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.

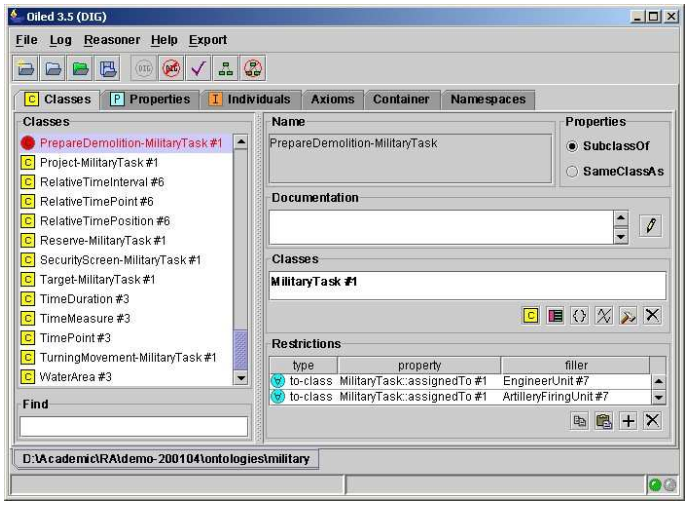

Figure 4.1: Discovery of an unsatisfiable concept by RACER

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.

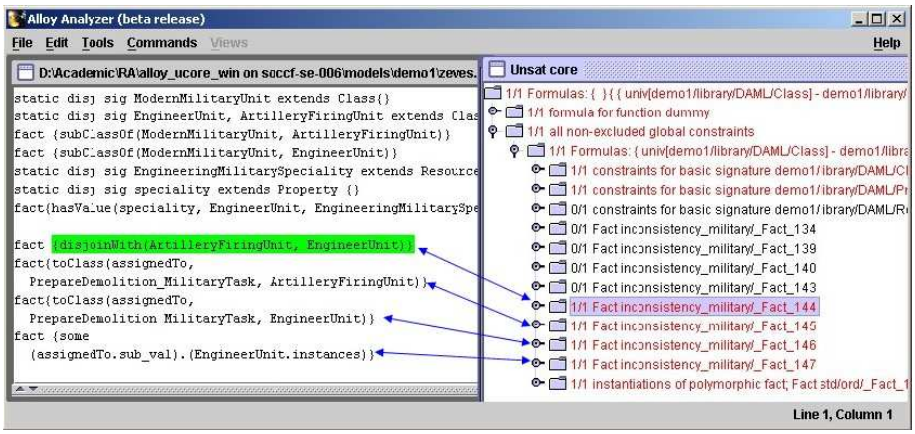

Figure 4.3: Alloy Analyzer showing the source of unsatisfiability

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.