MATEC Web of Conferences **76**, 04008 (2016) DOI: 10.1051/matecconf/20167604008
CSCC 2016

Derivation of Event-B Models from OWL Ontologies

Eman H. Alkhammash[1,a]

[1] *Department 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 ad 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

[a] e-mail: eman.kms@tu.edu.sa

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 : datatype =
                "&xsd; nonNegativeInteger" > 1
          < /owl : minCardinality >
        < /owl : Restriction >
    < /rdfs : subClassOf >
...
< /owl : Class >
```

Figure 1. Defintion 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 Atempto 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.

ACE-S1	Everybody who writes something is a human.
ACE-S2	Everything that somebody writes is a book.

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*:

ACE-S3	John who is a man writes a paper.
ACE-S4	Every man is a human.
ACE-S5	No paper is a book.

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

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*:

ACE-S7	Mary orders a book X.
ACE-S8	Bill orders the book X.
ACE-S10	Mary is not Bill.

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:

Figure 3. The architecture of the proposed approach

1. Convert OWL ontology to OWL/XML format using OWL Syntax Converter [6]
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 *or-der* 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*.

ACE	**Event-B modelling concepts**
R1 everything, something; thing.	*Thing* is a carrier set
R2 nothing.	no individuals $Thing = \{\}$
R3 If X orders Y then X is an orderer.	$dom(orders) = Orderer$
R4 If X orders Y then Y is an ordered-item.	$ran(orders) = Items$
R5 Everybody orders at most one thing.	$orders \in Orderer \rightarrow Items$
R6 Everything is something that at most one thing orders.	$order \in Orderer \rightarrowtail Items$
R7 Every millionaire owns at least 2 cars.	$owns \in millionaire \leftrightarrow car \wedge$ $\forall i.i \in dom(owns) \implies card(owns[\{i\}]) \geq 2$
R8 something that owns at least v.	$card(Thing) \geq v$
R9 something that owns at most v.	$card(Thing) \leq v$
R10 something that owns more than v.	$card(Thing) > v$
R11 something that owns less than v.	$card(Thing) < v$
R12 something that owns exactly v.	card(Thing)=v
R13 Common noun e.g. cat.	*cat* set
R14 If somebody X loves somebody Y then Y loves X.	$love = love^{-1}$
R15 Every narcissist likes himself.	$id(narcissist) \subseteq likes$
R16 Something that is not a cat.	$Thing \setminus cat$
R17 Something that is not a cat and something that is a camel.	individuals in $camel \setminus cat$
R18 something that is a cat or that is a camel or that ...	individuals in $cat \cup camel...$
R19 Everything that a carnivore eats is a meat.	$eats[\{carnivore\}] \subseteq meat$
R20 Every man is human.	$man \subseteq human$
R21 Every human is a male or is a female.	$human \subseteq (male \cup female)$
R22 No dog is a cat.	$dog \cap cat = \{\}$
R23 John is a student and Mary is a student.	$\{John, Mary\} \subseteq student$
R24 Nothing that is child-of something is spouse-of it. . . .	$\forall x.spouseOf[\{x\}] \cap childOf[\{x\}] = \{\}$
R25 (a) Everybody who hates somebody despises him/her. *(b)* Everybody who despises somebody hates him/her.	$\forall x, y.(x \mapsto y) \in hate \implies (x \mapsto y) \in despite$ $\forall x, y.(x \mapsto y) \in despite \implies (x \mapsto y) \in hate$

ACE	Event-B modelling concepts
R26 If something X is taller than something Y and Y is taller than something Z then X is taller than Z.	$taller; taller \subseteq taller$

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 *c* to *subsubclass*.

event *add_Member*
then
$superclass := superclass \cup \{c\}$
$subclass := subclass \cup \{c\}...$
$subsubclass := subsubclass \cup \{c\}...$
$property_i := property_i \cup \{c \mapsto a\}...$
$property_n := property_n \cup \{b \mapsto c\}$
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 ($property_i ... property_n$) are binary relations on individuals of *subsubclass*. *a* is an individual that belongs to the range of the property $property_i$ and *b* is an individual belongs to the domain of $property_n$.

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.

event *Delete_Member*
then
$superclass := superclass \setminus \{c\}$
$subclass := subclass \setminus \{c\}...$
$subsubclass := subsubclass \setminus \{c\}...$
$property_i := property_i \setminus \{c \mapsto a\}...$
$property_n := property_n \setminus \{b \mapsto c\}$
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 Protégé-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 *REQ1, REQ2* and *REQ3* can be mapped to the following Event-B set variables:

Food $\subseteq$ *Thing* where *Thing* is a set defined in the context *c*
Pizza $\subseteq$ *Food*
PizzaBase $\subseteq$ *Food*
PizzaTopping $\subseteq$ *Food*

Requirements *REQ4, REQ5* and *REQ6* are described as follows:

REQ4	Everything that hasIngredient something is a Food.
REQ5	Everything that is isIngredientOf by something is a Food.
REQ6	If X isIngredientOf something that isIngredientOf Y then X isIngredientOf 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:

$$hasIngredient \in Food \leftrightarrow Food$$

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:

$$hasIngredient; hasIngredient \subseteq hasIngredient$$

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:

$$hasTopping \in Pizza \leftrightarrow PizzaTopping$$

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:

$$hasTopping \subseteq hasIngredient$$

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:

$$FishTopping \subseteq PizzaTopping$$
$$MeatTopping \subseteq PizzaTopping$$
$$TomatoTopping \subseteq PizzaTopping$$
$$MozzarellaTopping \subseteq PizzaTopping$$
$$MushroomTopping \subseteq PizzaTopping$$

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* is as follows:

$$partition(Pizza, VegetarianPizza, NonVegetarianPizza)$$

Requirement *REQ19* and requirement *REQ20* introduce the definition of *VegetarianPizza*.

REQ19	Every VegetarianPizza is a Pizza that does not hasTopping a FishTopping and that does not hasTopping a MeatTopping.
REQ20	Every Pizza that does not hasTopping a FishTopping and that does not hasTopping a MeatTopping is a VegetarianPizza.

The Event-B modelling concepts correspond to requirement *REQ19* and requirement *REQ20* is as follows:

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

Requirement *REQ21* defines *Mushroom*.

REQ21	Every Mushroom is a NamedPizza.
REQ22	Every NamedPizza is a Pizza.

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

$$Mushroom \subseteq NamedPizza$$
$$NamedPizza \subseteq Pizza$$

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:

$$hasTopping[Mushroom] = (MozzarellaTopping \cup MushroomTopping \cup TomatoTopping)$$

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:

$partition(Food, partition(Pizza, VegetarianPizza, NonVegetarianPizza, ...)$
$partition(NamedPizza, Mushroom, ...),$
$partition(PizzaTopping, MushroomTopping, MozzarellaTopping,$
$\quad TomatoTopping)...$

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

[1] W.N. Borst, *Construction of Engineering Ontologies for Knowledge Sharing and Reuse* (Enschede, The Netherlands, 1997)

[2] N. Fuchs, K. Kaljur, G. Schneider, *Attempto Controlled English Meets the Challenges of Knowledge Representation, Reasoning, Interoperability and User Interfaces*, in *In Proceedings of 19th International Florida Artificial Intelligence Research Society Conference* (2006)

[3] J. Abrial, *Modeling in Event-B - System and Software Engineering* (Cambridge University Press, 2010), ISBN 978-0-521-89556-9

[4] V. Castaneda, L. Ballejos, L. Caliusco, M.R. Galli, Global Journal of Researches in Engineering specification **10, no 6**, 2 (2010)

[5] D. Dermeval, J. Vilela, I. Bittencourt, J. Castro, S. Isotani, P. Brito, A. Silva, Requirements Engineering pp. 1–33 (2015)

[6] U. of Manchester, *Wl syntax converter*, http://owl.cs.manchester.ac.uk/converter/

[7] K. Kaljurand, *Verbalizing OWL in (controlled) english*, http://attempto.ifi.uzh.ch/site/docs/verbalizing_owl_in_controlled_english.html (2007)

[8] J. Heflin, K. Words, *An introduction to the owl web ontology language* (2007)

[9] K. Kaljur, N. Fuchs, *Verbalizing OWL in Attempto Controlled English*, in *In Proceedings of OWLED07* (2007)

[10] *Writing OWL ontologies in ACE*, http://attempto.ifi.uzh.ch/site/docs/writing_owl_in_ace.html, accessed: 2016-07-17

[11] N.E. Fuchs, U. Schwertel, *Reasoning in Attempto Controlled English*, in *Principles and Practice of Semantic Web Reasoning, International Workshop PPSWR 2003* (2003), pp. 174–188, ISBN 978-3-540-20582-1

[12] J. Abrial, M. Butler, S. Hallerstede, T. Hoang, F. Mehta, L. Voisin, STTT **12**, 447 (2010)

[13] M. Horridge, *A practical guide to building OWL ontologies using Protege 4 and CO-ODE tools edition 1.3* (2011)