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.