Checking and Reasoning about Semantic Web through Alloy

Jin Song Dong, Jing Sun, and Hai Wang

School of Computing,
National University of Singapore,
dongjs,sunjing,wangh@comp.nus.edu.sg

Abstract. Semantic Web (SW), commonly regarded as the next generation of the Web, is an emerging vision of the new Web from the Knowledge Representation and the Web communities. The Formal Methods community can also play an important role to contribute to SW development. Reasoning and consistency checking can be useful at many stages during the design, maintenance and deployment of SW ontology. However the existing reasoning and consistency checking tools for SW are primitive. We believe that formal techniques and tools, such as Alloy, can provide automatic reasoning and consistency checking services for SW. In this paper, we firstly construct semantic models for the SW language (DAML+OIL) in Alloy, and these models form the semantic domain for interpreting DAML+OIL in Alloy. Then we develop the translation techniques and tools which can automatically map the SW ontology into the DAML+OIL semantic domain in Alloy. Furthermore, with the assistance of Alloy Analyzer (AA) we demonstrate that the consistency of the SW ontology can be checked automatically and different kinds of reasoning tasks can be supported.

keywords: Semantic Web, Alloy

1 Introduction

In recent years, researchers have begun to explore the potential of associating web content with explicit meaning so that the web content becomes more machine-readable and intelligent agents can retrieve and manipulate pertinent information readily. The Semantic Web (SW) [1] proposed by W3C is one of the most promising and accepted approaches. It has been regarded as the next generation of the Web. SW not only emerges from the Knowledge Representation and the Web Communities, but also brings the two communities closer together. We believe in the SW development process, there is a role for formal techniques and tools to play and make important contributions.

In the development of Semantic Web there is a pivotal role for ontology, since it provides a representation of a shared conceptualization of a particular domain that can be communicated between people and applications. Reasoning can be useful at many stages during the design, maintenance and deployment of

K. Araki, S. Gnesi, and D. Mandrioli (Eds.): FME 2003, LNCS 2805, pp. 796–813, 2003.

ontology. Because autonomous software agents may perform their reasoning and come to conclusions without human supervision, it is essential that the shared ontology is consistent. However, since the Semantic Web technology is still in the early stage, the reasoning and consistency checking tools are very primitive.

The software modeling language Alloy [9] is suitable for specifying structural properties of software. SW is a well suited application domain for Alloy because relationships between web resources are the focus points in SW and Alloy is a first order declarative language based on relations. Furthermore, Alloy specifications can be analyzed automatically using the Alloy Analyzer (AA) [10]. Given a finite scope for a specification, AA translates it into a propositional formula and uses SAT solving technology to generate instances that satisfy the properties expressed in the specification. We believe that if the semantics of the SW languages can be encoded into Alloy, then Alloy can be used to provide automatic reasoning and consistency checking services for SW. Various reasoning tasks can be supported effectively by AA.

The remainder of the paper is organized as follows. Section 2 briefly introduces the Semantic Web and Alloy. In section 3 semantic domain and functions for the DARPA Agent Markup Language (DAML+OIL) [14] constructs are defined in Alloy. Section 4 presents the transformation from DAML+OIL documents to an Alloy program. In section 5 different reasoning tasks are demonstrated. Section 6 concludes the paper.

2 Semantic Web and Alloy Overview

2.1 Semantic Web Overview

The Semantic Web is a vision for a new kind of Web with enhanced functionality which will require semantic-based representation and processing of Web information. W3C has proposed a series of technologies that can be applied to achieve this vision. The Semantic Web extends the current Web by giving the web content a well-defined meaning, better enabling computers and people to work in cooperation. XML is aimed at delivering data to systems that can understand and interpret the information. XML is focused on the syntax (defined by the XML schema or DTD) of a document and it provides essentially a mechanism to declare and use simple data structures. However there is no way for a program to actually understand the knowledge contained in the XML documents.

Resource Description Framework (RDF) [11] is a foundation for processing metadata; it provides interoperability between applications that exchange machine-understandable information on the Web. RDF uses XML to exchange descriptions of Web resources and emphasizes facilities to enable automated processing. The RDF descriptions provide a simple ontology system to support the exchange of knowledge and semantic information on the Web. RDF Schema [2] provides the basic vocabulary to describe RDF documents. RDF Schema can be used to define properties and types of the web resources. Similar to XML Schema which gives specific constraints on the structure of an XML document,

Table 1. DAML+OIL constructs (partial)

DAML+OIL constructs	**Description**
$DAML\_class$	classes
$DAML\_property$	properties
$DAML\_subclass[C]$	subclasses of C
$DAML\_subproperty[P]$	sub properties of P
$instanceof[C]$	instances of the DAML+OIL class C

RDF Schema provides information about the interpretation of the RDF statements. The DARPA Agent Markup Language (DAML) [14] is an AI-inspired description logic-based language for describing taxonomic information. DAML currently combines Ontology Interchange Language (OIL) [3] and features from other ontology systems. It is now called DAML+OIL and contains richer modelling primitives than RDF. The DAML+OIL language builds on top of XML and RDF(S) to provide a language with both a well-defined semantics and a set of language constructs including classes, subclasses and properties with domains and ranges, for describing a Web domain. DAML+OIL can further express restriction on membership in classes and restrictions on certain domains and ranges values.

Semantic Web is highly distributed, and different parties may have different understanding of the same concept. Ideally, the program must have a way to discover the common meanings from the different understandings. It is central to another important concept in Semantic Web service – ontology. The ontology for a Semantic Web service is a document or file that formally defines the relations among terms. The most typical kind of ontology for the Web has taxonomy and a set of inference rules. Ontology can enhance the functioning of the Web in many ways, and RDFS and DAML+OIL supply the language to define the ontology.

We summarize some essential DAML+OIL constructs in Table 1.

2.2 Alloy Overview

Alloy [9] is a structural modelling language based on first-order logic, for expressing complex structural constraints and behavior. Alloy treats relations as first class citizens and uses relational composition as a powerful operator to combine various structured entities. The essential constructs of Alloy are as follows:

Signature: A signature (`sig`) paragraph introduces a basic type and a collection of relation (called field) in it along with the types of the fields and constraints on their value. A signature may inherit fields and constraints from another signature.

Function: A function (`fun`) captures behaviour constraints. It is a parameterized formula that can be "applied" elsewhere,

Fact: Fact (`fact`) constrains the relations and objects. A *fact* is a formula that takes no arguments and need not to be invoked explicitly; it is always true.

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

The Alloy Analyzer (AA) is a tool for analyzing models written in Alloy. Given a formula and a scope – a bound on the number of atoms in the universe – AA determines whether there exists a model of the formula that uses no more atoms than the scope permits, and if so, return it. It supports two kinds of automatic analysis: simulation, in which the consistency of an invariant or operation is demonstrated by generating a state or transition, and checking, in which a consequence of the specification is tested by attempting to generate a counterexample.

3 DAML+OIL Semantic Encoding

DAML+OIL has a well-defined semantics which has been described in a set of axioms [7]. In this section based on the semantics of DAML+OIL, we define the semantic functions for some important DAML+OIL primitives in Alloy. The complete DAML+OIL semantic encoding can be found in the appendix.

3.1 Basic Concepts

The semantic models for DAML+OIL are encoded in the module `DAMLOIL`. Users only need to import this module to reason DAML+OIL ontology in Alloy.

```
module DAMLOIL
```

All the things described in Semantic web context are called resources. A basic type `Resource` is defined as:

```
sig Resource {}
```

All other concepts defined later are extended from the `Resource`. `Property`, which is a kind of `Resource` itself, relates `Resource` to `Resource`.

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

Each `Property` has a relation `sub_val` from set `<Property, Resource, Resource>` with type `<Resource, Resource, Resource>` (since in Alloy subsignature does not introduce a new type). This relation can be regarded as a RDF statement, i.e., a triple of the form
`<property(or predicate), subject, value(or object)>`.

The class corresponds to the generic concept of type or category of resource. Each `Class` maps a set of resources via the relation `instances`, which contains all the instance resources. The keyword `disj` is used to indicate the `Class` and `Property` are disjoint.

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

The DAML+OIL also allows the use of XML Schema datatypes to describe (or define) part of the datatype domain. However there are no predefined types in Alloy, so we treat Datatype as a special Class, which contains all the possible datatype values in the instances relation.

```
disj sig Datatype extends Class {}
```

3.2 Class Elements

The `subClassOf` is a relation between classes. The instances in a subclass are also in the superclasses. A parameterized formula (a function in Alloy) is used to represent this concept.

```
fun subClassOf(csup, csub: Class)
  {csub.instances in csup.instances}
```

The `disjointWith` is a relation between classes. It asserts that there are no instances common with each other.

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

3.3 Property Restrictions

A `toClass` function states that all instances of the class `c1` have the values of property P all belonging to the class `c2`.

```
fun toClass (p: Property, c1: Class, c2: Class)
 {all r1, r2: Resource | r1 in c1.instances <=> r2 in r1.(p.sub_val) =>
                                                r2 in c2.instances}
```

A `hasValue` function states that all instances of the class `c1` have the values of property `P` as resource `r`. The `r` could be an individual object or a datatype value.

```
fun hasValue (p: Property, c1: Class, r: Resource)
    {all r1: Resource | r1 in c1.instances => r1.(p.sub_val) = r}
```

A `cardinality` function states that all instances of the class `c1` have exactly `N` distinct values for the property `P`. The new version of Alloy supports some integer operations.

```
fun cardinality (p: Property, c1: Class, N: Int)
 {all r1: Resource| r1 in c1.instances <=> # r1.(p.sub_val) = int N}
```

3.4 Boolean Combination of Class Expressions

The `intersectionOf` function defines a relation between a class `c1` and a list of classes `clist`. The `List` is defined in the Alloy library. The class `c1` consists of exactly all the objects that are common to all class expressions from the list `clist`.

```
fun intersectionOf (clist: List, c1: Class)
     {all r: Resource| r in c1.instances <=>
         all ca: clist.*next.val | r in ca.instances}
```

The `unionOf` function defines a relation between a class `c1` and a list of classes `clist`. The class `c1` consists of exactly all the objects that belong to at least one of the class expressions from the list `clist`. It is analogous to logical disjunction;

```
fun unionOf (clist: List, c1: Class)
    {all r: Resource| r in c1.instances <=>
      some ca: clist.*next.val| r in ca.instances}
```

3.5 Property Elements

The `subPropertyOf` function states that `psub` is a subproperty of the property `psup`. This means that every pair (subject,value) that is in `psup` is in the `psub`.

```
fun subPropertyOf (psup, psub: Property) {psub.sub_val in psup.sub_val}
```

The `domain` function asserts that the property `P` only applies to instances of the class `c`.

```
fun domain (p: Property, c: Class) {(p.sub_val).Resource inc.instances}
```

The `inverseOf` function shows two properties are inverse.

```
fun inverseOf (p1, p2: Property) {p1.sub_val = ~(p2.sub_val)}
```

4 DAML+OIL to Alloy Transformation

In the previous section we defined the semantic model for the DAML+OIL constructs, so that analyzing DAML+OIL ontology in Alloy can be easily and effectively achieved. We also constructed a XSLT [15] stylesheet for the automatic transformation from DAML+OIL file to into Alloy program. [1]

A set of transformation rules transforming from DAML+OIL ontology to Alloy program are developed in the following presentation.

4.1 DAML+OIL Class Transformation

$$\frac{C \in DAML\_class}{static\ disj\ sig\ C\ extends\ Class\{\}}$$

A DAML_class `C` will be transferred into a scalar `C`, constrained to be an elements of the signature `Class`.

[1] The details of the XSLT program and other information on this project can be found at:
http://nt-appn.comp.nus.edu.sg/fm/alloy/

4.2 DAML+OIL Property Transformation

$$\frac{P \in DAML\_property}{static\ disj\ sig\ P\ extends\ Property\{\}}$$

A DAML_property `p` will be transferred into a scalar `P`, constrained to be an elements of the signature `Property`.

4.3 Instance Transformation

$$\frac{x \in instancesof[Y]}{\begin{array}{l} static\ disj\ sig\ x\ extends\ Resource\{\} \\ fact\{\ x\ in\ Y.instances\}\end{array}}$$

A DAML instance `x` of class `Y` will be transferred into a scalar `x`, constrained to be an element of the signature `Resource`. `x` is a subset of `Y.instances`.

4.4 Other Transformation

Other DAML+OIL constructs can be easily transferred into the Alloy function we defined in the previous section. For example the following rule shows how to transfer the DAML+OIL subclass relation into Alloy code.

$$\frac{subclass[X,Y], X \in DAML\_class, Y \in daml\_class}{fact\{subClassOf(X,Y)\}}$$

4.5 Case Study

A classical DAML+OIL ontology, "animal relation" is used to illustrate how the transformation and analysis could be achieved. The following DAML+OIL ontology defines two class `animal` and `plant` which are disjoint. The `eats` and `eaten_by` are two properties, which are inverse to each other. The domain of `eats` is `animal`. The `carnivore` is a subclass of animal which can only eat animals.

```
<daml:Class rdf:ID="animal">
  <rdfs:label>animal</rdfs:label> </daml:Class>
<daml:Class rdf:ID="plant">
  <rdfs:label>plant</rdfs:label>
  <daml:disjointWith rdf:resource="#animal"/></daml:Class>
<daml:ObjectProperty rdf:about="eaten_by">
  <rdfs:label>eaten_by</rdfs:label>
</daml:ObjectProperty>
<daml:ObjectProperty rdf:about="eats">
  <rdfs:label>eats</rdfs:label>
```

```
   <daml:inverseOf rdf:resource="#eaten_by"/>
   <rdfs:domain><daml:Class rdf:about="#animal"/>
   </rdfs:domain></daml:ObjectProperty>
 <daml:Class rdf:ID="carnivore">
   <rdfs:label>carnivore</rdfs:label>
   <rdfs:subClassOf rdf:resource="#animal"/>
   <rdfs:subClassOf>
     <daml:Restriction> <daml:onProperty rdf:resource="#eats"/>
       <daml:toClass rdf:resource="#animal"/>
     </daml:Restriction>
   </rdfs:subClassOf></daml:Class>
```

This DAML+OIL ontology will be transferred into Alloy as follow,

```
module animal
/*import the library module we defined*/
open DMALOIL
/* plant and animal are translated to two class instances, the key
 word static is used to a signature contains exactly one element.*/
static disj sig plant, animal extends Class {}

/* The disjoin element was transferred into fact in Alloy */
fact {disjointWith(plant, animal)}

/* eats, eaten_by are translated to two property instances */
static disj sig eats, eaten_by extends Property {}
fact {inverseOf(eats, eaten_by)}
fact {domain(eats, animal)}

static disj sig carnivore extends Class{}
fact{subClass(animal, carnivore)}
fact{toClass(eats, carnivore, animal)}
```

We can check the consistency of the DAML+OIL ontology and do some reasoning readily.

5 Analysing DAML+OIL Ontology

Reasoning is one of the key tasks for the semantic web. It can be useful at many stages during the design, maintenance and deployment of ontology.

There are two different levels of checking and reasoning, the conceptual level and the instance level. At the conceptual level, we can reason about class properties and subclass relationships. At the instance level, we can do the membership checking (instantiation) and instance property reasoning. The DAML+OIL reasoning tool, i.e. FaCT [8], can only provide conceptual level reasoning, while AA can perform both. The FaCt system originally is designed to be a terminological classifer (TBox) which concerns only about the concepts, roles and attributes, not instances. The semantic web reasoner based on the FaCT, like OILED, does not support instance level reasoning well.

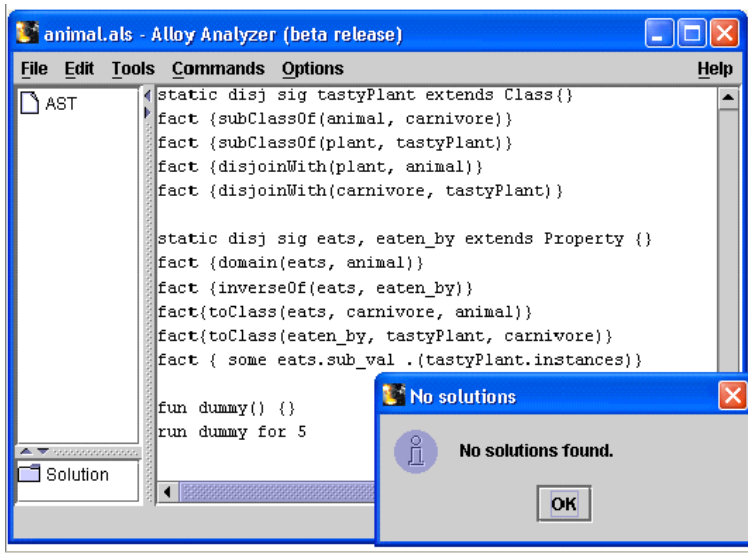

Fig. 1. Inconsistence example

5.1 Class Property Checking

It is essential that the ontology shared among autonomous software agents is conceptually consistent. Reasoning with inconsistent ontology may lead to erroneous conclusions. In this section we give some examples of inconsistent ontology that can arise in ontology development, and demonstrate how these inconsistencies can be detected by the Alloy analyzer. For example, we define another class `tastyPlant` which is a subclass of `plant` and eaten by the `carnivore`. There is an inconsistency since by the ontology definition carnivores can only eat animals. Animals and plants are disjoint.

```
<daml:Class rdf:ID="tastyPlant">
 <rdfs:label>tastyPlant</rdfs:label>
 <rdfs:subClassOf rdf:resource="#plant"/>
 <rdfs:subClassOf>
   <daml:Restriction>
     <daml:onProperty rdf:resource="#eat_by"/>
     <daml:toClass rdf:resource="#carnivore"/>
   </daml:Restriction></rdfs:subClassOf>
 </daml:Class>
```

We transform the ontology into an Alloy program, add some facts to remove the trivial models (like everything type is empty set) and load the program into the Alloy Analyzer. The Alloy Analyzer will automatically check the consistency. We conclude that there is an inconsistency in the animal ontology since Alloy can not find any solutions satisfying all facts within the scope (Figure 1). Note that when Alloy can not find a solution, it may also be due to the scope being too small. By picking a large enough scope, "no solution found' is very likely to mean that an inconsistency has occurred.

Let us take another example. Suppose we define that the `polyphagic_animal` `eats` at least two kind of things i.e `polyphagic_animal` objects have at least two distinct values for the property `eats`. There is also one kind of animal called `picky_animal` which only eats one other kind of animal. The ontology will be defined as follows:

```
<daml:Class rdf:ID="polyphagic_animal">
 <rdfs:label>polyphagic_animal</rdfs:label>
 <rdfs:subClassOf rdf:resource="#animal"/>
 <rdfs:subClassOf>
  <daml:Restriction>
    <daml:onProperty rdf:resource="#eats"/>
    <daml:minCardinality> 2 </daml:minCardinality>
  </daml:Restriction></rdfs:subClassOf></daml:Class>
<daml:Class rdf:ID="#picky_animal">
 <rdfs:label>picky_animal</rdfs:label>
 <rdfs:subClassOf rdf:resource="#animal"/>
 <rdfs:subClassOf>
   <daml:Restriction>
     <daml:onProperty rdf:resource="#eats"/>
     <daml:Cardinality> 1 </daml:Cardinality>
   </daml:Restriction></rdfs:subClassOf></daml:Class>
```

From the above ontology we can infer that the `picky_animal` is not a kind of `polyphagic_animal`, otherwise it would be an inconsistency that AA can easily pick up.

5.2 Subsumption Reasoning

The task of subsumption reasoning is to infer a DAML+OIL class is the subclass of another DAML+OIL class. We use the relationship between the fish, shark and dolphin as a example to demonstrate this kind of reasoning task. In the animal ontology a property `breathe_by` is defined. A `fish` class is a subclass of the `animal` which `breathe_by` the `gill`. Since the purpose of this paper is to demonstrate ideas, we keep the ontology simple. In reality there are some animals such as frogs and toads, which can respire by use of gills when they are young and by lungs when they reach adult stage. Also we do not consider the animals which respire by use of the pharyngeal lining or skin, like newborn Julia Creek dunnarts.

```
<daml:ObjectProperty rdf:ID="breathe_by"/>
<daml:Class rdf:ID="gill">
   <rdfs:label>gill</rdfs:label></daml:Class>
<daml:Class rdf:ID="fish">
 <rdfs:label>fish</rdfs:label>
 <rdfs:subClassOf rdf:resource="#animal"/>
 <rdfs:subClassOf>
  <daml:Restriction>
   <daml:onProperty rdf:resource="#breathe_by"/>
   <daml:toClass rdf:resource="#gill"/>
  </daml:Restriction></rdfs:subClassOf></daml:Class>
```

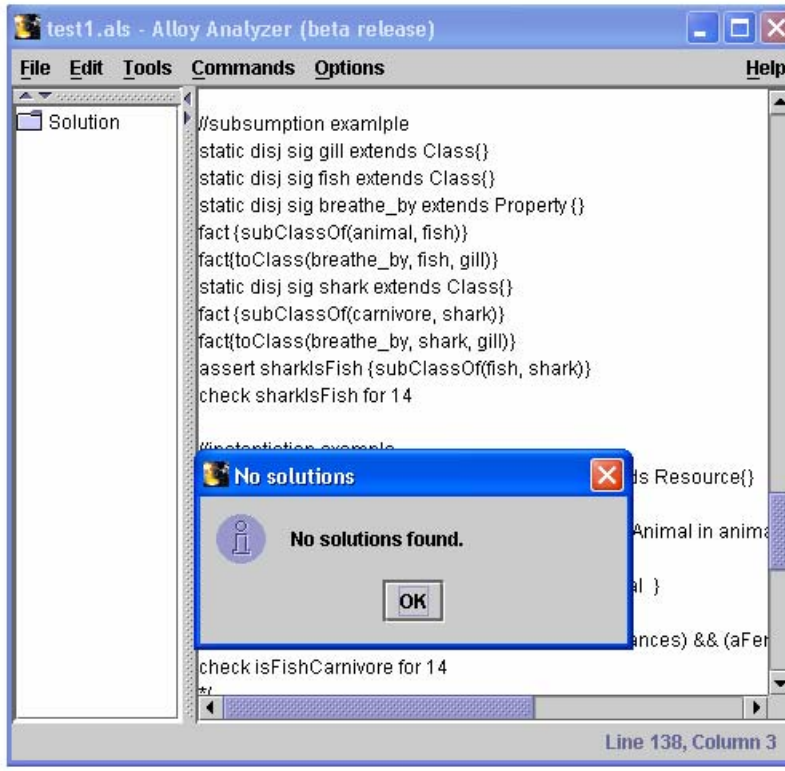

Fig. 2. Subsumption example

We also define a class `shark`, a subclass of `carnivore` which `breathe by` the gill.

```
<daml:Class rdf:ID="shark">
 <rdfs:label>shark</rdfs:label>
 <rdfs:subClassOf rdf:resource="#carnivore"/>
 <rdfs:subClassOf>
  <daml:Restriction>
   <daml:onProperty rdf:resource="#breathe_by"/>
   <daml:toClass rdf:resource="#gill"/>
  </daml:Restriction></rdfs:subClassOf></daml:Class>
```

Several of the classes were upgraded to being defined when their definitions constituted both necessary and sufficient conditions for class membership, e.g., a `animal` is a `fish` if and only if it breathes by the `gill`. Additional subclass relationships can be inferred i.e. the `shark` is also a subclass of `fish`. We transfer this ontology into an Alloy program and make an assertion that the `shark` is the subclass of `fish`. The Alloy analyzer will check the correctness of this assertion automatically (Figure 2). The Alloy Analyzer checks whether an assertion holds by trying to find a counterexample. Note that "no solution" means no

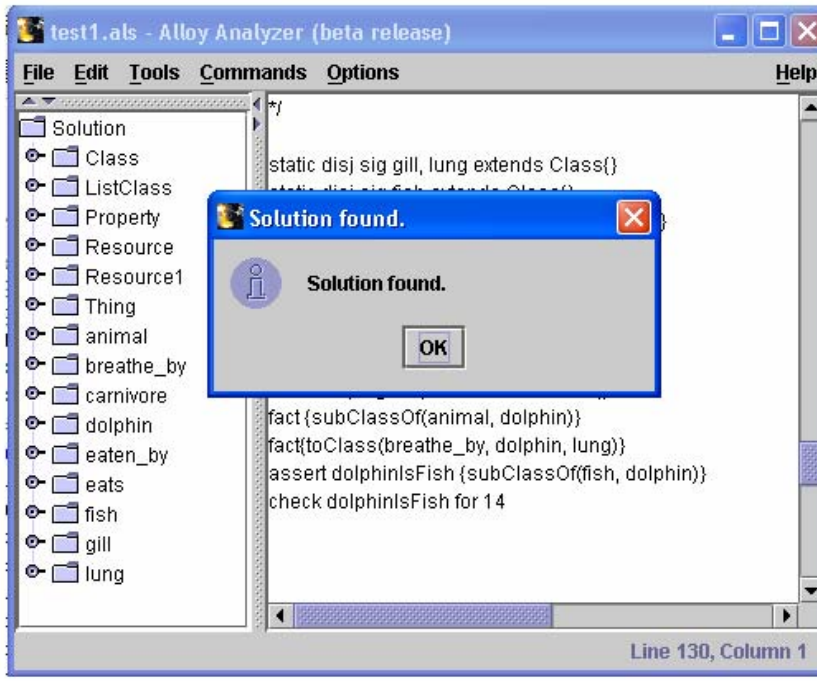

Fig. 3. Dolphin is not a fish

counterexample found, in this case, it indicates that the assertion is sound. To make it more interesting, we define classes `dolphin` and `lung`. The Dolphin is a kind of animal which breathe by lungs. The classes `gill` and `lung` are disjoint. Furthermore the `breathe_by` is a unique property.

```
<daml:Class rdf:ID="lung">
 <rdfs:label>lung</rdfs:label>
 <daml:disjointWith rdf:resource="#gill"/></daml:Class>
<daml:Class rdf:ID="dolphin">
 <rdfs:label>dolphin</rdfs:label>
 <rdfs:subClassOf rdf:resource="#animal"/> <rdfs:subClassOf>
  <daml:Restriction>
   <daml:onProperty rdf:resource="#breathe_by"/>
   <daml:toClass rdf:resource="#lung"/>
  </daml:Restriction></rdfs:subClassOf></daml:Class>
```

Suppose we make an assertion that the `dolphin` is a kind of `fish`, the Alloy Analyzer will refute it since some counterexample was found (Figure 3). If we add that dolphin is a fish as a fact in the module, the AA will conclude that an inconsistency has arisen.

5.3 Instantiation

Instance level reasoning is one of the main contributions for reasoning over DAML+OIL ontology using Alloy. Currently some successful DAML+OIL reasoners like FaCT are designed for description logics (DL) T-box reasoning, which lacks support for instances. In Alloy every expression denotes relations. The scalars will be represented by singleton unary relations - that is, relations with one column and one row. The instance level reasoning can be supported readily in Alloy.

Instantiation is a reasoning task which tries to check if an individual is an instance of a class. For example, we define two resources `aFeralAnimal` and `aMeekAminal` as the instances of class `animal`. `aGill` is an instance of class `gill`. `aFeralAnimal eats aMeekAnimal` and breathes by `aGill`. People may want to check if `aFeralAnimal` is a `carnivore` and a `fish`.

```
<animal rdf:ID="aMeekAnimal">
  <rdfs:label>aMeekAnimal</rdfs:label> </animal>
<gill rdf:ID="aGill"> <rdfs:label>aGill</rdfs:label></gill>
<animal rdf:ID="aFeralAnimal">
 <rdfs:label>aFeralAnimal</rdfs:label>
 <breathe_by rdf:resource="aGill"/>
 <eats rdf:resource="aMeekAnimal"/> </animal>
```

We transfer the ontology into an Alloy program and make an assertion as following:

```
static disj sig aFeralAnimal, aMeekAnimal extends Resource{}
static disj sig aGill extends Resource{}
fact {aFeralAnimal in animal.instances &&
      aMeekAnimal in animal.instances}
fact {aGill in gill.instances}
fact {(aFeralAnimal->aMeekAnimal) in eats.sub_val}
fact {(aFeralAnimal->aGill) in breathe_by.sub_val}
assert isFishCarnivore
  {(aFeralAnimal in fish.instances) &&
   (aFeralAnimal in carnivore.instances)}
check isFishCarnivore for 15
```

AA concludes that this assertion is correct.

5.4 Instance Property Reasoning

Instance property reasoning (often regarded as knowledge querying) is important in Semantic Web applications. Since one of the promising strengths of Semantic Web technology is that it gives the agents the capability to do more accurate and more meaningful searches. The agent can answer some questions for which the answer is not explicitly stored in the knowledge base.

For example, the `emerge_early` and `emerge_later` are two properties, which are inverse to each other. Animal `A` emerged early then `B` if the species of `A`

emerges earlier than the species of B on the earth. emerge_early is transitive. Three animal instances firstDinosaur, firstApe and firstHuman are defined. firstDinosaur emerge_early firstApe and firstApe emerge_early firstHuman. One possible question people may ask is that whether firstHuman is emerge_later firstDinosaur. With the assistance of Alloy reasoner, such questions can be answered.

```
fact{TransitiveProperty(emerge_early)}
static disj sig firstDinosaur, firstApe, firstHuman extends Resource{}
fact { firstDinosaur in animal.instances
    && firstApe in animal.instances
    && firstHuman in animal.instances}
fact {(firstDinosaur->firstApe) in emerge_early.sub_val}
fact {(firstApe->firstHuman) in emerge_early.sub_val}
assert hum {(firstHuman->firstDinosaur) in emerge_later.sub_val}
check hum for 14
```

AA concludes that this assertion is correct.

6 Related Work and Conclusion

The main contribution of this paper is that it develops the semantic models for DAML+OIL language constructs in Alloy and the systematic transformation rules and (XSLT) program which can translate DAML+OIL ontology to Alloy automatically. With the assistance of Alloy Analyzer (AA), we also demonstrated that the consistency of the SW ontology can be checked automatically and different kinds of reasoning tasks can be supported. Alloy is chosen over other modeling techniques because

- Alloy is based on relations, where relations between web resources are the focus issues in SW.
- Alloy has an impressive automatic tool support.

We believe SW is a new novel application domain for Alloy. Recently, the technique/tool developed in this paper was successfully applied to a military case study [6]. Alloy was used to check and reason a *plan ontology* [12] developed by a research team at DSO National Laboratories in Singapore.

Recently, some researchers have begun to explore the potential of combining Web technologies and SE technologies together, e,g. [13]. However there has not been much work done on the application of formal techniques for semantic-web. In our previous work [5] we tried to extract web ontology from Z requirement models, which is a very different approach from the techniques demonstrated in this paper – checking and reasoning web ontology by encoding the semantics of DAML+OIL into the Alloy system.

From a completely different direction, i.e., applying SW to build software modeling environment, we recently investigated how RDF and DAML+OIL can be used to construct a Semantic Web environment for supporting, extending and

integrating various specification languages [4]. We believe SW can contribute to the new developments for the software modeling environment.

In summary, there is a clear synergy between SW languages and software modeling techniques. The investigation of the links between those two paradigms will lead to great benefits for both areas.

Acknowledgements

We would like to thank Hugh Anderson, DSTA staffs and anonymous referees for many helpful comments. We also would thank Daniel Jackson and Ilya Shlyakhter for providing useful info and demo on Alloy. This work is supported by the Defence Innovative Research grant *Formal Design Methods and DAML* from Defence Science & Technology Agency (DSTA) Singapore.

References

1. T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. Scientific American, May 2001.
2. D. Brickley and R.V. Guha (editors). Resource description framework (rdf) schema specification 1.0. http://www.w3.org/TR/2000/CR-rdf-schema-20000327/, March, 2000.
3. 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.
4. J. S. Dong, J. Sun, and H. Wang. Semantic Web for Extending and Linking Formalisms. In L.-H. Eriksson and P. A. Lindsay, editors, *Proceedings of Formal Methods Europe: FME'02*, Copenhagen, Denmark, July 2002. Springer-Verlag.
5. J. S. Dong, J. Sun, and H. Wang. Z Approach to Semantic Web Services. In C. George and H. Miao, editors, *International Conference on Formal Engineering Methods (ICFEM'02)*. LNCS, Springer-Verlag, October 2002.
6. J. S. Dong, J. Sun, H. Wang, C. H. Lee, and H. B. Lee. Analysing web ontology in alloy: A military case study. In *Proc. 15th International Conference on Software Engineering and Knowledge Engineering: SEKE'03*, San Francisco, USA, July 2003.
7. Richard Fikes and Deborah L. McGuinness. An axiomatic semantics for rdf, rdf schema, and daml+oil. Technical Report KSL-01-01, Knowledge Systems Laboratory, 2001.
8. I. Horrocks. The FaCT system. *Tableaux'98, Lecture Notes in Computer Science*, 1397:307–312, 1998.
9. D. Jackson. Micromodels of software: Lightweight modelling and analysis with alloy. Available: http://sdg.lcs.mit.edu/alloy/book.pdf, 2002.
10. D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the alloy constraint analyzer. In *Proc. 22nd International Conference on Software Engineering: ICSE'2000*, pages 730–733, Limerick, Ireland, June 2000. ACM Press.
11. O. Lassila and R. R. Swick (editors). Resource description framework (rdf) model and syntax specification. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/, Feb, 1999.

12. C. H. Lee. Phase I Report for Plan Ontology. DSO National Labs, Singapore, 2002.
13. Cecilia Mascolo, Wolfgang Emmerich, and Anthony Finkelstein. XML technologies and software engineering. In *International Conference on Software Engineering*, pages 775–776, 2001.
14. F. van Harmelen, P. F. Patel-Schneider, and I. Horrocks (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, ..., March, 2001.
15. World Wide Web Consortium (W3C). Xsl transformations (xslt) version 1.0. http://www.w3.org/TR/xslt, 1999.

A Completed DAML+OIL Semantic Encoding

A.1 Basic concepts

The semantic models for DAML+OIL are encoded in the module `DAMLOIL`. The semantic encoding for the basic concepts was summarized in the table 2.

```
module DAMLOIL
```

Table 2. DAML+OIL Semantic encoding (basic concepts)

DAML+OIL primitive	**Alloy semantic function**
Resource	sig Resource {}
DAML_Property	disj sig Property extends Resource {sub_val: Resource → Resource}
DAML_Class	disj sig Class extends Resource {instances: set Resource}
Datatype	disj sig Datatype extends Class {}

All the things described in Semantic web context are called resources. All other concepts defined later like `Property` and `Class` are extended from the `Resource`.

A.2 Class Elements

The semantic encoding for the class elements was summarized in the table 3. It includes constructs like `subClassOf`, `disjointWith`, `disjointUnionOf` and `sameClassAs`.

A.3 Property Restrictions

The semantic encoding for the property restrictions was summarized in the table 4. A property restriction defines the class of all objects that satisfy the restriction. For example the `toClass` function states that all instances of the class `c1` have the values of property P all belonging to the class `c2`. The other constructs include `hasValue`, `hasClass`, `cardinality` etc..

Table 3. DAML+OIL Semantic encoding (class elements)

DAML+OIL primitive	**Alloy semantic function**
subClassOf	fun subClassOf(csup, csub: Class) {csub.instances in csup.instances}
disjointWith	fun disjointWith (c1, c2: Class) { no c1.instances & c2.instances}
disjointUnionOf	fun disjointUnionOf(clist: List, c1: Class) {c1.instances = clist.*next.val.instances all disj ca1, ca2: clist.*next.val | no ca1.instances & ca2.instances }
sameClassAs	fun sameClassAs(c1, c2: Class) {c1.instances = c2.instances}

Table 4. DAML+OIL Semantic encoding (Property restrictions)

DAML+OIL primitive	**Alloy semantic function**
toClass	fun toClass (p: Property, c1: Class, c2: Class) {all r1, r2: Resource | r1 in c1.instances <=> r2 in r1.(*p.sub_val*) => r2 in c2.instances}
hasValue	fun hasValue (p: Property, c1: Class, r: Resource) {all r1: Resource | r1 in c1.instances => r1.(*p.sub_val*)=r }
hasClass	fun hasClass (p: Property, c1: Class, c2: Class) {all r1: Resource | r1 in c1.instances => some r1.(*p.sub_val*) & c2.instances}
cardinality	fun cardinality (p: Property, c1: Class, N: Int) {all r1: Resource | r1 in c1.instances <=> # r1.(*p.sub_val*) = int N}
maxCardinality	fun maxCardinality (p: Property, c1: Class, N: Int) {all r1: Resource | r1 in c1.instances <=> # r1.(*p.sub_val*) =< int N }
minCardinality	fun minCardinality (p: Property, c1: Class, N: Int) {all r1: Resource | r1 in c1.instances <=> # r1.(*p.sub_val*) >= int N }
cardinalityQ	fun cardinalityQ (p: Property, c1: Class, N: Int, c2: Class) {all r1: Resource | r1 in c1.instances <=> # r1.(*p.sub_val*) & c2.instances = int N }
maxCardinalityQ	fun maxCardinalityQ (p: Property, c1: Class, N: Int, c2: Class) {all r1: Resource | r1 in c1.instances <=> # r1.(*p.sub_val*) & c2.instances =< int N }
minCardinalityQ	fun minCardinalityQ(p: Property, c1: Class, N: Int, c2: Class) {all r1: Resource | r1 in c1.instances <=> # r1.(*p.sub_val*) & c2.instances >= int N}

A.4 Boolean Combination of Class Expressions

The semantic encoding for the boolean combination of class expression was summarized in the table 5.

Table 5. DAML+OIL Semantic encoding (Boolean combination)

DAML+OIL primitive	Alloy semantic function
intersectionOf	fun intersectionOf (clist: List, c1: Class) {all r: Resource| r in c1.instances <=> all ca: clist.*next.val | r in ca.instances}
unionOf	fun unionOf (clist: List, c1: Class) {all r: Resource| r in c1.instances <=> some ca: clist.*next.val| r in ca.instances}

A.5 Property Elements

The semantic encoding for the property elements was summarized in the table 6. It includes `subPropertyOf`, `samePropertyAs` etc..

Table 6. DAML+OIL Semantic encoding (Property elements)

DAML+OIL primitive	Alloy semantic function
subPropertyOf	fun subPropertyOf (psup, psub: Property) {psub.sub_val in psup.sub_val }
domain	fun domain (p: Property, c: Class) {(p.sub_val).Resource in c.instances }
range	fun range (p: Property, c: Class) {Resource.(p.sub_val) in c.instances }
samePropertyAs	fun samePropertyAs(p1, p2: Property) {p1.sub_val=p2.sub_val }
inverseOf	fun inverseOf (p1, p2: Property) {p1.sub_val = ~(p2.sub_val)}
TransitiveProperty	fun TransitiveProperty(p: Property) {all x, y, z: Resource | y in (p.sub_val).x && z in (p.sub_val).y => z in (p.sub_val).x }
UniqueProperty	fun UniqueProperty (p: Property) {all x : Resource | sole x.(p.sub_val) }
UnambigousProperty	fun UnambigousProperty(p: Property) {all x : Resource | sole (p.sub_val).x}