizations of Web service concepts and descriptions of individual services. However, SWSL is domain-independent and it does not contain any constructs customized to Web services. SWSL has a layered structure. Unlike OWL, the layers of SWSL are not organized according to expressivity. Rather, the SWSL layers are orthogonal to each other and each introduces new features that enhance the modeling power of the language. Moreover, these layers can be implemented together or in any arbitrary combination so that users can implement the reasoning service according to features required. SWSL includes two sublanguages: SWSL-FOL, a full first-order logic language, which is used to specify the service ontology (SWSO), and SWSL-Rules, a rule-based sublanguage, which can be used both as a specification and an implementation language.

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

Recently, the Rule Interchange Format (RIF) working group [2] has been formed by the W3C with the aim to producing a "standard means for exchanging rules on the Web".

Tools in the Semantic Web

Besides ontology languages, we also witness the growth of ontology tools in the recent years. Various tools have been built to facilitate the diversified range of ontology development tasks, including creation, management, versioning, merging, querying, verification, etc. Here we briefly survey a few. An extensive survey was provided in [77].

Cwm (Closed world machine) [96] is a general-purpose data processor for the SW. Implemented in Python and command-line based, it is a forward chaining reasoner for RDF.

Triple [87] is an RDF query, inference and transformation language. It does not have a built-in semantics for RDF Schema, allowing semantics of languages to be defined with rules on top of RDF. This feature of Triple facilitates data aggregation as user can perform RDF reasoning and transformation under different semantics. The Triple tool supports DAML+OIL through external DAML+OIL reasoners such as FaCT and RACER.

Fast **C**lassification of **T**erminologies (FaCT) [45], developed at University of Manchester, is a TBox (terminology Box, concept-level) reasoner that supports automated concept-level reasoning, namely class subsumption and consistency reasoning. It does not support ABox (assertion Box, instance-level) reasoning. FaCT implements a reasoner for the description logic $\mathcal{SHIQ}$ [50]. It is implemented in Common Lisp and comes with a

[2] cf. http://www.w3.org/2005/rules/.

FaCT server, which can be accessed across network via its CORBA interface. Given a DAML+OIL/OWL ontology, it can classify the ontology (performs subsumption reasoning) to reduce redundancy and detects any inconsistency within it.

Recently a new version, the FaCT++ [44] system was released. It is an OWL Lite reasoner and introduced some new optimization techniques.

RACER, the **R**enamed **A**Box and **C**oncept **E**xpression **R**easoner [36], implements a TBox and ABox reasoner for the description logic $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D})^-$ [35]. It can be regarded as (a) a SW inference engine, (b) a description logic reasoning system capable of both TBox and ABox reasoning and (c) a prover for modal logic Km. In the SW domain, RACER's functionalities include creating, maintaining and deleting ontologies, concepts, roles and individuals; querying, retrieving and evaluating the knowledge base, etc. It supports RDF, DAML+OIL and OWL. The RACER system has recently been commercialized and it is now known as RacerPro[3].

Both FaCT (FaCT++) and RACER (RacerPro) perform their functions in full automation, which means by "pushing a button", these tools return a definitive answer without intermediate steps.

OilEd [4] is a visual DAML+OIL and OWL ontology editor developed by the University of Manchester. In OilEd, users can create new classes/properties, relate them using restrictions, view the hierarchy of classes and create instances of classes. Protégé [30] is a system for developing knowledge-based systems developed at Stanford University. It is an open-source, Java-based Semantic Web ontology editor that provides an extensible architecture, allowing users to create customized applications. In particular, the Protégé-OWL plugin [57] enables editing OWL ontologies and connecting to DIG [5]-compliant reasoning engines such as RACER [36] and FaCT++ [44]

[3] cf. http://www.racer-systems.com/

to perform tasks such as automated consistency checking and ontology classification.

Both of the above two editors support reasoners that conform to the DIG interface [5]. Next we briefly introduce a few that is relevant to the thesis.

2.2 Semantic Web Services Ontology OWL-S

Web Services [4] are a W3C coordinated effort to define a set of open and industry-supported specifications to provide a standard way of coordination between different software applications in a variety of environments. A Web service is defined as "a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL [14]). Other systems interact with the Web service in a manner prescribed by its description using SOAP [110] messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards" [10].

The various specifications in the Web services domains are all based on XML, making information processing and interchange easier. However, as XML Schema only defines the syntax of a document, it is hard for software agents to understand the *semantics* of a Web service described using these specifications. A language that is both syntactically well-formed and semantical is therefore desirable.

As introduced in the previous section, the Semantic Web [8] is an envisioned extension of the current Web where resources are given machine-understandable, unambiguous meaning so that software agents can cooperate to accomplish complex tasks without human supervision.

[4] cf. http://www.w3.org/2002/ws/

OWL Services (OWL-S) [95] is a Web services ontology in OWL DL. It supplies Web service producers/consumers with a core set of markup language constructs for describing the properties and capabilities of their Web services in an unambiguous, computer-interpretable form. OWL-S was expected to enable the tasks of "automatic Web service discovery", "automatic Web service invocation" and "automatic Web service composition and inter-operation". OWL-S consists of three essential types of knowledge about a service: the profile, the process model and the grounding. Figure 2.2 shows the high-level architecture of an OWL-S ontology.

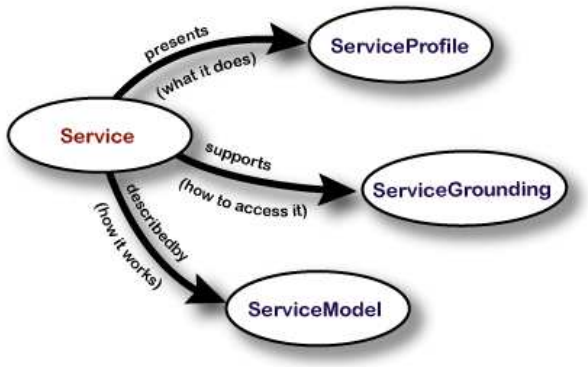

Figure 2.2: Architecture of the OWL-S ontology

A Web service consists of mainly three ingredients, a ServiceProfile, a ServiceGrounding and a ServiceModel. A ServiceProfile tells what the service does. It is the primary construct by which a service is advertised, discovered and selected. The ServiceGrounding tells how the service is used. It specifies how an agent can access a service by specifying, for example, communication protocol, message format, port numbers, etc.. The primary concern of our work in this paper is the OWL-S ServiceModel (also called process model), which tells how the service works. Thus, the OWL class Service is describedBy a ServiceModel. It includes information about the service's inputs, outputs, preconditions and effects. It also shows the component processes of a complex process and how the control flows between the components.

The OWL-S process model is intended to provide a basis for specifying the behaviors of a wide array of services. There are two chief components of an OWL-S process model – the process, and process control model. The process describes a Web Service in terms of its input, output, precondition, effects and, where appropriate, its component subprocesses. The process model enables planning, composition and agent/service inter-operation. The process control model – which describes the control flow of a composite process and shows which of various inputs of the composite process are accepted by which of its sub-processes – allows agents to monitor the execution of a service request. The constructs to specify the control flow within a process model include Sequence, Split, Split+Join, If-Then-Else, Repeat-While and Repeat-Until. The full list of control constructs in OWL-S and its semantics can be found in Chapter 7 and in the latest version of OWL-S [95].

2.3 Z & Alloy – Languages & Tools

The verification of Semantic Web ontologies to be presented in the following chapters involves the use of formal languages. In this section, we briefly introduce these languages, namely Z and Alloy, and their respective proof tool support.

2.3.1 Z

Z [107, 89] is a well-studied formalism based on ZF set theory and first-order predicate logic. Its formal semantics [106] and elegant modeling style encouraged an object-oriented extension, the Object-Z [28], and subsequently the Timed Communicating Object-Z (TCOZ) [67]. These additions greatly expand the expressivity of Z-family languages.

Z is specially suited to model system data and states. Z defines a number of language constructs including given type, abbreviation type, axiomatic definition, generic definition, state and operation schema definitions, etc. Besides, Z also defines a mathematical library, the *toolkit*, which gives definitions of commonly used concepts, symbols and operators, such as sets, set union, intersection, natural numbers, sequences, functions, relations, bags, etc.

Declarations

Z is a strictly-typed specification language. In Z, a name must be declared before it is referenced. Moreover, properties of systems being specified are stated using Z predicates. Hence, declarations and predicates are the building blocks of Z specifications.

The basic form of Z declarations is $x : A$, where x is the newly introduced variable of the type A. Moreover, this type A, which must be a set itself, should be defined previously too. A variable declared is either global or local. A global variable is visible from the point of declaration to the end of specification. A local variable's scope is the current enclosing environment. Interested readers may refer to [106, 88] for details.

Predicates

As in first-order logic, predicates in Z are Boolean-valued statements over a number of subjects. Z predicates allow the forms:

Equality & membership The basic Z predicates are equalities $=$ and membership relationships $\in$. For example, the predicate $x \in \mathbb{N}$ states that variable x is a member of natural numbers $\mathbb{N}$.

Set relationship operators such as subset can be derived using set membership. In general, the subset relationship $A \subseteq B$ can be expressed as $A \in \mathbb{P}\, B$, where $\mathbb{P}$ is the powerset symbol. The expression $\mathbb{P}\, B$ denotes all the sets that are subsets of B.

Propositional connectives These include the usual connectives in the propositional logic, namely $\neg$, $\vee$, $\wedge$, $\Rightarrow$ and $\Leftrightarrow$. They are used to connect simpler predicates to construct complex ones.

Quantifiers Based on first-order logic, Z also allows quantifiers in predicates. These include the universal quantifier $\forall$, the existential quantifier $\exists$ and the unique existential quantifier $\exists_1$. The predicate $\exists_1\, S \bullet P$ is true if there exists only one way of value assignment for the variables in S.

Note that the $\bullet$ symbol denotes "such that".

Let expressions The **let** expression constructs local definitions in a predicate. For example, in the predicate $\mathbf{let}\, x_1 == E_1;\ \ldots;\ x_n == E_n \bullet P$, the scope of variables $x_1, \ldots, x_n$ extends to the predicate P, but not into the bodies of expression $E_1, \ldots, E_n$.

The semantics of the **let** operator can be summarized as follows.

$$(\mathbf{let}\, x_1 == E_1;\ \ldots;\ x_n == E_n \bullet P) \\ \Leftrightarrow (\exists_1\, x_1 : t_1;\ \ldots;\ x_n : t_n \mid x_1 = E_1;\ \ldots;\ x_n = E_n \bullet P)$$

Note that the vertical bar | denotes the conditions that the expression in front of it must satisfy.

Relations Z also allows relation symbols to be used as predicates. The abstract syntax is defined as follows.

```
Predicate ::= Expression Rel Expression Rel ... Rel Expression
            | Pre-Rel Expression
```

Treated as predicates, relations denote relational memberships. For example, for a binary relation R, the predicate $E_1 \ R \ E_2$ denotes the membership predicate $(E_1, E_2) \in R$. The predicate $R \ E$, where R is a unary prefix symbol, denotes $E \in R$.

For the general form of chain of relations $E_1 \ R_1 \ E_2 \ R_2 \ E_3 \ R_3 \ldots E_{n-1} \ R_{n-1} \ E_n$, it is equivalent to the conjunction of individual relation predicate $E_1 \ R_1 \ E_2 \wedge E_2 \ R_2 \ E_3 \wedge \ldots \wedge E_{n-1} \ R_{n-1} \ E_n$.

Essential Language Constructs

In this subsection, we give a brief introduction to the more high-level Z language constructs relevant to this thesis. A more detailed introduction can be found in Appendix A.

***Given type*:**

A given type introduces uninterpreted basic types, which are treated as sets in Z. For example:

$[Resource]$

introduces one given type *Resource*, which is a set.

***Axiomatic definition*:**

An axiomatic definition defines global variables, and optionally constrains their values using predicates. These global variables cannot be globally redefined. For example, the following axiomatic definition defines two variables *Class* and *Property* as subsets of *Resource*. Furthermore, we assert that these two sets are mutually disjoint (their intersection is an empty set).

$$\begin{array}{|l} Class : \mathbb{P}\, Resource \\ Property : \mathbb{P}\, Resource \\ \hline Class \cap Property = \emptyset \end{array}$$

***Generic Definitions*:**

A generic definition is a generic form of axiomatic definition, parameterized by a formal parameter, a set X.

For example, in OWL DL, a datatype property relates some individuals to values of some data type. The mapping of such properties can be modeled by the following generic definition *sub_valD*. Note that in this definition the predicate part is empty.

$$\begin{array}{|l} [X] \\ \hline sub\_valD : DatatypeProperty \rightarrow (Individual \leftrightarrow X) \\ \hline \end{array}$$

***Constraints*:**

A constraint (predicate) constrains values of global variables that have been declared previously. For example, the following predicate states that the cardinality of the set is 2, implying that the two set members, which are both previously defined, are actually distinct.

$$\#\{PLAN\_P3\_P6\_P1, PLAN\_P3\_P6\} = 2$$

Ontology languages such as DAML+OIL and OWL are based on description logics, which are well known to be a subset of first-order logic [58]. Z, on the other hand, embraces expressivity from both first-order logic and schema calculus. Hence, Z is by nature more expressive than these languages. It is able to capture more complex properties pertaining to an ontology than ontology languages can.

Z/EVES [84] is an interactive system for composing, checking, and analyzing Z specifications. It supports the analysis of Z specifications in a number of ways: syntax and

type checking, schema expansion, precondition calculation, domain checking, general theorem proving, etc.

In Z/EVES, properties about a specification can be specified as *theorems*. These properties include facts and facts that one hopes to be facts. By proving theorems of a particular specification, we gain more confidence about its correctness.

The abstract syntax of theorems is defined as follows [71].

theorem ::= `\begin`[para-opt]{`theorem`}{[usage] theorem-name}[gen-formals]
predicate
[`\proof`
command sep ... sep command]
`\end{theorem}`

In the above abstract syntax, the keyword "para-opt" has two options: `disabled` or `enabled`, which indicate whether the theorem is to be automatically used by the theorem prover. The "gen-formals" keyword is an optional list of formal parameters appearing in the definition of the theorem.

The keyword "usage" have a number of options and it indicates the type of the theorem and consequently how it is to be used by Z/EVES. The options for this keyword are categorized as follows.

Facts The usage `axiom` indicates that the theorem is to be used by Z/EVES as a fact.

Rewrite rules The usage `rule` specifies that a theorem is to be used as a rewrite rule. Put it simply, a rewrite rule is a Z predicate, in the form of either a universal quantification, a logical implication, equivalence or an expression equality. If

such a theorem is used, Z/EVES will replace the left-hand side of the predicate or expression by its right-hand side during reduction and rewriting.

Forward rules The usage `frule` specifies that a theorem is to be used as a forward rule, which is in the form of an implication from a schema reference to a list of conjuncted predicates. A forward rule can be fired during simplification and it used to introduce predicates to Z/EVES.

Assumption rules The usage `grule` specifies that a theorem is to be used as an assumption rule. As its name suggests, an assumption rule is used to make Z/EVES assume some predicates. It can be used to introduce type information and inequalities into the proof context.

For example, the following theorem is a `disabled` assumption rule that states if a resource x is a member of *Class*, then it can be assumed that it is not a member of *Property*. Note that in theorems, variables can be used without declaration.

theorem disabled grule classPropertyDisjointRule
$x \in Class \Rightarrow x \notin Property$

In the ISO standard Z [52] and Z/EVES, Z specifications are organized into *sections* to improve specification clarity and reuse. The built-in section `toolkit`, as introduced above, defines basic constants and operators. Specifications are built hierarchically by including existing sections as their parents.

2.3.2 Alloy

Alloy [54] is a structural modeling language emphasizing on automated reasoning support. It treats relations as first-class citizens and uses relational composition as

a powerful operator to combine various structural entities. The design of Alloy was influenced by Z and it can be (roughly) viewed as a subset of Z.

Essential Alloy language constructs are presented below.

***Signatures*:**

A signature (`sig`) paragraph introduces a basic type and a collection of relations (called fields) in it along with types of the fields and constraints on their values. A signature may inherit fields and constraints from another signature. For example

```
sig Resource {}
```

defines a signature `Resource` with no relations associated with it.

The signature below defines a basic type `Class`, which is a subset of `Resource` defined above (`Class` extends `Resource`). Moreover, it has a field associated with it, the `instances`, that maps a class to the set of its instances, which are of the type `Resource`.

```
disj sig Class extends Resource
    {instances: set Resource}
```

The keyword `disj` preceding the definition asserts that this definition and other subsets of `Resource` are disjoint with each other.

***Functions*:**

A function (`fun`) captures behavior constraints. It is a parameterized formula that can be "applied" elsewhere. For example, in the following Alloy specification, `subClassOf` is a function that states for classes `c1` and `c2` to be of `subClassOf` relationship, the instances of `c1` must be a subset of the instances of `c2`.

```
fun subClassOf(c1, c2: Class)
    {c1.instances in c2.instances}
```

***Facts*:**

A fact (`fact`) constrains the relations and objects. A fact is a formula that takes no arguments and need not be invoked explicitly; it is always true. For example, the following fact states that `MilitaryTask` is a sub class of `MilitaryProcess`.

```
fact{subClassOf(MilitaryTask, MilitaryProcess)}
```

***Assertions*:**

An assertion (`assert`) specifies an intended property. It is a formula the correctness of which needs to be checked, assuming the facts in the model. For example:

```
assert PrepareDemolitionTaskIsMilProcess
    {subClassOf(PrepareDemolition_MilitaryTask, MilitaryProcess)}
```

Alloy Analyzer [55] is a constraint solver for Alloy that provides fully automated simulation and checking. Alloy Analyzer works as a compiler: it compiles a given problem into a (usually huge) boolean formula, which is subsequently solved by a SAT solver, and the solution is then translated back to Alloy Analyzer. Inevitably, a finite scope - a bound on the size of the domains - must be given to make the problem finite.

Alloy Analyzer determines whether there exists a model for the formula. When it finds an assertion to be false, it generates a counterexample, which makes tracing the error easier, compared to theorem provers. However, the capability of Alloy Analyzer is constrained by the way it works. Since Alloy Analyzer performs exhaustive search,

it does not scale very well. Similar to Z/EVES, Alloy specifications are in the form of *modules*, organized into a tree. Existing modules can be reused by commands `open` or `use`.

Besides Z/EVES and Alloy Analyzer, a number of Automated Theorem Provers have been implemented in the recent years [73, 82, 92] and Vampire [82] is one with very high performance. In [97], It has been chosen to make comparison with a DL reasoner FaCT++, the next-generation of the FaCT reasoner introduced above. In the comparison, core DL reasoning tasks, namely knowledge base classification and concept subsumption were considered. As the comparison turned out, Vampire is out performed by FaCT++. Based on the above result, the authors suggested that first-order reasoners, including Z/EVES and Alloy Analyzer, are best suited to be used in a hybrid way, performing some reasoning tasks DL Reasoners such as FaCT++ and RACER cannot deal with. This is exactly what we have done in our combined approach.

So far we have introduced several Semantic Web reasoning tools and software engineering proof tools. It is interesting to compare them. In Table 2.2, we summarize the strength and weakness of RACER, Z/EVES and Alloy Analyzer.

Table 2.2: Strength & weakness of the reasoning tools

Tool	**Strength**	**Weakness**
RACER	Fully automated reasoning	Kinds of reasoning tasks limited
Alloy Analyzer	Able to locate the source of the errors	Scope is limited
Z/EVES	Very expressive & powerful	Interactive proof process

2.4 Institutions & Institution Morphisms

Institutions and institution morphisms are used in this thesis to prove the correctness of the Z semantics of OWL in Chapter 5. In this section, we give a brief introduction to them. We assume the reader is familiar with the basics of category theory, including category, opposite category, functor, natural transformation, colimit, and the categories Set of sets and Cat of categories; e.g., see [59] for an introduction to this subject.

Institutions were introduced by Goguen and Burstall [31, 32] to formalize the notion of logical systems and to provide a basis for reasoning about software specifications independently of the underlying logical system chosen. The basic components of a logical system are *models* and *sentences*, related by the *satisfaction relation*. The compatibility between models and sentences is provided by *signatures*, which formalizes the notion of vocabulary from which the sentences are constructed. Modeling the signatures of a logical system as a category, we get the possibility to translate sentences and models across signature morphisms. The consistency between the satisfaction relation and this translation is given by the *satisfaction condition* which intuitively means that *the truth is invariant under the change of notation*.

Formally, an *institution* is a quadruple $\Im = (\mathsf{Sign}, \mathsf{sen}, \mathsf{Mod}, \models)$ where Sign is a category whose objects are called *signatures*, sen is a functor $\mathsf{sen} : \mathsf{Sign} \to \mathsf{Set}$ which associates with each signature Σ a set whose elements are called Σ-*sentences*, $\mathsf{Mod} : \mathsf{Sign}^{op} \to \mathsf{Cat}$ is a functor which associates with each signature Σ a category whose objects are called Σ-*models*, and $\models$ is a function which associates with each signature Σ a binary relation $\models_\Sigma \subseteq |\mathsf{Mod}(\Sigma)| \times \mathsf{sen}(\Sigma)$, called *satisfaction relation*, such that for each morphism $\phi : \Sigma \to \Sigma'$ the *satisfaction condition*

$$\mathsf{Mod}(\phi)(M') \models_\Sigma e \Leftrightarrow M' \models_{\Sigma'} \phi(e)$$

holds for each model $M' \in \mathsf{Mod}(\Sigma')$ and each sentence $e \in \mathsf{sen}(\Sigma)$. The functor sen abstracts the way the sentences are constructed from signatures (vocabularies). The functor Mod is defined over the opposite category Sign^{op} because a "translation between vocabularies" $\phi : \Sigma \to \Sigma'$ defines a forgetful functor $\mathsf{Mod}(\phi^{op}) : \mathsf{Mod}(\Sigma') \to \mathsf{Mod}(\Sigma)$ such that for each Σ'-model M', $\mathsf{Mod}(\phi^{op})(M')$ is M' viewed as a Σ-model. The satisfaction condition may be read as "M' satisfies the ϕ-translation of e iff M' viewed as a Σ-model satisfies e", i.e., the meaning of e is not changed by the translation ϕ.

We often use $\mathsf{Sign}(\Im)$, $\mathsf{sen}(\Im)$, $\mathsf{Mod}(\Im)$, $\models_{\Im}$ to denote the components of the institution $\Im$. If $\phi : \Sigma \to \Sigma'$ is a signature morphism, then the Σ-model $\mathsf{Mod}(\phi^{op})(M')$ is also denoted by $M'|_{\phi}$ and we call it *the ϕ-reduct of M'*.

The satisfaction relation is extended to sets of sentences and it is used to define the semantical consequence notion. If E is a set of Σ-sentences, then:

1. $M \models_{\Sigma} E$ if $M \models e$ for each $e \in E$.
2. $\mathsf{Mod}^{th}(\Sigma, E) = \{M \mid M \models_{\Sigma} E\}$.
3. $E \models_{\Sigma} e$ if $M \models e$ for each model $M \in \mathsf{Mod}^{th}(\Sigma, E)$. We say that e is a *semantical consequence* of E.

A specification (presentation) is a way to represent the properties of a system independent of model (= implementation). Formally, a *specification* is a pair (Σ, E), where Σ is a signature and E is a set of Σ-sentences. A (Σ, E)-*model* is a Σ-model M such that $M \models_{\Sigma} E$. We sometimes write $(\Sigma, E) \models e$ for $E \models_{\Sigma} e$.

The migration from one logical system to another is captured by institution morphism or institution comorphism. There are many variations on institution morphisms/comorphisms in the literature. We recommend [33, 94] for systematic investigations of these notions and the relations between them. Here we recall from

[33] the definition for simple theoroidal comorphism. Let $\Im = (\mathsf{Sign}, \mathsf{sen}, \mathsf{Mod}, \models)$ and $\Im' = (\mathsf{Sign}', \mathsf{sen}', \mathsf{Mod}', \models')$ be two institutions. We denote by Th the category of the specifications in $\Im$ and by Th' the category of the specifications in $\Im'$. Let $sign' : \mathsf{Th}' \to \mathsf{Sign}'$ be the forgetful functor which sends a specification (Σ', E') in Th' to its signature Σ'. A *simple theoroidal comorphism* $(\Phi, \alpha, \beta) : \Im \to \Im'$ consists of:

1. a functor $\Phi : \mathsf{Sign} \to \mathsf{Th}'$ such that there is a functor $\Phi^\diamond : \mathsf{Sign} \to \mathsf{Sign}'$ satisfying $\Phi\,;\ sign' = \Phi^\diamond$,
2. a natural transformation $\alpha : \mathsf{sen} \Rightarrow \Phi^\diamond\,;\mathsf{sen}'$, and
3. a natural transformation $\beta : \Phi^\diamond\,;\mathsf{Mod}' \Rightarrow \mathsf{Mod}$,

such that the following satisfaction condition holds:

$$M' \models'_{\Phi(\Sigma)} \alpha_\Sigma(e) \text{ iff } \beta_\Sigma(M') \models_\Sigma e$$

for any $\Phi(\Sigma)$-model M' of $\Im'$ and Σ-sentence e of $\Im$. We extend Φ to the functor $\Phi : \mathsf{Th} \to \mathsf{Th}'$ such that if $\Phi(\Sigma) = (\Sigma', E')$, then $\Phi(\Sigma, E) = (\Sigma', E' \cup \alpha_\Sigma(E))$. In other words, $\Phi(\Sigma, E)$ is $\Phi(\Sigma)$ to which we add the sentences $\alpha_\Sigma(E)$. The functor Φ associates with each signature Σ in $\Im$ a specification $(\Sigma^\emptyset, E^\emptyset)$ in $\Im'$; this means that the definition of vocabularies in $\Im$ includes properties which are expressed in $\Im'$ by $E^\emptyset$. The natural transformation α consists of a morphism $\alpha_\Sigma : \mathsf{sen}(\Sigma) \to \mathsf{sen}'(\phi^\diamond(\Sigma))$ for each signature Σ in $\Im$; α_Σ defines the translation of Σ-sentences in $\Im$ into $\phi^\diamond(\Sigma)$-sentences in $\Im'$. The natural transformation β consists of a functor $\beta_\Sigma : \mathsf{Mod}'(\phi^\diamond(\Sigma)) \to \mathsf{Mod}(\Sigma)$ for each signature Σ in $\Im$; β_Σ says how a $\phi^\diamond(\Sigma)$-model in $\Im'$ can be seen as a Σ-model in $\Im$. The meaning of the satisfaction condition is similar to that from the definition of the institution.

Remark 1 *The definition for simple theoroidal comorphism is slightly modified from that given in [33]. If*

– *we extend* Mod'^{th} *to a functor* $\mathsf{Mod}'^{th} : \mathsf{Th}'^{op} \to \mathsf{Cat}$ *similar to* Mod' *but defined over specifications, and*

– *we denote by* mod' *the natural transformation* $\mathsf{mod}' : \Phi^{op}\,;\, \mathsf{Mod}'^{th} \Rightarrow \Phi^{\diamond op};\ \mathsf{Mod}'$ *such that for each signature* Σ $\mathsf{mod}'_\Sigma : \mathsf{Mod}'^{th}(\Phi(\Sigma)) \to \mathsf{Mod}'(\Phi^\diamond(\Sigma))$ *is the inclusion, and*

– β^{th} *is the vertical composition* $\mathsf{mod}'\,;\, \beta$*,*

then $(\Phi, \alpha, \beta^{th})$ *is a simple theoroidal comorphism as in [33].*

Chapter 3

Checking Web Ontologies using Z/EVES

As stated in Chapter 2, ontology languages are the building blocks of the Semantic Web as they prescribe how Web resources are defined and related. The reasoning and verification tools for the Semantic Web are continually improving. However, due to the inherent expressivity limitation of main ontology languages such as DAML+OIL and OWL, the reasoning tools can only perform a very restricted set of tasks. It is hence our belief that the Semantic Web is a novel application domain for software modeling languages and tools.

Z [107, 89] is a formal modeling language for specifying software systems and Z/EVES [84] is an integrated proof environment for Z. In this chapter, we demonstrate how Z and Z/EVES can be used to represent and reason about DAML+OIL and RDF ontologies.

We begin by presenting the Z semantics for ontology language DAML+OIL in Section 3.1. This semantic model is embedded as a Z section in Z/EVES, which serves as an environment for checking and verifying Web ontologies. Following a brief introduc-

tion of the military plan ontologies in Section 3.3, we present a tool for automatically transforming DAML+OIL and RDF ontologies into Z specifications understood by Z/EVES in Section 3.4. Finally in Section 3.5, we use a recent real application, the military plan ontologies, to demonstrate the different reasoning tasks that Z/EVES can perform. Section 3.6 summarizes the main contributions of this chapter.

3.1 Z Semantics for DAML+OIL

This section presents (part of) the Z semantics for the DAML+OIL language. The full semantics can be found in Appendix B. The Z syntax used in this section are documented earlier in Section 2.3.

3.1.1 Basic Concepts

Everything in the Semantic Web is a *Resource*. So we model it as a given type in Z.

$$[Resource]$$

Class corresponds to a concept, which has a number of resources associated with it: the *instances* of this class. Hence, we model *Class* as a subset of resource and *instances* as a function from classes to sets of resources.

$$Class : \mathbb{P}\, Resource$$
$$instances : Class \rightarrow \mathbb{P}\, Resource$$

Property is also a subset of resource, disjoint with class. A property relates resources to resources. The function *sub_val* maps each property to the resources it relates.

$$Property : \mathbb{P}\, Resource$$
$$\hline$$
$$Property \cap Class = \emptyset$$

$$sub\_val : Property \rightarrow (Resource \leftrightarrow Resource)$$

The property *equivalentTo* relates two equivalent resources. It is used as a super property of *sameClassAs* and *samePropertyAs*.

$$
\begin{array}{|l}
equivalentTo : Resource \leftrightarrow Resource \\
\hline
\forall\, a, b : Resource \bullet a \ \underline{equivalentTo}\ b \Leftrightarrow a = b
\end{array}
$$

3.1.2 Class Elements

The property *subClassOf* is defined as a relation from class to class. For a class c_1 to be the sub class of class c_2, the instances of c_1 must be a subset of instances of c_2. Other properties such as *disjointWith* are similarly defined.

$$
\begin{array}{|l}
subClassOf : Class \leftrightarrow Class \\
disjointWith : Class \leftrightarrow Class \\
\hline
\forall\, c_1, c_2 : Class \bullet \\
\quad c_1 \ \underline{subClassOf}\ c_2 \Leftrightarrow instances(c_1) \subseteq instances(c_2) \\
\quad c_1 \ \underline{disjointWith}\ \ c_2 \Leftrightarrow instances(c_1) \cap instances(c_2) = \emptyset
\end{array}
$$

The properties *intersectionOf* and *unionOf* constructs a class from a list (sequence) of classes whose instances are the intersection/union of the sequence of classes.

$$
\begin{array}{|l}
intersectionOf : \mathrm{seq}\, Class \rightarrow Class \\
unionOf : \mathrm{seq}\, Class \rightarrow Class \\
\hline
\forall\, cl : \mathrm{seq}\, Class;\ c : Class \bullet \\
\quad intersectionOf(cl) = c \Leftrightarrow instances(c) = \bigcap \{x : \mathrm{ran}\, cl \bullet instances(x)\} \\
\quad unionOf(cl) = c \Leftrightarrow instances(c) = \bigcup \{x : \mathrm{ran}\, cl \bullet instances(x)\}
\end{array}
$$

3.1.3 Property Restrictions

Properties introduced in this section can be used in DAML+OIL restrictions to construct (anonymous) classes that are used to define other classes.

The property *toClass* attempts to establish a maximal possible set of resources as a class. It states that any resource a_1 is an instance of class c_2 if either: a_1 is defined for property p and $(a_1, a_2) \in sub\_val(p)$ implies that a_2 is an instance of class c_1; or that p is not defined for a_1 at all.

An example may better illustrate this property. Suppose that we want to define a class carnivore in DAML+OIL by stating that it only eats animals. This can be achieved by using the *toClass* property. Assuming that *eats* is a property and *Animal* and *Carnivore* are a DAML+OIL class, the following Z predicate indicates that *Carnivore* only *eats Animal*: $toClass(Animal, eats) = Carnivore$.

$$
\begin{array}{|l}
toClass : (Class \times Property) \rightarrow Class \\
\hline
\forall\, c_1, c_2 : Class;\ p : Property \bullet toClass(c_1, p) = c_2 \Leftrightarrow \\
\quad instances(c_2) = \\
\quad\quad \{a : Resource \mid sub\_val(p)(\!|\, \{a\} \,|\!) \subseteq instances(c_1)\}
\end{array}
$$

Property *hasValue* states that all instances of class c have resource r for property p.

$$
\begin{array}{|l}
hasValue : (Resource \times Property) \rightarrow Class \\
\hline
\forall\, r : Resource;\ p : Property;\ c : Class \bullet hasValue(r, p) = c \Leftrightarrow \\
\quad instances(c) = \\
\quad\quad \{a : Resource \mid r \in sub\_val(p)(\!|\, \{a\} \,|\!)\}
\end{array}
$$

There are also a number of cardinality-related properties in DAML+OIL that define a class through constraining the cardinality of the set of resources mapped by a property to its instances. For example, the *cardinality* property defines the class c of all resources that have exactly n distinct values for the property p, i.e. a is an

instance of the defined class if and only if there are n distinct values y such that (x, y) is an instance of p.

$$
\begin{array}{|l}
cardinality : (Property \times \mathbb{N}) \rightarrow Class \\
\hline
\forall\, n : \mathbb{N};\ p : Property;\ c : Class \bullet cardinality(n, p) = c \Leftrightarrow \\
\qquad instances(c) = \{a : Resource \mid \#(sub\_val(p)(\!|\ \{a\}\ |\!)) = n\}
\end{array}
$$

Other similar properties such as `minCardinality` and `maxCardinality` and their qualified variations can be similarly defined.

3.1.4 Property Elements

DAML+OIL also defines properties to restrict and relate existing properties.

The property *subPropertyOf* states that a property p_1 is a sub property of another property p_2 if and only if $sub\_val(p_1)$ is a subset of $sub\_val(p_2)$.

$$
\begin{array}{|l}
subPropertyOf : Property \leftrightarrow Property \\
\hline
\forall\, p_1, p_2 : Property \bullet p_1\ \underline{subPropertyOf}\ p_2 \Leftrightarrow \\
\qquad sub\_val(p_1) \subseteq sub\_val(p_2)
\end{array}
$$

The *inverseOf* property defines one property to be the inverse of another one by reversing the mappings these two properties define.

$$
\begin{array}{|l}
inverseOf : Property \leftrightarrow Property \\
\hline
\forall\, p1, p2 : Property \bullet p1\ \underline{inverseOf}\ p2 \Leftrightarrow \\
\qquad (sub\_val(p1)) = (sub\_val(p2))^{\sim}
\end{array}
$$

Similarly, *TransitiveProperty* defines the condition of a property being transitive.

$$TransitiveProperty : \mathbb{P}\, Property$$

$$\forall\, p : Property \bullet p \in TransitiveProperty \Leftrightarrow (\forall\, x, y, z : Resource \bullet (x, y) \in sub\_val(p) \land (y, z) \in sub\_val(p) \Rightarrow (x, z) \in sub\_val(p))$$

3.1.5 Instances

Properties under this section relate individuals in one way or the other. For example, *differentIndividualFrom* is a property over resources. It asserts that two individuals are different from each others.

$$differentIndividualFrom : Resource \leftrightarrow Resource$$

3.2 Import Mechanisms & Proof Support

The Z semantics is contained in a Z section `daml2z`, on top of the built-in section `toolkit`. As suggested in [85], definitions alone are not sufficient to exploit the full power of Z/EVES. An ample stock of rewrite rules, forward rules and assumption rules is needed to make proof processes more automated. Based on the semantic model, we constructed a Z section, called `DAML2ZRules`, of rules which describe the above definitions in more than one angle and are used to help Z/EVES to perform reasoning tasks. This section has `daml2z` as parent.

For example, `toClassDisjointWithRule1` is a rewrite rule relating two properties: *toClass* and *disjointWith*. It states that if classes c_3 and c_2 are disjoint and (c_1, p) is related by *toClass* to c_3, then (c_1, p) cannot be related by *toClass* to c_2.

theorem rule toClassDisjointWithRule1
$\forall c_1, c_2, c_3 : Class;\ p : Property \bullet$
$(c_2, c_3) \in disjointWith \land toClass(c_1, p) = c_3 \Rightarrow toClass(c_1, p) \neq c_2$

Ontologies in the Semantic Web are open, shared and reused. New ontologies are built on top of existing ones. Other domain specific ontologies are built in terms of basic concepts presented in this section and their corresponding Z models will have `DAML2ZRules` or its descendent sections as parents.

3.3 Military Plan Ontologies

DSO National Laboratories (DSO) Singapore developed a DAML+OIL military plan ontology [60], defining concepts in the military domain, including military organizations, specialities, geographic features, etc. For example, the class `MilitaryTask` is defined as follows. It is a sub class of `MilitaryProcess`,

```
<daml:Class rdf:about="http://www.dso.org.sg/PlanOntology#MilitaryTask">
  <rdfs:label>MilitaryTask</rdfs:label>
  <rdfs:subClassOf>
    <daml:Class rdf:about="http://www.dso.org.sg/PlanOntology#MilitaryProcess"/>
  </rdfs:subClassOf>
</daml:Class>
```

The military plan ontology contains 98 classes, 26 properties and 34 individuals. The OWL classes define the classification of military formations, military tasks, geographic features, etc. The properties relate military units to tasks, defines chain of command, etc. The individuals are mostly used to represent the military specialities.

A number of plan instances of this ontology were also generated from plain text by an information extraction (IE) engine developed by DSO. Military plans are typically prepared as both graphical overlays and textual documents detailing the plans. IE is

used to transform the textual documents into ontological data. A typical IE workflow consists of word segmentation & stemming, PoS (Part of Speech) tagging, Named Entity recognition, etc. With all information gathered from the various steps, the IE engine then fills the slots in pre-defined templates. Each template specifies the slots to be emitted and the semantic classes of the value used to fill each slot. The output of the IE engine is a document containing a set of records. Each record created based on the templates contains key–value pairs. The first word on each line is the *key* and the rest of the line is the *value* of the *key*. An example of the record emitted by the IE engine is given in Fig. 3.1. Basically, the above IE output describes a movement military plan, starting at time point 0 and ending at time point 1, of one infantry battalion (1 Inf BN) to EASTLAND.

Action	PLAN-P1-P1
Annotation	moving 1 x Bn (-) to EASTLAND
Location	EASTLAND
Name	moving
End	1
Begin	0
Actor	1 INF BN
SubAction	PLAN-P1-P1-P1
Next	PLAN-P1-P2

Figure 3.1: Sample IE output

The entities described in each record from the IE output is mapped to concepts and relations found in the plan ontology. For example the value `INF BN` has a mapping to the concept `InfantryBattalion`. When this value is found in the slot of a record, an instance of `InfantryBattalion` is created. The *key* of each record is mapped to a relation in the plan ontology. As the record references other records (e.g. actions and subactions) whose types are unknown at the point of processing, typeless instances are created. The types of these instances are revised when sufficient information are available to determine their types. Jena [51] is used to hold and output the instances into an RDF file, which usually comprises the following four parts:

- A set of military operations and tasks, defining their types, phases and the logic order.
- A set of military units, which are the participants of the military operations and tasks,
- A set of geographic locations, where such operations take place and
- A set of time points for constraining the timing of such operations.

3.4 Transformation from DAML+OIL/RDF to Z

We have developed a tool (a part of the SESeW tool suite to be presented later in Chapter 6) in Java to automatically transform ontologies into Z. Given a DAML+OIL or RDF ontology, it iterates through all elements and transforms them into Z definitions.

We used this tool to transform the military plan ontology into Z section `military`, with `DAML2ZRules` as parent. To better utilize Z/EVES's proof power, We made the following enhancements to the `military` section:

- During transformation, *labels* are systematically added to Z predicates, making them axioms (either rewrite rules or assumption rules) recognized by Z/EVES, which will assume an assumption rule to be true and rewrite the left-hand side of a rewrite rule to its right-hand side during the proof process.
- Since `MilitaryProcess` and its sub classes have a start and end time, `start` and `end` are modeled as functions from `MilitaryProcess` to integer, so that Z/EVES can perform reasoning over integer domain.
- A set of theorems specific to these military definitions are formulated. These theorems describe the relationships among the various military entities. For

example, we have theorems stating sub task relationship between different kinds of military tasks, transitivity of sub task relationship, etc.

For example, the class `MilitaryTask` presented earlier is transformed into the following axiomatic definition. Note that the predicate is marked as an assumption rule, which is automatically assumed to be true by Z/EVES during reduction and rewriting.

$$MilitaryTask : Class$$

$$\langle\!\langle \text{grule MilitaryTask\_subClassOf\_MilitaryProcess} \rangle\!\rangle$$
$$(MilitaryTask, MilitaryProcess) \in subClassOf$$

SESeW also transforms instance RDF ontologies into Z specifications, in which additional Z predicates are added to make the reasoning process of Z/EVES more automated.

In RACER and many other description logics reasoners, different names refer to different entities (Unique Name Assumption [36]). However, in Z, different names can refer to the same entity. We use cardinality of sets to make Z/EVES work the same way. For example, in the instance ontology, whenever two military tasks are related by sub task or super task relationship, we construct a set containing the two tasks and assume the cardinality of the set is two, as follows:

$$\langle\!\langle \text{grule ECA\_P3\_P13\_S1\_disj\_ECA\_P3\_P13} \rangle\!\rangle$$
$$\#\{ECA\_P3\_P13\_S1, ECA\_P3\_P13\} = 2$$

3.5 Checking DAML+OIL Ontologies using Z/EVES

This section gives a concise account of our work in checking DAML+OIL ontologies using Z/EVES [24]. The presentation is focused on performing the core Semantic Web reasoning tasks, namely inconsistency, subsumption, instantiation and instance-property reasoning, over the military plan ontology.

3.5.1 Inconsistency Checking

Ensuring the consistency each class is an important task as the overall ontology consistency can be reduced to class consistency problem [46].

After transforming the plan ontology into Z section `military`, We applied Z/EVES to section `military` to systematically check consistency for its classes. During checking, we identified the following closely-related Z definitions.

$$\frac{PrepareDemolition\_MilitaryTask : Class}{(PrepareDemolition\_MilitaryTask, MilitaryTask) \in subClassOf}$$

$$\frac{EngineerUnit : Class}{\begin{array}{l}(EngineerUnit, ModernMilitaryUnit) \in subClassOf \\ \langle\!\langle \text{grule EngineerUnitSpeciality} \rangle\!\rangle \\ ((EngineerUnit, speciality), EngineeringMilitarySpeciality) \in hasValue \\ \langle\!\langle \text{grule DemolitionAssignedtoEngin} \rangle\!\rangle \\ ((PrepareDemolition\_MilitaryTask, assignedTo), EngineerUnit) \in toClass\end{array}}$$

$$\frac{EngineerSection : Class}{\begin{array}{l}\langle\!\langle \text{grule SectionIsSubClassOfUnit} \rangle\!\rangle \\ (EngineerSection, EngineerUnit) \in subClassOf \\ ((EngineerSection, echelon), SECT) \in hasValue\end{array}}$$

$$
\begin{array}{|l}
ArtilleryFiringUnit : Class \\
\hline
\langle\!\langle \text{FUIsMUnit} \rangle\!\rangle \\
(ArtilleryFiringUnit, ModernMilitaryUnit) \in subClassOf \\
\langle\!\langle \text{grule FiringUnitDisjWithEngin} \rangle\!\rangle \\
(ArtilleryFiringUnit, EngineerUnit) \in disjointWith \\
\langle\!\langle \text{grule DemolitionAssignedToFU} \rangle\!\rangle \\
((PrepareDemolition\_MilitaryTask, assignedTo), ArtilleryFiringUnit) \in toClass
\end{array}
$$

With the assumption rule label DemolitionAssignedToFU removed, we issue the following command to test the consistency of the above definitions.

$$try\ (((PrepareDemolition\_MilitaryTask, assignedTo), ArtilleryFiringUnit) \in toClass);$$

We enter a sequence of commands into Z/EVES. The first 2 are axioms (labelled predicates) from the specification and the 3^{rd} is a theorem defined in section `DAML2ZRules`. The final command `reduce` performs simplification and rewriting.

Proof
use FiringUnitDisjWithEngin;
use DemolitionAssignedtoEngin;
*apply disjointWithRule*0;
reduce;

Z/EVES returns the following predicate as the remaining goal to be proven.

$$\neg\ (instances\ EngineerUnit \cap instances\ ArtilleryFiringUnit) = \{\}$$

We suspect that there is potentially an inconsistency since the disjointness of the above two classes is stated in the specification. Since it is very hard for a theorem prover to prove falsity, we use the usual trick: negate the goal and retry.

$try\ (\neg\ ((PrepareDemolition\_MilitaryTask, assignedTo), ArtilleryFiringUnit) \in toClass);$

With the same sequence of commands entered, Z/EVES manages to return `true`. Hence we know that the predicate is inconsistent with the section. After checking the original ontology, we found that there is indeed an inconsistency, which was intentionally inserted as a test case for our tool without our knowledge.

3.5.2 Subsumption Reasoning

The task of subsumption reasoning is to infer that a DAML+OIL class is a sub class of another class. It is supported by Z/EVES with a high degree of automation: usually a `reduce` command will prove the goal.

3.5.3 Instantiation Reasoning

Instantiation reasoning asserts that one resource is an instance of a class. Some Semantic Web reasoning tools, such as FaCT, are designed to only support TBox reasoning, hence reasoning involving instances cannot be performed. We demonstrate through an example that Z/EVES supports instance level reasoning.

In one of the instance ontologies, `planE.daml`, an instance of `ModernMilitaryUnit` is assigned to an instance of `PrepareDemolition_MilitaryTask`. We want to deduce that it is an instance of the class EngineerUnit (since we know from one assumption given in the previous section, that every instance of `EngineerUnit` is assigned to some instance of `PrepareDemolition_MilitaryTask`).

$$
\begin{array}{|l}
ModernMilitaryUnit\_8ad : Resource \\
\hline
\langle\!\langle \text{grule ModernMilitaryUnit\_8ad\_type} \rangle\!\rangle \\
ModernMilitaryUnit\_8ad \in instances(ModernMilitaryUnit)
\end{array}
$$

$$
\begin{array}{|l}
PLAN\_P2\_P4 : Resource \\
\hline
\langle\!\langle \text{grule PLAN\_P2\_P4\_type} \rangle\!\rangle \\
PLAN\_P2\_P4 \in instances(PrepareDemolition\_MilitaryTask) \\
\langle\!\langle \text{rule PLAN\_P2\_P4\_assignedTo} \rangle\!\rangle \\
(sub\_val(assignedTo))(\!| \{PLAN\_P2\_P4\} |\!) = \{ModernMilitaryUnit\_8ad\}
\end{array}
$$

$$try\ ModernMilitaryUnit\_8ad \in instances(EngineerUnit);$$

With two axioms from the specification and two theorems from section `DAML2ZRules` used, a final `prove` command cleans up the proof and Z/EVES returns `true`.

Proof

$$
\begin{array}{l}
use\ imageTupleRule[p := assignedTo, \\
\quad x := PLAN\_P2\_P4, y := ModernMilitaryUnit\_8ad]; \\
use\ DemolitionAssignedtoEngin; \\
use\ PLAN\_P2\_P4\_type; \\
use\ toClassInstanceRule2 \\
\quad [c_1 := PrepareDemolition\_MilitaryTask, \\
\quad c_2 := EngineerUnit, a_1 := PLAN\_P2\_P4, \\
\quad a_2 := ModernMilitaryUnit\_8ad, p := assignedTo]; \\
prove; \\
\blacksquare
\end{array}
$$

3.5.4 Instance Property Reasoning

Another important reasoning task in the Semantic Web domain is instance property reasoning, which is often regarded as knowledge base querying. In the Semantic Web, a promising vision is that intelligent agents can infer information that is not explicitly

stored in the knowledge base. We illustrate Z/EVES's capability of instance property reasoning using an example.

In the beginning of this section, we know that the speciality of `EngineerUnit` is `EngineeringMilitarySpeciality` and that `EngineerSection` is a sub class of `EngineerUnit`. We want to know whether `EngineeringMilitarySpeciality` is also a speciality of `EngineerSection`. The goal is established as follows:

$try\ ((EngineerSection, speciality), EngineeringMilitarySpeciality) \in hasValue;$

With the following commands issued, Z/EVES proves the goal to be `true`.

Proof
use $EngineerUnitSpeciality$;
use $SectionIsSubClassOfUnit$;
use $subClassHasValueRule1$
$[c_1 := EngineerSection, c_2 := EngineerUnit,$
$p := speciality, r := EngineeringMilitarySpeciality]$;
reduce;
■

As it can be seen, the highly interactive proof process and the potentially large size of ontologies make it difficult to be applicable in the SW environment. The work introduced in this chapter inspired us to propose the combined approach presented in the next chapter, which is more effective and efficient as it is able to check more complex properties and ontological properties with high automation.

3.6 Chapter Summary

The main contribution of this chapter can be summarized as follows.

- The Z semantics for the ontology language DAML+OIL is defined, which is the foundation for the later work on checking Web ontologies using Z/EVES, a theorem prover for Z language.

- A Java transformation tool from DAML+OIL and RDF to Z is developed, making this checking approach easier as large ontologies can be automatically transformed into Z specifications ready to be checked by Z/EVES.
- The checking of core Semantic Web reasoning tasks, including concept inconsistency, subsumption, instantiation, etc., by Z/EVES is another contribution of this chapter. It shows that software engineering languages and tools can contribute to the development of the Semantic Web.

As it can be seen from the last section, the proof process in this Z/EVES-only approach is very interactive and it requires substantial user expertise in interacting with the theorem prover.

Although Semantic Web reasoners such as RACER and FaCT++ can carry out only a limited number of types of reasoning tasks (concept consistency, subsumption and instantiation reasoning), due to the expressivity limitation of the ontology languages, they are fully automated reasoners. It is advantageous to use SW reasoners to perform reasoning tasks that can be automated.

Moreover, since ontology languages are based on description logics, certain complex properties cannot be represented in these languages. We need a way to express and verify the desirable properties, which may be critical to assuring the correctness of the ontology.

The above two requirements inspired us to harness the synergy of Semantic Web reasoners and software engineering proof tools for better automation, expressivity and debugging aid.

Chapter 4

A Combined Approach to Checking Web Ontologies

Ontology languages such as DAML+OIL and (a subset of) OWL were designed [40] to be decidable so that core reasoning tasks such as subsumption and instantiation can be carried out with full automation. However, decidability is achieved by limiting the expressivity of these languages. An obvious shortcoming with this design decision is that certain very desirable properties associated with ontologies cannot be expressed in these languages. Consequently, they cannot be checked by Semantic Web reasoning engines such as RACER or FaCT++. For example, in the military plan ontologies case study presented in the previous chapter, it is important to ensure that no single military unit is assigned to two different military tasks (that may be at different locations) at the same time. This property cannot be expressed in DAML+OIL or OWL but is very important to the validity of the military plan.

Based on the previous chapter, we observe that there is a complementary power between software engineering proof tools (Z/EVES and Alloy Analyzer) and Semantic

Web reasoning engines such as RACER and FaCT++. As formal languages such as Z and Alloy can express more complex properties ontology languages cannot, Z/EVES and Alloy Analyzer can be used to verify the correctness of these properties. Semantic Web reasoning engines can automatically detect any ontological inconsistencies. Moreover, Alloy Analyzer is able to locate the source of the error, making debugging inconsistent ontologies easier.

As introduced in Chapter 2, the proposed rules extension to OWL, the SWRL (ORL originally) partially solves the problem by adding Horn-style rules to OWL.

Although at the time of writing, the military ontologies were developed in DAML+OIL format, it is almost a trivial task to update it to OWL format. Hence, this does not present any challenge for incorporating SWRL into the picture.

In order to use software engineering tools such as Z/EVES and Alloy Analyzer to check SWRL and DAML+OIL ontology-related properties, it is the necessary first step to define Z and Alloy semantics for SWRL and DAML+OIL vocabularies. In this chapter, part of the Alloy semantics for DAML+OIL, given in `teletype` font, and Z semantics for SWRL will be presented. The full semantics can be found in [27, 24].

After introducing the semantics of DAML+OIL and SWRL in Sections 4.1 and 4.2, the transformation process from DAML+OIL to Alloy and SWRL to Z in Section 4.3, we proceed to present the combined approach using RACER, Z/EVES and Alloy Analyzer in Section 4.4. The approach will be illustrated in detail by presenting how it can be applied to verifying both plan ontology and instance ontologies.

4.1 Alloy Semantics for DAML+OIL

In this section, the Alloy semantics for DAML+OIL is briefly presented. More details can be found in [102]. The structure of this section closely follows that of Section 3.1 as the Alloy semantics for DAML+OIL is similar to that of Z.

Basic Concepts

We model `Resource` as a given type in Alloy.

```
sig Resource {}
```

In Alloy, we model `Class` as a subset of resource and `instances` a relation such that each `Class` maps a set of resources via the relation `instances`, which contains all the instance resources. The keyword `disj` is used to indicate that `Class` and `Property` are disjoint, meaning that any member of type `Class` is not a member of `Property`, and vice versa.

```
disj sig Class extends Resource
    {instances: set Resource}
```

As in Z, `Property` is model as another subset of `Resource`, which is disjoint with `Class`. In Alloy, the keyword `disj` is used to indicate that the types `Class` and `Property` are disjoint from each other, although both of them are sub types of `Resource`. In effect, this keyword ensures that any member of the type `Class` is not a member of type `Property` and vice versa.

```
disj sig Property extends Resource
    {sub_val: Resource -> Resource}
```

The property `equivalentTo` is a property that relates two equivalent resources. It is used as a super property of *sameClassAs* and *samePropertyAs*.

```
fun equivalentTo(a, b: Resource)
    {a = b}
```

Class relationships

In Alloy, a function is used to represent the `subClassOf` concept.

```
fun subClassOf(c1, c2: Class)
    {c1.instances in c2.instances}
fun disjointWith (c1, c2: Class)
    {no c1.instances & c2.instances}
```

Class & Property

The definitions of properties `toClass`, `hasClass` and `hasValue` closely mirror those in Z.

```
fun toClass (p:Property, c1:Class, c2:Class)
{all a1, a2: Resource | a1 in c1.instances <=>
    a2 in a1.(p.sub_val) => a2 in c2.instances}


fun hasValue (p:Property, c:Class, r:Resource)
    {all a:Resource |
    a in c.instances => a.(p.sub_val) = r}


fun hasClass(p: Property, c1: Class, c2: Class)
{all r1: Resource | r1 in c1.instances =>
     some r1.(p.sub_val) & c2.instances}
```

Property relationships

The function below models the Alloy semantics for property `subPropertyOf`.

```
fun subPropertyOf (p1, p2:Property)
    {p1.sub_val in p2.sub_val}
```

Individual relationships

`differentIndividualFrom` asserts that two individuals are different from each others.

```
fun differentIndividualFrom(a,b: Resource)
    {all a, b: Thing.instances | !a = b}
```

4.1.1 Import Mechanisms & Proof Support

The Alloy semantics is contained in a module called `DAML`. Similar to the Z/EVES approach, later Alloy models transformed from DAML+OIL ontologies will import this module or its descendants to make use of the language constructs in these modules.

4.2 Z Semantics for SWRL

As introduced in Chapter 2, SWRL is an extension of OWL towards first-order logic that improves its expressivity. As a result, SWRL is able to express some complex

properties inexpressible in OWL. This section presents the Z semantics for SWRL, making the combined approach more versatile by incorporating SWRL.

In SWRL [48], a rule consists of an antecedent and a consequent, each of which contains zero or more atoms. Atoms can be of the form $C(x)$, $P(x, y)$, $sameAs(x, y)$ or $differentFrom(x, y)$, where C is an OWL (class) description (class membership), P is an OWL property (property membership), and x, y are either OWL individuals, OWL data values or SWRL variables (variables are prefixed with a question mark "?"). Informally, an atom $C(x)$ holds if x is an instance of the class description C, an atom $P(x, y)$ holds if x is related to y by property P, an atom $sameAs(x, y)$ holds if x is interpreted as the same object as y, and an atom $differentFrom(x, y)$ holds if x and y are interpreted as different objects.

Multiple atoms in antecedent are treated as a conjunction, where empty antecedent is treated as trivially true. Multiple atoms in consequent are treated as separate consequents and an empty consequent is treated as trivially false. A rule may be read as to mean that if the antecedent holds (is "true"), then the consequent must also hold.

As a result, the Z semantics of an SWRL rule is encoded as a universally quantified implication predicate, with the atoms being $\wedge$-connected. The Z semantics of SWRL rules atoms can be found in Table 4.1. Since we will only be using Z/EVES to check SWRL rules, we do not construct the Alloy semantics for SWRL, which is similar to that of Z.

The properties *sameAs* and *differentFrom* are defined in OWL, which are equivalent to *equivalentTo* and *differentIndividualFrom* in DAML+OIL, respectively.

SWRL also defines a set of built-ins that can be used as atoms. These include built-ins for comparison(equal, less than or equal to, etc.), built-ins for mathematical

Table 4.1: SWRL rules atoms in Z

SWRL Atom	**Z semantics**
$C(x)$	$x \in instances(C)$
$P(x, y)$	$(x, y) \in sub\_val(P)$
$sameAs(x, y)$	$(x, y) \in sameAs$
$differentFrom(x, y)$	$(x, y) \in differentFrom$

operations (add, subtract,power, etc.), built-ins for Boolean values and built-ins for string operations (concatenation, substring, to upper case, etc.). Most of these built-ins can be directly translated into their Z counterparts.

4.3 Transformation from Web Ontologies to Z & Alloy

4.3.1 Transformation from SWRL to Z

An SWRL rule is transformed to a rewrite rule in Z/EVES format. During proof, a rewrite rule can be invoked in Z/EVES, with its left-hand side rewritten to its right-hand side of the formula.

For example, although the military plan ontology is in DAML+OIL but not SWRL syntax, it is very natural to model some domain-specific properties using SWRL rules. For example, we can use SWRL rules to specify that if two overlapping military tasks are at different locations, then they must be assigned to different military units.

$$overlaps(?a, ?b) \land differentFrom(?c, ?d) \land location(?a, ?c) \land location(?b, ?d) \land assignedTo(?a, ?e) \land assignedTo(?b, ?f)$$
$$\rightarrow$$
$$differentFrom(?e, ?f)$$

where all the variables are instances of appropriate classes, e.g., $?a$ and $?b$ are instances of `MilitaryTask`, $?c$, $?d$ are instances of `GeographicArea` and $?e$ and $?f$ are instances of `ModernMilitaryUnit`. This information does not need to be explicitly stated as the class membership can be automatically inferred according to the OWL and SWRL semantics.

The above rule is transformed as follows:

theorem rule durationOverlapRule
$\forall a, b, c, d, e, f : Resource \bullet$
$(a, b) \in sub\_val(overlaps) \land (c, d) \in sub\_val(differentFrom) \land$
$(a, c) \in sub\_val(location) \land (b, d) \in sub\_val(location) \land$
$(a, e) \in sub\_val(assignedTo) \land (b, f) \in sub\_val(assignedTo)$
$\Rightarrow$
$differentFrom(e, f)$

4.3.2 Transformation from DAML+OIL to Alloy

The transformation from DAML+OIL & RDF ontologies to Alloy is straightforward. Unlike Z, definitions of a name in Alloy does not need to appear before this name is referenced. Hence, only one pass is required to correctly transform the ontology into Alloy. The military ontology is transformed into a module `military`. The class `MilitaryTask` is transformed into the following Alloy definition. Note that it is a sub class of `MilitaryProcess`.

```
static disj sig MilitaryTask extends Class {}
fact{subClassOf(MilitaryTask, MilitaryProcess)}
```

4.4 The Combined Approach to Checking Web Ontologies

4.4.1 An Overview of the Combined Approach

In this section, we present the approach of checking DAML+OIL ontologies and using tools RACER, OilEd, Z/EVES and Alloy Analyzer in conjunction. Moreover, we also discuss how SWRL rules can be used to model properties that may be of interest in the military domain and how Z/EVES can be used to check these rules.

Given an ontology, the combined approach performs the following steps:

1. We transform it to a Z specification and use Z/EVES as a type checker to check for syntax and type errors. Any such error found by Z/EVES is corrected back in the original ontology. Z/EVES performs the type checking automatically.

   The purpose of this step is to remove trivial errors before actual checking is performed. Sometimes, type errors are caused by implicit facts in the ontology. Some properties are also redefined wrongly. For example, in the instance ontology (ABox) `planA.owl`, the datatype property `end`, which maps a military process to its end time point, is erroneously redefined as an object property. This kind of errors can be discovered automatically and corrected accordingly.

   For example, in the instance ontology planA.daml, the resource `ECA-P2-P7` is an instance of class `Thing`. However, it is defined for the property `start`, whose `domain` is instances of class `MilitaryProcess` and its sub classes. If RACER is queried whether `ECA-P2-P7` is an instance of `MilitaryProcess`, it will return true and hence this fact is implicit and assumed. However, if similar query is issued to Z/EVES, it will complain that `ECA-P2-P7` is not well typed. The revelation of implicit facts helps human to understand the ontology better.

2. We input the trivial-errors-corrected ontology into an ontology editor, such as OilEd, and connect it to RACER to *classify* it. In this step, RACER performs consistency, subsumption and instance checking, which automatically decides whether there are ontological inconsistencies.

   RACER reports any inconsistent classes. However, it is unable to tell where the error lies. OilEd as an ontology editor collects information related to each individual class and property, and that information about the inconsistent entity is used in the next step to guide the identification of possible source of the inconsistency (see next step).

3. For each inconsistency, as described in the previous step, OilEd returns a minimal set of classes, properties and instances that constrain the offending concept. Then we employ Alloy Analyzer to analyze the isolated ontology fragment to determine the source of the error.

   Our past experiences showed that the root cause of an ontological inconsistency can often be revealed within a few classes & properties. In most cases, Alloy Analyzer can pinpoint certain classes and properties which cause the inconsistency.

   If Alloy Analyzer does not detect an error, we need to iteratively augment the fragment ontology by referring to OilEd and including classes, properties and instances related to existing definitions. This step requires human interaction but it can be handled with relative ease.

   When Alloy Analyzer detects the inconsistency, it does more by indicating how it is caused. A number of statements related to the inconsistency in the Alloy specification, and possibly imported modules, are highlighted. With this help, we return to OilEd and RACER to correct the original ontology.

   If the fragment ontology is too large for Alloy Analyzer to analyze, we use Z/EVES as a theorem prover to determine the source of the inconsistency, which

requires substantial expertise in interacting with Z/EVES.

Steps 2) and 3) are iterated until no ontological inconsistencies are found. These steps are presented in detail in Section 4.4.2.

4. Finally, we use Z/EVES again to check properties beyond the modeling capability of DAML+OIL and Alloy. As stated in Chapter 2, Z is a superset of ontology languages and Alloy and it can capture a richer set of information, which is sometimes crucial to the correctness of the ontology.

   This step is domain-specific and it requires thorough understanding of the domain. For the military plan ontologies case study, we have constructed a set of theorems in Z/EVES and used them to systematically test the correctness of the instance.

   By capturing properties that cannot be expressed by DAML+OIL using Z, we actually treat Z as an ontology language but with increased expressiveness, at the cost of decidability and automation. The benefit of the gained expressiveness is domain-specific and it is exemplified in our case study in Section 4.4.3.

In the rest of this chapter, we use the military plan ontologies case study to demonstrate this approach.

4.4.2 Checking Military Plan Ontology

In this subsection, we illustrate the application of the combined approach on the military planning ontology introduced in Section 3.3.

Firstly, we transform this ontology into the corresponding Z section `military`. With order of some Z definitions swapped, Z/EVES accepts this Z section, which means, that the Z section does not contain any syntactic or type errors. The absence of such

errors is due to the reason that this ontology is visually developed with the help of OilEd, and is not produced by the IE engine.

Secondly, we open OilEd and connect it to RACER. We then load the ontology into OilEd and use RACER to *classify* it, as described in step 2) of Section 4.4.1. OilEd instantly reports one unsatisfiable concept, as Fig. 4.1 shows.

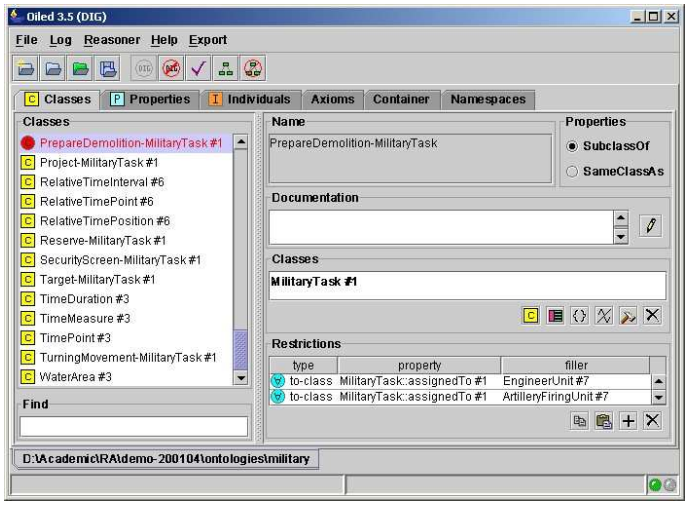

Figure 4.1: Discovery of an unsatisfiable concept by RACER

Shown in Fig. 4.1, `PrepareDemolition-MilitaryTask`, the first class on the left panel, is highlighted in red color by OilEd as an inconsistent class. Restrictions imposed on this class are displayed at the bottom on the right.

RACER flags the class `PrepareDemolition-MilitaryTask` as inconsistent. However, it cannot determine exactly where the inconsistency comes from. In the next step, we employ Alloy Analyzer to pinpoint the source of the inconsistency.

Thirdly, we extract a small ontology fragment containing definitions of the offending class and those classes, properties and instances appearing in the **Restrictions** panel, namely `assignedTo`, `EngineerUnit` and `ArtilleryFiringUnit`. This fragment is subsequently transformed into an Alloy module shown in Fig. 4.2, which is loaded

into Alloy Analyzer to check for inconsistency.

```
module inconsistency_military open demo1/library/DAML

static disj sig MilitaryTask extends Class {}
static disj sig PrepareDemolition_MilitaryTask extends Class {}
fact {subClassOf(PrepareDemolition_MilitaryTask, MilitaryTask)}
static disj sig assignedTo extends Property {}
static disj sig ModernMilitaryUnit extends Class{}
static disj sig EngineerUnit, ArtilleryFiringUnit extends Class{}
fact {subClassOf(ArtilleryFiringUnit, ModernMilitaryUnit)}
fact {subClassOf(EngineerUnit, ModernMilitaryUnit)}
static disj sig EngineeringMilitarySpeciality extends Resource {}
static disj sig speciality extends Property {}
fact{hasValue (speciality, EngineerUnit, EngineeringMilitarySpeciality)}

fact {disjoinWith(ArtilleryFiringUnit, EngineerUnit)}
fact {toClass(assignedTo, PrepareDemolition_MilitaryTask, ArtilleryFiringUnit)}
fact {toClass(assignedTo, PrepareDemolition_MilitaryTask, EngineerUnit)}
fact {some (PrepareDemolition_MilitaryTask.instances).(assignedTo.sub_val)}

fun dummy() {} run dummy for 15
```

Figure 4.2: Alloy concepts related to the inconsistency

Basically speaking, this fragment of ontology states the following facts.

1. `PrepareDemolotion_MilitaryTask` is a sub class of `MilitaryTask`.
2. Both DAML+OIL classes `ArtilleryFiringUnit` and `EngineerUnit` are sub classes of `ModernMilitaryUnit` and that they are *disjoint* with each other.
3. All instances of the class `PrepareDemolotion_MilitaryTask` are assigned to some instances of `ArtilleryFiringUnit`.
4. All instances of the class `PrepareDemolotion_MilitaryTask` are assigned to some instances of `EngineerUnit`.
5. There exist some instances of class `PrepareDemolotion_MilitaryTask` that have been `assignedTo` some units (the last fact). This fact is necessary because of the definition of `allValuesFrom` (see Section 3.1 for details), which

states that if a property (`assignedTo` in this case) is not defined for an individual, it *is* an instance of the target class (`PrepareDemolition_MilitaryTask` in this case). Hence, this fact rules out the individuals that are not in the domain of `assignedTo`.

In the military domain, the engineer units (represented by the DAML+OIL class `EngineerUnit` are solely responsible for the task of preparation of demolition of targets (represented by the DAML+OIL class `PrepareDemolition_MilitaryTask`) using explosives. Intuitively, a unit that is responsible for firing weapons such as large mounted guns and cannons should not be assigned to the above task. Hence, the DAML+OIL fragment is inconsistent because of the third fact above.

Alloy Analyzer detects the inconsistency by its inability to find a *solution* that satisfies all facts within the given scope. It may be due to the scope being too small. To determine the reason, we use Alloy Analyzer's utility "Determine unsat core" to trace the source of the error. In an unconvincing case, we increase the scope and run Alloy Analyzer again.

Fig. 4.3 shows how Alloy Analyzer determines which facts caused the problem. When a clause is clicked, Alloy Analyzer automatically highlights the corresponding statement in the left panel. Arrows are added in the figure to show this correspondence. After examining the clauses in red, we found that the 4 clauses (`_Fact_144` to `_Fact_147`) with arrows attached actually caused the problem. Hence, the lack of solution was indeed due to the inconsistency of the original ontology. The inconsistency is caused by assigning `PrepareDemolition_MilitaryTask` to both classes `ArtilleryFiringUnit` and `EngineerUnit`, which are `disjointWith` each other. Hence, by removing any of the two assignments, the fact of disjointness or the fact that some instances of `EngineerUnit` being assigned, the inconsistency can be eliminated. Since the source of the inconsistency is discovered by Alloy Analyzer, we need not return

to Z/EVES, in this case.

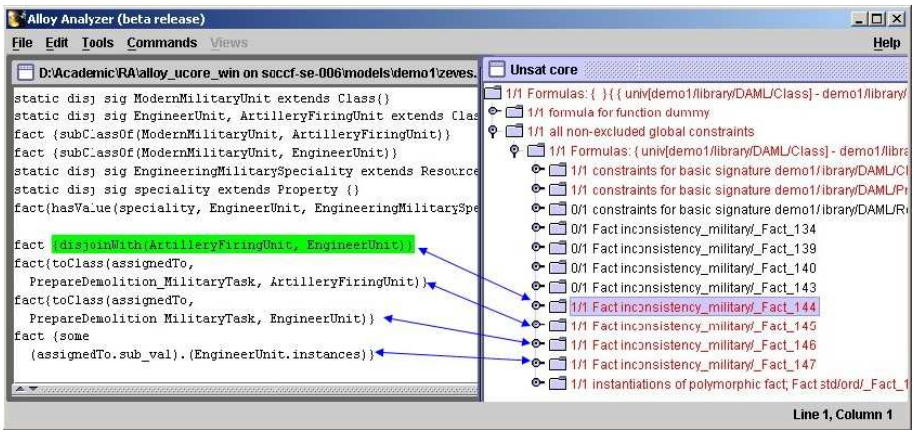

Figure 4.3: Alloy Analyzer showing the source of unsatisfiability

After checking the original ontology, we found that `ArtilleryFiringUnit` is mistakenly assigned to `PrepareDemolition_MilitaryTask`. After this fact is removed, RACER confirms that the ontology is satisfiable.

From this example, we can see that the fact, that an inconsistency is caused by two disjoint military unit classes being assigned to the same military task class, is rather implicitly captured. With the help of SWRL rules, this property can be expressed much explicitly.

$$\begin{aligned} &\mathit{EngineerUnit}(?a) \wedge \mathit{PrepareDemolition\text{-}MilitaryTask}(?b) \wedge \\ &\mathit{assignedTo}(?b, ?a) \wedge \mathit{assignedTo}(?b, ?c) \\ &\quad \rightarrow \\ &(\mathit{complementOf}\ \mathit{ArtilleryFiringUnit})(?c) \end{aligned}$$

This SWRL rule states that if individual $?a$ is an instance of `EngineerUnit`, $?b$ is an instance of `PrepareDemolition-MilitaryTask` and $?a$ and $?c$ are both assigned to $?b$, then we can conclude that $?c$ is not an instance of `ArtilleryFiringUnit`. Since SWRL disallows the use of negation, we cannot directly state that the consequent is $\neg\ \mathit{ArtilleryFiringUnit}(?c)$. Instead, we can use the OWL class description to construct an anonymous class to be the *complement of* `ArtilleryFiringUnit` and then

make individual $?c$ an instance of this class. The interpretation of the complement is the universal set of individuals (instances of class `Thing`) minus those belonging to the class `ArtilleryFiringUnit`. Although the idea of this SWRL rule can be captured by the DAML+OIL (OWL DL) ontology, the presence of this rule explicitly groups relevant information together, serving as a formal documentation to prevent the wrong assignment of `ArtilleryFiringUnit`.

Lastly, we use Z/EVES to model and prove the SWRL rule we just mentioned. This rule can be translated into a Z/EVES theorem as follows. Ten (parameterized) Z/EVES commands prove the theorem.

theorem rule PrepareDemolitionAssignmentRule
$\forall a, b, c : Resource \bullet$
$a \in instances(EngineerUnit) \land b \in instances(PrepareDemolition\_MilitaryTask) \land$
$(b, a) \in sub\_val(assignedTo) \land (b, c) \in sub\_val(assignedTo)$
$\Rightarrow$
$c \in (instances(Thing) \setminus instances(ArtilleryFiringUnit))$

It can be seen that the proof of this theorem requires user ingenuity and expertise. Compared to the RACER/Alloy reasoning, this certainly requires more time and manpower. The advantage of using Z/EVES is that the property can be explicitly stated for better management and documentation.

4.4.3 Reasoning About More Complex Properties

In this subsection, we discuss how Z/EVES is used to reason about more complex properties that DAML+OIL cannot express. This reasoning task is applied to an instance of the military plan ontology: `planA.daml`.

To ensure the correctness of military plan ontologies, it is not enough just to perform checking using Alloy Analyzer and RACER. One requirement in the military planning exercises is, for example, that no military unit is assigned to two or more military tasks

at the same time, and that no military task is a sub task of itself. By performing the last step of the approach, we discovered a number of such errors beyond the modeling capabilities of DAML+OIL and Alloy.

The first three steps are not shown in order to concentrate on the final step of our approach. In the first three steps, we performed the usual transformation and checking and obtained an ontological-error-free document. It was then transformed into a Z section. Part of this ontology and the corresponding Z definitions are shown below.

```
<rdf:Description rdf:about='ECA-P1-P2-P2-S1'>
  <NS4:subTaskOf rdf:resource='ECA-P1-P2'/>
  <NS4:subTaskOf rdf:resource='ECA-P1-P2-P2'/>
  <NS4:location rdf:resource='E. AFRICA'/>
  <NS4:target rdf:resource='E. AFRICA'/>
  <rdf:type rdf:resource='http://www.dso.org.sg/
     PlanOntology#HastyDefend-MilitaryTask'/>
  <NS0:start rdf:resource='0'/>
  <NS0:end rdf:resource='15'/>
  <NS4:assignedTo rdf:resource='InfantryBattalion_aa5'/>
</rdf:Description>
<rdf:Description rdf:about='G. SMILAX'>
  <rdf:type rdf:resource='http://www.dso.org.sg/
     PlanOntology#AxisOfAdvance'/>
</rdf:Description>
<rdf:Description rdf:about='InfantryBattalion_aa5'>
  <rdf:type rdf:resource='http://www.dso.org.sg/PlanOntology#InfantryBattalion'/>
</rdf:Description>
```

The above DAML+OIL ontology fragment describes an individual `ECA-P1-P2-P2-S1`, an instance of the class `HastyDefend-MilitaryTask`. Its start and end time points, location, relationships to other tasks and assignment information are also described. The fragment also describes a geographic feature `G. SMILAX` and an infantry battalion.

$$
\begin{array}{|l}
ECA\_P1\_P2\_P2\_S1 : Resource \\
\hline
\langle\!\langle \text{grule ECA\_P1\_P2\_P2\_S1\_type} \rangle\!\rangle \\
ECA\_P1\_P2\_P2\_S1 \in instances(HastyDefend\_MilitaryTask) \\
\langle\!\langle \text{rule ECA\_P1\_P2\_P2\_S1\_start} \rangle\!\rangle \\
start(ECA\_P1\_P2\_P2\_S1) = 0 \\
\langle\!\langle \text{rule ECA\_P1\_P2\_P2\_S1\_assignedTo} \rangle\!\rangle \\
(sub\_val(assignedTo))(\!| \{ECA\_P1\_P2\_P2\_S1\} |\!) = \\
\quad \{InfantryBattalion\_aa5\} \\
\langle\!\langle \text{rule ECA\_P1\_P2\_P2\_S1\_end} \rangle\!\rangle \\
end(ECA\_P1\_P2\_P2\_S1) = 15 \\
\langle\!\langle \text{rule ECA\_P1\_P2\_P2\_S1\_target} \rangle\!\rangle \\
(sub\_val(target))(\!| \{ECA\_P1\_P2\_P2\_S1\} |\!) = \{E\_AFRICA\} \\
\langle\!\langle \text{rule ECA\_P1\_P2\_P2\_S1\_location} \rangle\!\rangle \\
(sub\_val(location))(\!| \{ECA\_P1\_P2\_P2\_S1\} |\!) = \{E\_AFRICA\}
\end{array}
$$

$$
\begin{array}{|l}
G\_SMILAX : Resource \\
\hline
\langle\!\langle \text{grule G\_SMILAX\_type} \rangle\!\rangle \\
G\_SMILAX \in instances(AxisOfAdvance)
\end{array}
$$

$$
\begin{array}{|l}
InfantryBattalion\_aa5 : Resource \\
\hline
\langle\!\langle \text{grule InfantryBattalion\_aa5\_type} \rangle\!\rangle \\
InfantryBattalion\_aa5 \in instances(InfantryBattalion)
\end{array}
$$

$$
\begin{array}{l}
\langle\!\langle \text{rule ECA\_P1\_P2\_P2\_S1\_subTaskOf} \rangle\!\rangle \\
(sub\_val(subTaskOf))(\!| \{ECA\_P1\_P2\_P2\_S1\} |\!) = \\
\quad \{ECA\_P1\_P2, ECA\_P1\_P2\_P2\}
\end{array}
$$

It may be noted that the `subTaskOf` statement is modeled in a separate Z predicate at the end. Actually all `subTaskOf` statement are extracted and put to the end of the Z specification to prevent circular or advance reference of military tasks.

The brief statistics of the ontology and the Z section is shown in Table 4.2.

Note that there is a decrease in number of Z predicates from that of RDF statements. There are two reasons: (1) statements with properties `comment` and `label` are not transformed to Z since they are just textual descriptions of the subject; (2) statements

Table 4.2: Statistics of the ontology planA.daml

Items	**Numbers**
Resources	195
Operations, tasks, phases	78
Units	69
Geographic areas	48
Statements (in RDF)	954
Transformed Axiomatic Defns (in Z)	195
Transformed Predicates (in Z)	766
Type errors	8
Hidden errors	***12***

such as `subTaskOf` and `assignedTo` for any one instance are grouped to form one Z predicate, as shown in the above rewrite rule `ECA_P1_P2_P2_S1_subTaskOf`.

Firstly, twenty-eight type errors were discovered by Z/EVES in step 1. Most of these errors are caused by the inaccuracy of the IE engine. For example, `Coastal_Hook_Force` is defined as a class in the plan ontology; it is redefined as a resource of type `Thing` in this instance ontology. Although the user may have wanted to redefine `Coastal_Hook_Force` as `Thing`, it is very unlikely since no semantic significance is added and the ontology becomes harder to comprehend. Conservatively, we treat this redefinition as an error.

In step 1, implicit facts are also made explicit by Z/EVES. For example, the type of one of the military tasks `ECA-P1-P4-P1` was `Thing` in the instance ontology, it is reported by Z/EVES as a type error and corrected to be `MilitaryProcess`. The reason is that `ECA-P1-P4-P1` has `start` and `end` time points associated with it and the domains of these two functions are restricted to instances of `MilitaryProcess`.

Note that in ontological sense, the above errors are not treated as inconsistencies: in description logics, implicit information can be inferred and if there is no conflict, it is assumed true. Hence, if RACER is queried whether `ECA-P2-P9` is an instance of

`MilitaryProcess`, it will return true based on other facts present in the ontology. However, Z/EVES is more restrictive in treating types of Z language constructs and would not make such deductions, e.g., it will not assume `ECA-P2-P9` to be an instance of `MilitaryProcess` given the facts that it has a start and end time point.

Secondly, the ontology is opened in OilEd and RACER does not detect any ontological inconsistency.

Thirdly, since there is no ontological inconsistency, this step is skipped and we proceed to the final step of the combined approach.

Lastly, we apply the Z/EVES theorem prover to check for complex properties that cannot be expressed in OWL or Alloy.

Before applying Z/EVES, we study the plan ontology and gain some insights of military domain, based on which we formulate a number of theorems to test the correctness of instance ontologies.

Generally speaking, the formulation of the Z/EVES theorems is done through the interactions between domain experts, ontology developers and software engineering (and formal methods in particular) practitioners. The domain experts state desired properties, requirements while ontology developers and software engineering practitioners decide which of the above can be part of the ontology and which are too complex and can only be stated as Z/EVES theorems.

After a systematic checking of the ontology against this set of theorems, 14 *hidden errors* are discovered.

- 2 are caused by military tasks having start time greater than end time,
- 4 are caused by military tasks that do not have end time defined,

- 3 are caused by military units being assigned to different tasks simultaneously, and
- 5 are caused by military tasks having more than one `start` or `end` time points.

In the rest of this subsection, we demonstrate how various kinds of checking can be performed by Z/EVES.

In the first place, we test the *local* consistency of each military task. Two conditions are to be satisfied for each such task. Firstly, its start time must be less than or equal to its end time and secondly, it is not a sub task of itself.

In SWRL, these conditions can be expressed in the following two rules. In the first rule, we specify that the start time of any instance of `MilitaryTask` is less than or equal to its end time. Note that the less than or equals to operator $\leq$ is a SWRL built-in comparison operator. In the second rule, we specify that any such instance is not a `subTaskOf` itself. The second rule has an empty consequent, meaning that it is trivially false. In this way we express negation in SWRL.

$$MilitaryTask(?x) \wedge start(?x, ?s) \wedge end(?x, ?e) \rightarrow ?s \leq ?e$$

$$subTaskOf(?x, ?x) \rightarrow$$

The above two SWRL rules can be combined to a Z theorem, as shown below. The relational image $(\!|\ x\ |\!)$ returns the set of `Resources` mapped by a property, in this case $subTaskOf$, for x. By ensuring that x is not itself a member of this set, we ensure that no instance of `MilitaryTask` is a sub task of itself.

theorem MilitaryTaskTimeSubTaskTest1
$\forall x : instances(MilitaryTask) \bullet$
$start(x) < end(x) \wedge x \notin (sub\_val(subTaskOf))(\!|\ \{x\}\ |\!)$

We systematically test all instances of military tasks (including sub classes) for the above theorem. For example, one such instance, `ECA_P1_P2_P1_S1`, is tested as follows. It is an instance of class `HastyDefend_MilitaryTask` and it has two super tasks: `ECA_P1_P2` and `ECA_P1_P2_P1`.

Proof
*try lemma MilitaryTaskTimeSubTaskTest*1;
split $x = ECA\_P1\_P2\_P1\_S1$;
cases;
use cardCup [*Resource*] [$S := \{ECA\_P1\_P2\_P1\_S1\}$, $T := \{ECA\_P1\_P2\}$];
reduce;
use cardCup [*Resource*] [$S := \{ECA\_P1\_P2\_P1\_S1\}$, $T := \{ECA\_P1\_P2\_P1\}$];
reduce;
...

The proof process is intuitive: we consider the super tasks of x (`ECA_P1_P2_P1_S1` in this case) one at a time as sub goals. When all sub goals are completed, the current goal is proven. Defined in the built-in section `toolkit`, the rule *cardCup* is used here, with `Resource` as the actual parameter, to make the two military tasks distinct, as we stated in the end of Section 3.4. The last command `reduce` returns `true`, which means that the *current* sub goal is proven, not the whole theorem.

We show the proof process for another military task: `ECA_P3_P3_S1`. This time, after issuing similar commands, the remaining goal is of the form:

$$\neg\, x = ECA\_P3\_P3\_S1$$

This is an obvious contradiction to the 2^{nd} step of the proof: instantiation of x to `ECA_P3_P3_S1`. Hence we know for sure there is something wrong with this instance. Since it is very hard for theorem provers to prove falsity, we need to negate the theorem and show that the negated theorem can be proved to be `true`.

theorem negatedMilitaryTaskTimeSubTaskTest1
$\exists\, x : instances(MilitaryTask) \bullet$
$\neg\, (start(x) < end(x) \land x \notin (sub\_val\, (subTaskOf))(\!|\, \{x\} \,|\!))$

By negating the theorem and trying again, Z/EVES does return `true`. After checking the ontology, we found that start time is 7 but end time is 4, hence it is indeed an error, which was not discovered by RACER or Alloy Analyzer.

Two such instances failed this theorem. These errors may be caused by the inaccuracy of the IE engine; or they may be human error. After checking with the developers at DSO, it was found out that the errors were in the original textual document, which is the input of the IE engine. Hence in this case, it is human error.

After ensuring that all instances of `MilitaryTask` (and sub classes) are *locally* correct, we proceed to express and check the inter-task temporal relationship. It is required that for any instance $?x$ of `MilitaryTask`, any super tasks $?y$ of $?x$ must satisfy $start(?y) \leq start(?x) \land end(?y) \geq end(?x)$. That means, the start time of a super task must be less than or equal to that of its sub task, and the end time of a super task must be greater than or equal to that of its sub task. Since we have ensured that start time is before the end time for each military task, the above predicate suffices to prove the correctness. This can be expressed in the following SWRL rule.

$$MilitaryTask(?x) \land subTaskOf(?x, ?y) \rightarrow start(?y) \leq start(?x) \land end(?y) \geq end(?x)$$

As above, the following Z theorem basically states the above SWRL rule.

theorem subTaskOfTimingTest2
$\forall x : instances(MilitaryTask) \bullet$
$\forall y : \mathbb{P}(instances(MilitaryTask)) \mid y = (sub\_val(subTaskOf))(\!| \{x\} |\!) \bullet$
$\forall z : y \bullet start(z) \leq start(x) \land end(z) \geq end(x)$

A systematical application of this theorem against all appropriate military tasks show that there is no such kind of errors in this ontology.

The next SWRL rule tests the relationship between a military unit and the military tasks assigned to it. It states that for any given military unit and two military

tasks assigned to this unit, the durations of the two tasks do not overlap. As we have proved the *local* consistency of each military task, the predicate $end(?y) \leq start(?z) \vee end(?z) \leq start(?y)$ is sufficient.

$$ModernMilitaryUnit(?x) \wedge MilitaryTask(?y) \wedge MilitaryTask(?z) \wedge assignedTo(?y, ?x) \wedge assigned(?z, ?x)$$
$$\rightarrow$$
$$end(?y) \leq start(?z) \wedge end(?z) \leq start(y)$$

As above, this rule is also transformed into a Z theorem.

theorem MilitaryUnitTest
$$\forall x : instances(ModernMilitaryUnit) \bullet \forall y, z : instances(MilitaryTask) \bullet x \in (sub\_val(assignedTo))(\!| \{y\} |\!) \wedge x \in (sub\_val(assignedTo))(\!| \{z\} |\!) \wedge (end(y) \leq start(z) \vee end(z) \leq start(y))$$

We exhaustively and systematically apply this theorem to appropriate military units and tasks. During transformation process, we have collected information about what tasks each military unit executes; it is easy to proceed in this case. The proof process of one such combination is shown below.

Proof
try lemma MilitaryUnitTest;
split $x = CHF\_1$;
cases;
split $y = ECA\_P3\_P5\_S1$;
cases;
split $z = ECA\_P3\_P5\_S3$;
cases;
reduce;

After the last command `reduce` is entered, the following remaining goal is returned by Z/EVES:

$$z = ECA\_P3\_P5\_S1 \wedge y = ECA\_P3\_P5\_S3$$
$$\Rightarrow \neg\, x = CHF\_1$$

This is an obvious contradiction to the instantiation of quantified variables x, y and z. Hence we suspect that there is an error with this combination of instances. So we negate the theorem again and try to prove this negated theorem.

theorem negatedMilitaryUnitTest
$\exists\, x : instances(ModernMilitaryUnit) \bullet \exists\, y, z : instances(MilitaryTask) \bullet$
$\neg\, (x \in (sub\_val(assignedTo))(\!|\, \{y\} \,|\!) \land x \in (sub\_val(assignedTo))(\!|\, \{z\} \,|\!) \land$
$(end(y) \leq start(z) \lor end(z) \leq start(y)))$

After issuing similar commands, we proved the negated theorem. We found in the original ontology that the start and end time of these two military tasks are the same. Hence there is indeed an error that cannot be discovered by RACER or Alloy Analyzer.

4.5 Chapter Summary

The main contribution of this chapter is the combined approach of checking DAML+OIL and RDF ontologies using the complementary reasoning power of Semantic Web reasoning engines such as RACER and software engineering proof tools Z/EVES and Alloy Analyzer.

The combined approach was based on the Z and Alloy semantics for DAML+OIL and SWRL, which is the foundation of the respective transformation from DAML+OIL and RDF ontologies to Z and Alloy specifications.

In our approach, Z/EVES is firstly deployed to check for type errors in the (transformed) ontology. This step serves as a pre-processing so that unintended or unnecessary instantiation or subsumption can be removed, making the ontology easier to understand by human. The type-correct ontology is then checked by RACER fully automatically. If any inconsistency is detected, a fragment of the ontology relevant

to the inconsistency is then extracted and analyzed using Alloy Analyzer, which can give the exact location of the error in the transformed Alloy specification, helping debugging the original ontology. Finally, the theorem proving capability of Z/EVES is used to check for more complex properties inexpressible in DAML+OIL/OWL or Alloy.

Although expressible in SWRL, there has not been any tool support for SWRL. Moreover, since SWRL FOL expands the expressivity of ontology languages more into the first-order domain, Z and Z/EVES is a natural candidate for reasoning more complex ontology languages such as SWRL and SWRL FOL.

This approach has been applied to a military planning ontology case study, where one ontological inconsistency was detected and located and 14 errors inexpressible in DAML+OIL were found by Z/EVES.

This chapter focuses on the practical aspect of the combined approach. However, its validity relies on the correctness of the Z and Alloy semantics for DAML+OIL (hence OWL) since obviously if the semantic library is incorrect, wrong conclusion may be drawn from interacting with the various proof tools. In the next chapter, this issue will be addressed.

Chapter 5

Z Semantics for OWL: Soundness Proof Using Institution Morphisms

As mentioned in the previous chapter, the validity of the combined approach depends on the correctness of the Z/Alloy semantics of the ontology languages. Since the Z and Alloy semantics are very similar to each other, we will focus on one of these, i.e., Z.

As OWL has become the W3C recommendation as the ontology language for the Semantic Web, it is necessary to extend the Z/Alloy support from DAML+OIL to OWL. In the OWL species, OWL DL retains decidability and is more expressive than OWL Lite, we have constructed the Z semantics for OWL DL, which can be found in Appendix C.

Institutions and institution morphisms are a powerful tool to abstract and reason about software systems without any assumption about the underlying logical systems. They make a perfect candidate to reason about the relationship between OWL and Z as they are based on description logics and first-order predicate logic respectively.

In this chapter, we use institutions to investigate the Z semantics of OWL DL. It is proved at the end of the chapter, by making use of the Z semantics for OWL, that there exists a comorphism between OWL DL and Z, meaning that Z is more expressive than OWL DL.

This chapter is divided into four parts. In Sections 5.1 and 5.2, we construct the institutions for OWL and Z, respectively. Section 5.3 is devoted to relating the two institutions. Finally, Section 5.4 concludes the chapter.

5.1 The OWL Institution $\mathfrak{O}$

In this section we briefly introduce the definition of the logic underlying the Web Ontology Language OWL DL. We note that in OWL DL there is mutual disjointness between classes, properties, and individuals.

We suppose that all the OWL specifications share the same data types. Therefore we consider given a set $\mathbb{D}$ of *data type names*, a set $\mathcal{V}$ of *data values*, and a function $[\![\_]\!]$ which associates a subset $[\![D]\!] \subseteq \mathcal{V}$ with each data type name D. The set of *data expressions* is defined as follows:

$$\mathcal{D} ::= D \mid \{v_1, \ldots, v_n\}$$

where D ranges over data type names and v_i ranges over data values. We extend the definition of $[\![\_]\!]$ by setting $[\![\{v_1, \ldots, v_n\}]\!] = \{v_1, \ldots, v_n\}$. In OWL definition [80] a data type D is characterized by a lexical space, $L(D)$, a value space, $V(D)$, and a mapping $L2V(D) : L(D) \rightarrow V(D)$. We represent a data type in a more abstract way by forgetting the lexical space. $V(D)$ is denoted here by $[\![D]\!]$. For instance, $(\mathbb{D}, [\![\_]\!])$ might be the set of the XML data types and/or the set of the OWL built-in types. We separate the data world from the world over which we define ontologies.

A first reason for this separation is that the specification of the data types is quite different from that of ontologies. Another reason is that we get more flexibility in relating web ontologies with various formalisms. For instance, we may use directly the built-in implementations of the data types in these formalisms and focus only on the translation of the taxonomy and its sentences.

An *OWL signature* consists of a quadruple $\mathcal{O} = (\mathbb{C}, \mathbb{R}, \mathbb{U}, \mathbb{I})$, where $\mathbb{C}$ is the set of the *concept (class) names*, $\mathbb{R}$ is the set of the *individual-valued property names*, $\mathbb{U}$ is the set of the *data-valued property names*, and $\mathbb{I}$ is the set of *individual names*. We suppose that $\mathbb{D}$, $\mathbb{C}$, $\mathbb{R}$, $\mathbb{U}$, and $\mathbb{I}$ are pairwise disjoint. We denote by $\mathcal{N}(\mathcal{O})$ the set $\mathbb{C} \cup \mathbb{R} \cup \mathbb{U} \cup \mathbb{I}$. An *OWL signature morphism* $\phi : (\mathbb{C}, \mathbb{R}, \mathbb{U}, \mathbb{I}) \to (\mathbb{C}', \mathbb{R}', \mathbb{U}', \mathbb{I}')$ consists of a quadruple of functions $\phi = (\phi_{co}, \phi_{op}, \phi_{dp}, \phi_{in})$ where $\phi_{co} : \mathbb{C} \to \mathbb{C}'$, $\phi_{op} : \mathbb{R} \to \mathbb{R}'$, $\phi_{dp} : \mathbb{U} \to \mathbb{U}'$, and $\phi_{in} : \mathbb{I} \to \mathbb{I}'$. Sometimes we see ϕ as a function $\phi : \mathcal{N}(\mathcal{O}) \to \mathcal{N}(\mathcal{O}')$. We denote by $\mathsf{Sign}(\mathfrak{O})$ the category of the OWL signatures. Given an OWL signature $\mathcal{O} = (\mathbb{C}, \mathbb{R}, \mathbb{U}, \mathbb{I})$, an *$\mathcal{O}$-structure (model)* is a tuple $A = (\Delta_A, [\![\_]\!]_A, Res_A, res_A)$ consisting of a set of *resources* Res_A, a subset $\Delta_A \subseteq Res_A$ called *domain*, a function $res_A : \mathcal{N}(\mathcal{O}) \to Res_A$ associating a resource to each name in $\mathcal{O}$, and an interpretation function $[\![\_]\!]_A : \mathbb{C} \cup \mathbb{R} \cup \mathbb{U} \to \mathcal{P}(Res) \cup P(Res) \times P(Res)$ such that the following conditions hold:

$\mathcal{V} \subseteq Res_A$,
$\Delta_A \cap \mathcal{V} = \emptyset$,
$[\![C]\!]_A \subseteq \Delta_A$ for each $C \in \mathbb{C}$,
$[\![R]\!]_A \subseteq \Delta_A \times \Delta_A$ for each $R \in \mathbb{R}$,
$[\![U]\!]_A \subseteq \Delta_A \times \mathcal{V}$ for each $U \in \mathbb{U}$,
$res_A(o) \in \Delta_A$ for each $o \in \mathbb{I}$.

In order to have a uniform notation, we often write $[\![o]\!]_A$ for $res_A(o)$.

The definition above corresponds to that of abstract interpretation of an OWL vocabulary given by the direct model-theoretic semantics [80]. In particular we have $\Delta_A = O$, $[\![\_]\!]_A \mid_{\mathbb{C}} = EC$, $[\![\_]\!]_A \mid_{\mathbb{R}\cup\mathbb{U}} = ER$, and $res_A = S$. Here $[\![\_]\!]_A \mid_X$ denotes the restriction of the function $[\![\_]\!]_A$ to the subset X.

Given two $\mathcal{O}$-structures $A = (\Delta_A, [\![\_]\!]_A, Res_A, res_A)$ and $A' = (\Delta_{A'}, [\![\_]\!]_{A'}, Res_{A'}, res_{A'})$, an *$\mathcal{O}$-homomorphism* $h : A \to A'$ is a function $h : Res_A \to Res_{A'}$ such that:

1. $h(\Delta_A) = \Delta_{A'}$;
2. $res_{A'} = res_A;\ h$;
3. for each $C \in \mathbb{C}$ and $x \in \Delta_A$, $x \in [\![C]\!]_A$ iff $h(x) \in [\![C]\!]_{A'}$;
4. for each $R \in \mathbb{R}$ and $x, y \in \Delta_A$, $(x, y) \in [\![R]\!]_A$ iff $(h(x), h(y)) \in [\![R]\!]_{A'}$;
5. for each $U \in \mathbb{U}$, $x \in \Delta_A$, and $v \in \mathcal{V}$, $(x, v) \in [\![U]\!]_A$ iff $(h(x), v) \in [\![U]\!]_{A'}$.

Let $\mathsf{Mod}(\mathfrak{O})(\mathcal{O})$ denote the category of the $\mathcal{O}$-models. If $\phi : \mathcal{O} \to \mathcal{O}'$ is an OWL signature morphism and $A' = (\Delta_{A'}, [\![\_]\!]_{A'}, Res_{A'}, res_{A'})$ an $\mathcal{O}'$-structure, then the ϕ-reduct $A'{\restriction_\phi}$ is the $\mathcal{O}$-structure $A = (\Delta_A, [\![\_]\!]_A, Res_A, res_A)$, where $Res_{A\restriction\phi} = Res_{A'}$, $\Delta_{A\restriction\phi} = \Delta_{A'}$ and $res_A(N) = res_{A'}(\phi(N))$ for each name $N \in \mathcal{N}(\mathcal{O})$, and the interpretation function $[\![\_]\!]_A$ is defined as follows:

$[\![C]\!]_A = [\![\phi_{co}(C)]\!]_{A'}$ for each $C \in \mathbb{C}$;
$[\![R]\!]_A = [\![\phi_{op}(R)]\!]_{A'}$ for each $R \in \mathbb{R}$;
$[\![U]\!]_A = [\![\phi_{dp}(U)]\!]_{A'}$ for each $U \in \mathbb{U}$.

If $h' : A' \to A''$ is an $\mathcal{O}'$-homomorphism, then the reduct along ϕ of h' is the $\mathcal{O}$-homomorphism $h'{\restriction_\phi}$: $A'{\restriction_\phi} \to A''{\restriction_\phi}$ given by $h'{\restriction_\phi}(x') = h'(x')$. It is a matter of routine to check that $h'{\restriction_\phi}$ is indeed an $\mathcal{O}$-homomorphism. We may now consider the functor $\mathsf{Mod}(\mathfrak{O}) : \mathsf{Sign}(\mathfrak{O})^{\mathrm{op}} \to \mathsf{Cat}$ mapping each OWL signature $\mathcal{O}$ to the category of its models $\mathsf{Mod}(\mathfrak{O})(\mathcal{O})$ and each OWL signature morphism $h : \mathcal{O} \to \mathcal{O}'$ to the forgetful

functor $\mathsf{Mod}(\mathfrak{O})(\phi^{\mathrm{op}}) : \mathsf{Mod}(\mathfrak{O})(\mathcal{O}') \to \mathsf{Mod}(\mathfrak{O})(\mathcal{O})$ defined by $\mathsf{Mod}(\mathfrak{O})(\phi^{\mathrm{op}})(h') = h'{\restriction_\phi}$.

The set of the $\mathcal{O}$-*expressions* is defined by:

$$\begin{aligned}\mathcal{C} ::= & \bot \mid \top \mid C \mid \mathcal{C} \sqcap \mathcal{C} \mid \mathcal{C} \sqcup \mathcal{C} \mid \neg\, \mathcal{C} \\ & \mid \forall\, \mathcal{R}.\mathcal{C} \mid \exists\, \mathcal{R}.\mathcal{C} \mid \leqslant n\, \mathcal{R} \mid \geqslant n\, \mathcal{R} \mid R : o \\ & \mid \forall\, U.\mathcal{D} \mid \exists\, U.\mathcal{D} \mid \leqslant n\, U \mid \geqslant n\, U \mid U : v \\ & \mid \{o_1, \ldots, o_n\} \\ \mathcal{R} ::= & R \mid \mathsf{Inv}(R)\end{aligned}$$

where C ranges over concepts names, R ranges over individual-valued properties names, U over data-valued properties, v over $\mathcal{V}$, and o_i over individuals names.

The set of OWL $\mathcal{O}$-*sentences* is defined by:

$$\begin{aligned}\mathcal{F} ::= & \mathcal{C} \sqsubseteq \mathcal{C} \mid \mathcal{C} \equiv \mathcal{C} \mid \mathsf{Disjoint}(\mathcal{C}, \ldots, \mathcal{C}) \\ & \mid \mathsf{Tr}(R) \mid \mathcal{R} \sqsubseteq \mathcal{R} \mid \mathcal{R} \equiv \mathcal{R} \\ & \mid U \sqsubseteq U \mid U \equiv U \\ & \mid o : \mathcal{C} \mid (o, o') : \mathcal{R} \mid (o, v) : U \mid o \equiv o' \mid o \not\equiv o'\end{aligned}$$

where n ranges over natural numbers, o and o' over individuals names, and v over data values. We denote by $\mathsf{sen}(\mathfrak{O})(\mathcal{O})$ the set of the OWL $\mathcal{O}$-sentences. If $\phi : \mathcal{O} \to \mathcal{O}'$ is an OWL signature morphism, then $\mathsf{sen}(\mathfrak{O})(\phi) : \mathsf{sen}(\mathfrak{O})(\mathcal{O}) \to \mathsf{sen}(\mathfrak{O})(\mathcal{O}')$ is the function translating the OWL $\mathcal{O}$-sentences in OWL $\mathcal{O}'$-sentences in the standard way; for instance,

$$\mathsf{sen}(\mathfrak{O})(\phi)(\forall\, R.C \sqcap C') = \forall\, \phi_{op}(R).\phi_{co}(C) \sqcap \phi_{co}(C').$$

We have now defined the functor

$$\mathsf{sen}(\mathfrak{O}) : \mathsf{Sign}(\mathfrak{O}) \to \mathsf{Set}.$$

Example 1 *Here is a very simple example of OWL specification:*

$$
\begin{array}{rcl}
\mathbb{C} & = & \{Author,\ FamousAuthor,\ Paper\}, \\
\mathbb{R} & = & \{writtenBy,\ citedBy\}, \\
\mathbb{U} & = & \{noOfPages\}, \\
\mathbb{I} & = & \{Kleene, Mathematical\ Logic\}, \\
\\
\mathcal{F} & = & \{FamousAuthor \sqsubseteq Author, \\
 & & Paper \sqsubseteq \ \geqslant 1\ writtenBy, \\
 & & \top \sqsubseteq \forall\ writtenBy.Author, \\
 & & Paper \sqsubseteq \ \geqslant 1\ citedBy, \\
 & & \top \sqsubseteq \forall\ citedBy.Author, \\
 & & \geqslant 1\ noOfPages \sqsubseteq Paper, \\
 & & \top \sqsubseteq \forall\ noOfPages.integer, \\
 & & (Mathematical\ Logic, Kleene) : writtenBy, \\
 & & Kleene : FamousAuthor\}
\end{array}
$$

The first sentence asserts that any famous author is an author. The second one asserts that `Paper` *is included in the domain of the individual-valued property* `writtenBy`*. The third sentence asserts that the range (codomain) of* `writtenBy` *is included in* `Author`*. We show the validity of these two assertions later when we give the semantics for expressions and sentences. The next four sentences are similar to the second and the third, respectively. The last two sentences are self-explanatory.*

The semantics of the $\mathcal{O}$-expressions is given by:

$$
\begin{array}{l}
[\![\bot]\!]_A = \emptyset, \\
[\![\top]\!]_A = \Delta_A, \\
[\![\mathsf{Inv}(R)]\!]_A = \{(y, x) \mid (x, y) \in [\![R]\!]_A\}, \\
[\![\mathcal{C} \sqcap \mathcal{C}']\!]_A = [\![\mathcal{C}]\!]_A \cap [\![\mathcal{C}']\!]_A, \\
[\![\mathcal{C} \sqcup \mathcal{C}']\!]_A = [\![\mathcal{C}]\!]_A \cup [\![\mathcal{C}']\!]_A, \\
[\![\neg\ \mathcal{C}]\!]_A = \Delta_A \setminus [\![\mathcal{C}]\!]_A,
\end{array}
$$

$$[\![\forall \mathcal{R}.\mathcal{C}]\!]_A = \{x \mid (\forall\, y)(x,y) \in [\![\mathcal{R}]\!]_A \Rightarrow y \in [\![\mathcal{C}]\!]_A\},$$
$$[\![\exists \mathcal{R}.\mathcal{C}]\!]_A = \{x \mid (\exists\, y)(x,y) \in [\![\mathcal{R}]\!]_A \wedge y \in [\![\mathcal{C}]\!]_A\},$$
$$[\![\leqslant n\, \mathcal{R}]\!]_A = \{x \mid \#(\{y \mid (x,y) \in [\![\mathcal{R}]\!]_A\}) \leqslant n\},$$
$$[\![\geqslant n\, \mathcal{R}]\!]_A = \{x \mid \#(\{y \mid (x,y) \in [\![\mathcal{R}]\!]_A\}) \geqslant n\},$$
$$[\![R : o]\!]_A = \{x \mid (x, [\![o]\!]_A) \in [\![R]\!]_A\},$$
$$[\![\forall\, U.\mathcal{D}]\!]_A = \{x \mid (\forall\, v)(x,v) \in [\![U]\!]_A \Rightarrow v \in [\![\mathcal{D}]\!]\},$$
$$[\![\exists\, U.\mathcal{D}]\!]_A = \{x \mid (\exists\, v)(x,v) \in [\![U]\!]_A \wedge v \in [\![\mathcal{D}]\!]\},$$
$$[\![\leqslant n\ U]\!]_A = \{x \mid \#(\{v \mid (x,v) \in [\![U]\!]_A\}) \leqslant n\},$$
$$[\![\geqslant n\ U]\!]_A = \{x \mid \#(\{v \mid (x,v) \in [\![U]\!]_A\}) \geqslant n\},$$
$$[\![U : v]\!]_A = \{x \mid (x,v) \in [\![U]\!]_A\},$$
$$[\![\{o_1, \ldots, o_n\}]\!]_A = \{res_A(o_1), \ldots, res_A(o_n)\}.$$

The satisfaction relation between $\mathcal{O}$-structures and $\mathcal{O}$-sentences is defined as follows:

$A \models_{\mathcal{O}} \mathcal{C} \sqsubseteq \mathcal{C}'$ iff $[\![\mathcal{C}]\!]_A \subseteq [\![\mathcal{C}']\!]_A$,
$A \models_{\mathcal{O}} \mathcal{C} \equiv \mathcal{C}'$ iff $[\![\mathcal{C}]\!]_A = [\![\mathcal{C}']\!]_A$,
$A \models_{\mathcal{O}} \mathsf{Disjoint}(\mathcal{C}_1, \ldots, \mathcal{C}_n)$, iff $[\![\mathcal{C}_i]\!]_A \cap [\![\mathcal{C}_j]\!]_A = \emptyset$ for all $i \neq j$,
$A \models_{\mathcal{O}} \mathsf{Tr}(\mathcal{R})$ iff $[\![\mathcal{R}]\!]_A$ is transitive,
$A \models_{\mathcal{O}} \mathcal{R} \sqsubseteq \mathcal{R}'$ iff $[\![\mathcal{R}]\!]_A \subseteq [\![\mathcal{R}']\!]_A$,
$A \models_{\mathcal{O}} \mathcal{R} \equiv \mathcal{R}'$ iff $[\![\mathcal{R}]\!]_A = [\![\mathcal{R}']\!]_A$,
$A \models_{\mathcal{O}} U \sqsubseteq U'$ iff $[\![U]\!]_A \subseteq [\![U']\!]_A$,
$A \models_{\mathcal{O}} U \equiv U'$ iff $[\![U]\!]_A = [\![U']\!]_A$,
$A \models_{\mathcal{O}} o : \mathcal{C}$ iff $[\![o]\!]_A \in [\![\mathcal{C}]\!]_A$,
$A \models_{\mathcal{O}} (o, o') : \mathcal{R}$ iff $([\![o]\!]_A, [\![o']\!]_A) \in [\![\mathcal{R}]\!]_A$,
$A \models_{\mathcal{O}} (o, v) : U$ iff $([\![o]\!]_A, v) \in [\![U]\!]_A$,
$A \models_{\mathcal{O}} o \equiv o'$ iff $[\![o]\!]_A = [\![o']\!]_A$,
$A \models_{\mathcal{O}} o \not\equiv o'$ iff $[\![o]\!]_A \neq [\![o']\!]_A$.

Example 2 *We have:*

$$\begin{aligned}
A \models\ & \mathit{Paper} \sqsubseteq\ \geqslant 1\, \mathit{writtenBy} \quad \mathit{iff} \\
& [\![\mathit{Paper}]\!]_A \subseteq [\![\geqslant 1\, \mathit{writtenBy}]\!]_A \quad \mathit{iff} \\
& [\![\mathit{Paper}]\!]_A \subseteq \{x \mid \#(\{y \mid (x,y) \in [\![\mathit{writtenBy}]\!]_A\}) \geqslant 1\} \\
& \mathit{iff} \\
& [\![\mathit{Paper}]\!]_A \subseteq \{x \mid (\exists\, y)(x,y) \in [\![\mathit{writtenBy}]\!]_A\} \quad \mathit{iff} \\
& [\![\mathit{Paper}]\!]_A \subseteq \mathrm{dom}\, [\![\mathit{writtenBy}]\!]_A
\end{aligned}$$

and

$$
\begin{aligned}
A \models\ & \top \sqsubseteq \forall\, writtenBy.Author \quad \textit{iff} \\
& [\![\top]\!]_A \subseteq [\![\forall\, writtenBy.Author]\!]_A \quad \textit{iff} \\
& \Delta_A \subseteq \{x \mid (\forall\, y)(x,y) \in [\![writtenBy]\!]_A \Rightarrow y \in [\![Author]\!]_A\} \\
& \textit{iff} \\
& (\forall\, x, y \in \Delta_A)(x,y) \in [\![writtenBy]\!]_A \Rightarrow y \in [\![Author]\!]_A \ \textit{iff} \\
& \operatorname{ran} writtenBy \subseteq [\![Author]\!]_A
\end{aligned}
$$

Theorem 1 $\mathfrak{O} = (\mathsf{Sign}(\mathfrak{O}), \mathsf{sen}(\mathfrak{O}), \mathsf{Mod}(\mathfrak{O}), \models_{\mathfrak{O}})$, *where* $\models_{\mathfrak{O}}$ *associates with each OWL signature* $\mathcal{O}$ *the relation* $\models_{\mathcal{O}}$ *defined as above, is an institution.*

The next result proves the first main feature of the OWL institution.

Theorem 2 *The category of OWL signatures* $\mathsf{Sign}(\mathfrak{O})$ *is cocomplete.*

The proof of the next corollary follows from Theorem 27 in [31] and it supplies the mathematical support for putting together smaller ontologies to form larger ones.

Corollary 1 *The category* $\mathsf{Th}_{\mathfrak{O}}$ *is cocomplete.*

The second main feature of the OWL institution is given by the following result and it shows that there is a sound way to amalgamate consistent OWL models (worlds of resources).

Theorem 3 *The functor* $\mathsf{Mod}(\mathfrak{O}) : \mathsf{Sign}(\mathfrak{O})^{\mathrm{op}} \rightarrow \mathsf{Cat}$ *preserves pullbacks.*

5.1.1 The Grothendieck Institution of OWL

Since the institution we defined above is strongly dependent on the data type $(\mathbb{D}, [\![\_]\!])$, it follows that we should denote it by $\mathfrak{O}(\mathbb{D}, [\![\_]\!])$. The data type can be organized into a category DT as follows:

– the objects are pairs of the form $(\mathbb{D}, [\![\_]\!] : \mathbb{D} \to |\ \mathsf{Set}\ |)$

– the arrows $u : (\mathbb{D}, [\![\_]\!]) \to (\mathbb{D}', [\![\_]\!]')$ are functions $u : \mathbb{D} \to \mathbb{D}'$ such that $[\![D]\!] = [\![u(D)]\!]$ for all $D \in \mathbb{D}$.

We define the functor $\mathsf{owl} : \mathsf{DT}^{op} \to \mathsf{Ins}$ as follows:

– $\mathsf{owl}(\mathbb{D}, [\![\_]\!]) = \mathfrak{O}(\mathbb{D}, [\![\_]\!])$;

– if $u : (\mathbb{D}, [\![\_]\!]) \to (\mathbb{D}', [\![\_]\!]')$, then $\mathsf{owl}(u)$ is the institution morphism $(\phi^u, \alpha^u, \beta^u)$ where ϕ^u is the identity, $\alpha^u_{\mathcal{O}} : \mathsf{sen}(\mathfrak{O}(\mathbb{D}, [\![\_]\!]))(\mathcal{O}) \to \mathsf{sen}(\mathfrak{O}(\mathbb{D}', [\![\_]\!]'))(\mathcal{O})$ maps each $\mathcal{O}$-sentence F over $\mathbb{D}$ to an $\mathcal{O}$-sentence F' over $\mathbb{D}'$ obtained from F by replacing the occurrences of $D \in \mathbb{D}$ with $u(D)$, and $\beta^u_{\mathcal{O}}$ is identity.

The general institution of the web ontologies $\mathfrak{O}$ can now be defined as the Grothendieck institution $\mathsf{owl}^{\#}$.

The Grothendieck construction can be done in a more general framework. Let $\widehat{\underline{\mathrm{DT}}}$ be an institution of data types. The signature category of the predefined types is the Grothendieck category $\mathsf{Mod}(\widehat{\underline{\mathrm{DT}}})^{\#}$. The institution $\mathfrak{O}$ is now the Grothendieck institution of the indexed institution $\mathsf{owl} : (\mathsf{Mod}(\widehat{\underline{\mathrm{DT}}})^{\#})^{op} \to \mathsf{Ins}$. A main consequence of this fact is that we can change the syntactical notation for the data values or the implementation of the same abstract data type without changing the properties of the ontologies.

5.2 The Institution $\mathfrak{Z}$

Z [107, 89] is a formal specification language based on first-order predicate logic and ZF set theory. It is well suited for modeling system data and states. Z has a rich set of language constructs including given type, abbreviation type, axiomatic definition, state and operation schema definitions, etc.

We briefly recall from [3] the institution $\mathfrak{Z}$, denoted by $\boldsymbol{\mathcal{S}}$ in [3], formalizing the logic underlying the specification language Z.

A *Z signature* $\mathcal{Z}$ is a triple (G, Op, τ) where G is the set of the *given-sets names*, Op is a set of the *identifiers*, and τ is a function mapping the names in Op into types $\mathcal{T}(G)$, where $\mathcal{T}(G)$ is inductively defined by:

1. $G \subseteq \mathcal{T}(G)$,
2. $T_1 \times \cdots \times T_n \in \mathcal{T}(G)$ for $T_i \in \mathcal{T}(G)$, $i = 1, \ldots, n$,
3. $\mathcal{P}(T) \in \mathcal{T}(G)$ for $T \in \mathcal{T}(G)$,
4. $\langle x_1 : T_1, \ldots, x_n : T_n \rangle \in \mathcal{T}(G)$ for $T_i \in \mathcal{T}(G)$ and x_i is a variable name, $i = 1, \ldots, n$, such that $i \neq j \Rightarrow x_i \neq x_j$.

A *Z signature morphism* $\phi : (G, Op, \tau) \to (G', Op', \tau')$ is a pair of functions $\phi_{gs} : G \to G'$ and $\phi_{op} : Op \to Op'$ such that $\tau ; \mathcal{T}(\phi_{gs}) = \phi_{op} ; \tau'$. $\mathcal{T}(\phi_{gs})$ is the standard extension of ϕ_{gs} to $\mathcal{T}(\phi_{gs}) : \mathcal{T}(G) \to \mathcal{T}(G')$. We denote by $\mathsf{Sign}(\mathfrak{Z})$ the category of Z signatures. Given a Z signature $\mathcal{Z} = (G, Op, \tau)$, a *$\mathcal{Z}$-structure (model)* is a pair (A_G, A_{Op}) where A_G is a functor from G, viewed as a discrete category, to Set, and A_{Op} is a set $\{(o, v) \mid o \in Op\}$ where $v \in \overline{A}_G(\tau(o))$. The functor $\overline{A}_G$ is the standard extension of A_G to $\overline{A}_G : \mathcal{T}(G) \to \mathsf{Set}$. A *$\mathcal{Z}$-homomorphism* $h : (A_G, A_{Op}) \to (B_G, B_{Op})$ is a natural transformation $h : A_G \Rightarrow B_G$ given by $\overline{h}_{\tau(o)}(v) = v'$, where $(o, v) \in A_{Op}$ and $(o, v') \in B_{Op}$; again, $\overline{h}$ is the usual extension of h to $\overline{h} : \overline{A}_G \Rightarrow \overline{B}_G$. We denote by $\mathsf{Mod}(\mathfrak{Z})(\mathcal{Z})$ the category of $\mathcal{Z}$-structures. Given a Z signature morphism

$\phi : \mathcal{Z} \to \mathcal{Z}'$ and a $\mathcal{Z}'$-structure $A' = (A'_{G'}, A'_{O'})$, the ϕ-reduct $A'\restriction_\phi$ is the $\mathcal{Z}$-structure $A = (A_G, A_{Op})$ given by $A_G = \phi_{gs} ; A'_{G'}$ and $A_{Op} = \{(o, v) \mid (\phi_{op}(o), v) \in A'_{Op'}, o \in Op\}$.

Given a Z signature $\mathcal{Z}$, the sets of *$\mathcal{Z}$-expressions* E, *$\mathcal{Z}$-schema-expressions* S, and (part) of *$\mathcal{Z}$-formulas* P are defined by:

$$
\begin{aligned}
E ::= {} & id \mid x \mid (E, \ldots, E) \mid E.i \mid \langle x_1 \mapsto E, \ldots, x_n \mapsto E \rangle \\
& \mid E.x \mid E(E) \mid \{E, \ldots, E\} \mid \{S \bullet E\} \mid \mathcal{P}(E) \\
& \mid E \times \cdots \times E \mid S \\
S ::= {} & x_1 : E; \ \ldots; \ x_n : E \mid (S \mid P) \mid \neg\, S \mid S \vee S \mid S \wedge S \\
& \mid S \Rightarrow S \mid \forall\, S.S \mid \exists\, S.S \mid S \setminus [x_1, \ldots, x_n] \\
& \mid S[x_1/y_1, \ldots, x_n/y_n] \mid S \ \mathit{Decor} \mid E \\
P ::= {} & \mathtt{true} \mid \mathtt{false} \mid E \in E \mid E = E \mid \neg\, P \mid P \vee P \mid P \wedge P \\
& P \Rightarrow P \mid \forall\, S.P \mid \exists\, S.P
\end{aligned}
$$

Example 3 *The following simple Z specification:*

$[Class, Resource]$

ClassesAsResources

$$
\begin{array}{|l}
\hline
instances : Class \to \mathbb{P}\, Resource \\
res : Class \rightarrowtail Resource \\
\hline
\forall\, c, c' : Class;\ r : Resource;\ pr : \mathbb{P}\, Resource \bullet \\
\quad c \mapsto r \in res \Rightarrow \neg(r \in pr \wedge c' \mapsto pr \in instances) \\
\hline
\end{array}
$$

is described in the terms of the institution $\mathfrak{Z}$ *as* $\mathsf{CR} = ((G, Op, \tau), P)$ *where* $G = \{Class, Resource\}$, $Op = \{instances, res\}$, $\tau(instances) = \mathcal{P}(Class \times \mathcal{P}(Resource))$, $\tau(res) = \mathcal{P}(Class \times Resource)$, *and* P *includes the formulas expressing the functionality of the relation instances, the functionality and the injectivity of the relation res, together with the invariant of the state schema ClassesAsResources. It is easy to see, e.g., that* $c \mapsto r \in res$ *is a* CR*-expression and* $c, c' : Class;\ r : Resource;\ pr : \mathbb{P}\, Resource$ *is a* CR*-schema-expression.*

An *environment* $(\mathcal{Z}, (X, \tau_X))$ consists of a Z signature $\mathcal{Z}$, a set of variables $X = \{x_1, \ldots, x_n\}$, and a function $\tau_X : X \rightarrow \mathcal{T}(G)$ which associates a type with each variable. The sets of expressions and formulas are restricted to those well-formed w.r.t. an environment. Intuitively, an expression is *well-formed w.r.t. the environment* $(\mathcal{Z}, (X, \tau_X))$ iff we can uniquely associate to it a type which can be deduced from τ and τ_X. A $\mathcal{Z}$-formula P is *well defined w.r.t. the environment* $(\mathcal{Z}, (X, \tau_X))$ iff all its operators and quantifiers are given over expressions having the types compatible with their definition. For instance, if $X = \{c, r\}$, $\tau_X(c) = \mathit{Class}$, $\tau_X(r) = \mathit{Resource}$, then $c \mapsto r \in \mathit{res}$ is well defined w.r.t. the environment $(\mathsf{CR}, (X, \tau_X))$ whereas $c \mapsto r \in \mathit{instances}$ is not. Given a Z signature $\mathcal{Z}$ and an environment $(\mathcal{Z}, (X, \tau_X))$, a *variable binding* $\beta = (A, A_X)$ consists of a $\mathcal{Z}$-structure A and a set $A_X = \{(x_1, v_1), \ldots, (x_n, v_n)\}$ with $v_i \in \overline{A}_G(\tau_X(x_i))$ for $i = 1, \ldots, n$. The satisfaction relation between variable bindings and $\mathcal{Z}$-expressions and $\mathcal{Z}$-formulas is defined as expected (see [3] for details). For instance, if we consider the variable binding $\beta = (A, A_X)$, where $A_X = \{(c, v_c), (r, v_r)\}$, then $\beta \models c \mapsto r \in \mathit{res}$ iff $(v_c, v_r) \in w$ and $(\mathit{res}, w) \in A_{Op}$. The *$\mathcal{Z}$-sentences* are the $\mathcal{Z}$-formulas well defined with the environment $(\mathcal{Z}, (\{\,\}, \tau_{\{\,\}}))$. A $\mathcal{Z}$-structure A *satisfies* a $\mathcal{Z}$-sentence P, written $A \models_{\mathfrak{Z},\mathcal{Z}} P$, iff $(A, \{\,\}) \models P$.

The institution $\mathfrak{Z}$ is given by $\mathfrak{Z} = (\mathsf{Sign}(\mathfrak{Z}), \mathsf{sen}(\mathfrak{Z}), \mathsf{Mod}(\mathfrak{Z}), \models_{\mathfrak{Z}})$, where $\mathsf{Sign}(\mathfrak{Z})$ is the category of Z signatures, the functor $\mathsf{sen}(\mathfrak{Z})$ maps each Z signature $\mathcal{Z}$ to its set of $\mathcal{Z}$-sentences, the functor $\mathsf{Mod}(\mathfrak{Z})$ maps each Z signature $\mathcal{Z}$ to the category of $\mathcal{Z}$-structures, and $\models_{\mathfrak{Z},\mathcal{Z}}$ is defined as above.

5.2.1 The Use of the Mathematical Tool-kit

Many Z specifications use mathematical definitions included in so-called the Mathematical Tool-kit or standard library [89]. The use of these definitions can be formally

described in terms of the structured specifications. We show that by means of an example. Let Z be the following Z specification:

$[Resource]$

$$\begin{array}{|l} Class : \mathbb{P}\, Resource \\ Property : \mathbb{P}\, Resource \\ \hline Class \cap Property = \emptyset \end{array}$$

In terms of the institution theory, the above specification is represented by $(\mathcal{Z}_0, P_0)$, where $G_0 = \{\text{Resource}\}$, $Op_0 = \{\text{Class}, \text{Property}\}$, $\tau_0(\text{Class}) = \tau_0(\text{Property}) = \mathcal{P}(\text{Resource})$, and $P_0 = \{\text{Class} \cap \text{Property} = \emptyset\}$. The definitions for $\emptyset$, meaning $\emptyset[\text{Resource}]$, and $\_ \cap \_$ are included in the standard library:

$\emptyset[X] ::= \{x : X \mid false\}$

$$\begin{array}{|l} [X] \\ \hline \_ \cap \_ : \mathbb{P}\, X \times \mathbb{P}\, X \to \mathbb{P}\, X \\ \hline \forall\, S, T : \mathbb{P}\, X \bullet S \cap T = \{x : X \mid x \in S \wedge x \in T\} \end{array}$$

The full description $(\mathcal{Z}, P)$ of the initial Z specification is obtained as the vertex of the following pushout:

$$\begin{array}{ccc} \emptyset[\text{Resource}] & \longrightarrow & \_ \cap \_\,[\text{Resource}] \\ \downarrow & & \downarrow \\ (\mathcal{Z}_0, P_0) & \longrightarrow & (\mathcal{Z}, P) \end{array}$$

5.3 Encoding $\mathfrak{o}$ in $\mathfrak{z}$

In previous two chapters, we developed the semantics for DAML+OIL language in formal language Z as a extension of the standard library. This semantic library was later on revised for the new ontology language OWL, incorporating changes incurred

in OWL from DAML+OIL. In this Section, we will demonstrate, through institutions comorphisms, that the Z encoding of OWL is indeed sound.

The main idea is to associate a Z specification $\Phi(\mathcal{O}, F)$ with each OWL specification $(\mathcal{O}, F)$ such that an $(\mathcal{O}, F)$-model can be extracted from each $\Phi(\mathcal{O}, F)$-model. The construction of $\Phi(\mathcal{O}, F)$ is given in two steps: we first associate a Z specification $\Phi(\mathcal{O})$ with each OWL signature $\mathcal{O}$ and then we add to it the sentences F translated via a natural transformation.

Since $\Phi(\mathcal{O}, F)$ can be seen as a Z semantics of $(\mathcal{O}, F)$, it includes a distinct subspecification $(\mathcal{Z}^{\emptyset}, P^{\emptyset})$ defining the main OWL concepts and the operations over sets. More precisely, we consider $(\mathcal{Z}^{\emptyset}, P^{\emptyset})$ as being the vertex of the colimit having as base the standard library, the specification of the data types, together with the following Z specification:

given sets:
`Resource`;

identifiers:

- $\surd$ corresponding to OWL signatures:
  `Class`, `Property`, `ObjectProperty`, `DatatypeProperty`, `Individual`, `Thing`, `Nothing`
- $\surd$ giving Z semantics to OWL signatures:
  `instances`, `subVal`
- $\surd$ corresponding to OWL class axioms:
  `disjointClasses`, `equivalentClasses`, `subClassOf`
- $\surd$ corresponding to OWL descriptions and restrictions:
  `unionOf`, `intersectionOf`, `complementOf`, `oneOf`, `allValuesFrom`, `someValuesFrom`, `minCardinality`, `maxCardinality`, `cardinality`
- $\surd$ corresponding to OWL property axioms:
  `domain`, `range`, `functional`, `inverseOf`, `symmetric`, `transitive`, `inverseFunctional`, `equivalentProperties`, `subPropertyOf`

$\tau^{\emptyset}$ for the new identifiers:

$\surd$ corresponding to OWL signatures:
$\tau^{\emptyset}(\texttt{Class}) = \tau^{\emptyset}(\texttt{Property}) = \tau^{\emptyset}(\texttt{ObjectProperty}) =$
$\tau^{\emptyset}(\texttt{DatatypeProperty}) = \mathcal{P}(\texttt{Resource})$
$\tau^{\emptyset}(\texttt{Thing}) = \tau^{\emptyset}(\texttt{Nothing}) = \texttt{Resource}$
$\surd$ giving Z semantics to OWL signatures:
$\tau^{\emptyset}(\texttt{instances}) = \mathcal{P}(\texttt{Resource} \times \mathcal{P}(\texttt{Resource}))$
$\tau^{\emptyset}(\texttt{subVal}) = \mathcal{P}(\texttt{Resource} \times \mathcal{P}(\texttt{Resource} \times \texttt{Resource}))$
$\surd$ corresponding to OWL class axioms:
$\tau^{\emptyset}(\texttt{disjointClasses}) = \tau^{\emptyset}(\texttt{Class} \times \texttt{Class})$
$= \mathcal{P}(\texttt{Resource} \times \texttt{Resource})$
$\tau^{\emptyset}(\texttt{equivalentClasses}) = \mathcal{P}(\texttt{Resource} \times \texttt{Resource})$
$\tau^{\emptyset}(\texttt{subClassOf}) = \mathcal{P}(\texttt{Resource} \times \texttt{Resource})$
$\surd$ corresponding to OWL descriptions, restrictions
$\tau^{\emptyset}(\texttt{unionOf}) = \tau^{\emptyset}((\texttt{Class} \times \texttt{Class}) \times \texttt{Class})$
$= \mathcal{P}((\texttt{Resource} \times \texttt{Resource}) \times \texttt{Resource})$
$\tau^{\emptyset}(\texttt{intersectionOf}) = \mathcal{P}((\texttt{Resource} \times \texttt{Resource}) \times \texttt{Resource})$
$\tau^{\emptyset}(\texttt{complementOf}) = \mathcal{P}(\texttt{Resource} \times \texttt{Resource})$
$\tau^{\emptyset}(\texttt{oneOf}) = \mathcal{P}(\mathcal{P}(\texttt{Resource}) \times \texttt{Resource})$
$\tau^{\emptyset}(\texttt{allValuesFrom}) = \tau^{\emptyset}((\texttt{Resource} \times \texttt{Property}) \times \texttt{Class})$
$= \mathcal{P}((\texttt{Resource} \times \texttt{Resource}) \times \texttt{Resource})$
...
$\surd$ corresponding to OWL property axioms:
$\tau^{\emptyset}(\texttt{domain}) = \tau^{\emptyset}(\texttt{Property} \times \texttt{Resource})$
$= \mathcal{P}(\texttt{Resource} \times \texttt{Resource})$
$\tau^{\emptyset}(\texttt{range}) = \mathcal{P}(\texttt{Resource} \times \texttt{Resource})$
$\tau^{\emptyset}(\texttt{inverseOf}) = \tau^{\emptyset}(\texttt{ObjectProperty} \times \texttt{ObjectProperty})$
$= \mathcal{P}(\texttt{Resource} \times \texttt{Resource})$
$\tau^{\emptyset}(\texttt{functional}) = \tau^{\emptyset}(\texttt{Property}) = \mathcal{P}(\texttt{Resource})$
...

sentences :

$\surd$ corresponding to OWL signatures:
$\texttt{Class} \cap \texttt{Property} = \emptyset$
$\texttt{Class} \cap \texttt{Individual} = \emptyset$
$\texttt{Property} \cap \texttt{Individual} = \emptyset$
$\texttt{ObjectProperty} \cap \texttt{DatatypeProperty} = \emptyset$
$\texttt{Property} = \texttt{ObjectProperty} \cup \texttt{DatatypeProperty}$
$\surd$ giving Z semantics to OWL signatures:
$\texttt{instances}(\texttt{Thing}) = \texttt{Individual}$
$\texttt{instances}(\texttt{Nothing}) = \emptyset$
$\forall\, c : \texttt{Class} \bullet \texttt{instances}(c) \subseteq \texttt{Individual}$

$\forall\, p : \texttt{Property} \bullet \texttt{subVal}(p) \subseteq \mathcal{P}(\texttt{Individual} \times \texttt{Resource})$

...

$\surd$ corresponding to OWL class axioms:

$\forall\, c_1, c_2 : \texttt{Class} \bullet c_1 \mapsto c_2 \in \texttt{disjointClasses} \Leftrightarrow$
$\texttt{instances}(c_1) \cap \texttt{instances}(c_2) = \emptyset$

$\forall\, c_1, c_2 : \texttt{Class} \bullet c_1 \mapsto c_2 \in \texttt{subClassOf} \Leftrightarrow$
$\texttt{instances}(c_1) \subseteq \texttt{instances}(c_2)$

$\forall\, c_1, c_2 : \texttt{Class} \bullet c_1 \mapsto c_2 \in \texttt{equivalentClasses} \Leftrightarrow$
$\texttt{instances}(c_1) = \texttt{instances}(c_2)$

$\surd$ corresponding to OWL descriptions, restrictions:

$\forall\, c, c_1, c_2 : \texttt{Class} \bullet (c_1, c_2) \mapsto c \in \texttt{unionOf} \Leftrightarrow$
$\texttt{instances}(c) = \texttt{instances}(c_1) \cup \texttt{instances}(c_2)$

$\forall\, p : \texttt{Property};\ c_1, c : \texttt{Class} \bullet$
$(p, c_1) \mapsto c \in \texttt{allValuesFrom} \Leftrightarrow$
$\texttt{instances}(c) = \{x : \texttt{Individual} \mid \forall\, y : \texttt{Individual} \bullet$
$(x, y) \in \texttt{subVal}(p) \Rightarrow y \in \texttt{instances}(c_1)\}$

$\forall\, p : \texttt{Property};\ n : \mathbb{N};\ c : \texttt{Class} \bullet$
$(p, n) \mapsto c \in \texttt{minCardinality} \Leftrightarrow$
$\texttt{instances}(c) =$
$\{x : \texttt{Individual} \mid \#(\texttt{subVal}(p)(\!|\ \{x\}\ |\!)) \leq n\}$

...

$\surd$ corresponding to OWL property axioms:

$\forall\, p_1, p_2 : \texttt{Property} \bullet p_1 \mapsto p_2 \in \texttt{subPropertyOf} \Leftrightarrow$
$\texttt{subVal}(p_1) \subseteq \texttt{subVal}(p_2)$

$\forall\, p : \texttt{Property};\ c : \texttt{Class} \bullet p \mapsto c \in \texttt{domain} \Leftrightarrow$
$\operatorname{dom} \texttt{subVal}(p) \subseteq \texttt{instances}(c)$

$\forall\, p : \texttt{Property} \bullet p \in \texttt{functional} \Leftrightarrow$
$\forall\, x, y, z : \texttt{Resource}(x, y) \in \texttt{subVal}(p) \wedge$
$(x, z) \in \texttt{subVal}(p) \Rightarrow y = z$

...

We define $\Phi^{\diamond} : \mathsf{Sign}(\mathfrak{OD}) \rightarrow \mathsf{Sign}(\mathfrak{Z})$ as follows. Let $\mathcal{O} = (\mathbb{C}, \mathbb{R}, \mathbb{U}, \mathbb{I})$ be an OWL signature. Then $\Phi^{\diamond}(\mathcal{O}) = (G, Op, \tau)$ is defined as follows:

$G = G^{\emptyset}$;

$Op = Op^{\emptyset} \cup \mathbb{C} \cup \mathbb{R} \cup \mathbb{U} \cup \mathbb{I}$;

$\tau(C) = \texttt{Resource}$ for each $C \in \mathbb{C}$,

$\tau(R) = \texttt{Resource}$ for each $R \in \mathbb{R}$,

$\tau(U) = \texttt{Resource}$ for each $U \in \mathbb{U}$,

$\tau(o) = \texttt{Resource}$ for each $o \in \mathbb{I}$.

If $\varphi : \mathcal{O} \to \mathcal{O}'$ is an OWL signature morphism and $\Phi^\diamond(\mathcal{O}) = (G^\emptyset, Op, \tau)$ and $\Phi^\diamond(\mathcal{O}') = (G^\emptyset, Op', \tau')$, then $\Phi^\diamond(\varphi) : \Phi(\mathcal{O}) \to \Phi(\mathcal{O}')$ is the Z signature morphism $(\mathrm{id} : G^\emptyset \to G^\emptyset, \Phi^\diamond(\varphi)_{op} : Op \to Op')$ such that $\Phi^\diamond(\varphi)_{Op}$ is the identity over the subset $Op^\emptyset$ and $\Phi^\diamond(\varphi)_{op}(N) = \varphi(N)$ for each name N in $\mathcal{O}$. It is easy to check that $\tau\,;\mathcal{T}(\mathrm{id}) = \Phi^\diamond(\varphi)_{op}\,;\tau'$.

We extend $\Phi^\diamond$ to $\Phi : \mathsf{Sign}(\mathfrak{O}) \to \mathsf{Th}(\mathfrak{Z})$ by defining $\Phi(\mathcal{O}) = (\Phi^\diamond(\mathcal{O}), P)$, where P is $P^\emptyset$ together with the following sentences:

$$\{C \in \texttt{Class}) \mid C \in \mathbb{C}\} \cup$$
$$\{R \in \texttt{ObjectProperty} \mid R \in \mathbb{R}\} \cup$$
$$\{U \in \texttt{DatatypeProperty} \mid U \in \mathbb{U}\} \cup$$
$$\{o \in \texttt{Individual} \mid o \in \mathbb{I}\}.$$

If $\mathcal{O}$ is an OWL signature, then

$$\alpha_\mathcal{O} : \mathsf{sen}(\mathfrak{O})(\mathcal{O}) \to \mathsf{sen}(\mathfrak{Z})(\Phi(\mathcal{O}))$$

is defined by:

$$\alpha_\mathcal{O}(\bot) = \texttt{Nothing},\ \alpha_\mathcal{O}(\top) = \texttt{Thing},$$
$$\alpha_\mathcal{O}(N) = N \text{ for each name } N \text{ in } \mathcal{O}$$
$$\alpha_\mathcal{O}(C_1 \sqcap C_2) = \texttt{intersectionOf}(\alpha_\mathcal{O}(C_1), \alpha_\mathcal{O}(C_2)),$$
$$\ldots$$
$$\alpha_\mathcal{O}(\forall\, R.C) = \texttt{allValuesFrom}(\alpha_\mathcal{O}(R), \alpha_\mathcal{O}(C)),$$
$$\ldots$$
$$\alpha_\mathcal{O}(\leqslant n\, R) = \texttt{maxCardinality}(\alpha_\mathcal{O}(R), n), \ldots$$
$$\alpha_\mathcal{O}(C_1 \sqsubseteq C_2) = \alpha_\mathcal{O}(C_1) \mapsto \alpha_\mathcal{O}(C_2) \in \texttt{subClassOf},$$
$$\ldots$$
$$\alpha_\mathcal{O}(E) = \{\alpha_\mathcal{O}(e) \mid e \in E\}.$$

Example 4 *Let $\mathcal{O}$ be that defined in Example 1.*

$\alpha_{\mathcal{O}}(Paper \sqsubseteq \geqslant 1\, writtenBy) =$
$Paper \mapsto \mathit{minCardinality}(writtenBy, 1) \in \mathit{subclassOf}$
which is equivalent to
$\mathit{instances}(Paper) \subseteq \mathrm{dom}\ \mathit{subVal}(writtenBy)$

$\alpha_{\mathcal{O}}(\top \sqsubseteq \forall\, writtenBy.Author) =$
$\mathit{Resource} \mapsto \mathit{allValuesFrom}(\mathit{Author}, \mathit{writtenBy})$
$\in \mathit{subclassOf}$
which is equivalent to
$\mathrm{ran}\ \mathit{subVal}(writtenBy) \subseteq \mathit{instances}(Author)$

Lemma 1 *$\alpha = \{\alpha_{\mathcal{O}} \mid \mathcal{O} \in \mathsf{Sign}(\mathfrak{O})\}$ is a natural transformation $\alpha : \mathsf{sen}(\mathfrak{O}) \Rightarrow \Phi^{\diamond}; \mathsf{sen}(\mathfrak{Z})$.*

Proof: Let $\varphi : \mathcal{O} \to \mathcal{O}'$ be an OWL signature morphism. Then it is a matter of routine to check that the following diagram commutes:

$$\begin{array}{ccc} \mathsf{sen}(\mathfrak{O})(\mathcal{O}) & \xrightarrow{\alpha_{\mathcal{O}}} & \mathsf{sen}(\mathfrak{Z})(\Phi^{\diamond}(\mathcal{O})) \\ {\scriptstyle \mathsf{sen}(\mathfrak{O})(\varphi)}\downarrow & & \downarrow{\scriptstyle \mathsf{sen}(\mathfrak{Z})(\Phi(\varphi))} \\ \mathsf{sen}(\mathfrak{O})(\mathcal{O}') & \xrightarrow[\alpha_{\mathcal{O}'}]{} & \mathsf{sen}(\mathfrak{Z})(\Phi^{\diamond}(\mathcal{O}')) \end{array}$$

For instance, if $C_1, C_2 \in \mathbb{C}$, then $\alpha_{\mathcal{O}}(C_1 \sqsubseteq C_2) = (C_1 \mapsto C_2 \in \texttt{subClsassOf})$ and $\mathsf{sen}(\mathfrak{Z})(\Phi^{\diamond}(\phi))(\alpha_{\mathcal{O}}(C_1 \sqsubseteq C_2)) = (\phi(C_1) \mapsto \phi(C_2) \in \texttt{subClsassOf})$. On the other hand, $\mathsf{sen}(\mathfrak{O})(\phi)(C_1 \sqsubseteq C_2) = (\phi(C_1) \sqsubseteq \phi(C_2))$ and $\alpha_{\mathcal{O}}(\phi(C_1) \sqsubseteq \phi(C_2)) = (\phi(C_1) \mapsto \phi(C_2) \in \texttt{subClsassOf})$. □

If $\mathcal{O} = (\mathbb{C}, \mathbb{R}, \mathbb{U}, \mathbb{I})$ is an OWL signature and $A' = (A'_G, A'_{Op})$ a $\Phi^{\diamond}(\mathcal{O})$-model, then $\beta_{\mathcal{O}}(A')$ is the $\mathcal{O}$-model $A = (\Delta_A, [\![\_]\!]_A, Res_A, res_A)$ defined as follows:

$Res_A = A'_G(\texttt{Resource})$,
$res_A(N) = v$ where $(N, v) \in A'_{Op}$ for each name $N \in \mathcal{O}$,
$\Delta_A = v$ where $(\texttt{Thing}, v) \in A'_{Op}$,
if $C \in \mathbb{C}$, then $[\![C]\!]_A = v_C$ where $(\texttt{instances}, v) \in A'_{Op}$ and $(C, v_C) \in v$,

if $R \in \mathbb{R}$, then $[\![R]\!]_A = v_R$ where $(\texttt{subVal}, v) \in A'_{Op}$ and $(R, v_R) \in v$,
if $U \in \mathbb{U}$, then $[\![U]\!]_A = v_U$ where $(\texttt{subDVal}, v) \in A'_{Op}$ and $(U, v_U) \in v$.

A is indeed an $\mathcal{O}$-model. For instance, if $(\texttt{instances}, v) \in A'_{Op}$, then v is the graph of the function defined in A' by `instances` and v_C is just the value of this function for the argument C. Since $\tau^{\emptyset}(\texttt{instances}) = \mathcal{P}(\texttt{Resource} \times \mathcal{P}(\texttt{Resource}))$, it follows that $v_C \subseteq A'_G(\texttt{Resource})$. We obtain $[\![C]\!]_A \subseteq \Delta_A$ applying the sentences in $P^{\emptyset}$. We extend $\beta_{\mathcal{O}}$ to a functor $\beta_{\mathcal{O}} : \mathsf{Mod}'(\Phi^{\diamond}(\mathcal{O})) \to \mathsf{Mod}(\mathcal{O})$ as follows: if $h : A' \to B'$ is a $\Phi^{\diamond}(\mathcal{O})$-homomorphism, then $\beta_{\mathcal{O}}(h)$ is the $\mathcal{O}$-homomorphism $\beta_{\mathcal{O}}(h) : \beta_{\mathcal{O}}(A') \to \beta_{\mathcal{O}}(B')$ given by $\beta_{\mathcal{O}}(h) = h_{\texttt{Resource}}$.

Lemma 2 *$\beta = \{\beta_{\mathcal{O}} \mid \mathcal{O} \in \mathsf{Sign}(\mathfrak{O})\}$ is a natural transformation $\beta : \Phi^{\diamond\, op};\ \mathsf{Mod}(\mathfrak{Z}) \Rightarrow \mathsf{Mod}(\mathfrak{O})$.*

Proof: Let $\varphi : \mathcal{O} \to \mathcal{O}'$ be an OWL signature morphism. The commutativity of the diagram:

$$\begin{array}{ccc} \mathsf{Mod}(\mathfrak{Z})(\Phi^{\diamond\, op}(\mathcal{O}')) & \xrightarrow{\beta_{\mathcal{O}'}} & \mathsf{Mod}(\mathfrak{O})(\mathcal{O}') \\ {\scriptstyle \mathsf{Mod}_{(\mathfrak{Z})(\Phi^{\diamond\, op}(\varphi))}}\downarrow & & \downarrow{\scriptstyle \mathsf{Mod}_{(\mathfrak{O})(\varphi^{op})}} \\ \mathsf{Mod}(\mathfrak{Z})(\Phi^{\diamond\, op}(\mathcal{O})) & \xrightarrow{\beta_{\mathcal{O}}} & \mathsf{Mod}(\mathfrak{O})(\mathcal{O}) \end{array}$$

follows by checking that $\beta_{\mathcal{O}}(A'\restriction_{\Phi^{\diamond\, op}(\varphi)}) = \beta_{\mathcal{O}'}(A')\restriction_{\varphi}$ for each $\Phi^{\diamond}(\mathcal{O})$-model A'. □

Theorem 4 *$(\Phi, \alpha, \beta) : \mathfrak{O} \to \mathfrak{Z}$ is a simple theoroidal comorphism.*

Proof: We already proved that α and β are natural transformations. We have to prove the satisfaction condition. Let $\mathcal{O}$ be an OWL signature, e an $\mathcal{O}$-sentence, and A' a $\mathsf{Mod}(\mathfrak{Z})(\Phi(\mathcal{O}))$-model. We suppose first that $A' \models_{\Phi(\Sigma)} \alpha_{\mathcal{O}}(e)$. We prove

that $\beta_{\mathcal{O}}(A') \models_{\mathcal{O}} e$ by structural induction on e. For instance, we suppose that e is $C_1 \sqsubseteq C_2$. We have:
$A' \models_{\Phi(\mathcal{O})} \alpha_{\mathcal{O}}(C_1 \sqsubseteq C_2)$ iff $A' \models_{\Phi(\mathcal{O})} C_1 \mapsto C_2 \in \texttt{subClassOf}$
Since $A' \models P^{\emptyset}$ (we recall that $\Phi(\mathcal{O}) = (\Phi^{\diamond}(\mathcal{O}), P^{\emptyset})$), it follows that $A' \models \forall\, c_1, c_2 : \texttt{Class}\bullet c_1 \mapsto c_2 \in \texttt{subClassOf} \Rightarrow \texttt{instances}(c_1) \subseteq \texttt{instances}(c_2))$ which implies $[\![C_1]\!]_{\beta_{\mathcal{O}}(A')} \subseteq [\![C_2]\!]_{\beta_{\mathcal{O}}(A')}$, i.e., $\beta_{\mathcal{O}}(A') \models C_1 \sqsubseteq C_2$. The inverse implication is proved in a similar way. □

5.4 Chapter Summary

The main contribution of this chapter is the formal proof of the soundness of the Z semantics of ontology language OWL DL, which is the semantical foundation of the combined approach presented in the previous chapter.

As ontology languages and Z (and Alloy) are based on different logical systems (description logics vs first-order predicate logic), the proof of semantical equivalence between the OWL language constructs and Z semantics has to resort to a higher-level device that is able to reason with different logical systems.

In this chapter, we used the notion of institutions and institution comorphisms to represent the two logical systems underlying OWL DL and Z. Two institutions, $\mathfrak{O}$ (for OWL DL) and $\mathfrak{Z}$ (for Z) were defined and we proved that there is a simple theoroidal comorphism $(\Phi, \alpha, \beta) : \mathfrak{O} \rightarrow \mathfrak{Z}$ between $\mathfrak{O}$ and $\mathfrak{Z}$. Hence, we proved the soundness of the Z semantics for OWL DL.

Chapter 6

The Tools Environment: SESeW

The combined approach presented in Chapter 4 is an effective way of verifying correctness of the Semantic Web ontologies. However, it was also pointed out in Chapter 4 that there are a number rather involved steps in this approach. Moreover, there are some other functionalities, such as ontology querying, that the users may desire but not covered in the combined approach. An implementation of the ontology development methodology, the Methontology [29], is incorporated to facilitate systematic ontology creation.

For these reasons, we have developed a prototype of an integrated tools environment, the Software Engineering for the Semantic Web (SESeW), that facilitates the application of the combined approach and supports a number of other functionalities.

This chapter is devoted to an introduction of our integrated tools environment for developing and reasoning DAML+OIL and OWL ontologies. It is divided into the following parts. In Section 6.1, we present SESeW in brief. In subsequent Sections, we present the ontology creation process, querying, transformation and connection with external tools. Finally, Section 6.6 summarizes the chapter.

6.1 Overview of SESeW

Figure 6.1 shows the main window of SESeW, with a military-domain OWL ontology opened. It has four tabbed text areas for ontologies, Z, Alloy and PVS [78] [1] specifications respectively. Transformed Z, Alloy and PVS specifications are displayed in the respective text areas.

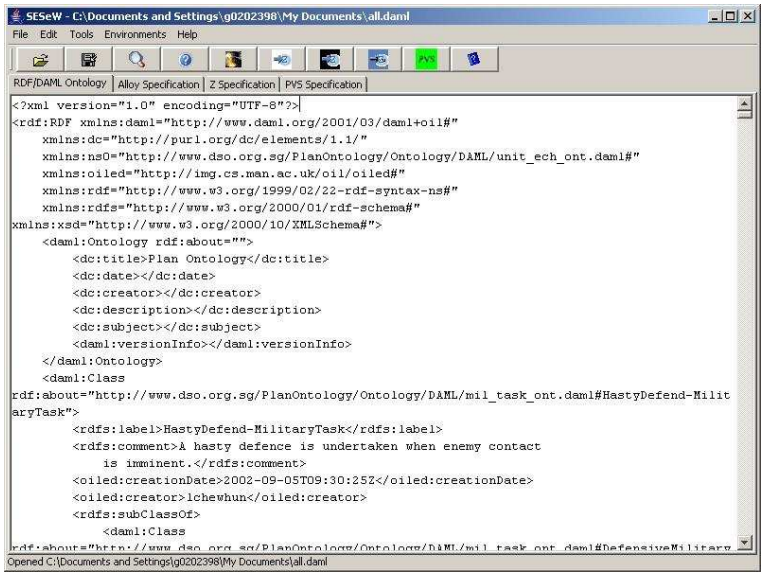

Figure 6.1: Main Window of SESeW

A user may load an existing ontology created using other editors like Protégé [30], in which case SESeW provides a standard text editing environment and functionality for editing the ontologies. Simple validation functions like well-formedness checking are offered to make sure the syntactical correctness of the ontologies.

[1] The PVS text area is for research work [21].

6.2 Ontology Creation

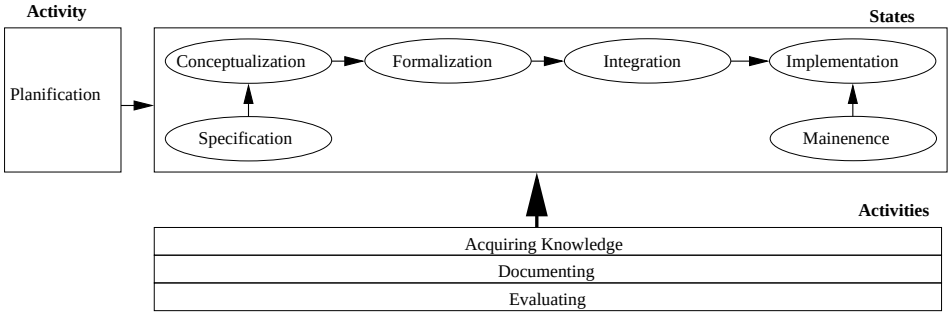

Figure 6.2: Flow of Ontology Creation

We implemented a systematic methodology for creating ontologies, namely Methontology [29]. Basically, Methontology is a set of activities in ontology development process, a life cycle to build ontologies based on evolving prototypes, and a well-structured methodology used to build ontologies from scratch. The ontology life cycle contains the following states: specification, conceptualization, integration, implementation, and maintenance (Figure 6.2, borrowed from [29]). The specification phase is to produce either an informal, semi-formal or formal ontology specification document in natural language, as a set of intermediate representations or using competency questions. In the conceptualization phase, the domain knowledge is structured in a conceptual model. A complete glossary of terms, i.e. concepts, instances, verbs, and properties, is built. The integration phase speeds up the construction of the ontology by considering reuse of definitions already built in other ontologies. Ontology implementation requires the use of an environment that supports the meta-ontology and ontologies selected at the integration phase. Knowledge acquisition is an independent activity in the ontology development process. Experts, books, handbooks, figures, tables and even other ontologies are sources of knowledge from which the knowledge can be elucidated.

In SESeW, we assume the existence of a list of gathered terms from text files or other ontologies generated using knowledge acquisition techniques such as text analysis, structured interview or brainstorming. In the conceptualization phase, users are required to identify the classes from a list of possible classes, the instances of each class, and the relationships between individuals and classes.

Properties are distinguished by whether they relate individuals to individuals or individuals to datatypes. Datatype properties may range over RDF literals or simple XML Schema datatypes. To create a datatype property, users are required to provide a property name by either selecting one from the Glossary of Terms or typing a name into the text field. The user then selects the property domain and range. Figure 6.3 shows the window for introducing new datatype properties. A datatype property can be a *FunctionalProperty*.

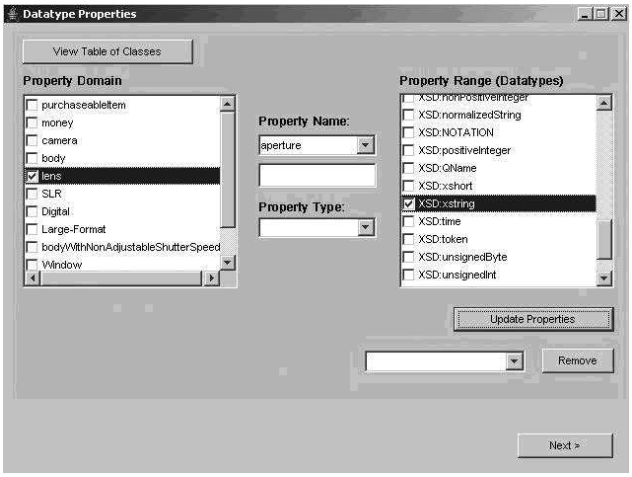

Figure 6.3: Creating Datatype Property

The creation of object properties is similar to the creation of datatype properties, except that an object property has classes as its range (instead of datatypes). Besides *FunctionalProperty*, it may be of three other property types: *TransitiveProperty*, *SymmetricProperty*, and *InverseFunctionalProperty*.

In addition to designating property characteristics, it is possible to further constrain classes with property restrictions. The six types of restrictions defined in OWL are all supported in SESeW:

- *hasValue*: which allows users to restrict classes by requiring the existence of particular property values.
- *allValuesFrom*: which requires that for every instance of the class that has the specified property, the values of the property are all members of the class indicated by the *allValuesFrom* clause.
- *someValuesFrom*: which requires that for every instance of the class that has the specified property, at least one value of the property is a member of the class indicated by the *someValueFrom* clause.
- *cardinality*: which requires the specification of the exact number of elements in a relation.
- *minCardinality*: which permits the specification of the minimum number of elements in a relation.
- *maxCardinality*: which permits the specification of the maximum number of elements in a relation.

After the user has fully specified the ontology, it is automatically generated, making use of the Jena Framework [51] (bundled with SESeW), shown in its text area and saved into the file designated.

6.2.1 Performance Evaluation

To evaluate the performance of ontology generation in SESeW, experimental performance monitors are included to find out the memory and computational time used

for creation of ontologies.

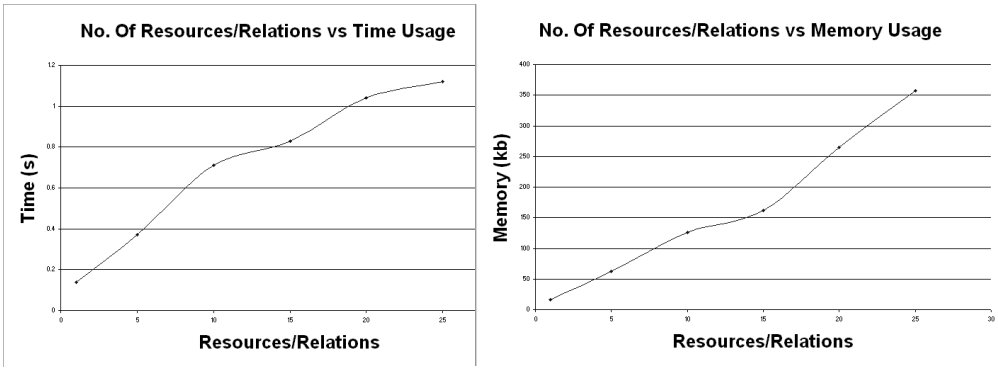

Figure 6.4: Performance of Ontology Creation

Figure 6.4 shows how the increase in the number of ontology resources and relations affects the time and memory usage of the tool, assuming that in an ontology building process, each resource/relation costs same amount of time and consumes same amount of memory. Approximately, the time and memory usage increases linearly as the number of resources/relations increases. The average time needed for creating one resource/relation decreases slowly.

6.3 Ontology Querying

A friendly user interface, shown in Figure 6.5, is provided for querying a given ontology. A user may input queries in an SQL-like language RDQL [86]. The query engine is a part of the ontology toolkit, the Jena Framework. An RDF model can be viewed as a graph, often expressed as a set of triples. An RDQL consists of a graph pattern, expressed as a list of triple patterns. Each triple pattern is comprised of named variables and RDF values (URIs and literals). An RDQL query can additionally have a set of constraints on the values of those variables, and a list of the variables required in the answer set. A typical query has the structure "`SELECT...WHERE...USING...`",

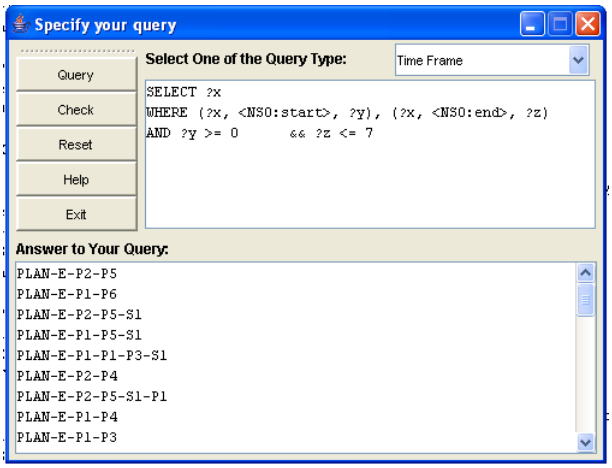

Figure 6.5: The Query Interface

where ontology entities of interest are specified after the keyword `SELECT` along with constraints after the keyword `WHERE` in the namespaces given after the keyword `USING`.

As SESeW was initially developed as part of a military-related research project, frequently used query patterns in the military planning domain are categorized and templates are created to ease the creation of such queries. For example, the category "instantiation" provides a template to create queries to find out all instances of a particular class. Once a user selects a query type, the text area for typing in query is updated with the corresponding templates. After a query is entered, user may perform syntax checking of the query before submitting it. As the target users may not have the required expertise to identify the namespaces of a given ontology, the namespaces in an ontology are automatically recognized and extracted for user's convenience.

For the military plans case study, we have developed a set of 14 query templates, including queries to find the sub-task/super-task relationship with regard to a particular military task, queries to find all military tasks whose start and end time fall

into a particular time frame, queries to find all military tasks that proceeds/follows a given task, and queries to find a military unit assigned to execute a given task, etc. This set of templates greatly eases the querying and understanding of the ontology.

6.4 Ontology Transformation

The main purpose of the SESeW is to realize our approach of using software engineering techniques and tools such as model-checking and theorem proving to verify DAML+OIL/OWL/RDF ontologies. Thus, SESeW provides fully automated transformation from ontologies to Alloy, Z and PVS specifications.

The transformation from DAML+OIL/OWL ontologies to Z specifications was discussed in detail in 3.4. Originally the transformation from DAML+OIL to Alloy was accomplished with an XSLT [109] stylesheet. To be integrated into the SESeW framework, the transformation program has been re-written using Java language. The transformation is based on the semantics library for DAML+OIL built in Alloy and Z. The semantical libraries are straightforwardly extended to OWL by defining the Alloy and Z semantics for the OWL language. The Z semantics is contained in a section $owl2z$, on top of *toolkit*. Similarly, the Alloy semantics is contained in a module *owl*.

The transformed Alloy or Z specification is presented in its own text area. The first lines of the transformed specification imports the Alloy or Z library for DAML+OIL or OWL constructs. The transformed specification is ready to be imported to Alloy Analyzer or Z/EVEs for various reasoning tasks.

The transformation from DAML+OIL/OWL/RDF to Z has been fine-tuned to make the proof using Z/EVES more automated. In Z/EVES, a name must be declared

before it is used. Hence, the transformation program extracts all the names of declared classes, properties and individuals first, put them at the beginning of the generated Z specification. In subsequent passes, predicates about these names are then grouped and generated. As these predicates are used in proof process, *labels* are systematically added to all the predicates for easy referencing later on.

In addition, SESeW also includes the fully automated transformation from ontology languages to PVS so as to verify both OWL and SWRL ontologies. In order to use PVS to verify and reason about ontologies with SWRL axioms, it is necessary to define the PVS semantics for OWL and SWRL. This semantic model forms the reasoning environment for verification using PVS theorem prover. The complete PVS semantics for OWL language primitives and the newly proposed SWRL are available online[2]. To make the proving process of PVS more automated, a set of rewrite rules and theorems are defined. They aim to hide certain amount of underlying model from the verification and reasoning and to achieve abstraction and automation. Usually these rules relate several classes and properties by defining the effect of using them in a particular way. PVS is used for standard SW reasoning like inconsistency checking, subsumption reasoning, instantiation reasoning as well as checking SWRL and beyond. For instance, OWL and SWRL cannot deal with the concrete domains: it can only make assertions about linear (in)equalities of cardinalities of property instances over integer. PVS, on the other hand, can perform basic arithmetic operations and comparisons.

[2]cf. http://nt-appn.comp.nus.edu.sg/fm/ORL2PVS/OWL2PVS.pvs.txt

6.5 External Tools Connection

SESeW also integrates existing tools for developing and reasoning about ontologies so that a user may choose his/her favorite tool(s) to prepare the ontology before using our approach for reasoning about or verifying the finished ontology to obtain confidence. The following tools are bundled with, or connected to SESeW with shortcuts:

- Alloy Analyzer: It is bundled with SESeW and can be invoked directly
- Z/EVES: A shortcut to Z/EVES previously installed in a machine is provided in SESeW to invoke it.
- RACER: Acting as a background reasoner, RACER is bundled with SESeW so that its reasoning functionality can be directly tapped whenever required. Moreover, RACER also acts as a background reasoner for ontology editors.
- OilEd: Being an ontology editor for DAML+OIL, OilEd is bundled with SESeW so that ontologies can be developed, visualized and reasoned about.

Alloy Analyzer is also developed in Java so that it is possible to develop programmatic ways of accessing functionalities of Alloy Analyzer if the API is provided. In this way, SESeW becomes a more integrated formal environment. As Alloy Analyzer can pin point the source of identified error, it will be more user-friendly if SESeW can directly command Alloy Analyzer to bring up the identified erroneous source statements. We are currently involving people to explore the source code of Alloy Analyzer for this purpose. Z/EVES is developed in Allegro Common Lisp and it presents a more complicated challenge for integration with SESeW.

6.6 Chapter Summary

As we have shown in previous chapters, formal methods can be successfully applied to the Semantic Web domain to improve the quality of ontologies. To advocate this application, we developed an integrated tools environment, the SESeW, so that different tools from both the SW and formal methods communities can be grouped together and used in combination more efficiently. In a nutshell, SESeW allows systematic creation as well as effective querying, transformation, verification and reasoning about DAML+OIL/OWL/RDF ontologies.

The SESeW includes functionalities such as ontology creation, querying, transformation, etc. It also links with a number of external tools to visualize and reason about ontologies.

By implementing a systematic approach of ontology creation, the Methontology, and supporting ontology querying and the combined approach of verifying ontology correctness, the SESeW supports a complete ontology life cycle.

So far, the chapters are only focused on transforming and verifying static Web resources. The dynamic aspect of the Web, the Web services, will take stage in the next chapter.

Chapter 7

Simulating Semantic Web Services with LSCs and Play-Engine

As introduced in Chapter 1, the full potential of the Web is realized when not only static information, but also dynamic Web services, are processable by software agents.

Web Services provide a standard way of interoperation between applications that may be running on a variety of platforms. The interoperability is achieved by the development of employment of a set of XML-based open standard protocols/languages such as WSDL [14], SOAP [110] and UDDI [99]. Web services encoded in such protocols can be autonomously understood by applications.

Although the above languages are still in evolution, it has been recognized that there is a growing need for semantically richer specification languages. Such *semantical* specifications can further automate various activities of the life cycle of a Web service, such as service invocation, selection, composition, negotiation, etc. For these reasons the Semantic Web Services ontology was developed.

The Semantic Web Services ontology, called OWL-S, is an ontology in OWL DL

language. As introduced previously, it contains essential mark-ups for describing a Semantic Web service. Such markups can be categorized into three parts: service grounding, service profile and service model. The details can be found in Chapter 2.2. The service model component describes the how the service works, detailing its inputs, outputs, preconditions, effects, control flow, etc. Hence, it is essential to the selection and invocation of a service. It is important to ensure the correctness of such models as erroneous service descriptions will give rise to invocation of wrong services, with wrong parameters, resulting in undesired outcome.

In this chapter, we demonstrate how to encode Semantic Web service models as Live Sequence Charts (LSCs) and how to simulate them using Play-Engine.

The chapter is divided into four sections. Section 7.1 is devoted to an introduction to the LSCs and Play-Engine, the visualization and simulation tool support for LSC. In Section 7.2, we introduce how OWL-S ontologies are transformed into LSCs. In Section 7.3, we demonstrate the simulation process through a case study of an online holiday booking system. Finally, Section 7.4 summarizes the chapter.

7.1 LSCs & Play-Engine

Live Sequence Charts (LSCs) [18] are a powerful visual formalism which serves as an enriched requirements specification language. LSCs are a broad extension of the classic Message Sequence Charts (MSCs [53]). They capture communicating scenarios between system components rigorously. LSCs distinguish scenarios that must happen from scenarios that may happen, conditions that must be fulfilled from conditions that may be fulfilled, etc.

There are two kinds of charts in LSCs: existential charts and universal charts. Exis-

tential charts are mainly used to describe possible interactions between participants in early stages of system design. At a later stage, knowledge becomes available about when a system run has progressed far enough for a specific usage of the system to become relevant. Universal charts are then used to specify behaviors that should always be exhibited. A universal chart may be preceded by a pre-chart, which serves as the activation condition for executing the main chart. Whenever a communication sequence matches a pre-chart, the system must proceed as specified by the main chart. A chart typically consists of multiple instances, which are represented as vertical lines. Along with each line, there are a finite number of locations (i.e., the joint points of instances and messages). A location carries the temperature annotation for progress within an instance. Message passing between instances is represented as horizontal lines. Cold conditions are used to assistant specifying complex control structures like guarded-choice, do-while. Hot conditions are asserted to assure critical properties at certain point of execution. Typically, a system is described by a set of LSCs, both universal charts and existential charts. LSCs support advanced MSC features like co-region, hierarchy, etc. For details on features of LSCs, refer to [37]. LSCs are far more expressive than MSCs, which makes them capable of expressing complicated inter-objects system requirements.

An interaction-based model specifies the desired inter-object relationships before a system is actually constructed. It is beneficial if the model can be simulated and tested so as to detect inconsistencies and under-specification. One of the significance of LSCs is that descriptions in the LSC language can be executed by Play-Engine [38] without implementing the underlying object system. Play-Engine is a tool recently developed to support an approach to the specification, validation, analysis and execution of LSCs, called “play-in” and “play-out”. Behaviors are “played in” directly from the system’s user interface, and as this is being done the Play-Engine continuously constructs LSCs. Later, behaviors can be “played out” freely from the user inter-

face, and the tool executes the LSCs directly, thus driving the system's behaviors. When "playing out", Play-Engine computes a "maximal response" to a user-provided event, called a super-step. During the computation of a super-step, hot conditions are evaluated. If any hot condition evaluates to false, a violation is caught. Otherwise, simulation continues with the user provided events. This way, users may detect undesired behaviors allowed by the specification early in the development. The basic play-out engine arbitrarily explores a single super-step, hence possibly running into problems. The smart play-out approach uses model checking to compute a valid super-step if it exists. Alternatively, test cases may be supplied by the users as existential charts so that Play-Engine may guide the system accordingly to verify that a scenario of interactions between the user and system is possible.

7.2 Modeling OWL-S with LSCs

7.2.1 Basics

The work in this chapter is concentrated on the process model of OWL-S and we abstract away the service profile and grounding details. The key idea of using LSCs to visualize and simulate the OWL-S process models is to use an LSC universal chart to capture a process model. In other words, each process is viewed as describing a possible communicating scenario between a service-using agent and the service-providing agent. For each process model, we assume there is a pre-service request from the service-using agent to the service-providing agent that identifies the service to perform, which corresponds to the service grounding phase that we ignore in this work. For instance, the *request*() message in Figure 7.2 is a pre-service request from a *HolidayBookingAgent* to a *BdgtChker*. Once a pre-service request is exchanged between the service-using agent and the service-providing agent, subsequent interactions

follow precisely as defined in the service definition (the process model).

In OWL-S, processes are modeled as OWL classes and they are sub classes of one of the three mutually disjoint OWL classes: *AtomicProcess*, *SimpleProcess* and *CompositeProcess*.

Processes can have inputs, outputs, preconditions, effects (IOPEs) and results, which are also defined as OWL classes. A result bundles (conditioned) effects and outputs.

Besides defining these classes, the OWL-S ontology also defines a number of object properties that defines the IOPEs of a process. The following list briefly explains these properties.

- *hasInput*: It specifies one of the inputs of the service.
- *hasLocal*: It specifies one of the local parameters. Local parameters are only used in atomic processes.
- *hasOutput*: It specifies one of the outputs of the service.
- *hasPrecondition*: It specifies one of the preconditions of the service. Preconditions are evaluated with respect to the client environment before the process is invoked.
- *hasResult*: It specified one of the *Results* of the service. Results can be associated with post-conditions by the property *inCondition*. Result conditions are effectively meant to be 'evaluated' in the server context after the process has executed. The outputs and effects of a result can only occur if its conditions are evaluated to true.

  Post-condition of the *inCondition* properties in *hasResult* are conjoined and identified with a shared hot condition at the end of the chart so that if the post-condition is violated, an error is raised by Play-Engine. The *withOutput* properties are then identified with communications after the hot condition.

7.2.2 Processes

An atomic process corresponds to the actions that a service can perform by engaging it in a single interaction, i.e., a one-step service that expects a bundle of inputs and produces a bundle of outputs. An atomic process is a "black box" representation; that is, no description is given of how the process works (apart from IOPEs).

Basically, a service defined by an atomic process is translated to an LSC universal chart preceded by a pre-chart containing only the pre-service request. An atomic process has always two participants, i.e., a service-using agent and a service-providing agent if the participants are skipped in the OWL-S ontology. Otherwise, participants in an ontology are translated to instances in the chart. According to [95], "inputs and outputs specify the data transformation produced by the process", hence they are identified with communication between different participants in the main chart. If a process has a precondition, it cannot be performed successfully unless the precondition is true. Precondition of a service is, therefore, identified with a shared cold condition (among all participants) at the very beginning of the main chart. Thus, if the condition is violated, the chart terminates and hence the process (service) is not performed.

The data bindings are analyzed to identify the correspondence between different inputs and outputs and local variables (if there are). Besides, built-in functions in the process models are translated to external functions in LSC (Play-Engine) and local variables are identified with variables associated with the instances in the chart.

Composite processes are composed of sub-processes, and specify constraints on the ordering and conditional execution of these sub-processes. These constraints are captured by the *composedOf* property. Composite processes are constructed using control constructs and references to processes called *Performs*. These are analogous

to function calls in procedural language function bodies. *Perform* itself is a kind of control construct specifying where the client should invoke a process provided by some server. *Perform* may be references to atomic or other composite processes. *Perform*s are composed using other control constructs. The minimal initial set includes *Sequence*, *Split*, *Split+Join*, *Any-Order*, *Condition*, *If-Then-Else*, *Iterate*, *Repeat-While* and *Repeat-Until*. We summarize the list of control constructs in Table 7.1 (according to OWL-S 1.1).

Table 7.1: A Partial Summary of the OWL-S constructs

OWL-S Constructs	**Description**
Sequence	Executes a list of processes in order.
Split	Executes a bag of processes concurrently.
Split+Join	Executes a bag of processes concurrently with barrier synchronization.
Any-Order	Execute a bag of processes in any order but not concurrently.
Choice	Chooses between alternatives and executes.
If-Then-Else	Tests the *if-condition*. If *true* executes the "Then" branch, if *false* executes the "Else" branch.
iterate	Serves as the common superclass of *Repeat-While* and *Repeat-Until* and potentially other specific iteration constructs.
Repeat-While	Iterates execution of a bag of processes until the *while* Condition becomes true.
Repeat-Until	Iterates execution of a bag of processes until the *until* Condition becomes true.
timeout	Interval of time allowed for completion of the process component (relative to the start of process component execution).

In the following, we discuss how composite services are systematically transformed to LSCs. We present the transformation in the following as transformation rules for each and every control construct in Table 7.1.

- *Sequence*: It is naturally translated to sequential communication along the vertical lines in a chart. If a sub-process itself is composed by other processes, the sub-process is transformed to a sub-chart or a pre-service request in case the sub-process is reused in other processes. Variables in the output bindings are parameterized with the message so that they are unified with the variables in the invoked processes.
- *Split*: Because no specification about waiting or synchronization is made among the bag of process components, processes in *Split* correspond to multiple pre-service requests grouped as a co-region so that the ordering of the execution of the components are not constrained. Each pre-service request will in turn activate an LSC modeling the corresponding service.
- *Split+Join*: Because of the possible barrier synchronization, it is transformed to LSCs similarly as *Split* with additional 0-buffered communication corresponding to the barrier synchronization. The 0-buffered communication events are shared by all LSCs modeling the invoked services. Therefore, the synchronization is made among all sub-processes. Moreover, the location where the co-region is set to be hot so that completion of all components are guaranteed.
- *Any-Order*: All components of an *Any-Order* control construct must be executed, but not concurrently. This requires that no execution of any two processes can overlap. This is transformed to LSCs exactly as *Split* except all locations in LSCs corresponding to the components are set to be hot so that completion of all components are guaranteed.
- *Choice*: This corresponds to the *Select-Case* construct in LSCs. Thus, a choice in OWL-S is transformed to a *Select-Case* sub-chart with equally distributed possibility.
- *If-Then-Else*: The exact same construct *if-then-else* is available in LSCs. The

If-condition and *Else-condition* are mapped to cold conditions in the respective sub-chart. The only problem is to syntax-rewrite the logical expression used in OWL-S (represented in SWRL [48], DRS[1] or KIF[2]) properly to logical expression in LSCs.

- *Repeat-While and Repeat-Until*: Both these two constructs are sub classes of the *abstract* control construct *Iterate*, whereas the former is transformed to a looping sub-chart in LSCs with a shared cold condition (corresponding to the condition in the service definition) at the end of the sub-chart and the latter is transformed to a looping sub-chart in LSCs with a cold condition (corresponding to the negation of the condition in the service definition) at the end of the sub-chart.
- *timeout*: *timeout* is defined as an object property on the above control constructs, each of which can have at most 1 such timeout instance. It is mapped to a timer set event followed by a timeout event in LSCs containing the respective process components.

The transformation rules for composite processes are applied inductively. One of the difficulties of using LSCs to simulate the OWL-S process models is to perform correct data binding and data computation. We assume that a simple underlying data and functional model of the system is supplied by the users, i.e. the underlying system variables and the implementation of the external functions and so on. To simulate the set of process models interactively, we may build a simple user interface to trigger environmental events manually. A simple user-interface is built with a button for triggering every process model. Play-Engine supports building such user-interface with Visual Basic, and "playing-out" the corresponding LSCs according the

[1] cf. http://www.daml.org/services/owl-s/1.0/conditions.html
[2] cf. http://logic.stanford.edu/kif/dpans.html

user interaction through the interface.

7.3 Case Study

This section illustrates the approach with an example of an online holiday booking system.

7.3.1 System scenario

The holiday booking system is a Web portal offering access to information about air tickets and hotels. This Web portal provides automated air ticket and hotel booking services to users who are planning their holidays.

In the course of operation, the customer submits a request, which includes the information about the destination, travelling time and maximum budget, to the holiday booking agent. Upon receiving the request, the holiday booking agent tries to find the most suitable air ticket and hotel based on information in the customer's preferences, which have been obtained from his online, OWL-encoded profile. The preferences may include the preferred airlines, hotels, etc. Following that, the holiday booking agent calculates if the total cost overruns the budget limit. If the total cost is more than customer's budget, the holiday booking agent tries to find another cheaper hotel or ticket. If there is no ticket and hotel combination that can be found within the budget, the customer will be notified. Otherwise the booking agent shows the information about the matched ticket and hotel to the customer. If the customer is satisfied, he/she submits his/her credit card information to the holiday booking agent. The holiday booking agent asks a third-part credit checking agent to check if the card is valid with sufficient credit. If it is, the booking will be made.

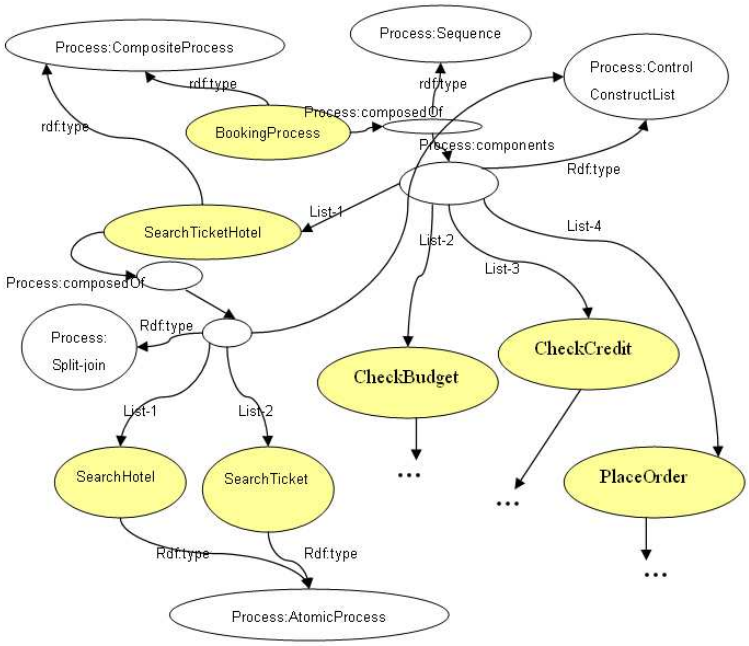

Figure 7.1: Holiday booking System

Figure 7.1 is an RDF graph of the service model ontology. It shows part of the OWL-S process model for the holiday booking agent[3]. The holiday booking service has a composite process *BookingProcess* which sequentially performs four sub-processes – *SearchTicketHotel*, *CheckBudget*, *CheckCredit* and *PlaceOrder*. *SearchTicketHotel* is a composite process as well, which performs two atomic process, *SearchHotel* and *SearchTicket*, in parallel. The complete OWL-S process model can be found at `http://www.comp.nus.edu.sg/~liyf/booking.xml`.

Being part of our case study, the following is the process model of an atomic OWL-S service ontology that checks whether the current air ticket and hotel prices are within user budget, given as inputs the air ticket price (variable `X1`), hotel accommodation cost (variable `X2`) and the user's budget (variable `X3`)[4]. As output, this atomic service

[3]The diagram has been slightly revised for presentation purpose.

[4]These variables are represented as `budget_ticket_Cost`, `budget_hotel_Cost` and

returns `true` for variable `Check_Budget_result` if `X3` $\leq$ `X1` + `X2`, and false otherwise. For atomic processes, the inputs must come from the service-using agent.

```
<process:AtomicProcess rdf:ID="CheckBudget">
  <process:hasInput><process:Input rdf:ID="budget_hotel_Cost">
    <process:parameterType rdf:datatype="&xsd;#nonNegativeInteger"/>
  </process:Input></process:hasInput>
  <process:hasInput><process:Input rdf:ID="budget_ticket_Cost">
    <process:parameterType rdf:datatype="&xsd;#nonNegativeInteger"/>
  </process:Input></process:hasInput>
  <process:hasInput><process:Input rdf:ID="budget_total_Cost">
    <process:parameterType rdf:datatype="&xsd;#nonNegativeInteger"/>
  </process:Input></process:hasInput>
  <process:hasOutput><process:Output rdf:ID="Check_Budget_result">
    <process:parameterType rdf:datatype="&xsd;#anyURI">&xsd;#boolean
  </process:parameterType></process:Output></process:hasOutput>
  <process:hasResult>
    <process:Result rdf:ID="Within_budget">
      <process:withOutput>
        <process:OutputBinding>
          <process:toParam rdf:resource="#Check_Budget_result"/>
          <process:valueData rdf:datatype="&xsd;#boolean">true
          </process:valueData></process:OutputBinding></process:withOutput>
      <process:inCondition>
        <expr:KIF-Condition>
          <expr:expressionBody>
              (>= ?budget_total_Cost
                  (+ ?budget_ticket_Cost ?budget_hotel_Cost))
          </expr:expressionBody>
        </expr:KIF-Condition>
      </process:inCondition>
    </process:Result>
  </process:hasResult>
  <process:hasResult>
    <process:Result rdf:ID="beyond_budget">
      ...
    </process:Result>
  </process:hasResult>
</process:AtomicProcess>
```

Figure 7.2 shows an LSC universal chart capturing the necessary interactions between a service-using agent and a budget-checking agent cooperating in the above

`budget_total_Cost` in the ontology, respectively.

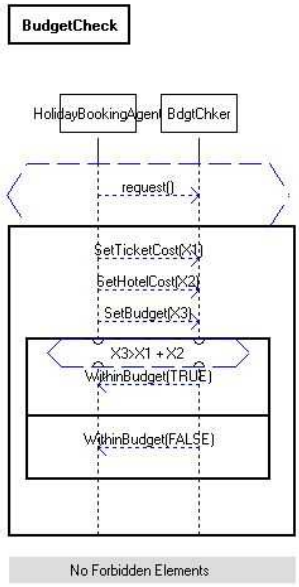

Figure 7.2: LSC Example: Budget checking

atomic service. Once the service-using agent requests the service *CheckBudget* (after determining whether the service meets its needs by exploring the service profile), necessary information like `budget_ticket_Cost` and `budget_hotel_Cost` is supplied by the service-using agent. The budget-checking agent replies with true, if the budget is at least as much as the sum of the air ticket and hotel prices, and false otherwise.

7.3.2 Simulation

Figure 7.3 shows in Play-Engine part of the LSC of the *HolidayBooking* process model. Given a set of inputs including departure and destination cities, outbound and inbound dates, budgets, etc., the service searches for valid air tickets and hotels. Finally if such flights and hotel accommodation are available, it proceeds to book the flight and room.

Our simulation begins with building a simple Graphical User Interface (GUI) for

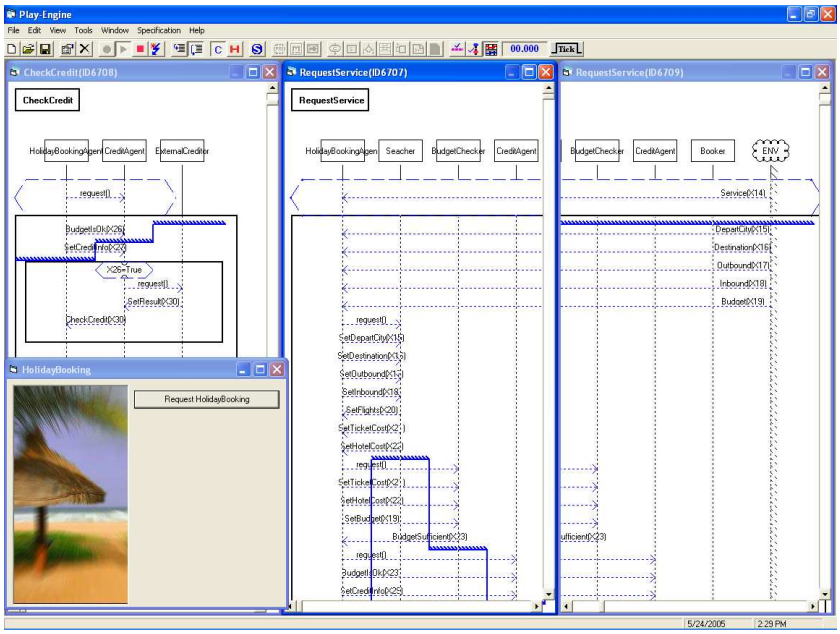

Figure 7.3: Simulation Screen Shot

interactively introducing external events. A systematic approach is to build one GUI component for each user-accessible Web service. In our example, only one Web service is accessible to service-using agents, namely *HolidayBooking*. The simple GUI is shown in the left bottom corner of Figure 7.3. Play-Engine allows user-defined variables and external function through ActiveX DLLs. For the purpose of simulation before actual implementation, an abstract "implementation" capturing only necessary details of the system is sufficient. However, if the underlying data and functional system is implemented using techniques compatible with ActiveX DLLs, e.g. ASP, .NET, Play-Engine may import the actual implementation of the underlying system and perform the simulation.

From our experiences, symbolic messages and instances are very helpful for capturing the OWL-S process models compactly. After building the LSC model, a user may interactively play out the system by initiating an (or a series of) external event

and check how the system proceeds step-by-step. Assertion can be inserted freely by introducing hot conditions in the LSCs. During simulation, a violation of the hot condition will be caught by Play-Engine. This way, inconsistency and under-specification is detected intuitively. In case an external process (to be offered by third party) is assumed, the user may specify the possible output of the process manually or Play-Engine would use model-checking techniques to automatically find a valid value (if the variables have finite domain). In our example, during simulation, windows pop up for the user to specify the ticket price and the hotel price. Alternatively, a user may build a test case of the system as an existential chart (with assertions) and let Play-Engine do the guided play-out according the existential chart.

In Figure 7.3, the *HolidayBooking* process is invoked by two different service-using agents. Hence, two copies of the chart *HolidayBooking* (according to the *HolidayBooking* process) are monitored. With simulation run of this scenario, where a number of service-using agent are using the ticket-booking service, we gain confidence that the same shared resource (e.g. ticket vacancy) is accessed exclusively.

7.4 Chapter Summary

In this chapter, we propose to use LSCs and Play-Engine to visualize and simulate OWL-S process models. The significance and novel aspects can be summarized as follows. Firstly, by transforming an OWL-S service model ontology into an LSC, service developer can design the services in a more visual and intuitive manner. In XML format, the LSCs can be easily transformed back to OWL-S. Secondly, we may simulate the interactions without implementing the Web service (exploring the service grounding), and be able to gain confidence of the service models. The key point of this approach is that a Web service can be naturally viewed as a desired usage of the web

agent, i.e., a scenario of the interaction between the service-using agent the service-providing agent. Thirdly, as Play-Engine supports dynamic linked libraries such as COM and ActiveX Controls, Web services written in these libraries can be more easily transformed to LSCs, from which the OWL-S service model may be derived. Hence, our approach also facilitates the integration of Web services with OWL-S. Moreover, we presented a travel booking case study to demonstrate our approach.

There are a number of future work directions that we deem as worthwhile to pursue. First of all, it is necessary to develop programs to automatically construct LSCs from the OWL-S process models to make this approach more practical. Recently an OWL-S editor has been developed[5] as a plug-in for the Protégé OWL Editor [57]. It will be valuable for OWL-S developers if they can obtain feedback, in terms of simulation results, from Play-Engine simulations directly to the editor. Hence, such a deep linking between Play-Engine and the OWL-S editor is desirable. Besides LSC and Play-Engine, formal languages such as CSP [42] can also be considered to represent OWL-S ontologies and their tool support, such as the FDR [83] or SPIN [43] model checkers, may also be used to perform verification tasks. They are part of the future research plan that will be detailed in the next chapter.

We foresee that Web Services will be a new and fruitful application domain of Software Engineering (SE) methods and tools. Our approach, along with other approaches on applying SE methods to the Web domain, offers both experience and possible tool supports for developing Web services languages and techniques.

[5] cf. http://owlseditor.semwebcentral.org/index.shtml

Chapter 8

Conclusion

This chapter serves two purposes. Firstly, a conclusion of the whole thesis is given, summarizing the main contributions and secondly, a discussion on future work directions is also presented.

8.1 Main Contributions of the Thesis

Ontology languages form the foundation of the Semantic Web and they are of utter importance to the upper-layer technologies in the Semantic Web, such as Web services, trust modeling, etc.

As the Semantic Web is envisioned as a ubiquitous network for humans as well as machines, software agents can cooperate and aggregate Web resources from different sites to carry out complex tasks autonomously. Hence, automation of core reasoning tasks performed by agents is very important. It is for this reason ontology languages such as DAML+OIL and OWL are designed to be decidable.

Decidability is achieved by limiting the expressivity of ontology languages.

Being based on description logics, a subset of first-order predicate logic, DAML+OIL and OWL statements can only express properties with a limited degree of complexity. Many desirable properties cannot be represented in these languages. Such a challenge is often faced by Semantic Web developers as it is often the case that complex properties capture vital information pertaining to the validity of the ontology are too complex to be modeled in DAML+OL or OWL, even in its most expressive species OWL Full.

The newly proposed rules extension to OWL, the SWRL, partially solves the problem by incorporating Horn-style clauses into OWL.

Being able to represent the complex properties is only the first step. The ability to reason about ontologies and associated complex properties efficiently is at least as important. However, as SWRL is undecidable, there is unlikely that a proof tool can support all reasoning tasks for SWRL ontologies.

This thesis presents my research works in answering some of these challenges. The five main contributions of this thesis can be summarized as follows.

- We identified the expressivity limitation of ontology languages and defined a Z semantics (in Chapter 3) for the ontology languages DAML+OIL and OWL, making it possible to use software engineering proof tools such as Z/EVES to perform complex reasoning tasks on Semantic Web ontologies. We have shown that properties crucial to the validity of an ontology can be checked by Z/EVES. Some of these properties are inexpressible in ontology languages, even in SWRL.
- Based on the above work, we proposed a combined approach in Chapter 4, exploiting the complementary power of software engineering proof tools such as Z/EVES and Alloy Analyzer and Semantic Web reasoning engines such as

RACER and FaCT++. The application of these tools in combination can verify the correctness of DAML+OIL/RDF ontologies and debug inconsistent ontologies more effectively.

RACER and other Semantic Web reasoning engines are fully automated. Given an ontology, these reasoners can judge whether it is consistent without user interaction. However, as stated previously, the automation is based on the fact that the expressivity of ontology languages is limited. Hence, complex properties inexpressible in these languages are certainly un-checkable by these tools. Moreover, these description logics-based tools can only detect that there is an inconsistency in the ontology, they cannot tell where and how this is caused, making debugging large ontologies very hard.

Alloy Analyzer is an automated constraint solver with the ability of finding the source of the error if there is one. This ability is achieved by giving a finite scope to each Alloy specification to be solved by Alloy Analyzer. This fits naturally with Semantic Web reasoning engines as Alloy Analyzer can be used like a surgery tool to precisely locate the source of the inconsistencies found by Semantic Web reasoning engines.

Theorem provers such as Z/EVES are very powerful and they can prove complex properties that ontology languages and Alloy cannot represent. Hence, Z language is used to represent complex properties about ontologies and Z/EVES is used to perform a final proof of such complex properties interactively.

The above combined approach has been successfully applied to a military planning ontologies case study, where one ontological inconsistency was discovered and located and a number of errors undetected by RACER and Alloy Analyzer were found by Z/EVES.

- The applicability of the above combined approach largely relies on the soundness of the Z/Alloy semantics for DAML+OIL and OWL DL. Hence, it is impor-

tant to formally prove the soundness of these semantics. As OWL is based on DAML+OIL and it has been recommended as *the* ontology language, we have developed a Z semantics for OWL DL, a sub language of OWL that is most comparable with DAML+OIL.

As ontology languages and Z are based on different logical systems, a more abstract device that is able to represent and inter-relate different logical systems is needed to formally investigate their relationship. Institutions were introduced to formalize the notion of logical systems. Institution morphisms provide means of translating signatures of different institutions while preserving *truth*. Hence, institutions and institution morphisms are natural candidates to prove the soundness of Z semantics for OWL DL (hence DAML+OIL). In Chapter 5, we have defined institutions $\mathfrak{OD}$ (for OWL DL) and $\mathfrak{Z}$ (for Z) and used institution comorphisms to prove the soundness of the above semantics.

- To ease the application of the combined approach, we have developed a tools environment, the SESeW. Chapter 6 presented this environment in detail.

  SESeW implements the ontology development methodology, the Methontology [29] to systematically create an ontology. Given a number of terms in a particular domain, a user can create an OWL ontology by following some simple steps to designate terms to OWL classes, properties and individuals and relate them.

  With an ontology, SESeW can perform a number of tasks. Firstly, a user can transform it into specifications in various formal languages such as Alloy, Z and PVS [79] fully automatically. Secondly, a user can query the ontology by issuing RDQL [86] queries in the friendly interface provided by SESeW. A number of query templates in the military domain have been created for non-expert users. These templates simplifies the querying process by hiding non-necessary technical details.

  Moreover, SESeW serves as a point of contact to the various external editing

and reasoning tools such as OilEd, RACER, Alloy Analyzer and Z/EVES. It can also invoke functionalities of RACER directly to check the consistency of a given DAML+OIL or OWL ontology.

- In the development of the Semantic Web, a services ontology, the OWL Services (OWL-S), has been developed to add semantic information to the Web services. This is one step closer to realize the full potential of the Web.

  This thesis presented an approach to visualize and simulate Semantic Web services (OWL-S) [95] ontologies using Live Sequence Charts (LSCs) [18] and Play-Engine [38].

  The OWL-S ontology was developed to complement Web Services standards such WSDL [14] to semantically markup the capabilities, requirements, control constructs, inputs/outputs, preconditions and effects of Web services.

  As OWL-S ontologies capture dynamic aspects of Web services, the core reasoning services, namely subsumption, consistency and instantiation, are no longer adequate to ensure their correctness.

  In Chapter 7, we translate OWL-S process models to Live Sequence Charts and use Play-Engine to visualize and simulate them. By "playing out" the charts, potential undesired scenarios can be detected early, without actually implementing the services.

In summary, our research in this thesis attempts to answer some of the challenges in the realization of the Semantic Web vision by representing and proving complex ontology-related properties using a combination of software engineering and Semantic Web techniques synergistically. It also opens up a new application domain for software engineering languages and tools.

8.2 Future Work Directions

Based on the works in this thesis, there are a number of directions of future research that may be beneficial to the Semantic Web community. In this section, some of these possible research works are briefly discussed.

8.2.1 Further Development of SESeW

As presented in Chapter 6, the SESeW tools environment is developed to ease the application of the combined approach. Still in a prototype stage, there is room for improvement. Based on the feedback from users, we will further improve it in the following aspects:

Support of up-to-date RDF query engine Recently, a more sophisticated query RDF language, the SPARQL [81] has been developed to replace RDQL. How SESeW can support this query language is also a future research work.

Support of rules extension of OWL As we mentioned in the overview in Chapter 2, SWRL [48] has been accepted by the W3C as a member submission. It is layered on top of OWL to improve the expressivity of the Semantic Web languages. It is very likely for SWRL to be officially integrated into the Semantic Web. Hence, it is necessary to keep SESeW updated with the technology trend. The Z semantics for SWRL has been developed in Chapter 4. The Alloy and PVS semantics for SWRL can be similarly defined. By incorporating transformation procedures into SESeW, SWRL ontologies can be checked using software engineering tools such as Z/EVES, Alloy Analyzer and the PVS theorem prover [78]. With the support of SWRL, we can look into SWRL FOL [9], an extension of SWRL towards full first-order logic. With SWRL FOL, being a part of the

combined approach and SESeW, expressive power of Z and Alloy can be tapped by translating Z theorems and/or Alloy assertions and facts into SWRL FOL ontologies. By doing so, software engineering practitioners can work with ontologies with greater ease.

Tighter integration with external tools Some of the external reasoners used in the combined approach such as Alloy Analyzer and RACER provide Java-based APIs, which can be used to make direct function calls from within SESeW, e.g., calling reasoning functions from SESeW directly without invoking the GUI of RACER to determine the consistency a given ontology.

This improvement has already been experimented where from SESeW, we can already invoke RACER's methods to check the consistency of a given DAML+OIL/OWL ontology.

More flexible support for ontology query Currently SESeW supports ontology query with built-in query templates particularly geared towards the military plan ontologies. It is our development plan that users are able to create, modify and delete query templates in a future version.

8.2.2 Verification of Web Ontologies – Beyond Static Data

Semantically marked-up data on the Web alone cannot fulfill the full potential of the Semantic Web. These data must be machine-interpretable and machine-processable. Web Services, enable users to effect changes in the world. Built on top of OWL, the OWL Services ontology OWL-S [95] provides semantic markup for low-level service description languages. Looking into the issue on how software engineering techniques and tools can benefit SW Services is another promising future research direction.

Chapter 7 presented our research of using Live Sequence Charts and Play-Engine to

model, visualize and simulate OWL-S process models. With no open XML textual representation of LSCs, the transformation from OWL-S to LSC is a manual process.

Model checking techniques [15] may prove to be applicable in this domain. Berghofer and Nipkow [75] have recently developed a tool for Isabelle/HOL [76] that supports random testing of specifications, which may be useful in specifying and verifying Web services ontologies. The Communicating Sequential Processes (CSP) [42] is a well-known event-based formal notation primarily aimed at describing the sequencing of behavior within a process and the synchronization of behavior between different processes. FDR (Failures-Divergence Refinement) [83] is a CSP model checker that verifies CSP models automatically. It also provides a graphical interface for determining the source of errors by analyzing the trace of events that led up to the error. Other model checkers such as SPIN [43] may also be used.

Symbolic Analysis Laboratory (SAL) [6] is a framework for combining different tools for abstraction, program analysis, theorem proving and model checking. towards the symbolic analysis of concurrent systems expressed as transition systems.

SAL defines a a common intermediate language to describe transition systems. This intermediate language serves as a common medium from which various analysis tools such as the PVS theorem prover and SMV [70] model checker can be invoked by translating the intermediate language to the specific language used by these tools.

We believe that SAL can be a candidate environment for reasoning Web service ontologies. Besides theorem proving and model checking, SAL specifications can also be translated to Java code for animation purposes. By developing translators to translate Web service ontologies to the SAL common intermediate language, the above reasoning services can all be readily deployed.

8.2.3 Augmenting the Semantic Web with Belief

As the Web is a constantly evolving and totally distributed environment, software agents may from time to time face incomplete, incoherent or incomplete data. This is especially the case when the agent needs to aggregate data developed or maintained by different sites. It will be valuable for agents in these situations to associate belief with Web resources.

Currently, all ontology languages in the Semantic Web stack, such as RDF Schema, OWL and SWRL, are based on crisp logics, in which all statements are interpreted to be either true or false. Hence, the lack of the ability of associating confidence factors with ontology statements is another prominent expressivity limitation of the current ontology languages.

We believe that by extending ontology languages to allow fuzzy or belief-based interpretations of statements will help to resolve the above problem. Belief Augmented Frames (BAF) [93] is an extension to the Minsky knowledge representation systems [72]. In BAF, concepts are represented by frames and relations between concepts are represented by slots. We associate a pair of values representing belief/disbelief values with each frame and slot. BAF-logic defines how the two values are calculated and combined to give the confidence factor of a certain frame/slot. The belief/disbelief values are obtained independent from each other, allowing for greater flexibility in modeling ignorance and confidence.

The other future research direction that is worth to pursuit is to integrate BAF with OWL and RDF to incorporate belief factors into the Semantic Web stack.

Bibliography

[1] J. Angele, H. Boley, J. de Bruijn, D. Fensel, P. Hitzler, M. Kifer, R. Krummenacher, H. Lausen, A. Polleres, and R. Studer. Web Rule Language (WRL), Version 1.0, 2005. http://www.wsmo.org/wsml/wrl/wrl.html.

[2] S. Battle, A. Bernstein, H. Boley, B. Grosof, M. Gruninger, R. Hull, M. Kifer, D. Martin, S. McIlraith, D. McGuinness, J. Su, and S. Tabet. Semantic Web Services Language (SWSL). http://www.daml.org/services/swsf/1.0/swsl/, 2005.

[3] H. Baumeister. Relating abstract datatypes and Z-schemata. In D. Bert and C. Choppy, editors, *Recent Trends in Algebraic Development Techniques - Selected Papers*, volume 1827 of *Lect. Notes in Comput. Sci.*, pages 366–382, Bonas, France, 2000. Springer-Verlag.

[4] S. Bechhofer, I. Horrocks, C. Goble, and R. Stevens. OilEd: a reason-able ontology editor for the semantic web. In *Proceedings of KI2001, Joint German/Austrian conference on Artificial Intelligence*, number 2174 in Lecture Notes in Computer Science, pages 396–408, Vienna, September 2001. Springer-Verlag.

[5] S. Bechhoffer. The dig description logic interface: Dig/1.1. Technical report, The University Of Manchester, The University Of Manchester, Oxford Road, Manchester M13 9PL, 2003.

[6] S. Bensalem, V. Ganesh, Y. Lakhnech, C. M. noz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In C. M. Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, jun 2000. NASA Langley Research Center.

[7] T. Berners-Lee. Uniform Resource Identifiers (URI): Generic Syntax. http://www.ietf.org/rfc/rfc2396.txt, 1998.

[8] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):35–43, 2001.

[9] H. Boley, M. Dean, B. Grosof, I. Horrocks, P. Patel-Schneider, S. Tabet, and G. Wagner. SWRL FOL (November 2004). http://www.daml.org/2004/11/fol/, Nov. 2004.

[10] D. Booth, M. Champion, C. Ferris, F. McCabe, E. Newcomer, and D. Orchard. Web Services Architecture. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/, Feb. 2004.

[11] J. P. Bowen and M. G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, 1995.

[12] J. P. Bowen and M. G. Hinchey. Ten Commandments of Formal Methods. *IEEE Computer*, 28(4):56–63, 1995.

[13] J. Broekstra, M. Klein, S. Decker, D. Fensel, and I. Horrocks. Adding formal semantics to the web: building on top of rdf schema. In *ECDL Workshop on the Semantic Web: Models, Architectures and Management*, 2000.

[14] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. *Web Services Description Language (WSDL) 1.1.* W3C, 1.1 edition, March 2001. http://www.w3c.org/TR/wsdl.

[15] E. Clarke. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *Proc. 10th ACM Symp. on Princ. Prog. Lang.*, pages 117–127, 1983.

[16] E. Clarke and J. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, Dec. 1996. Other working group members: R. Alur, R. Cleaveland, D. Dill, A. Emerson, S. Garland, S. German, J. Guttag, A. Hall, T. Henzinger, G. Hozmann, C. Jones, R. Kurshan, N. Leveson, K. McMillan, J. Moore, D. Peled, A. Pnueli, J. Rushbby, N. Shankar, J. Sifakis, P. Sistla, B. Steffen, P. Wolper, J. Woodcock, and P. Zave.

[17] D. Brickley and R.V. Guha (editors). Resource description framework (rdf) schema specification 1.0. http://www.w3.org/TR/rdf-schema/, Feb. 2004.

[18] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. In *Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, page 451. Kluwer, B.V., 1999.

[19] J. de Bruijn, D. Fensel, P. Hitzler, M. Kifer, and A. Polleres. Relationship of WRL to relevant other technologies. http://www.w3.org/Submission/WRL-related/, Sept. 2005.

[20] J. de Bruijn, R. Lara, A. Polleres, and D. Fensel. Owl dl vs. owl flight: conceptual modeling and reasoning for the semantic web. In A. Ellis and T. Hagino, editors, *WWW*, pages 623–632. ACM, 2005.

[21] J. S. Dong, Y. Feng, and Y. F. Li. Reasoning Support for the OWL Rules Language. In *Proceedings of First International Colloquium on Theoretical Aspects of Computing (ICTAC'04)*, Guiyang, China, Sept. 2004.

[22] J. S. Dong, Y. Feng, Y. F. Li, and J. Sun. A tools environment for developing and reasoning about ontologies. In *Proc. of 12th Asia-Pacific Software*

Engineering Conference (APSEC'05), Taipei, Taiwan, Dec. 2005.

[23] J. S. Dong, C. H. Lee, Y. F. Li, and H. Wang. A Combined Approach to Checking Web Ontologies. In *Proceedings of 13th World Wide Web Conference (WWW'04)*, pages 714–722, New York, USA, May 2004.

[24] J. S. Dong, C. H. Lee, Y. F. Li, and H. Wang. Verifying DAML+OIL and beyond in Z/EVES. In *Proceedings of 26th International Conference on Software Engineering (ICSE'04)*, pages 201–210, Edinburgh, Scotland, May 2004.

[25] J. S. Dong, Y. F. Li, J. Sun, J. Sun, and H. Wang. XML-based static type checking and dynamic visualization for TCOZ. In *International Conference on Formal Engineering Methods (ICFEM'02)*, pages 311–322, Shanghai, China, Oct. 2002. LNCS, Springer-Verlag.

[26] J. S. Dong, Y. F. Li, and H. Wang. TCOZ Approach to Semantic Web Services Design. In *Proceedings of 13th World Wide Web Conference (WWW'04)*, pages 442–443, New York, USA, May 2004.

[27] J. S. Dong, J. Sun, and H. Wang. Checking and Reasoning about Semantic Web through Alloy. In *Proceedings of Formal Methods Europe: FME'03*, volume 2805 of *Lect. Notes in Comput. Sci.*, pages 796–814, Pisa, Italy, Sept. 2003. LNCS, Springer-Verlag.

[28] R. Duke, G. Rose, and G. Smith. Object-Z: a Specification Language Advocated for the Description of Standards. *Computer Standards and Interfaces*, 17:511–533, 1995.

[29] M. Fernandez, A. Gomez-Perez, and N. Juristo. METHONTOLOGY: from Ontological Art towards Ontological Engineering. In *Proceedings of the AAAI97 Spring Symposium Series on Ontological Engineering*, pages 33–40, Stanford, USA, Mar. 1997.

[30] J. Gennari, M. A. Musen, R. W. Fergerson, W. E. Grosso, M. Crubézy, H. Eriksson, N. F. Noy, and S. W. Tu. The evolution of protégé: An environment for knowledge-based systems development. Technical Report SMI-2002-0943, Stanford Medical Informatics, Stanford University, 2002.

[31] J. Goguen and R. M. Burstall. Introducing institutions. In *Proc. Logics of Programming Workshop*, number 164 in Lect. Notes in Comput. Sci., pages 221–256. Springer-Verlag, 1984.

[32] J. Goguen and R. M. Burstall. Institutions: Abstract Model Theory for Specification and Programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, Jan. 1992. Predecessor in: LNCS 164, 221–256, 1984.

[33] J. Goguen and G. Roşu. Institution morphisms. Formal Aspects of Computing, 2002.

[34] T. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[35] V. Haarslev and R. Möller. Practical Reasoning in Racer with a Concrete Domain for Linear Inequations. In I. Horrocks and S. Tessaris, editors, *Proceedings of the International Workshop on Description Logics (DL-2002)*, Toulouse, France, Apr. 2002. CEUR-WS.

[36] V. Haarslev and R. Möller. *RACER User's Guide and Reference Manual: Version 1.7.6*, Dec. 2002.

[37] D. Harel and R. Marelly. Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach. Technical Report MCS01-15, The Weizmann Institute of Science Rehovot, Israel, 2002.

[38] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.

[39] I. Hayes, editor. *Specification Case Studies*. International Series in Computer Science. Prentice-Hall, 1987.

[40] J. Heflin. OWL Web Ontology Language Use Cases and Requirements. http://www.w3.org/TR/2004/REC-webont-req-20040210/, Feb. 2004.

[41] M. Hinchey and J. P. Bowen, editors. *Applications of Formal Methods*. Prentice Hall, 1995.

[42] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[43] G. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[44] I. Horrocks. Fact++ web site. http://owl.man.ac.uk/factplusplus/.

[45] I. Horrocks. The FaCT system. *Tableaux'98, LNCS*, 1397:307–312, 1998.

[46] I. Horrocks. DAML+OIL: a description logic for the semantic web. *IEEE Data Engineering Bulletin*, 25(1):4–9, 2002.

[47] I. Horrocks and P. F. Patel-Schneider. A proposal for an owl rules language. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 723–731, New York, USA, May 2004. ACM. http://www.cs.man.ac.uk/~horrocks/DAML/Rules/.

[48] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/, May 2004.

[49] I. Horrocks, P. F. Patel-Schneider, and F. van Harmelen. From $\mathcal{SHIQ}$ and RDF to OWL: The making of a web ontology language. *J. of Web Semantics*, 1(1):7–26, 2003.

[50] I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for very expressive description logics. *Logic Journal of the IGPL*, 8(3):239–263, 2000.

[51] HP Labs. Jena Semantic Web Toolkit - version 1. http://www.hpl.hp.com/semweb/jena1.htm.

[52] ISO/IEC 13568:2002. Information technology—Z formal specification notation—syntax, type system and semantics, 2002. International Standard.

[53] ITU. *Message Sequence Chart(MSC)*, Nov 1999. Series Z: Languages and general software aspects for telecommunication systems.

[54] D. Jackson. Micromodels of software: Lightweight modelling and analysis with Alloy. Available: http://sdg.lcs.mit.edu/alloy/book.pdf (an early version has been published in TOSEM Vol-11), 2002.

[55] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the Alloy Constraint Analyzer. In *The 22nd International Conference on Software Engineering (ICSE'00)*, pages 730–733, Limerick, Ireland, June 2000. ACM Press.

[56] Jos de Bruijn and Axel Polleres and Rubén Lara and Dieter Fensel. OWL$^-$. http://www.wsmo.org/wsml/wrl/wrl.html, 2004. Deliverable D20.1v0.2, WSML.

[57] H. Knublauch, R. W. Fergerson, N. F. Noy, and M. A. Musen. The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications. In *Proceedings of the Third International Semantic Web Conference (ISWC 2004)*,, Hiroshima, Japan, Nov. 2004.

[58] P. Lambrix. Description Logics home page. http://www.ida.liu.se/labs/iislab/people/patla/DL/index.html.

[59] S. M. Lane. *Categories for the Working Mathematician*. Springer-Verlag, New York, second edition, 1985.

[60] C. H. Lee. Phase I Report for Plan Ontology. DSO National Labs, Singapore, 2002.

[61] Y. F. Li, J. Sun, G. Dobbie, J. Sun, and H. Wang. Validating Semistructured Data using OWL. In *Proceedings of the 7th International Conference on Web-Age Information Management (WAIM'06)*, Hong Kong, China, June 2006.

[62] S. Liu. *A Structured and Formal Requirements Analysis Method Based on Data Flow Analysis and Rapid Prototyping.* PhD thesis, The University of Manchester, 1992.

[63] D. Lucanu, Y. F. Li, and J. S. Dong. Soundness proof of Z semantics of OWL using institutions. In *Fourteenth International Conference on World Wide Web (WWW'05)*, pages 1048–1049, Chiba, Japan, May 2004.

[64] D. Lucanu, Y. F. Li, and J. S. Dong. Institution Morphisms for Relating OWL and Z. In *Proc. of The 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'05)*, Taipei, Taiwan, July 2005.

[65] D. Lucanu, Y. F. Li, and J. S. Dong. Semantic web languages – towards an institutional perspective. In K. F. et al., editor, *Algebra, Meaning and Computation, Festschrift in Honor of Prof. Joseph Goguen*, volume 4060 of *Lect. Notes in Comput. Sci.*, pages 99–123. Springer-Verlag, 2006. to appear.

[66] M. Dean and G. Schreiber (editors). OWL Web Ontology Language Reference. http://www.w3.org/TR/2004/REC-owl-ref-20040210/, Feb. 2004.

[67] B. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, Feb. 2000.

[68] F. Manola and E. M. (editors). RDF Primer. http://www.w3.org/TR/rdf-primer/, Feb. 2004.

[69] D. L. McGuinness and F. van Harmelen (editors). OWL Web Ontology Language Overview. http://www.w3.org/TR/2003/PR-owl-features-20031215/, Dec. 2003.

[70] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[71] I. Meisels and M. Saaltink. The Z/EVES Reference Manual (for Version 1.5). Technical Report TR-97-5493-03d, ORA Canada, One Nicholas Street, Suite 1208 - Ottawa, Ontario K1N 7B7 - CANADA, Sept. 1997.

[72] M. Minsky. A framework for representing knowledge. In J. Haugeland, editor, *Mind Design: Philosophy, Psychology, Artificial Intelligence*, pages 95–128. MIT Press, Cambridge, MA, 1981.

[73] M. Moser, O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann, and K. Mayr. SETHEO and E-SETHEO – the CADE-13 systems. *Journal of Automated Reasoning*, 18:237–246, 1997.

[74] D. Nardi and R. J. Brachman. An introduction to description logics. In F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors, *The description logic handbook: theory, implementation, and applications*, pages 1–40. Cambridge University Press, 2003.

[75] S. B. T. Nipkow. Random testing in Isabelle/HOL. In *Proceedings of the 2nd Software Engineering and Formal Methods (SEFM 2004)*, Beijing, China, Sept. 2004. IEEE Computer Society Press. To appear.

[76] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[77] Ontoweb Ontology-Based Information. Deliverable 1.3: A survey on ontology tools. http://ontoweb.aifb.uni-karlsruhe.de/About/Deliverables/D13_v1-0.zip.

[78] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[79] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, Dec. 2001.

[80] P. F. Patel-Schneider and I. Horrocks (editors). OWL: Direct Model-Theoretic Semantics. http://www.w3.org/TR/owl-semantics/direct.html.

[81] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. http://www.w3.org/TR/2006/CR-rdf-sparql-query-20060406/, Apr. 2006.

[82] A. Riazanov and A. Voronkov. The design and implementation of vampire. *AI Communications*, 15(2-3):91–110, 2002.

[83] A. W. Roscoe. *Theory and Practice of Concurrency*. International Series in Computer Science. Prentice-Hall, 1997.

[84] M. Saaltink. The Z/EVES system. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *ZUM'97: Z Formal Specification Notation*, volume 1212 of *Lect. Notes in Comput. Sci.*, pages 72–85. Springer-Verlag, 1997.

[85] M. Saaltink. The Z/EVES 2.0 User's Guide. Technical Report TR-99-5493-06a, ORA Canada, One Nicholas Street, Suite 1208 - Ottawa, Ontario K1N 7B7 - CANADA, Oct. 1999.

[86] A. Seaborne. RDQL - A Query Language for RDF, Jan. 2004. http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/.

[87] M. Sintek and S. Decker. TRIPLE—A query, inference, and transformation language for the semantic web. In I. Horrocks and J. Hendler, editors, *The Semantic Web — ISWC 2002. Proceedings of the First International Semantic Web Conference*, volume 2348 of *Lect. Notes in Comput. Sci.*, pages 364–378, Sardinia, Italy, June 2002. Springer-Verlag.

[88] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1988.

[89] J. M. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice-Hall, 2nd edition, 1992.

[90] J. Sun, Y. F. Li, H. Wang, and J. Sun. 'Visualizing and Simulating Semantic Web Services Ontologies. In *Proc. of 7th International Conference on Formal Engineering Methods (ICFEM'05)*, pages 439–445, Manchester, UK, Nov. 2005. LNCS, Springer-Verlag.

[91] J. Sun, H. Zhang, Y. F. Li, and H. Wang. Formal Semantics and Verification for Feature Modeling. In *Proc. of 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*. IEEE Press, June 2005.

[92] T. Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, Apr. 1997.

[93] C. K. Y. Tan. *Belief Augmented Frames*. PhD thesis, National University of Singapore, 2003.

[94] A. Tarlecki. Moving between logical systems. In M. Haveraaen, O. Owe, and O.-J. Dahl, editors, *Recent Trends in Data Type Specifications. 11th Workshop on Specification of Abstract Data Types*, volume 1130 of *Lect. Notes in Comput. Sci.*, pages 478–502. Springer-Verlag, 1996.

[95] The OWL Services Coalition. OWL-S: Semantic Markup for Web Services. http://www.daml.org/services/owl-s/, 2004.

[96] Tim Berners-Lee. cwm - a general purpose data processor for the semantic web. http://www.w3.org/2000/10/swap/doc/cwm, 2004.

[97] D. Tsarkov and I. Horrocks. DL reasoner vs. first-order prover. In *Proc. of the 2003 Description Logic Workshop (DL 2003)*, volume 81 of *CEUR (http://ceur-ws.org/)*, pages 152–159, 2003.

[98] D. Tsarkov and I. Horrocks. Efficient reasoning with range and domain constraints. In *Proc. of the 2004 Description Logic Workshop (DL 2004)*, pages 41–50, 2004.

[99] UDDI. *Universal Description, Discovery, and Integration of Business for the Web*, October 2001. http://www.uddi.org.

[100] F. van Harmelen and D. Fensel. Formal methods in knowledge engineering. *The Knowledge Engineering Review*, 10(4):345–360, 1995.

[101] F. van Harmelen, P. F. Patel-Schneider, and I. H. (editors). Reference description of the DAML+OIL ontology markup language. Contributors: T. Berners-Lee, D. Brickley, D. Connolly, M. Dean, S. Decker, P. Hayes, J. Heflin, J. Hendler, O. Lassila, D. McGuinness, L. A. Stein, et. al., March, 2001.

[102] H. Wang. *Semantic Web and Formal Design Methods*. PhD thesis, National University of Sinagpore, 2004.

[103] H. Wang, J. S. Dong, J. Sun, and Y. F. Li. TCOZ Approach to OWL-S Process Model Design. In *Proc. of The 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'05)*, Taipei, Taiwan, July 2005.

[104] H. Wang, Y. F. Li, J. Sun, and H. Zhang. Verify Feature Models using Protégé-OWL. In *Fourteenth International Conference on World Wide Web (WWW'05)*, pages 1038–1039, Chiba, Japan, May 2004.

[105] H. Wang, Y. F. Li, J. Sun, H. Zhang, and J. Pan. A Semantic Web Approach to Feature Modeling and Verification. In *1st Workshop on Semantic Web Enabled Software Engineering (SWESE'05)*, Galway, Ireland, Nov 2005. LNCS, Springer-Verlag. accepted.

[106] J. Woodcock and S. Brien. $\mathcal{W}$: A Logic for Z. In *Proceedings of Sixth Annual Z-User Meeting*, University of York, Dec 1991.

[107] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall International, 1996.

[108] World Wide Web Consortium (W3C). Extensible Markup Language (XML). http://www.w3.org/XML.

[109] World Wide Web Consortium (W3C). XSL Transformations (XSLT) Version 1.0. http://www.w3.org/TR/xslt, 1999.

[110] World Wide Web Consortium (W3C). *Simple Object Access Protocol (SOAP) 1.1*, 2000. http://www.w3c.org/TR/SOAP.

Appendix A

Glossary of Z Notation

This appendix presents a glossary of the Z notation used in this thesis. The glossary is based on the glossary of Z notation presented in Hayes [39] with modifications to reflect more closely the more recent Z notation of Spivey [89].

Mathematical Notation

A.1 Definitions and Declarations

Let x, x_k be identifiers and let T, T_k be non-empty, set-valued expressions.

$LHS == RHS$	Definition of LHS as syntactically equivalent to RHS.
$LHS[X_1, X_2, \ldots, X_n] == RHS$	Generic definition of LHS, where $X_1, X_2, \ldots, X_n$ are variables denoting formal parameter sets.
$x : T$	A declaration, $x : T$, introduces a new variable x of type T.
$x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n$	List of declarations.
$x_1, x_2, \ldots, x_n : T$	$\widehat{=}\ x_1 : T;\ x_2 : T;\ \ldots;\ x_n : T$

$[X_1, X_2, \ldots, X_n]$	Introduction of free types named $X_1, X_2, \ldots, X_n$.

A.2 Logic

Let P, Q be predicates and let D be a declaration or a list of declarations.

$true, false$	Logical constants.
$\neg\, P$	Negation: "not P".
$P \land Q$	Conjunction: "P and Q".
$P \lor Q$	Disjunction: "P or Q or both".
$P \Rightarrow Q$	$\widehat{=}\, (\neg\, P) \lor Q$ Implication: "P implies Q" or "if P then Q".
$P \Leftrightarrow Q$	$\widehat{=}\, (P \Rightarrow Q) \land (Q \Rightarrow P)$ Equivalence: "P is logically equivalent to Q".
$\forall\, x : T \bullet P$	Universal quantification: "for all x of type T, P holds".
$\exists\, x : T \bullet P$	Existential quantification: "there exists an x of type T such that P holds".
$\exists_1\, x : T \bullet P$	Unique existence: "there exists a unique x of type T such that P holds".
$\forall\, x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n \bullet P$	"For all x_1 of type T_1, x_2 of type T_2, …, and x_n of type T_n, P holds."
$\exists\, x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n \bullet P$	Similar to $\forall$.
$\exists_1\, x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n \bullet P$	Similar to $\forall$.
$\forall\, D \mid P \bullet Q$	$\Leftrightarrow \forall\, D \bullet P \Rightarrow Q$
$\exists\, D \mid P \bullet Q$	$\Leftrightarrow \exists\, D \bullet P \land Q$
$t_1 = t_2$	Equality between terms.
$t_1 \neq t_2$	$\Leftrightarrow \neg\, (t_1 = t_2)$

A.3 Sets

Let X be a set; S and T be subsets of X; t, t_k terms; P a predicate; and D declarations.

$t \in S$	Set membership: "t is a member of S".
$t \notin S$	$\Leftrightarrow \neg\, (t \in S)$
$S \subseteq T$	$\Leftrightarrow (\forall\, x : S \bullet x \in T)$ Set inclusion.
$S \subset T$	$\Leftrightarrow S \subseteq T \land S \neq T$ Strict set inclusion.
$\emptyset$	The empty set.
$\{t_1, t_2, \ldots, t_n\}$	The set containing the values of terms $t_1, t_2, \ldots, t_n$.
$\{x : T \mid P\}$	The set containing exactly those x of type T for which P holds.
$(t_1, t_2, \ldots, t_n)$	Ordered n-tuple of $t_1, t_2, \ldots, t_n$.
$T_1 \times T_2 \times \ldots \times T_n$	Cartesian product: the set of all n-tuples such that the kth component is of type T_k.
$\mathit{first}(t_1, t_2, \ldots, t_n)$	$\widehat{=}\ t_1$ Similarly, $\mathit{second}(t_1, t_2, \ldots, t_n) \widehat{=} t_2$, etc.
$\{x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n \mid P\}$	The set of all n-tuples $(x_1, x_2, \ldots, x_n)$ with each x_k of type T_k such that P holds.
$\{D \mid P \bullet t\}$	The set of values of the term t for the variables declared in D ranging over all values for which P holds.
$\{D \bullet t\}$	$\widehat{=} \{D \mid \mathit{true} \bullet t\}$
$\mathbb{P}\, S$	Powerset: the set of all subsets of S.
$\mathbb{P}_1\, S$	$\widehat{=}\ \mathbb{P}\, S \setminus \{\emptyset\}$ The set of all non-empty subsets of S.

$\mathbb{F}\, S$	$\widehat{=} \{ T : \mathbb{P}\, S \mid T \text{ is finite} \}$ Set of finite subsets of S.
$\mathbb{F}_1\, S$	$\widehat{=} \mathbb{F}\, S \setminus \{\emptyset\}$ Set of finite non-empty subsets of S.
$S \cap T$	$\widehat{=} \{x : X \mid x \in S \land x \in T\}$ Set intersection.
$S \cup T$	$\widehat{=} \{x : X \mid x \in S \lor x \in T\}$ Set union.
$S \setminus T$	$\widehat{=} \{x : X \mid x \in S \land x \notin T\}$ Set difference.
$\bigcap SS$	$\widehat{=} \{x : X \mid (\forall\, S : SS \bullet x \in S)\}$ Intersection of a set of sets; SS is a set containing as its members subsets of X, i.e. $SS : \mathbb{P}(\mathbb{P}\, X)$.
$\bigcup SS$	$\widehat{=} \{x : X \mid (\exists\, S : SS \bullet x \in S)\}$ Union of a set of sets; $SS : \mathbb{P}(\mathbb{P}\, X)$.
$\#S$	Size (number of distinct members) of a finite set.

A.4 Numbers

$\mathbb{R}$	The set of real numbers.
$\mathbb{Z}$	The set of integers (positive, zero and negative).
$\mathbb{N}$	$\widehat{=} \{n : \mathbb{Z} \mid n \geq 0\}$ The set of natural numbers (non-negative integers).
$\mathbb{N}_1$	$\widehat{=} \mathbb{N} \setminus \{0\}$ The set of strictly positive natural numbers.
$m \mathinner{..} n$	$\widehat{=} \{k : \mathbb{Z} \mid m \leq k \land k \leq n\}$ The set of integers between m and n inclusive.
$\min S$	Minimum of a set; for $S : \mathbb{P}_1\, \mathbb{Z}$, $\min S \in S \land (\forall\, x : S \bullet x \geq \min S)$.
$\max S$	Maximum of a set; for $S : \mathbb{P}_1\, \mathbb{Z}$, $\max S \in S \land (\forall\, x : S \bullet x \leq \max S)$.

A.5 Relations

A binary relation is modelled by a set of ordered pairs hence operators defined for sets can be used on relations. Let X, Y, and Z be sets; $x : X$; $y : Y$; S be a subset of X; T be a subset of Y; and R a relation between X and Y.

$X \leftrightarrow Y$
$\widehat{=} \mathbb{P}(X \times Y)$
The set of relations between X and Y.

$x \mathrel{\underline{R}} y$
$\widehat{=} (x, y) \in R$
x is related by R to y.

$x \mapsto y$
$\widehat{=} (x, y)$

$\{x_1 \mapsto y_1, x_2 \mapsto y_2, \ldots, x_n \mapsto y_n\}$
$\widehat{=} \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$
The relation relating x_1 to y_1, x_2 to y_2, ..., and x_n to y_n.

$\operatorname{dom} R$
$\widehat{=} \{x : X \mid (\exists\, y : Y \bullet x \mathrel{\underline{R}} y)\}$
The domain of a relation: the set of x components that are related to some y.

$\operatorname{ran} R$
$\widehat{=} \{y : Y \mid (\exists\, x : X \bullet x \mathrel{\underline{R}} y)\}$
The range of a relation: the set of y components that some x is related to.

$R_1 \fatsemi R_2$
$\widehat{=} \{x : X;\ z : Z \mid (\exists\, y : Y \bullet x\, R_1\, y \land y\, R_2\, z)\}$
Forward relational composition; $R_1 : X \leftrightarrow Y$; $R_2 : Y \leftrightarrow Z$.

$R_1 \circ R_2$
$\widehat{=} R_2 \fatsemi R_1$
Relational composition. This form is primarily used when R_1 and R_2 are functions.

$R^{\sim}$
$\widehat{=} \{y : Y;\ x : X \mid x \mathrel{\underline{R}} y\}$
Transpose of a relation R.

$\operatorname{id} S$
$\widehat{=} \{x : S \bullet x \mapsto x\}$
Identity function on the set S.

R^k
The homogeneous relation R composed with itself k times: given $R : X \leftrightarrow X$,
$R^0 = \operatorname{id} X$ and $R^{k+1} = R^k \fatsemi R$.

R^+ $\quad \widehat{=} \bigcup\{n : \mathbb{N}_1 \bullet R^n\}$
$= \bigcap\{Q : X \leftrightarrow X \mid R \subseteq Q \land Q \fatsemi Q \subseteq Q\}$
Transitive closure.

R^* $\quad \widehat{=} \bigcup\{n : \mathbb{N} \bullet R^n\}$
$= \bigcap\{Q : X \leftrightarrow X \mid \mathrm{id}\, X \subseteq Q \land R \subseteq Q \land Q \fatsemi Q \subseteq Q\}$
Reflexive transitive closure.

$R(\!| S |\!)$ $\quad \widehat{=} \{y : Y \mid (\exists x : S \bullet x \mathrel{\underline{R}} y)\}$
Image of the set S through the relation R.

$S \lhd R$ $\quad \widehat{=} \{x : X;\ y : Y \mid x \in S \land x \mathrel{\underline{R}} y\}$
Domain restriction: the relation R with its domain restricted to the set S.

$S \lhd\!\!\!\!- R$ $\quad \widehat{=} (X \setminus S) \lhd R$
Domain subtraction: the relation R with the elements of S removed from its domain.

$R \rhd T$ $\quad \widehat{=} \{x : X;\ y : Y \mid x \mathrel{\underline{R}} y \land y \in T\}$
Range restriction to T.

$R \rhd\!\!\!\!- T$ $\quad \widehat{=} R \rhd (Y \setminus T)$
Range subtraction of T.

$R_1 \oplus R_2$ $\quad \widehat{=} (\mathrm{dom}\, R_2 \lhd\!\!\!\!- R_1) \cup R_2$
Overriding; $R_1, R_2 : X \leftrightarrow Y$.

A.6 Functions

A function is a relation with the property that each member of its domain is associated with a unique member of its range. As functions are relations, all the operators defined above for relations also apply to functions. Let X and Y be sets, and T be a subset of X (i.e. $T : \mathbb{P}\, X$).

$f\ t$ $\quad$ The function f applied to t.

$X \pfun Y$ $\quad \widehat{=} \{f : X \leftrightarrow Y \mid (\forall x : \mathrm{dom}\, f \bullet (\exists_1 y : Y \bullet x \mathrel{\underline{f}} y))\}$
The set of partial functions from X to Y.

$X \to Y$ $\quad \widehat{=} \{f : X \pfun Y \mid \dom f = X\}$
The set of total functions from X to Y.

$X \rightarrowtail Y$ $\quad \widehat{=} \{f : X \pfun Y \mid (\forall y : \ran f \bullet (\exists_1 x : X \bullet x \mathrel{f} y))\}$
The set of partial one-to-one functions (partial injections) from X to Y.

$X \rightarrowtail Y$ $\quad \widehat{=} \{f : X \rightarrowtail Y \mid \dom f = X\}$
The set of total one-to-one functions (total injections) from X to Y.

$X \twoheadrightarrow Y$ $\quad \widehat{=} \{f : X \pfun Y \mid \ran f = Y\}$
The set of partial onto functions (partial surjections) from X to Y.

$X \to\!\!\!\!\!\twoheadrightarrow Y$ $\quad \widehat{=} (X \twoheadrightarrow Y) \cap (X \to Y)$
The set of total onto functions (total surjections) from X to Y.

$X \rightarrowtail\!\!\!\!\!\twoheadrightarrow Y$ $\quad \widehat{=} (X \to\!\!\!\!\!\twoheadrightarrow Y) \cap (X \rightarrowtail Y)$
The set of total one-to-one onto functions (total bijections) from X to Y.

$X \fpfun Y$ $\quad \widehat{=} \{f : X \pfun Y \mid f \in \mathbb{F}(X \times Y)\}$
The set of finite partial functions from X to Y.

$X \rightarrowtail\!\!\!\!\!\fpfun Y$ $\quad \widehat{=} \{f : X \rightarrowtail Y \mid f \in \mathbb{F}(X \times Y)\}$
The set of finite partial one-to-one functions from X to Y.

$(\lambda x : X \mid P \bullet t)$ $\quad \widehat{=} \{x : X \mid P \bullet x \mapsto t\}$
Lambda-abstraction: the function that, given an argument x of type X such that P holds, gives a result which is the value of the term t.

$(\lambda x_1 : T_1;\ \ldots;\ x_n : T_n \mid P \bullet t)$
$\quad \widehat{=} \{x_1 : T_1;\ \ldots;\ x_n : T_n \mid P \bullet (x_1, \ldots, x_n) \mapsto t\}$

$\mathsf{disjoint}\ [I, X]$ $\quad \widehat{=} \{S : I \pfun \mathbb{P} X \mid \forall i, j : \dom S \bullet i \neq j \Rightarrow S(i) \cap S(j) = \emptyset\}$
Pairwise disjoint; where I is a set and S an indexed family of subsets of X (i.e. $S : I \pfun \mathbb{P} X$).

$S \mathrel{\underline{\mathsf{partition}}} T$ $\quad \widehat{=} S \in \mathsf{disjoint} \land \bigcup \ran S = T$

A.7 Sequences

Let X be a set; A and B be sequences with elements taken from X; and $a_1, \ldots, a_n$ terms of type X.

$\operatorname{seq} X$	$\widehat{=} \{A : \mathbb{N}_1 \pfun X \mid (\exists\, n : \mathbb{N} \bullet \operatorname{dom} A = 1..n)\}$ The set of finite sequences whose elements are drawn from X.
$\operatorname{seq}_\infty X$	$\widehat{=} \{A : \mathbb{N}_1 \pfun X \mid A \in \operatorname{seq} X \lor \operatorname{dom} A = \mathbb{N}_1\}$ The set of finite and infinite sequences whose elements are drawn from X.
$\#A$	The length of a finite sequence A. (This is just '#' on the set representing the sequence.)
$\langle\rangle$	$\widehat{=} \{\}$ The empty sequence.
$\operatorname{seq}_1 X$	$\widehat{=} \{s : \operatorname{seq} X \mid s \neq \langle\rangle\}$ The set of non-empty finite sequences.
$\langle a_1, \ldots, a_n\rangle$	$= \{1 \mapsto a_1, \ldots, n \mapsto a_n\}$
$\langle a_1, \ldots, a_n\rangle \frown \langle b_1, \ldots, b_m\rangle$	$= \langle a_1, \ldots, a_n, b_1, \ldots, b_m\rangle$ Concatenation. $\langle\rangle \frown A = A \frown \langle\rangle = A$.
head A	The first element of a non-empty sequence: $A \neq \langle\rangle \Rightarrow \mathit{head}\ A = A(1)$.
tail A	All but the head of a non-empty sequence: $\mathit{tail}\ (\langle x\rangle \frown A) = A$.
last A	The final element of a non-empty finite sequence: $A \neq \langle\rangle \Rightarrow \mathit{last}\ A = A(\#A)$.
front A	All but the last of a non-empty finite sequence: $\mathit{front}\ (A \frown \langle x\rangle) = A$.
$\mathit{rev}\ \langle a_1, a_2, \ldots, a_n\rangle$	$= \langle a_n, \ldots, a_2, a_1\rangle$ Reverse of a finite sequence; $\mathit{rev}\ \langle\rangle = \langle\rangle$.

$\frown / AA$	$= AA(1) \frown \ldots \frown AA(\#AA)$ Distributed concatenation; where $AA : \mathrm{seq}(\mathrm{seq}(X))$. $\frown/\langle\rangle = \langle\rangle$.
$A \subseteq B$	$\Leftrightarrow \exists\, C : \mathrm{seq}_\infty X \bullet A \frown C = B$ A is a prefix of B. (This is just '$\subseteq$' on the sets representing the sequences.)
$squash\ f$	Convert a finite function, $f : \mathbb{N} \pfun X$, into a sequence by squashing its domain. That is, $squash\ \{\} = \langle\rangle$, and if $f \neq \{\}$ then $squash\ f = \langle f(i)\rangle \frown squash\ (\{i\} \ndres f)$, where $i = \min(\mathrm{dom}\, f)$. For example, $squash\ \{2 \mapsto A, 27 \mapsto C, 4 \mapsto B\} = \langle A, B, C\rangle$.
$A \upharpoonright T$	$\mathrel{\widehat=}\ squash\ (A \rhd T)$ Restrict the range of the sequence A to the set T.

A.8 Bags

$\mathrm{bag}\, X$	$\mathrel{\widehat=} X \pfun \mathbb{N}_1$ The set of bags whose elements are drawn from X. A bag is represented by a function that maps each element in the bag onto its frequency of occurrence in the bag.
$[\![\,]\!]$	The empty bag $\emptyset$.
$[\![x_1, x_2, \ldots, x_n]\!]$	The bag containing $x_1, x_2, \ldots, x_n$, each with the frequency that it occurs in the list.
$items\ s$	$\mathrel{\widehat=} \{x : \mathrm{ran}\, s \bullet x \mapsto \#\{i : \mathrm{dom}\, s \mid s(i) = x\}\}$ The bag of items contained in the sequence s.

A.9 Axiomatic Definitions

Let D be a list of declarations and P a predicate.

The following axiomatic definition introduces the variables in D with the types as declared in D. These variables must satisfy the predicate P. The scope of the variables

is the whole specification.

$$D \\ \hline P$$

A.10 Generic Definitions

Let D be a list of declarations, P a predicate and $X_1, X_2, \ldots X_n$ variables.

The following generic definition is similar to an axiomatic definition, except that the variables introduced are generic over the sets $X_1, X_2, \ldots X_n$.

$$[X_1, X_2, \ldots X_n] \\ D \\ \hline P$$

The declared variables must be uniquely defined by the predicate P.

Schema Notation

A.11 Schema Definition

A schema groups together a set of declarations of variables and a predicate relating the variables. If the predicate is omitted it is taken to be true, i.e. the variables are not further restricted. There are two ways of writing schemas: vertically, for example,

$$
\begin{array}{|l}
\hline
S \\
\hline
x : \mathbb{N} \\
y : \operatorname{seq} \mathbb{N} \\
\hline
x \leq \# y \\
\hline
\end{array}
$$

and horizontally, for the same example,

$$S \mathrel{\widehat=} [x : \mathbb{N};\ y : \operatorname{seq} \mathbb{N} \mid x \leq \# y]$$

Schemas can be used in signatures after $\forall$, λ, $\{...\}$, etc.:

$$(\forall S \bullet y \neq \langle\rangle) \Leftrightarrow (\forall x : \mathbb{N};\ y : \operatorname{seq} \mathbb{N} \mid x \leq \# y \bullet y \neq \langle\rangle)$$

$\{S\}$ Stands for the set of objects described by schema S. In declarations $w : S$ is usually written as an abbreviation for $w : \{S\}$.

A.12 Schema Operators

Let S be defined as above and $w : S$.

$w.x$ $\mathrel{\widehat=} (\lambda S \bullet x)(w)$
Projection functions: the component names of a schema may be used as projection (or selector) functions, e.g. $w.x$ is w's x component and $w.y$ is its y component; of course, the predicate '$w.x \leq \# w.y$' holds.

θS — The (unordered) tuple formed from a schema's variables, e.g. θS contains the named components x and y.

Compatibility — Two schemas are compatible if the declared sets of each variable common to the declaration parts of the two schemas are equal. In addition, any global variables referenced in predicate part of one of the schemas must not have the same name as a variable declared in the other schema; this restriction is to avoid global variables being *captured* by the declarations.

Inclusion — A schema S may be included within the declarations of a schema T, in which case the declarations of S are merged with the other declarations of T (variables declared in both S and T must have the same declared sets) and the predicates of S and T are conjoined. For example,

$$
\begin{array}{|l}
T \\
\hline
S \\
z : \mathbb{N} \\
\hline
z < x \\
\hline
\end{array}
$$

is equivalent to

$$
\begin{array}{|l}
T \\
\hline
x, z : \mathbb{N} \\
y : \operatorname{seq} \mathbb{N} \\
\hline
x \leq \# y \land z < x \\
\hline
\end{array}
$$

The included schema (S) may not refer to global variables that have the same name as one of the declared variables of the including schema (T).

Decoration — Decoration with subscript, superscript, prime, etc: systematic renaming of the variables declared in the schema. For example, S' is
$[x' : \mathbb{N};\ y' : \operatorname{seq} \mathbb{N} \mid x' \leq \# y']$.

$\neg\, S$ — The schema S with its predicate part negated. For example, $\neg\, S$ is $[x : \mathbb{N};\ y : \operatorname{seq} \mathbb{N} \mid \neg\, (x \leq \# y)]$.

$S \land T$ — The schema formed from schemas S and T by merging their declarations and conjoining (and-ing) their predicates. The

two schemas must be compatible (see above).
Given $T \mathrel{\widehat=} [x : \mathbb{N};\ z : \mathbb{P}\,\mathbb{N} \mid x \in z]$, $S \land T$ is

$$\begin{array}{|l} S \land T \\ \hline x : \mathbb{N} \\ y : \operatorname{seq} \mathbb{N} \\ z : \mathbb{P}\,\mathbb{N} \\ \hline x \leq \# y \land x \in z \end{array}$$

$S \lor T$ The schema formed from schemas S and T by merging their declarations and disjoining (or-ing) their predicates. The two schemas must be compatible (see above). For example, $S \lor T$ is

$$\begin{array}{|l} S \lor T \\ \hline x : \mathbb{N} \\ y : \operatorname{seq} \mathbb{N} \\ z : \mathbb{P}\,\mathbb{N} \\ \hline x \leq \# y \lor x \in z \end{array}$$

$S \Rightarrow T$ The schema formed from schemas S and T by merging their declarations and taking ‘ pred $S \Rightarrow$ pred T’ as the predicate. The two schemas must be compatible (see above). For example, $S \Rightarrow T$ is

$$\begin{array}{|l} S \Rightarrow T \\ \hline x : \mathbb{N} \\ y : \operatorname{seq} \mathbb{N} \\ z : \mathbb{P}\,\mathbb{N} \\ \hline x \leq \# y \Rightarrow x \in z \end{array}$$

$S \Leftrightarrow T$ The schema formed from schemas S and T by merging their declarations and taking ‘ pred $S \Leftrightarrow$ pred T’ as the predicate. The two schemas must be compatible (see above). For example, $S \Leftrightarrow T$ is

$$\begin{array}{|l} S \Leftrightarrow T \\ \hline x : \mathbb{N} \\ y : \operatorname{seq} \mathbb{N} \\ z : \mathbb{P}\,\mathbb{N} \\ \hline x \leq \# y \Leftrightarrow x \in z \end{array}$$

$S \setminus (v_1, v_2, \ldots, v_n)$

Hiding: the schema S with variables $v_1, v_2, \ldots, v_n$ hidden – the variables listed are removed from the declarations and are existentially quantified in the predicate. The parantheses may be omitted when only one variable is hidden.

$S \upharpoonright (v_1, v_2, \ldots, v_n)$

Projection: The schema S with any variables that do not occur in the list $v_1, v_2, \ldots, v_n$ hidden – the variables are removed from the declarations and are existentially qualified in the predicate. For example, $(S \land T) \upharpoonright (x, y)$ is

$$
\begin{array}{|l}
\hline
(S \land T) \upharpoonright (x, y) \\
x : \mathbb{N} \\
y : \operatorname{seq} \mathbb{N} \\
\hline
(\exists\, z : \mathbb{P}\, \mathbb{N} \bullet \\
\quad x \leq \# y \land x \in z) \\
\hline
\end{array}
$$

The list of variables may be replaced by a schema; the variables declared in the schema are used for projection.

$\exists\, D \bullet S$

Existential quantification of a schema.
The variables declared in the schema S that also appear in the declarations D are removed from the declarations of S. The predicate of S is existentially quantified over D. For example, $\exists\, x : \mathbb{N} \bullet S$ is the following schema.

$$
\begin{array}{|l}
\hline
\exists\, x : \mathbb{N} \bullet S \\
y : \operatorname{seq} \mathbb{N} \\
\hline
\exists\, x : \mathbb{N} \bullet \\
\quad x \leq \# y \\
\hline
\end{array}
$$

The declarations may include schemas. For example,

$$
\begin{array}{|l}
\hline
\exists\, S \bullet T \\
z : \mathbb{N} \\
\hline
\exists\, S \bullet \\
\quad x \leq \# y \land z < x \\
\hline
\end{array}
$$

$\forall D \bullet S$ — Universal quantification of a schema.
The variables declared in the schema S that also appear in the declarations D are removed from the declarations of S. The predicate of S is universally quantified over D. For example, $\forall x : \mathbb{N} \bullet S$ is the following schema.

$$\begin{array}{|l} \forall x : \mathbb{N} \bullet S \\ \hline y : \operatorname{seq} \mathbb{N} \\ \hline \forall x : \mathbb{N} \bullet \\ \quad x \leq \# y \\ \hline \end{array}$$

The declarations may include schemas. For example,

$$\begin{array}{|l} \forall S \bullet T \\ \hline z : \mathbb{N} \\ \hline \forall S \bullet \\ \quad x \leq \# y \land z < x \\ \hline \end{array}$$

A.13 Operation Schemas

The following conventions are used for variable names in those schemas which represent operations, that is, which are written as descriptions of operations on some state,

undashed state before the operation,

dashed state after the operation,

ending in "?" inputs to (arguments for) the operation, and

ending in "!" outputs from (results of) the operation.

The basename of a name is the name with all decorations removed.

ΔS — $\widehat{=} S \land S'$
Change of state schema: this is a default definition for ΔS. In some specifications it is useful to have additional constraints

on the change of state schema. In these cases ΔS can be explicitly defined.

ΞS $\widehat{=} [\Delta S \mid \theta S' = \theta S]$
No change of state schema.

A.14 Operation Schema Operators

pre S Precondition: the after-state components (dashed) and the outputs (ending in "!") are hidden, e.g. given,

S
$x?, s, s', y! : \mathbb{N}$

$s' = s - x? \land y! = s'$

pre S is,

pre S
$x?, s : \mathbb{N}$

$\exists\, s', y! : \mathbb{N} \bullet$
$\quad s' = s - x? \land y! = s'$

$S \fatsemi T$ Schema composition: if we consider an intermediate state that is both the final state of the operation S and the initial state of the operation T then the composition of S and T is the operation which relates the initial state of S to the final state of T through the intermediate state. To form the composition of S and T we take the pairs of after-state components of S and before-state components of T that have the same basename, rename each pair to a new variable, take the conjunction of the resulting schemas, and hide the new variables. For example, $S \fatsemi T$ is,

$S \fatsemi T$
$x?, s, s', y! : \mathbb{N}$

$(\exists\, ss : \mathbb{N} \bullet$
$\quad ss = s - x? \land y! = ss$
$\quad \land\ ss \leq x? \land s' = ss + x?)$

Appendix B

Z Semantics for DAML+OIL

In this appendix, we present the complete Z semantics for the ontology language DAML+OIL. As DAML+OIL emphasizes on the description of abstract concepts, discussion of concrete (data type-related) properties are not considered in the Z semantics.

B.1 Basic Concepts

Everything in DAML+OIL (and RDF) is regarded a Web resource, hence, we make *Resource* a given type, which is not interpreted.

In DAML+OIL, resources are grouped under various *classes*, which are related to each other via *properties*. Hence, we model *Class* and *Property* as subsets of *Resource*. Moreover, these two sets are disjoint.

$$[Resource]$$

$$\begin{array}{|l} Class : \mathbb{P}\, Resource \\ Property : \mathbb{P}\, Resource \\ \hline Class \cap Property = \emptyset \end{array}$$

To link the members of a class to itself and the pairs of resources a property maps

back to this property, we define two important auxiliary functions: *instances* and *sub_val*.

$$instances : Class \rightarrow \mathbb{P}\, Resource$$

$$sub\_val : Property \rightarrow (Resource \leftrightarrow Resource)$$

In DAML+OIL, there are two pre-defined special classes: *Thing* and *Nothing*, which is the super class/sub class of all classes, respectively. In other words, the instances of *Thing* is the whole set *Resource* whereas *Nothing* does not hold any instance.

$$Thing, Nothing : Class$$

$$instances(Thing) = Resource$$
$$instances(Nothing) = \emptyset$$

B.2 Class Elements

DAML+OIL defines a number of properties to relate classes. In this section, we present the definitions of their Z counterparts. Note that these properties are translated as Z relations since they are *meta-level* properties.

The Z relations *subClassOf*, *disjointWith*, *sameClassAs* are all binary relations that apply to two classes.

$$subClassOf, disjointWith, sameClassAs : Class \leftrightarrow Class$$

$$\forall\, c_1, c_2 : Class \bullet$$
$$\quad c_1 \ \underline{subClassOf}\ c_2 \Leftrightarrow instances(c_1) \subseteq instances(c_2)$$
$$\forall\, c_1, c_2 : Class \bullet$$
$$\quad c_1 \ \underline{disjointWith}\ c_2 \Leftrightarrow instances(c_1) \cap instances(c_2) = \emptyset$$
$$\forall\, c_1, c_2 : Class \bullet$$
$$\quad c_1 \ \underline{sameClassAs}\ c_2 \Leftrightarrow instances(c_1) = instances(c_2)$$

Besides these binary relations, DAML+OIL also defines a number of *boolean combinations* of classes. These include *intersectionOf*, *unionOf* and *complementOf*. The relation *disjointUnionOf* combines *disjiontWith* and *unionOf*.

$$
\begin{array}{|l}
intersectionOf, unionOf : \operatorname{seq} Class \rightarrow Class \\
\hline
\forall\, cl : \operatorname{seq} Class;\ c : Class \bullet \\
\quad intersectionOf(cl) = c \Leftrightarrow instances(c) = \bigcap \{x : \operatorname{ran} cl \bullet instances(x)\} \\
\forall\, cl : \operatorname{seq} Class;\ c : Class \bullet \\
\quad unionOf(cl) = c \Leftrightarrow instances(c) = \bigcup \{x : \operatorname{ran} cl \bullet instances(x)\}
\end{array}
$$

$$
\begin{array}{|l}
complementOf : Class \leftrightarrow Class \\
\hline
\forall\, c_1, c_2 : Class \bullet \\
\quad c_1 \ \underline{complementOf}\ c_2 \Leftrightarrow Resource \setminus instances(c_1) = instances(c_2)
\end{array}
$$

$$
\begin{array}{|l}
disjointUnionOf : \operatorname{seq} Class \rightarrow Class \\
\hline
\forall\, cl : \operatorname{seq} Class;\ c : Class \bullet disjointUnionOf(cl) = c \Leftrightarrow \\
\quad unionOf(cl) = c \land \\
\quad (\forall\, x, y : cl \mid x.1 \neq y.1 \bullet x.2\ \underline{disjointWith}\ y.2)
\end{array}
$$

B.3 Class Enumeration

The class enumeration relation *oneOf* enumerates all the instances of a class.

$$
\begin{array}{|l}
oneOf : \mathbb{P}\, Resource \rightarrow Class \\
\hline
\forall\, x : \mathbb{P}\, Resource;\ c : Class \bullet oneOf(x) = c \Leftrightarrow x = instances(c)
\end{array}
$$

B.4 Property Restriction

Besides a class denoted by its name and class enumeration introduced above, class expressions include also *property restrictions*.

A *toClass* element defines the class c_2 of all objects for which the values of property p all belong to the class expression c_1.

$$
\begin{array}{|l}
toClass : (Class \times Property) \rightarrow Class \\
\hline
\forall\, c_1, c_2 : Class;\ p : Property \bullet toClass(c_1, p) = c_2 \Leftrightarrow \\
\quad instances(c_2) = \\
\quad\quad \{a : Resource \mid sub\_val(p)(\!|\, \{a\} \,|\!) \subseteq instances(c_1)\}
\end{array}
$$

A *hasValue* element defines the class c of all objects for which the property p has at least one value equal to the named object r or data type value (and perhaps other values as well).

$$
\begin{array}{|l}
hasValue : (Resource \times Property) \rightarrow Class \\
\hline
\forall\, r : Resource;\ p : Property;\ c : Class \bullet hasValue(r, p) = c \Leftrightarrow \\
\quad instances(c) = \\
\quad\quad \{a : Resource \mid r \in sub\_val(p)(\!|\, \{a\} \,|\!)\}
\end{array}
$$

A *hasClass* element defines the class c_2 of all objects for which at least one value of the property p is a member of the class expression or data type c_1.

$$
\begin{array}{|l}
hasClass : (Class \times Property) \rightarrow Class \\
\hline
\forall\, c_1, c_2 : Class;\ p : Property \bullet hasValue(c_1, p) = c_2 \Leftrightarrow \\
\quad instances(c_2) = \\
\quad\quad \{a : Resource \mid sub\_val(p)(\!|\, \{a\} \,|\!) \cap instances(c_1) \neq \emptyset\}
\end{array}
$$

DAML+OIL also defines a number of (qualified) cardinality-related constraints. For example, the *cardinality* relation defines the class c of all objects that have exactly n distinct values for the property p, i.e., a is an instance of c if and only if there are exactly n distinct values mapped to a by p. Other relations are similarly defined.

$$cardinality, minCardinality, maxCardinality : (\mathbb{N} \times Property) \rightarrow Class$$

$$\begin{aligned}
&\forall\, n : \mathbb{N};\ p : Property;\ c : Class \bullet cardinality(n, p) = c \Leftrightarrow \\
&\qquad instances(c) = \{a : Resource \mid \#(sub_val(p)(\!|\ \{a\}\ |\!)) = n\} \\
&\forall\, n : \mathbb{N};\ p : Property;\ c : Class \bullet minCardinality(n, p) = c \Leftrightarrow \\
&\qquad instances(c) = \{a : Resource \mid \#(sub_val(p)(\!|\ \{a\}\ |\!)) \geq n\} \\
&\forall\, n : \mathbb{N};\ p : Property;\ c : Class \bullet maxCardinality(n, p) = c \Leftrightarrow \\
&\qquad instances(c) = \{a : Resource \mid \#(sub_val(p)(\!|\ \{a\}\ |\!)) \leq n\}
\end{aligned}$$

The qualified cardinality constraints are similarly defined, except that the quantified elements must be from a specific class expression.

$$cardinalityQ, minCardinalityQ, maxCardinalityQ : (\mathbb{N} \times Class \times Property) \rightarrow Class$$

$$\begin{aligned}
&\forall\, n : \mathbb{N};\ c_1, c_2 : Class;\ p : Property \bullet cardinalityQ(n, c_1, p) = c_2 \Leftrightarrow \\
&\qquad instances(c_2) = \{a : Resource \mid \#(sub_val(p)(\!|\ \{a\}\ |\!) \cap instances(c_1)) = n\} \\
&\forall\, n : \mathbb{N};\ c_1, c_2 : Class;\ p : Property \bullet minCardinalityQ(n, c_1, p) = c_2 \Leftrightarrow \\
&\qquad instances(c_2) = \{a : Resource \mid \#(sub_val(p)(\!|\ \{a\}\ |\!) \cap instances(c_1)) \geq n\} \\
&\forall\, n : \mathbb{N};\ c_1, c_2 : Class;\ p : Property \bullet maxCardinalityQ(n, c_1, p) = c_2 \Leftrightarrow \\
&\qquad instances(c_2) = \{a : Resource \mid \#(sub_val(p)(\!|\ \{a\}\ |\!) \cap instances(c_1)) \leq n\}
\end{aligned}$$

B.5 Property Elements

In this section, we present the Z semantics of DAML+OIL language constructs for describing properties. These constructs, such as *domain*, *range*, *subPropertyOf*, etc., are translated into Z functions or relations.

The following three relations model the relationship between two properties. They are similar to those defined in Section B.2.

$$subPropertyOf, samePropertyOf, inverseOf : Property \leftrightarrow Property$$

$$\forall p_1, p_2 : Property \bullet$$
$$p_1 \ \underline{subPropertyOf} \ p_2 \Leftrightarrow sub\_val(p_1) \subseteq sub\_val(p_2)$$
$$\forall p_1, p_2 : Property \bullet$$
$$p_1 \ \underline{samePropertyOf} \ p_2 \Leftrightarrow sub\_val(p_1) = sub\_val(p_2)$$
$$\forall p_1, p_2 : Property \bullet$$
$$p_1 \ \underline{inverseOf} \ p_2 \Leftrightarrow sub\_val(p_1) = (sub\_val(p_2))^{\sim}$$

The relations *domain* and *range* maps a property to its domain and range, respectively.

$$domain, range : Property \rightarrow Class$$

$$\forall p : Property;\ c : Class \bullet$$
$$domain(p) = c \Leftrightarrow \mathrm{dom}(sub\_val(p)) \subseteq instances(c)$$
$$\forall p : Property;\ c : Class \bullet$$
$$range(p) = c \Leftrightarrow \mathrm{ran}(sub\_val(p)) \subseteq instances(c)$$

In DAML+OIL, a property can be transitive, unique (functional), or unambiguous (inverse functional). Three properties are defined to model these characteristics.

$$TransitiveProperty, UniqueProperty, UnambiguousProperty : \mathbb{P}\, Property$$

$$\forall p : Property \bullet$$
$$p \in TransitiveProperty \Leftrightarrow (\forall x, y, z : Resource \bullet$$
$$(x, y) \in sub\_val(p) \land (y, z) \in sub\_val(p) \Rightarrow (x, z) \in sub\_val(p))$$
$$\forall p : Property \bullet$$
$$p \in UniqueProperty \Leftrightarrow (\forall x, y, z : Resource \bullet$$
$$(x, y) \in sub\_val(p) \land (x, z) \in sub\_val(p) \Rightarrow y = z)$$
$$\forall p : Property \bullet$$
$$p \in UnambiguousProperty \Leftrightarrow (\forall x, y, z : Resource \bullet$$
$$(x, z) \in sub\_val(p) \land (y, z) \in sub\_val(p) \Rightarrow x = y)$$

B.6 Instances

DAML+OIL defines two properties to relate pairs of instances: *sameIndividualAs* and *differentIndividualFrom*. They are modeled as relations in Z.

$$sameIndividualAs : Resource \leftrightarrow Resource$$
$$differentIndividualFrom : Resource \leftrightarrow Resource$$

Appendix C

Z Semantics for OWL DL

In this chapter, we present the complete Z semantics for the ontology language OWL DL. The presentation in this chapter will be divided into subsections roughly according to [66].

C.1 Basic Concepts

As in the Z semantics for DAML+OIL, we model *Resource* as a given type.

$$[Resource]$$

In OWL, the instances of classes are grouped under one concept called *Individual*, which is modeled as a subset of *Resource*.

$$Individual : \mathbb{P}\, Resource$$

As in DAML+OIL, *Class* and *Property* are similarly defined. Moreover, *Class*, *Property* and *Individual* are mutually disjoint.

$$
\begin{array}{|l}
Class : \mathbb{P}\, Resource \\
Property : \mathbb{P}\, Resource \\
\hline
Individual \cap Class = \emptyset \\
Property \cap Class = \emptyset \\
Property \cap Individual = \emptyset
\end{array}
$$

Every class holds a number of individuals as its members. The function *instances* maps a class to the set of *Individual*s it holds.

$$
\begin{array}{|l}
instances : Class \rightarrow \mathbb{P}\, Individual
\end{array}
$$

As in DAML+OIL, OWL also defines the two special classes, *Thing* and *Nothing*.

$$
\begin{array}{|l}
Thing : Class \\
Nothing : Class \\
\hline
instances(Thing) = Individual \\
instances(Nothing) = \emptyset
\end{array}
$$

In OWL DL, support for data types are more elaborate than in DAML+OIL. Hence, we also tailor the Z semantics towards data types that might appear in the OWL ontologies.

First of all, properties are further divided into 2 broad categories, those relate an individual to another individual and those relate an individual to a value of a particular data type. These two types of properties are called *ObjectProperty* and *DatatypeProperty*, which are disjoint with each other.

$$
\begin{array}{|l}
ObjectProperty : \mathbb{P}\, Property \\
DatatypeProperty : \mathbb{P}\, Property \\
\hline
ObjectProperty \cap DatatypeProperty = \emptyset
\end{array}
$$

Before presenting the definitions of these properties, define how these properties are mapped to the pairs of resources that they relate. Two functions, *sub_val* and *sub_valD*, are defined in Z.

The function *sub_val* is almost identical to that defined in Appendix B, except that the domain is updated to *ObjectProeprty* and *Resource* is replaced by *Individual*.

$$sub\_val : ObjectProperty \rightarrow (Individual \leftrightarrow Individual)$$

The function *sub_valD* caters for the case where an individual is related to a data item by a property. It is defined as a generic definition where the data type is represented by the generic type X and the domain is changed to *DatatypeProperty*.

$$[X]$$
$$sub\_valD : DatatypeProperty \rightarrow (Individual \leftrightarrow X)$$

C.2 Classes

C.2.1 Class Descrpitions

In this section, we will present the class descriptions, the building blocks for constructing OWL classes.

The simplest form of class description is, according to [66], by referring to the name of the class. In Z, a concept must be declared before it is used. Hence, a class is defined by using an axiomatic definition. When it is referred subsequently, its name will be used, such as the class *Individual* when defining *Class* and *Property* in the previous section.

Enumeration

As in DAML+OIL, OWL defines a class property *oneOf* that completely defines a class by enumerating its instances.

$$oneOf : \mathbb{P}\, Individual \rightarrow Class$$

$$\forall\, x : \mathbb{P}\, Individual;\ y : Class \bullet oneOf(x) = y \Leftrightarrow x = instances(y)$$

Property Restrictions

In OWL, a property restriction usually describes an anonymous class by constraining its membership through the use of a property. Two kinds of property restrictions are defined: value constraints and cardinality constraints.

In OWL, the value constraints include three properties, namely *allValuesFrom*, *someValuesFrom* and *hasValue*, which are similar to *toClass*, *hasClass* and *hasValue* defined in Appendix B.4.

Since these properties cater for both abstract and concrete values, we transform them into different Z definitions, as detailed below.

$$allValuesFrom : (Class \times ObjectProperty) \rightarrow Class$$

$$\forall\, c_1 : Class;\ p : ObjectProperty;\ c_2 : Class \bullet allValuesFrom(c_1, p) = c_2 \Leftrightarrow instances(c_2) = \{a : Individual \mid sub\_val(p)(\!|\ \{a\}\ |\!) \subseteq instances(c_1)\}$$

The above axiomatic definition of *allValuesFrom* handles the case where an OWL class is constrained by another class and an object property. The following generic definition *allValuesFromD* handles the case where an OWL class is constrained by a (generic) data type and a data type property.

$$[X]$$

$$allValuesFromD : (\mathbb{P}\, X \times DatatypeProperty) \rightarrow Class$$

$$\forall\, d : \mathbb{P}\, X;\ c : Class;\ p : DatatypeProperty \bullet allValuesFromD(d, p) = c \Leftrightarrow instances(c) = \{a : Individual \mid (sub\_valD(p)(\!|\ \{a\}\ |\!)) \subseteq d\}$$

The treatment of *someValuesFrom* and *hasValue* are similar to that of *allValuesFrom*.

$$someValuesFrom : (Class \times ObjectProperty) \rightarrow Class$$

$$\forall c_1, c_2 : Class;\ p : ObjectProperty \bullet someValuesFrom(c_1, p) = c_2 \Leftrightarrow instances(c_2) = \{a : Individual \mid sub\_val(p)(\!|\ \{a\}\ |\!) \cap instances(c_1) \neq \emptyset\}$$

$$[X]$$

$$someValuesFromD : (\mathbb{P}\, X \times DatatypeProperty) \rightarrow Class$$

$$\forall t : \mathbb{P}\, X;\ p : DatatypeProperty;\ c : Class \bullet someValuesFromD(t, p) = c \Leftrightarrow instances(c) = \{a : Individual \mid sub\_valD(p)(\!|\ \{a\}\ |\!) \cap t \neq \emptyset\}$$

$$hasValue : (Individual \times ObjectProperty) \rightarrow Class$$

$$\forall r : Individual;\ p : ObjectProperty;\ c : Class \bullet hasValue(r, p) = c \Leftrightarrow instances(c) = \{a : Individual \mid r \in sub\_val(p)(\!|\ \{a\}\ |\!)\}$$

$$[X]$$

$$hasValueD : (X \times DatatypeProperty) \rightarrow Class$$

$$\forall r : X;\ p : DatatypeProperty;\ c : Class \bullet hasValueD(r, p) = c \Leftrightarrow instances(c) = \{a : Individual \mid r \in sub\_valD(p)(\!|\ \{a\}\ |\!)\}$$

The cardinality property constraints are updated based on those defined in DAML+OIL. In OWL, qualified cardinality constraints are removed as they can be expressed by using unqualified cardinality constraints and value constraints in conjunction.

To cater for data types, each of the cardinality constraints are also transformed into two versions.

$$maxCardinality : (\mathbb{N} \times ObjectProperty) \rightarrow Class$$

$$\forall c : Class;\ p : ObjectProperty;\ n : \mathbb{N} \bullet maxCardinality(n, p) = c \Leftrightarrow instances(c) = \{x : Individual \mid \#(sub\_val(p)(\!|\ \{x\}\ |\!)) \leq n\}$$

$$[X]$$

$$maxCardinalityD : (\mathbb{N} \times DatatypeProperty) \rightarrow Class$$

$$\forall\, c : Class;\ p : DatatypeProperty;\ n : \mathbb{N} \bullet maxCardinalityD(n, p) = c \Leftrightarrow instances(c) = \{x : Individual \mid \#[X](sub\_valD(p)(\!|\ \{x\}\ |\!)) \leq n\}$$

$$minCardinality : (\mathbb{N} \times ObjectProperty) \rightarrow Class$$

$$\forall\, c : Class;\ p : ObjectProperty;\ n : \mathbb{N} \bullet minCardinality(n, p) = c \Leftrightarrow instances(c) = \{x : Individual \mid \#(sub\_val(p)(\!|\ \{x\}\ |\!)) \geq n\}$$

$$[X]$$

$$minCardinalityD : (\mathbb{N} \times DatatypeProperty) \rightarrow Class$$

$$\forall\, c : Class;\ p : DatatypeProperty;\ n : \mathbb{N} \bullet minCardinalityD(n, p) = c \Leftrightarrow instances(c) = \{x : Individual \mid \#[X](sub\_valD(p)(\!|\ \{x\}\ |\!)) \geq n\}$$

$$cardinality : (\mathbb{N} \times ObjectProperty) \rightarrow Class$$

$$\forall\, c : Class;\ p : ObjectProperty;\ n : \mathbb{N} \bullet cardinality(n, p) = c \Leftrightarrow instances(c) = \{x : Individual \mid \#(sub\_val(p)(\!|\ \{x\}\ |\!)) = n\}$$

$$[X]$$

$$cardinalityD : (\mathbb{N} \times DatatypeProperty) \rightarrow Class$$

$$\forall\, c : Class;\ p : DatatypeProperty;\ n : \mathbb{N} \bullet cardinalityD(n, p) = c \Leftrightarrow instances(c) = \{x : Individual \mid \#[X](sub\_valD(p)(\!|\ \{x\}\ |\!)) = n\}$$

Boolean Combinations

A class description can also be one of the three boolean combinations, namely class intersection, union and complement. The property *disjointUnionOf* defined in DAML+OIL is removed from OWL as its effect can be achieved by using *disjointWith* and *unionOf* in conjunction.

$$intersectionOf : \text{seq}\ Class \rightarrow Class$$

$$\forall\, cl : \text{seq}\ Class;\ c : Class \bullet intersectionOf(cl) = c \Leftrightarrow instances(c) = \bigcap\{x : \text{ran}\ cl \bullet instances(x)\}$$

$$unionOf : \text{seq}\ Class \rightarrow Class$$

$$\forall\, cl : \text{seq}\ Class;\ c : Class \bullet unionOf(cl) = c \Leftrightarrow instances(c) = \bigcup\{x : \text{ran}\ cl \bullet instances(x)\}$$

$$complementOf : Class \leftrightarrow Class$$

$$\forall\, c1, c2 : Class \bullet c1\ \underline{complementOf}\ c2 \Leftrightarrow Individual \setminus instances(c1) = instances(c2)$$

C.2.2 Class Axioms

This section contains three properties that state the inter-class relationship, namely *subClassOf*, *equivalentClass* and *disjointWith*.

The *subClassOf* is identical to that in DAML+OIL, which states that class c_1 is a sub class of c_2 if its instances is a subset of c_2.

$$subClassOf : Class \leftrightarrow Class$$

$$\forall\, c1, c2 : Class \bullet c1\ \underline{subClassOf}\ c2 \Leftrightarrow instances(c1) \subseteq instances(c2)$$

As the name suggests, *eqivalentClass* states the conditions under which two classes are equivalent.

$$equivalentClass : Class \leftrightarrow Class$$

$$\forall\, c1, c2 : Class \bullet c1\ \underline{equivalentClass}\ c2 \Leftrightarrow instances(c1) = instances(c2)$$

Two classes are disjoint with each other if and only if the intersection of their instances is an empty set.

$$disjointWith : Class \leftrightarrow Class$$

$$\forall\, c1, c2 : Class \bullet c1 \ \underline{disjointWith}\ c2 \Leftrightarrow instances(c1) \cap instances(c2) = \emptyset$$

C.3 Properties

C.3.1 RDF Schema Property Constructs

As stated in Section 2.1, RDF Schema can be regarded as the first ontology language. It defines a number of language constructs for describing properties. In this section, we present the transformation of these constructs, namely *subPropertyOf*, *domain* and *range*. In OWL DL, all these three properties can be applied to both object properties and datatype properties.

$$[X]$$

$$subPropertyOf : Property \leftrightarrow Property$$

$$\begin{aligned}\forall\, p_1, p_2 : Property \bullet\ & p_1\ \underline{subPropertyOf}\ p_2 \Leftrightarrow \\ & (p_1 \in ObjectProperty \land p_2 \in ObjectProperty) \Rightarrow sub\_val(p_1) \subseteq sub\_val(p_2) \land \\ & (p_1 \in DatatypeProperty \land p_2 \in DatatypeProperty) \Rightarrow \\ & \quad sub\_valD[X](p_1) \subseteq sub\_valD[X](p_2)\end{aligned}$$

The following two properties return the domain and range of a property respectively.

$$[X]$$

$$domain : Property \rightarrow Class$$

$$\begin{aligned}\forall\, p : Property;\ c : Class \bullet\ & domain(p) = c \Leftrightarrow \\ & p \in ObjectProperty \Rightarrow \mathrm{dom}(sub\_val(p)) \subseteq instances(c) \land \\ & p \in DatatypeProperty \Rightarrow \mathrm{dom}(sub\_valD[X](p)) \subseteq instances(c)\end{aligned}$$

The property *range* defined in RDF Schema returns the range of the property. Since

OWL DL allows this property to be applied to both object and datatype properties, as before, we transform it to two versions in Z, one for each kind of properties.

$$range : ObjectProperty \rightarrow Class$$

$$\forall p : ObjectProperty;\ c : Class \bullet range(p) = c \Leftrightarrow \operatorname{ran}(sub\_val(p)) \subseteq instances(c)$$

$$[X]$$

$$rangeD : DatatypeProperty \rightarrow \mathbb{P}\, X$$

$$\forall p : DatatypeProperty;\ d : \mathbb{P}\, X \bullet rangeD(p) = d \Leftrightarrow \operatorname{ran}(sub\_valD(p)) \subseteq d$$

C.3.2 Relations to Other Properties

A property is equivalent to another property if its property extension is the same as that of the other. This property is also defined for both object and datatype properties.

$$[X]$$

$$equivalentProperty : Property \leftrightarrow Property$$

$$\forall p_1, p_2 : Property \bullet p_1 \ \underline{equivalentProperty}\ p_2 \Leftrightarrow ((p_1 \in ObjectProperty \land p_2 \in ObjectProperty) \Rightarrow sub\_val(p_1) = sub\_val(p_2)) \land ((p_1 \in DatatypeProperty \land p_2 \in DatatypeProperty) \Rightarrow sub\_valD[X](p_1) = sub\_valD[X](p_2))$$

The inverse of a property is another property with their domains and ranges flipped. It is only applicable to object properties as the domain and range of such properties must be of the same type.

$$inverseOf : ObjectProperty \leftrightarrow ObjectProperty$$

$$\forall p_1, p_2 : ObjectProperty \bullet p_1 \ \underline{inverseOf}\ p_2 \Leftrightarrow sub\_val(p_1) = (sub\_val(p_2))^{\sim}$$

C.3.3 Global Cardinality Constraints on Properties

A functional property is a property that can have only one (unique) value in its range for each instance in its domain.

$$
\begin{array}{|l}
[X] \\
\hline
functionalProperty : \mathbb{P}\ Property \\
\hline
\forall\, p : Property \bullet p \in functionalProperty \Leftrightarrow \\
\quad (p \in ObjectProperty \Rightarrow (\forall\, a : \mathrm{dom}(sub\_val(p)) \bullet \\
\quad\quad \#(sub\_val(p)(\!|\ \{a\}\ |\!)) = 1)) \land \\
\quad (p \in DatatypeProperty \Rightarrow (\forall\, a : \mathrm{dom}(sub\_valD[X](p)) \bullet \\
\quad\quad \#(sub\_valD[X](p)(\!|\ \{a\}\ |\!)) = 1)) \\
\hline
\end{array}
$$

An object property can be declared to be inverse-functional. If a property is declared to be inverse-functional, the object of a property statement uniquely determines the subject (some individual).

$$
\begin{array}{|l}
InverseFunctionalProperty : \mathbb{P}\ ObjectProperty \\
\hline
\forall\, p : ObjectProperty \bullet p \in InverseFunctionalProperty \Leftrightarrow \\
\quad (\forall\, a, b, c : Individual \mid (a, c) \in sub\_val(p) \land (b, c) \in sub\_val(p) \bullet a = b)
\end{array}
$$

C.3.4 Logical Characteristics of Properties

An object property can also be declared as being transitive. Formally speaking, if pairs of individuals (a, b) and (b, c) are instances (members of the property extension) of property p, then we can infer that (a, c) is also an instance of p.

$$
\begin{array}{|l}
TransitiveProperty : \mathbb{P}\ ObjectProperty \\
\hline
\forall\, p : ObjectProperty \bullet p \in TransitiveProperty \Leftrightarrow \\
\quad (\forall\, a, b, c : Individual \bullet (a, b) \in sub\_val(p) \land (b, c) \in sub\_val(p) \Rightarrow \\
\quad\quad (a, c) \in sub\_val(p))
\end{array}
$$

A symmetric property is a property for which holds that if the pair (x, y) is an instance of a property p, then the pair (y, x) is also an instance of p. As the same reason above, *SymmetricProperty* is a sub set of *ObjectProperty*.

$$SymmetricProperty : \mathbb{P}\ ObjectProperty$$

$$\forall\, p : ObjectProperty \bullet p \in SymmetricProperty \Leftrightarrow (\forall\, a, b : Individual \bullet (a, b) \in sub\_val(p) \Rightarrow (b, a) \in sub\_val(p))$$

C.4 Individuals

This section describes the properties that OWL defines for individuals.

C.4.1 Individual Identity

OWL provides three properties for stating the identity of an individual.

In OWL DL, the *sameAs* property states that two individuals are same as each other.

$$sameAs : Individual \leftrightarrow Individual$$

On the contrary to *sameAs*, *differentFrom* states that two individuals are actually different.

$$differentFrom : Individual \leftrightarrow Individual$$

The property *AllDifferent* is defined in OWL for convenience to state the pairwise disjointness among a list of individuals.

$$AllDifferent : \mathbb{P}(\mathrm{seq}\ Individual)$$

$$\forall\, ins : \mathrm{seq}\ Individual \bullet ins \in AllDifferent \Leftrightarrow (\forall\, x, y : ins \mid x.1 \neq y.1 \bullet x.2\ \underline{differentFrom}\ y.2)$$