A UML ontology and derived content language for a travel booking scenario

Stephen Cranefield, Jin Pan and Martin Purvis

Department of Information Science University of Otago PO Box 56, Dunedin, New Zealand scranefield@infoscience.otago.ac.nz

ABSTRACT

This paper illustrates an approach to combining the benefits of a multi-agent system architecture with the use of industry-standard modelling techniques using the Unified Modeling Language (UML). Using a UML profile for ontology modelling, an ontology for travel booking services is presented and the automatic derivation of an object-oriented content language for this domain is described. This content language is then used to encode example messages for a simple travel booking scenario, and it is shown how this approach to agent messaging allows messages to be created and analysed using a convenient object-oriented application-specific application programmer interface.

1. INTRODUCTION

This paper is a response to the challenge problem for the AAMAS 2003 Workshop on Ontologies in Agent Systems. The challenge problem [1] was based on the description of a travel agent domain previously developed for an ontology tool assessment exercise organised by the Special Interest Group on Enterprise-Standard Ontology Environments within the European Union's OntoWeb research network [2]. The OAS'03 challenge was to "describe the design and (preferably) an implementation of a multi-agent system in that domain" with emphasis on "the ways in which ontological information is referenced, accessed and used by agents".

In this paper we illustrate the application of our previous work on the use of the Unified Modeling Language (UML) for ontology and content language modelling [3] and the automatic generation of Java classes from these models [4]. This work rests on four observations:

- The Unified Modeling Language is a widely known and standardised modelling language with a compact graphical notation, an XML-based serialisation format, and a lot of existing tool support. We believe that the use of UML for ontology modelling has great benefits in terms of industry acceptance of agent technology. Its principal weakness is the lack of (official) formal semantics, but we believe that ongoing efforts in this direction will remove this shortcoming.
- Much current software development is done using the Java programming language, and the majority of widely used agent development tools are based on Java. Programmers using these tools are most familiar with the use of object-oriented representations and application programmer interfaces (APIs).



Figure 1: Overview of our approach

- The use of object-oriented structures to refer to domain objects within messages is convenient, but must be restricted to precise well understood usages in order to avoid semantic problems. [5].
- Multi-agent systems must coexist and interact with other distributed systems (both technological and human). These other systems have existing techniques for referring to domain objects using reference schemes such as World Wide Web Uniform Resource Identifiers (URIs). This style of reference goes beyond the notion of "standard names" (logical constants that denote each domain object) that lie behind the semantics of FIPA ACL's query-ref communicative act, and it is desirable to allow agents to use a more general notion of object reference when answering queries.

These observations have led us to develop our UML-based modeldriven approach to implementing multi-agent systems. In Section 2 we give a brief overview of this approach, before presenting a simple UML travel booking ontology in Section 3, a discussion of the automatically generated ontology-specific content language in Section 4 and an illustration of its use in an agent application in Section 5. The paper closes with some comments on the applicability of this techique and some areas for future work.

2. OVERVIEW OF OUR APPROACH

Figure 1 presents a schematic overview of our approach to designing and implementing the message-handling component of agent systems. The designer of an agent must have a mental model of the conceptual structure of the domain (the ontology) as well as an understanding of the structure of information describing instances of these concepts and their relationships. We believe the graphical nature of UML makes it a powerful tool for visualising these models: an ontology can be represented by a UML class diagram and instance information can be conveyed as a UML object diagram that shows the values of object attributes and the links (instances of associations) that exist between objects.

When creating the agent application, the programmer must translate these mental models into structures that can be manipulated within a programming language. When using Java, the natural counterpart to a concept in an ontology is a Java class. Although other representations can be used, such as string-based encodings of languages defined by grammars, the most convenient representation for a Java programmer is to have Java classes corresponding directly to the concepts that the agent will need to refer to when manipulating information about the world. To make this possible, we have defined XSLT [6] stylesheets that produce Java class definitions from an XMI [7] serialisation of a UML model (currently we support XMI 1.0 for UML 1.3) [4].

As agents need to communicate information about the world, it is beneficial to provide a straightforward mapping from the progammer's model of the domain (inter-related Java objects in our case) and the content language used to encode information within messages. However, standard agent content languages such as FIPA SL and KIF use a string-based logical representation. These are also generic and weakly typed languages in which domain concepts can only be referred to by name, rather than by more stongly typed mechanisms such as instantiation, and thus messages that do not conform to the agent's known ontologies can only be detected by run-time analysis. As an alternative to this approach, our Java classes generated from the ontology have a built-in serialisation mechanism that allows networks of inter-related objects describing domain objects to be included within messages. The serialisation uses the XML encoding of the Resource Description Framework (RDF) [8], which makes reference to concepts defined in an RDF schema that is also generated automatically from the ontology in UML [9].

This mechanism can also be used to serialise entire messages, including the outer agent communication language (ACL) layer. By defining the ACL in UML as well, and defining a set of UML 'marker' interfaces representing the concepts (such as *predicate* and *action description*) that comprise the required argument types for the ACL's various communicative acts, it is possible to conceptualise messages with arbitrary content languages (if modelled in UML) as object diagrams (see Figures 5 and 6 later in the paper).

Figure 2 illustrates how this technology can be integrated with a Java-based agent platform, and highlights a crucial aspect that addresses the third observation from the introduction: the need for careful use of object-oriented representations within messages. The figure shows a number of UML models: an ontology (top left), ACL and generic (i.e. SL-like) content language definitions, and an ontology-specific content language (top right). The ACL and the content languages are given as input to the XMI-to-Java transformation, and this results in Java classes that provide an object-oriented application programming interface that sits on top of the platform's built-in messaging system classes. We regard an ontology



Figure 2: Integration with an agent platform

as a model of the problem domain, not as a model of the language used to encode *information* about the domain. In other words, an instance of an ontological class Dog would be an actual dog, not a description of a dog. When a structured expression corresponding to the structure of the Dog class appears within a message, this cannot be taken to be playing the role of a logical *term* (which always has a unique denotation), but instead might (depending on the context) play the role of a *proposition* (stating that an object with the specified properties exists) or an *identifying reference expression* (a reference to a possibly non-existent or non-unique object by describing its attribute values) [5, 3].

To avoid any confusion between the notions of ontology and content language, we provide the facility to use domain-specific objectoriented expressions within messages by generating from the ontology a UML model representing a specialised ontology-specific content language. From this, Java classes can be generated as for the ACL and generic content languages models. The generated ontology-specific content language for the travel booking domain is described in Section 4 and its use to create messages is illustrated in Section 5.

3. A TRAVEL BOOKING ONTOLOGY IN UML

Figure 3 shows a simple ontology in UML for the travel booking scenario. This uses two stereotypes, «resourceType» and «value-Type» from a UML profile for ontology modelling that has been presented previously [3]. A resource type is a type of class for which the instances have an intrinsic identity, i.e. two instances with the same attribute values can be distinguished from each other. There is a possibility that an object of that class might be referred to using an identifier such as a unique name in some naming system, a UUID, or a World Wide Web Uniform Resource Identifier (URI). The semantics of the stereotype declare that the class has an additional optional association with a class representing some type of reference (e.g. the concept of a URI). This type is declared using a tagged value in the resource type class declaration, but this feature will not be used in this paper. The resource types in the travel booking ontology are Customer, Consultation, and Place and its subclasses Hotel, City and Airport.



Figure 3: A travel booking ontology

A value type is a class with the opposite property: two instances with the same attribute values cannot be distingushed. Essentially it defines a type for (potentially complex) structured values that can be treated as logical terms within messages. Although there may be concepts included in an ontology which intrinsically seem to have this property, in many other cases the labelling of a class as a value type is a pragmatic decision about how instances of that type will be treated during inter-agent communication. It is a declaration that the Semantic Web principle that anything can be referred to using a URI will not be applied to instances of this class. Agents can expect to receive values of these types explicitly within messages, rather than have them referenced using URIs or other reference types. Also, they do not need to include mechanisms to keep track of references for those types. For example, in the ontology shown, the Itinerary class is declared to be a value type. Neither party in a travel booking conversation needs to be prepared to store references associated with itineraries, whereas it is expected that customers and consultations may be referred to by ID codes. This does not mean that an agent cannot make a query about an existing itinerary, but it must be done indirectly, e.g. by using an identifying reference expression that means "the itinerary associated with the consultation beginning on 15 July 2003 for the customer with code C05321".

The ontology defines a class TravelComponent which represents both customer requirements and the proposed components of an itinerary returned by the travel agent. This dual use is achieved by defining the attributes of the TravelComponent class and the associations of its subclasses Stay, Journey and AirJourney to be optional. A requirement can then be vaguely specified by providing only some of the possible information about a travel component. In an extreme case, only a value for the description string attribute might be provided (although this paper does not attempt to explain how a software agent might understand a textual description of the customer's requirements). For a travel component that is associated with an itinerary, it is expected that all information is provided, with the possible exception of the description attribute (this constraint could be included in the ontology, but is not modelled at present). Note that a consultation object may be linked directly with travel components representing the customer requirements as well as indirectly with other, different, travel component objects via an itinerary. The latter represent the final bookings.

The ontology includes a number of constraints presented as notes in dog-eared rectangles. These could be defined in more detail using the UML's Object Constraint Language, but are shown here in English for clarity. It is not intended that these constraints be used for inference in the current design—rather they serve as part of the specification for the correct implementation of agents using this ontology.

The ontology is not intended to be a complete model of the travel booking domain. It does not include many concepts needed for a realistic account (including the cost for a given itinerary). Also, to keep the model simple it does not use some features of UML that could provide a better model, such as the definition of an enumerated type defining a set of allowed values for the Consulation class's status attribute. For simplicity we regard the types String, Date and DateTime as being 'built in' primitive types in our UML profile which are handled specially during the generation of Java classes.



Figure 4: Derived classes in an ontology-specific content language

4. THE ONTOLOGY-SPECIFIC CONTENT LANGUAGE

Figure 4 presents an overview of the classes that are generated from the ontology to form the ontology-specific content language. The UML package in the middle of the diagram ("Ontology-Specific CL") contains the generated classes. This also includes classes from two other packages: SL (a UML model of a generic content language based on FIPA SL) and ACL (a UML model of a FIPAstyle agent communication language). The inclusion of these additional classes allows complex statements to be formed using connectives from the SL language and also the use of ACL expressions to represent communicative actions (the details of this are beyond the scope of this paper). The CL package contains the set of marker interfaces that represent the generic types of expression that content languages are designed to describe (such as propositions and action descriptions). There is also a package OOCL shown. This defines some support classes used to create identifying reference expressions as networks of inter-connected "pattern nodes". These pattern node networks are used to describe an object by its properties and (possibly complex) inter-relationships with other objects.

To illustrate the nature of the derived classes in the ontology-specific content language we show the classes that correspond to two particular classes in the ontology: one that is a value type (Air Journey) and one that is a resource type (City). Each of these classes results in three generated classes in the ontology-specific content language. (Note that the dashed arrows labelled «derive» are UML dependencies, so they are directed from each derived class back to the one it depends on.)

As discussed in Section 3, an instance of a valuetype can be treated as a logical term within a content language, and so a corresponding class with the same name and structure (e.g. AirJourney) is generated and declared to implement the CL::ValueTerm interface. Some associations between AirJourney and other classes may need to be modified when translated to the new content language, e.g. a reference to a resource type must be replaced by a reference to a derived ... Description class for that resource type (this type of class is discussed below). However, the details of the mapping rules for value types and for resource types are beyond the scope of this paper.

An agent might also want to refer to a value type instance using an identifying reference expression. Therefore, for each value type class there are two corresponding generated classes that can be used for this purpose: a simple ... Description class and a more complex ... Pattern class. The description class (e.g. Air Journey Description) implements the interface CL:: IRE to show that that this can be used as an identifying reference expression (in particular, as a definite description—the only type of IRE currently supported). Under the mapping, all attributes and associations become optional because (for example) although an air journey in real life must necessarily have at least one flight segment, it is possible to refer to an air journey simply by specifying its date or departure and arrival cities.

The ...Pattern class is the same as the ...Description class, except it also extends the class OOCL::PatternNode and any association with another class must be changed to be an association with the appropriate ...Pattern class. The use of this type of class is illustrated in Figure 5 (which is discussed later in the paper).

For resource type classes, there can be no derived class that implements the CL::ValueTerm interface as it is not semantically meaningful to embed instances of that type within a message¹. Instead, corresponding ...Description and ...Pattern classes are generated, as for value types. In addition, a class implementing CL::Proposition is generated in order to allow a convenient object-oriented form of proposition about objects to be used within messages. For this generated class, all attributes and associations become optional.

Further details of this approach to generating ontology-specific content languages can be found elsewhere [3], although the presentation here takes account of some subsequent minor updates to that previous work.

5. USING THE GENERATED CONTENT LANGUAGE

In this section we illustrate the use of an ontology-specific content language generated from the travel booking ontology. Figures 5 and 6 show UML object diagrams representing (respectively) query and response messages in a conversation between a customer and a travel booking agent. The query is an instance of the class QueryRef (predefined in the ACL package). This corresponds to the FIPA query-ref message type which represents a question asking another agent to identify an entity that satisfies particular properties. The content part of the message (represented by a link from the QueryRef object in Figure 5) is a instance of the class OODefDescription shown at the bottom of Figure 4. This class represents an object-oriented version of the iota binding operator from FIPA SL. It has an attribute boundVarName representing a variable name to be used to refer to the subject of the query-ref. This object is then linked to a network of typed pattern nodes, each of which describes some object in terms of its attributes and relationships with other objects. One of these pattern nodes is expected to have a varName attribute value matching the boundVarName value of the OODefDescription object. The other nodes may also have variable names specified, and these may be referred to within Object Constraint Language expressions appearing as the values of the optional constraint attribute of other nodes (this feature is not used in Figure 4). The message in the figure represents the following query:

Given a customer named Stephen Cranefield having a consultation with the requirements of flying from Dunedin to Melbourne on 14 July 2003 and needing accommodation there from the 14th until the 19th, what is the associated itinerary? (For simplicity, we assume our hero wishes to remain uncommitted after the 19th).

Figure 6 shows a UML object diagram representing a possible reply to this request. The message is an instance of the InformRef class from the ACL package. This is a structured version of FIPA ACL's inform-ref message type with two content expressions: a definite description (generally this will be the one that was included in the preceding query-ref) and an expression that identifies the entity that satisfies the query—this may be a value of a primitive type or a value type, a reference to an object (e.g. a URI), or another, hopefully more detailed, definite description. In the case of Figure 6, the definite description (not depicted in full) is the same as the one contained in the query message, with two additional links that provide references for the customer and the consultation objects.

The bottom part of Figure 6 represents the answer to the query and contains an instance of the Itinerary value type that comprises fully detailed value type instances for the air journey and the stay. The details about the hotel, airports and cities are encoded by links to ... Description objects, which describe those external instances of resource types in terms of their attribute values and some required relationships between the objects.

Note that these diagrams conceptualise the messages as UML object diagrams. As shown in Figure 1, the messages are physically realised as Java objects within the agent at run time and as RDF documents when being transported between agents. The Java classes are generated using an XSLT stylesheet [4] and they include code that handles the marshalling and unmarshalling of messages between the in-memory Java representation and the RDF serialisation format [9].

Figure 7 shows an example of how the query message can be created and sent from Java code, using the generated classes for the ACL and the content language (this is based on a simplication of the ACL model presented previously [3]).

There is no doubt that using this object-oriented API to construct messages is far more cumbersome for the programmer than writing a string in FIPA SL. However, it is likely that most messages will be constructed dynamically within code rather than by a static sequence of Java statements as shown in the figure. This approach also has the benefits of being strongly typed and model-driven: support for new ACLs and content languages can quickly be provided once they have been defined using UML. Furthermore, any disadvantages for the creation of messages are balanced by advantages in the analysis of incoming messages: it is much easier to examine a message using its object structure than by performing string matching operations.

¹We would argue that even electronic entities such as instances of electronic currency are best regarded as external objects that agents refer to using references.



Figure 5: A travel booking request message

6. CONCLUSION

This paper has addressed the OAS'03 workshop challenge problem by illustrating the application of our UML-based model-driven approach to defining ontologies and then automatically generating related ontology-specific content languages along with corresponding Java classes and an RDF-based serialisation mechanism. We believe this approach has strong benefits for the software engineering task of designing and implementing agents to peform particular tasks in a given domain. Our work does not currently provide support for the construction of agents that are expected to have more general abilities, where inference may be required in order to determine how to respond to messages (although inference mechanisms based on object networks could be developed).

This technique is being incorporated into the Opal FIPA-compliant agent platform developed at the University of Otago.

An important avenue for future work is the enhancement of the API offered to programmers for constructing messages by providing a larger range of constructors in the generated classes. It would also be highly desirable to develop a technique for annotating the UML definition of ACLs and content languages with information that describes a concrete string-based syntax in such a way that parsers for this language can be generated automatically. This will allow interoperation with traditional FIPA agent platforms and will also give programmers the option of using the string-based syntax for creating messages within agent application code.

7. **REFERENCES**

 OAS 2003 Committee. OAS'03 challenge problem. http://oas.otago.ac.nz/OAS2003/Challenge/challenge.html, 2003.

- [2] OntoWeb project. Project Web pages. http://www.ontoweb.org, 2003.
- [3] S. Cranefield and M. Purvis. A UML profile and mapping for the generation of ontology-specific content languages. *Knowledge Engineering Review*, 17(1):21–39, 2002.
- [4] S. Cranefield, M. Nowostawski, and M. Purvis. Implementing agent communication languages directly from UML specifications. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems* (AAMAS 2002), volume 2, pages 553–554. ACM Press, 2002.
- [5] Stephen Cranefield and Martin Purvis. Referencing objects in FIPA SL: An analysis and proposal. In Proceedings of the Workshop on Agentcities: Challenges in open agent environments, 2nd International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2003), 2003.
- [6] XSL transformations (XSLT) version 1.0. World Wide Web Consortium Web page, 2003. http://www.w3.org/TR/xslt.
- [7] XML metadata interchange specifications. Object Management Group, 2003. http://www.omg.org/technology/ documents/modeling_spec_catalog.htm#XMI.
- [8] Resource Description Framework. World Wide Web Consortium Web page, 2003. http://www.w3.org/RDF/.
- [9] S. Cranefield. UML and the Semantic Web. In I. Cruz, S. Decker, J. Euzenat, and D. McGuiness, editors, *The emerging Semantic Web*, pages 3–20. IOS Press, Amsterdam, 2002.



Figure 6: A reponse message from the travel booking agent

```
// Construct query structure containing variable i
CustomerPattern cust = new CustomerPattern();
ConsultationPattern cons = new ConsultationPattern();
ItineraryPattern itin = new ItineraryPattern();
StringPattern custName = new StringPattern();
AirJourney journ = new AirJourney();
Stay stay = new Stay();
Date monday = new Date (2003, 7, 14);
Date saturday = new Date(2003, 7, 19);
PlacePattern dunedin = new PlacePattern();
PlacePattern melbourne = new PlacePattern();
custName.setValue("Stephen Cranefield");
cust.setName(custName);
dunedin.setName("Dunedin");
melbourne.setName("Melbourne");
journ.setStartDate(monday);
journ.setEndDate(monday);
journ.setFrom(dunedin);
journ.setTo(melbourne);
stay.setStartDate(monday);
stay.setEndDate(saturday);
stay.setPlace(melbourne);
itin.setVarName("i");
Set reqs = new HashSet(); reqs.add(journ); reqs.add(stay)
cons.setRequirement(reqs);
cons.setItinerary(itin);
// Construct message object
Message m = new QueryRef(new AgentRef("agent1"), // Sender
                         // Recipients:
                         Collections.singleton(new AgentRef("agent2")),
                         // Content:
                         new OODefDescription("i", patternNetwork));
// Send message
```

m.send();

```
Figure 7: Using the generated Java code to create and serialise a message
```