Derivation of Event-B Models from OWL Ontologies

Eman H. Alkhammash1,a
1Department of Computers and Information Technology, University of Taif, Taif, Saudi Arabia

Abstract. The derivation of formal specifications from large and complex requirements is a key challenge in systems engineering. In this paper we present an approach that aims to address this challenge by building formal models from OWL ontologies. An ontology is used in the field of knowledge representation to capture a clear view of the domain and to produce a concise and unambiguous set of domain requirements. We harness the power of ontologies to handle inconsistency of domain requirements and produce clear, concise and unambiguous set of domain requirements for Event-B modelling. The proposed approach works by generating Attempto Controlled English (ACE) from the OWL ontology and then maps the ACE requirements to develop Event-B models. ACE is a subset of English that can be unambiguously translated into first-order logic. There is an injective mapping between OWL ontology and a subset of ACE. ACE is a suitable interlingua for producing the mapping between OWL and Event-B models for many reasons. Firstly, ACE is easy to learn and understand, it hides the math of OWL and would be natural to use by everybody. Secondly ACE has a parser that converts ACE texts into Discourse Representation Structures (DRS). Finally, ACE can be extended to target a richer syntactic subset of Event-B which ultimately would facilitate the translation of ACE requirements to Event-B.

1 Introduction

The derivation of formal specifications from informal requirements is difficult. Informal requirements are often vague, incomplete, and ambiguous. To bridge the gap between informal requirements and formal specification, we propose an approach that makes use of OWL ontologies to describe requirements in a more precise way and to map OWL ontologies into Event-B models using ACE. An ontology is a formal specification of the concepts in a given domain and the relationships between them [1]. ACE is a subset of natural language that can be transformed to discourse representation structures [2]. Event-B is a formal method for modeling and verifying consistency of systems [3].

Ontologies have been used in different activities in requirements engineering and found to be beneficial in reducing ambiguities, inconsistency, and incompetence of requirements. Many studies have explored the use of ontologies in requirement engineering [4, 5]. Some studies adopted ontologies during the elicitation process to reduce ambiguous and incomplete requirements [4, 5]. Other studies build ontologies for describing the structure of requirements specification documents in order to reduce the insufficient requirements specification [4, 5]. Furthermore, application domain ontologies have been used to represent the application domain knowledge and enable the reuse of the requirements for the applications of the same domain [4, 5].

In this paper, we propose an approach that use ontologies to derive clear, concise, consistent and unambiguous requirements to develop Event-B formal models. There are three stages of this approach. Firstly, we convert OWL ontology to OWL/XML using OWL syntax parser [6]. Secondly, we convert OWL/XML into Attempto Controlled English (ACE) text using OWL verbalizer Converter [7]. Thirdly, we produce the Event-B modelling concepts that correspond to the ACE representation.

This paper is structured as follows: Sect. 2 gives an overview of OWL, OWL verbalization, and Event-B formal method. Sect. 3 introduces the methodology employed in this paper. Sect. 4 illustrates the application of the presented approach to a case study. Conclusions and future work are drawn in Sect. 5.

2 Preliminaries

2.1 OWL

The worldwide web consortium (W3C) created OWL that became a W3C recommendation in 2004. OWL is an important language in semantic web, which facilitates creating, modifying, linking and importing ontologies in different environments [8]. OWL is derived from descriptive logic that uses formal semantics and vocabulary to allow machines to perform automatic reasoning [8]. OWL extends the Resource Description Framework (RDF) and RDF Schema and can be processed by the wide range of XML and RDF tools already available [8]. OWL ontologies consist of concepts (classes), roles (properties), and set of individuals (instances). Every individual in the
OWL world is a member of the class called Thing, thus each user-defined class is implicitly a subclass of Thing. There are various OWL language constructs such as class disjointness, functional constraint, class union and intersection, property restrictions, and the like. Figure 1 shows a small example that demonstrates the syntax of OWL:

```
<owl:Class rdf:ID = "OrangeJuice" >
< rdfs:subClassOf rdf:resource = "&food; PotableLiquid"/>
< rdfs:subClassOf >
  < owl:Restriction >
    < owl:onProperty rdf:resource = "madeFromOrange"/>
    < owl:minCardinality rdf:resource = "1"/>
    < rdfs:nonNegativeInteger >1</ rdfs:nonNegativeInteger >
  < /owl:Restriction >
< /rdfs:subClassOf >
< /owl:Class >
```

Figure 1. Definition of the class “OrangeJuice”

This example gives definition for the class OrangeJuice. OrangeJuice is a PotableLiquid (i.e OrangeJuice is a subclass of PotableLiquid class). The cardinality constraint provided in owl : minCardinality says that OrangeJuice is made from at least one Orange.

### 2.2 OWL Verbalization

OWL verbalization is the process of transforming OWL into Ateempto Controlled English (ACE) [9]. ACE is a controlled natural language designed to serve as expressive knowledge representation language [2]. The mapping of OWL constructs into ACE constructs is injective (i.e. ACE can be parsed and converted back to OWL) [9]. ACE can be converted into Discourse Representation Structures (DRS) and automatically reasoned about. ACE includes some construction rules that define its syntax and some of interpretation rules that disambiguate constructs. The verbalization assumes that all names used in the ontology are English words [9]. The individuals are denoted by singular proper names (preferably capitalized). The classes are denoted by singular countable nouns, and (object) properties are map to active and passive verbs. Statement ACE-S1 and statement ACE-S2 are examples of ACE that describe the domain and range of properties.

<table>
<thead>
<tr>
<th>ACE-S1</th>
<th>Everybody who writes something is a human.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACE-S2</td>
<td>Everything that somebody writes is a book.</td>
</tr>
</tbody>
</table>

ACE statements perceive the reasoning results obtained on the OWL ontology [10, 11]. Statement ACE-S1 and statement ACE-S2 are valid but inconsistency is caused when we add the following ACE statements ACE-S3-ACE-S5:

<table>
<thead>
<tr>
<th>ACE-S3</th>
<th>John who is a man writes a paper.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACE-S4</td>
<td>Every man is a human.</td>
</tr>
<tr>
<td>ACE-S5</td>
<td>No paper is a book.</td>
</tr>
</tbody>
</table>

Inconsistency is caused because statement ACE-S5 says that the input argument paper is outside the rangebook.

Statement ACE-S6 describes inverse functional property. Inverse functional property means that every instance in the range is associated with at most one instance of the domain.

| ACE-S6       | Everything is ordered by at most one thing. |

Inconsistency can be seen if we add the following statements ACE-S7-ACE-S10:

<table>
<thead>
<tr>
<th>ACE-S7</th>
<th>Mary orders a book X.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACE-S8</td>
<td>Bill orders the book X.</td>
</tr>
<tr>
<td>ACE-S10</td>
<td>Mary is not Bill.</td>
</tr>
</tbody>
</table>

### 2.3 Event-B

Event-B [3] is a formal method developed by Jean-Raymond Abrial, which uses set theory and predicate logic to provide a formal notation for the creation of models of discrete systems and the undertaking of refinement steps. Event-B is supported by the Rodin toolset [12], which includes various plugins for features such as theorem-proving, model-checking, model composition and decomposition and translation of diagrammatic representations to textual representation. An abstract Event-B specification can be refined by adding more detail and bringing it closer to an implementation. A refined model in Event-B is verified through a set of proof obligations expressing that it is a correct refinement of its abstraction. Event-B may be used for parallel, reactive or distributed system development. Event-B models contain two constructs: a context and a machine. The context is the static part of a model in which fixed properties of the model (sets, constants, axioms) are defined. The dynamic functional behaviour of a model is represented in the machine part, which includes variables to describe the states of the system, invariants to constrain variables, and events to specify ways in which the variables can change. The outline of an event used in this paper is shown in Figure 2. The event takes parameters \( t \) that satisfies the guard and then executes the body. The guard is predicate on the machine variables and event parameters and an action updates machine variables atomically.

```
any t where guard then action end
```

Figure 2. The outline of an event
3 Methodology

This paper proposes an approach that uses clear, concise, consistent and unambiguous requirements extracted from OWL ontologies to build Event-B formal models. We transform OWL ontologies into Event-B formal models where it can be verified, developed and consecutively transformed into executable code. The steps used in the proposed approach are:

2. Convert OWL/XML into Attempto Controlled English (ACE) text using OWL verbalizer Converter [7]
3. Produce the Event-B modelling concepts that correspond to the ACE representation (requirements)

Table 1 shows some ACE statements (requirements) that parsed using OWL verbalizer and the corresponding Event-B modelling concepts for these sentences. The given ACE statements are borrowed from several tutorials that describe the process of writing OWL ontologies in ACE [9]. They are extracted from various OWL constructs. We use them to give an overview about the proposed approach. ACE can express a domain and a range of a property. Requirement R3 expresses domain of other property and requirement R4 expresses the range of the same property. A property can be functional and inverse functional, functionality in ACE means that for a given subject, the object of the property (verb) is always the same whereas inverse functionality means that for a given object the subject must always be the same. Requirement R5 expresses functionality property whereas requirement R6 expresses inverse functional property. A symmetric property is a property for which holds that if the pair (x, y) is an instance of P, then the pair (y, x) is also an instance of P. Requirement R14 is example of a symmetric property. Equivalent properties state that two properties have the same property extension. They are expressed by declaring the two properties to be superproperties of each other. Requirements R25(a, b) are examples of equivalent properties. The transitivity property means that if a pair (x, y) is an instance of P, and the pair (y, z) is also instance of P, then we can infer the the pair (x, z) is also an instance of P. Requirement R26 is an example of transitivity property. A property can be disjoint from another property as shown in R24. Disjointness of classes can be expressed via ACE as shown in requirement R22. Cardinality expressed in ACE as shown in requirements R8-12. Individuals of a class are expressed in ACE as in R23.
### Event-B modelling concepts

| Event-B Modelling Concepts | ACE
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>taller; taller ⊆ taller</td>
<td>R26 If something X is taller than something Y and Y is taller than something Z then X is taller than Z.</td>
</tr>
</tbody>
</table>

**Table 1.** Table 1: ACE requirements and the corresponding Event-B modelling concepts

The Event-B modelling concepts described in Table 1 represent data types. In Event-B, events are used to specify ways in which the variables can change. Suppose that we have intersecting sets which are **superclass** which is the parent of class **subclass** and **subsubclass** is a subclass of **subclass**. Then we introduce the following event to add an individual to **subsubclass**.

```plaintext
event add_Member
then
  superclass := superclass ∪ {c}
  subclass := subclass ∪ {c}...
  subsubclass := subsubclass ∪ {c}...
  property := property ∪ (c → a)...
  propertya := propertya ∪ (b → c)
end
```

Any properties that represent a relation of members (individuals) of a class of which the individual c is going to be added, should be updated as well. Properties (**property1**,...,**propertya**) are binary relations on individuals of **subsubclass**. **a** is an individual that belongs to the range of the property **property1**, and **b** is an individual belongs to the domain of property **a**.

To add an individual to a class that is disjoint from other classes, then we only need to add the individual to that class and its superclass and not to the disjoint classes.

To delete an individual from a class we introduce **Delete_Member** event.

```plaintext
event Delete_Member
then
  superclass := superclass \ {c}
  subclass := subclass \ {c}...
  subsubclass := subsubclass \ {c}...
  property := property \ (c → a)...
  propertya := propertya \ (b → c)
end
```

### 4 The application of the proposed approach to an OWL ontology

In this section we apply the proposed approach to Pizza ontology. Pizza ontology is well-known ontology developed in the Protége-OWL tutorial [13]. Pizza classifies several types of pizzas. In this case study we focus on two classes: **Food** and **PizzaTopping**. **Food** class has subclasses **VegetarianPizza** and **NonVegetarianPizza**. The **PizzaTopping** has subclasses: **MushroomTopping**, **MozzarellaTopping** and **TomatoTopping**. The properties are: **hasIngredient** which is a transitive property and **hasTopping** which is an inverse property that maps **Pizza** class to **PizzaTopping** class. In order to apply our approach to transform the Pizza ontology to Event-B formal models, we convert Pizza ontology to controlled English using the ACE converter using step 1 and 2 mentioned in Section 3 and then we select the following requirements to formalize them in Event-B models (step 3 in Section 3).

Requirements **REQ1**, **REQ2** and **REQ3** are described as follows:

**REQ1** Every Pizza is a Food.

**REQ2** Every PizzaBase is a Food.

**REQ3** Every PizzaTopping is a Food.

Requirements **REQ4**, **REQ5** and **REQ6** can be mapped to the following Event-B set variables:

**REQ4** Everything that hasIngredient something is a Food.

**REQ5** Everything that is IngredientOf something is a Food.

**REQ6** If X is IngredientOf something that is IngredientOf Y then X is IngredientOf Y.

Requirements **REQ4** and **REQ5** define **hasIngredient** and its inverse **isIngredientOf** and identify the domain and the range. In Event-B we can use relation/function to define the domain and the range of a property and to infer the inverse property. There is no requirement says that **hasIngredient** is functional. Therefore, we introduce relation called **hasIngredient** and define it as follows:

```plaintext
hasIngredient ∈ Food ↔ Food
```

The domain of relation **hasIngredient** is the set of **Food** parts of all the pairs in **hasIngredient** (i.e. dom( hasIngredient )). Whereas, the range of relation **hasIngredient** is the set of second parts of all the pairs in **hasIngredient** (i.e. ran( hasIngredient )). We can use relational inverse of **hasIngredient** to obtain **hasIngredient** inverse as follows: **hasIngredient**⁻¹

Requirement **REQ6** shows that **hasIngredient** is transitive. Therefore we add the following predicate:

```plaintext
hasIngredient; hasIngredient ⊆ hasIngredient
```
Requirements **REQ7-REQ10** are described as follows:

- **REQ7**: If $X$ hasTopping $Y$ then $Y$ isToppingOf $X$.
- **REQ8**: If $X$ isToppingOf $Y$ then $Y$ hasTopping $X$.
- **REQ9**: Everything that hasTopping something is a Pizza.
- **REQ10**: Everything that is hasTopping by something is a PizzaTopping.

Requirements **REQ7** and **REQ8** identify two properties: hasTopping and its inverse isTopping. **REQ9** identifies the domain of hasTopping. Requirement **REQ10** identifies that PizzaTopping is the range of hasTopping. Therefore we define hasTopping as relation property as follows:

$$\text{hasTopping} \in \text{Pizza} \leftrightarrow \text{PizzaTopping}$$

The domain of relation hasTopping is the set of Pizza parts of all the pairs in hasTopping (i.e. dom(hasTopping)). The range of relation hasTopping is the set of second parts of all the pairs in hasTopping (i.e. ran(hasTopping)). The relational image of set Pizza under relation hasTopping (hasTopping$^{-1}$(Pizza)) gives the image of a set Pizza under the relation hasTopping (i.e. determine all the toppings associated with set Pizza).

Requirement **REQ11** identifies that hasIngredient is a subproperty of hasTopping.

- **REQ11**: If $X$ hasTopping $Y$ then $X$ hasIngredient $Y$.

The corresponding Event-B predicate is as follows:

$$\text{hasTopping} \subseteq \text{hasIngredient}$$

Requirements **REQ12-REQ16** introduce different types of pizza toppings.

- **REQ12**: Every FishTopping is a PizzaTopping.
- **REQ13**: Every MeatTopping is a PizzaTopping.
- **REQ14**: Every TomatoTopping is a PizzaTopping.
- **REQ15**: Every MozzarellaTopping is a PizzaTopping.
- **REQ16**: Every MushroomTopping is a PizzaTopping.

The Event-B modelling concepts correspond to requirements **REQ12-REQ16** are as follows:

- $\text{FishTopping} \subseteq \text{PizzaTopping}$
- $\text{MeatTopping} \subseteq \text{PizzaTopping}$
- $\text{TomatoTopping} \subseteq \text{PizzaTopping}$
- $\text{MozzarellaTopping} \subseteq \text{PizzaTopping}$
- $\text{MushroomTopping} \subseteq \text{PizzaTopping}$

Requirements **REQ17** and **REQ18** introduce the definition of NonVegetarianPizza.

- **REQ17**: Every NonVegetarianPizza is a Pizza that is not a VegetarianPizza.
- **REQ18**: Every Pizza that is not a VegetarianPizza is a NonVegetarianPizza.

The Event-B modelling concepts correspond to requirement **REQ17** and requirement **REQ18** are as follows:

$$\text{VegetarianPizza} = \text{Pizza} \setminus (\text{hasTopping}^{-1}(\text{FishTopping}) \cup \text{hasTopping}^{-1}(\text{MeatTopping}))$$

Requirement **REQ19** defines Mushroom.

- **REQ19**: Every Mushroom is a NamedPizza.
- **REQ20**: Every NamedPizza is a Pizza.

The Event-B modelling concepts correspond to requirement **REQ21** and requirement **REQ22** is:

$$\text{Mushroom} \subseteq \text{NamedPizza} \subseteq \text{Pizza}$$

Requirements **REQ23-REQ26** are:

- **REQ23**: Every Mushroom hasTopping an MozzarellaTopping.
- **REQ24**: Every Mushroom hasTopping a MushroomTopping.
- **REQ25**: Every Mushroom hasTopping a TomatoTopping.
- **REQ26**: Everything that is hasTopping by a Mushroom is something that is an MozzarellaTopping and that is a MushroomTopping and that is a TomatoTopping.

The Event-B modelling concepts correspond to requirements **REQ23-REQ26** are:

$$\text{hasTopping(Mushroom)} = (\text{MozzarellaTopping} \cup \text{MushroomTopping}) \cup \text{TomatoTopping}$$
In Event-B the partition predicate is introduced to identify enumerated sets. Therefore, we can use partition predicate in the machine level to identify the enumerated sets as follows:

```
partition(Food, partition(Pizza, VegetarianPizza, NonVegetarianPizza, ...))
partition(NamePizza, Mushroom, ...)
partition(PizzaTopping, MushroomTopping, MozzarellaTopping, TomatoTopping)...
```

The first argument in the `partition` predicate is the superclass. Other classes are subsets of the superclass class. Partition predicate can be nested to show that subclasses can be parent of other classes as shown in `Pizza` partition that has `Food` as parent class.

Now, we need to define the events. We can introduce `portobello_mushrooms` as a new member in `Mushroom` class. This can be achieved using the following event:

```
event add_portobello_mushrooms
then
Mushroom := Mushroom \cup \{portobello_mushrooms\}
NamedPizza := NamedPizza \cup \{portobello_mushrooms\}
Food := Food \cup \{portobello_mushrooms\}
MozzarellaTopping := MozzarellaTopping \cup \{portobello\}
hasTopping := hasTopping \cup \{portobello_mushrooms \mapsto portobello\}
end
```

5 Conclusions and future work

In this paper we proposed an approach that extract clear, concise, consistent and unambiguous requirements from OWL ontologies and transform them to Event-B formal models. The approach consists of three steps: the first step is based on converting the ontology into OWL format. The second step is based on converting the OWL ontology into ACE texts. The third step links ACE requirements to Event-B modelling concepts. We applied this approach to convert Pizza ontology to Event-B models. In the future we are going to develop a tool to automatically transform ACE texts generated from OWL ontologies to Event-B models. Another direction for future work is to work on ACE texts and convert large subset of ACE’s texts into Event-B modelling concepts. Moreover, we plan to explore an approach of deciding the construction of refinements steps from ACE requirements extracted from OWL ontologies.

References