Implementing Agent Communication Languages Directly from UML Specifications

Stephen Cranefield
Mariusz Nowostawski
Martin Purvis

The Information Science Discussion Paper Series

Number 2001/03
February 2001
ISSN 1172-6024

Implementing agent communication languages directly from UML specifications

Stephen Cranefield[†], Mariusz Nowostawski and Martin Purvis

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

ABSTRACT

This paper proposes the use of the Unified Modelling Language (UML) as a formalism for defining an abstract syntax for Agent Communication Languages (ACLs) and their associated content languages. It describes an approach supporting an automatic mapping from high-level abstract specifications of language structures to specific computer language bindings that can be directly used by an agent platform. Some advantages of this approach are that it provides a framework for specifying and experimenting with alternative agent communication languages and reduces the error-prone manual process of generating compatible bindings and grammars for different syntaxes. A prototype implementation supporting an automatic conversion from an abstract communication language expressed in UML to a native Java API and a Resource Description Framework (RDF) serialisation format is described.

Keywords

Agent communication languages, abstract syntax, UML, XMI, Java binding, marshalling, RDF

1. INTRODUCTION

Following the introduction of agent communication languages based on speech acts and declarative content languages [8], a number of agent development toolkits became available supporting agent communication using variants of the Knowledge Query and Manipulation Language (KQML) [6]. Being the product of academic research laboratories, this software, understandably, has often not been particularly robust and has not provided a high level of support for the string processing required to parse and access the contents of messages. Creating agents using these toolkits, although preferable to starting from scratch, requires time-consuming and error-prone implementation of parsers and pattern matchers for KIF or other content languages, or the implementation of specific solutions for the types of messages that the application was designed to process.

Since the establishment of the Foundation for Intelligent Agents (FIPA) [7], several industrial research laboratories have developed implementations of FIPA agent platforms that are more robust and provide a higher level of support to the agent application writer than previously available agent toolkits (at least for core functionality such as agent messaging rather than the extensions and enhancements of the state of the art that are usually the focus of academic research). However, while standards organisations such as FIPA have a vital role to play in encouraging industrial adoption of agent technology, there is a danger that they can become bound by the inertia of their existing standards and find it difficult to incorporate any promising advances that may emerge from the research community.

[†]The work described was performed while the first author was visiting the Network Computing Group at the Institute for Information Technology, National Research Council of Canada in Ottawa. Thanks are due to host Larry Korba and to the University of Otago for granting leave and providing financial support.

This paper proposes an approach to support experimentation with alternative agent communication languages (both ACLs and content languages) by providing a way of mapping from high-level abstract specifications of language structures to specific computer language bindings that can be "plugged into" an agent platform. In particular, the paper discusses a procedure for automatically mapping from UML [14] class diagrams describing an ACL and a content language to a set of Java classes representing the concepts in these languages. In addition, Resource Description Framework (RDF) schemas [21] corresponding to the languages are produced and the Java classes include code to marshal and unmarshal messages to and from an RDF [23] representation (serialised using the RDF XML syntax).

This technique also allows descriptions of domain objects that are instances of an ontology expressed in UML to be marshalled and sent by value within the content of messages, however a discussion of the application of this idea is beyond the scope of this paper.

The structure of the paper is as follows: Section 2 describes how agent communication and content languages can be given an abstract syntax using UML [14]. Section 3 briefly summarizes the implications of using abstract syntax, and discusses the major benefits of such a modelling technique. Section 4 presents a proof-of-concept infrastructure for automatic generation of a concrete representation from abstract syntax (UML), and gives an example of a mapping to an object-oriented language (Java). Section 5 uses the same framework to generate a message serialisation format based on RDF, used for marshalling and unmarshalling in-memory object structures. Section 6 briefly discusses the marshalling support provided and Section 7 provides a summary and suggestions for future work.

2. AGENT COMMUNICATION LANGUAGES AND UML

Agent communication languages such as KQML and the FIPA ACL are based on the notion of exchanging information represented as sentences in a logic-based *content language* such as the Knowledge Interchange Language (KIF) [11] or FIPA's Semantic Language (SL). The agent communication language has an outer layer that specifies information needed for routing the message and understanding its conversational context, and also indicates the type of the communicative act represented by the message (e.g. *inform* or *request*), the language in which the content is expressed, and the name of the ontology defining the meanings of symbols appearing in the content. The message's *content* field is then used to store the parameters of the act, which must be a well-formed formula in the content language used and must be parsed by the receiving agent. This paper proposes the use of UML class diagrams to define the abstract syntax of ACLs and content languages. In this approach, an ACL defines interface types corresponding to the concepts required by the communicative acts supported by the ACL.

In the case of FIPA, the ACL is specified by a list of standard communicative acts and the names and descriptions in English of the allowable message parameters and their meanings. There are specific representations defined for the ACL in several formats: two string-based and defined by grammars (one designed for human readability and the other for efficient transmission over wireless devices) and one based on an XML encoding and defined by a document type definition (DTD).

Similarly, FIPA currently has experimental specifications for four different string-based content languages. These all share a common core of basic concepts, but there is no formal or semi-formal framework in which these languages can be related to each other, or which can specify the relationship between concepts in the content languages and those in the ACL. The interface between the two types of language is only implicitly defined in the specifications. From the ACL point of view, the parameters defined for each type of communicative act (CA) specify the number and types of content expression needed for that CA (these must be contained within a tuple occupying a single content field allowed by the FIPA ACL). From the content language point of view, the FIPA SL specification mentions the "grammar entry point"—an SL content expression, and then goes on to list three cases: a proposition, an action and an identifying referring expression (a term identifying an object in the domain of discourse). Thus a message must, depending on the type of the communicative act, have a correctly-typed tuple of expressions within its content field. For a given content language to be used with a particular communicative act, the content language must include some representation for the concepts that must be communicated within that type of message, e.g. only content languages that can describe actions can be used within a request message.

In the proposed approach the ACL model defines interface types corresponding to the concepts required by the communicative acts supported by the ACL. In the case of FIPA ACL these are the concepts of a proposition, an action description and a definite description (a reference to an unknown object by describing a proposition that it must uniquely satisfy, e.g. "the father of John Brown"). These are modelled by *marker interfaces* (i.e. interfaces with no operations), which just declare that the concepts exist. It can then be declared that a content language includes representations of one or more of these concepts by using a UML realisation relationship between classes in the content language model and interfaces from the ACL model. This formalises the relationship between an ACL and a content language and explains how a well-formed message can be constructed by nesting a content language expression within an ACL wrapper.

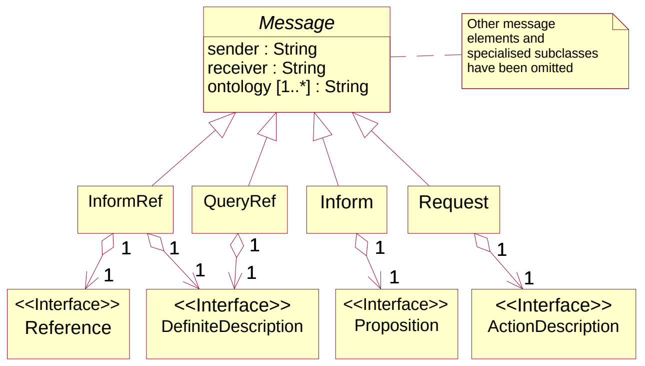
Message
sender : String
receiver : String
ontology [1..*] : String
Other message elements and specialised subclasses have been omitted
InformRef
QueryRef
Inform
Request
1
<<Interface>> Reference
<<Interface>> DefiniteDescription
<<Interface>> Proposition
<<Interface>> ActionDescription

Figure 1: A UML class diagram for an agent communication language

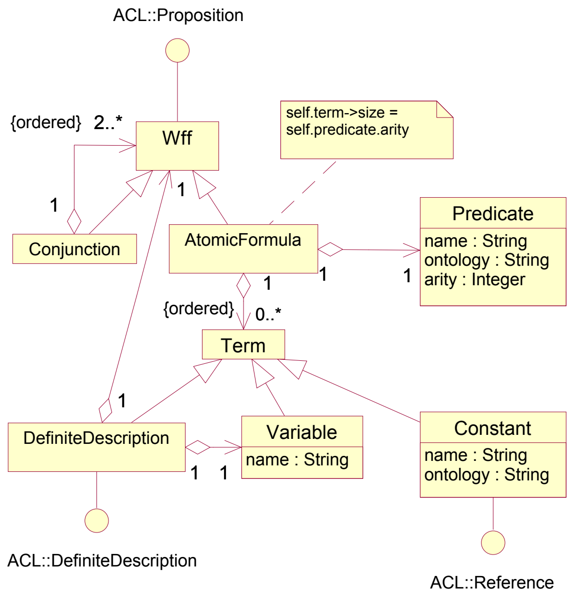

Figure 2: A UML class diagram for a fragment of a content language

Figures 1 and 2 give examples of UML class diagrams defining subsets of an ACL and a content language based on FIPA ACL and FIPA SL respectively.

In order to understand these diagrams, it is sufficient to know the following:

> Rectangles depict classes with the class name in the top section of the box (in italics in the case of an abstract class) and the attributes declared in a separate compartment (if applicable). Lines between classes represent association relationships. A diamond shape may appear at one end of an association; this means that objects of the class beside the diamond symbol are aggregates of objects at the other end of the association (although parts may be shared if the diamond is not filled in). Numbers at the end of an association indicate how many objects may have this association with instances of the class at the other end ('`0..*`' means "zero or more" and '`1..*`' means "one or more"). If an "ordered" constraint is present it means that the set of objects at that end of the association that are related to a common object at

the other end has an ordering relation on it. In implementation terms this means that the association end is represented by a list data structure within the class at the other end, rather than a set.

A solid line with a triangular arrowhead indicates a generalisation/specialisation relationship, with the arrow pointing to the more general class. A named 'lollipop' symbol attached to a class indicates that the class implements the named interface.

The dog-eared rectangle in Figure 2 represents a constraint on the class AtomicFormula. This is expressed using the Object Constraint Language (OCL) [14] which can be used in conjunction with UML in order to constrain the possible models of a specification in ways that cannot be achieved using the UML structural elements alone. In this case the constraint states that the number of Term objects associated with an AtomicFormula object must be equal to the value of the arity attribute of the associated Predicate object.

A similar rectangle can be used for notes attached to model objects. These give informal documentation about model elements.

It is important to note that an abstract syntax as shown in Figure 2 is not a substitute for semantics. It is still necessary to define the semantics of the structures modelled in these class diagrams, for example, the feasibility preconditions and the rational effect of the various communicative acts. We assume the standard FIPA semantics except for the following differences:

- Figure 1 includes a specialised message class `InformRef`. This is not the same as the FIPA ACL `inform-ref` *macro action*, which takes a single parameter—a definite description —and allows an agent to reason about how another agent should respond to a `query-ref` message, without knowing in advance the answer to the query. When (and if) that query is answered, it will be in the form of an `inform` message containing an equality statement that equates the query's definite description and the answer to the query (the name of some domain entity). However, requiring the use of the generic `inform` message type for this communication needlessly requires agents to have a notion of equality in order to be able to answer `query-ref` messages. The ACL in Figure 1 therefore adds a concrete `InformRef` message class that separately identifies the definite description and the returned answer instead of requiring the content language to relate these within an equality statement.

- The ACL also deviates from the FIPA ACL by allowing more than one ontology to be associated with a message and, to simplify the discussion, only allows a single message recipient to be named and ignores message envelope issues such as specific transport protocol addresses for agents.

Note also that Figure 2 only shows a partial model of an SL-like content language—it only includes the concepts necessary to express the example discussed below. A full version would include negation, disjunction, implication, action and belief expressions.

Figure 3 shows an example message in FIPA ACL and the corresponding representation as a UML object diagram (the "iota" expression in the upper part of the figure represents the definite description "the entity x that is the parent of Mary and is a doctor by profession").

```
(inform :sender agent1
        :receiver agent2
        :ontology (sequence (Family Professions People))
        :language FIPA-SL
        :content ((= (iota ?x (and (parent Mary ?x)
                                   (profession ?x doctor)))
                  Mavis))
)
```

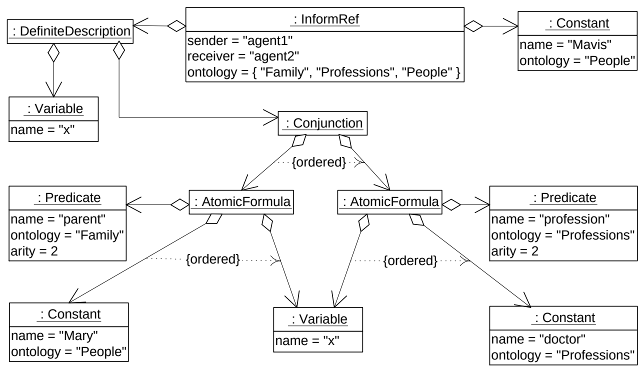

Figure 3: A message in FIPA syntax and its representation as a UML object diagram

In object diagrams, rectangles denote objects, specifying their class (after an optional name and a colon) and the object's attribute values. The lines connecting objects show *links*: instances of associations between classes. In Figure 3, where there was an ordered end to the corresponding association in Figure 2, an "ordered" constraint is shown between links to specify the ordering (this is not standard UML, but as of UML 1.3 no notation has been defined to convey this information).

For diagramming convenience, Figure 3 includes two separate variable objects, each having "x" as the value of the name attribute. One of these objects is shared by two AtomicFormula objects. In fact, it would make no difference if three separate variable instances were used or if one was referenced three times—the intended semantics of the content language in Figure 2 is that variables are identified by name, not by object identity.

3. BENEFITS OF AN ABSTRACT SYNTAX

The previous section discussed the advantages of UML abstract models of ACLs and content languages for clarifying the current specifications and the relationships between related languages. However, there are also more pragmatic advantages to this approach for language specification. High-level abstract models provide a common representation from which automated mappings can be defined to produce programming language bindings (which specify how to construct and, in some cases, interact with instances of the models from within a given programming language) and structured transport formats based on the Extensible Markup Language (XML) [13, 22] for serialising instances of models. While these automatically generated mappings may not produce results as elegant or concise as hand-crafted ones designed for specific problems, they can provide a standard and easy way to produce application code specialised to a new problem domain (in the case when the UML model represents a problem domain ontology).

In the case of agent communication languages, applying these techniques provides a way of rapidly defining new and extended languages and generating infrastructure code to support the use of those languages. For example, there are many interesting ways in which current agent communication languages could be extended: examples include extending the notion of reference to integrate object-oriented and World Wide Web technologies with agent systems, and incorporating social notions such as commitments and trust. To support and encourage experimentation with such extended languages it would be desirable to provide ways of customising existing agent platforms to provide application programmer interfaces allowing these languages to be used, while maintaining the rest of the agent platform infrastructure.

In order to increase the extensibility of agent platforms based on the proposed UML-based approach to language specification, the following technology is needed: (i) for each programming language of interest, the implementation of mappings from UML class diagrams to constructs of that programming language, and (ii) a marshalling framework that allows messages to be serialised to and from in-memory data structures. After determining that these technologies were not yet publicly available, such a mapping was implemented for the Java language using a XSLT stylesheet [20] applied to UML class diagrams[1] encoded in XMI [15] (version 1.0 for UML 1.3) documents—an export format supported by the CASE tools Argo/UML 0.8 and Rational Rose 2000 (with the Unisys XMI add-in). Another stylesheet was implemented to generate an RDF Schema document corresponding to a class diagram, and the Java binding was extended to include methods for marshalling and unmarshalling instances of the model to an instance of the RDF schema, using the RDF XML serialisation. This technology is not specific to agent communication—it can be used for any application where there are models represented as class diagrams. However, the next section will discuss some details of the implementation and illustrate its application to agent messaging.

4. XMI TO JAVA

Extensible Stylesheet Language Transformations (XSLT) [20] is a language for transforming XML documents into other documents. An XSLT stylesheet is composed of a set of templates that match nodes in the input document (represented internally as a tree) and transform them (possibly via the application of other templates) to produce an result tree. The result tree can then be output as text or as an HTML or XML document. Starting from an existing stylesheet for displaying class information from an XMI file as a table in HTML [16], this was first updated to work with XMI 1.0 files based on the UML 1.3 meta-model (the existing stylesheet was designed for UML 1.1). The resulting stylesheet was then converted and extended to produce a new stylesheet to produce Java source code corresponding to the model in the XMI files.

The mapping between UML and Java is mostly straightforward: classes and interfaces map to their equivalents in Java and attributes and associations map to Java fields, which may be implemented using an array or `java.util.Set` depending on the multiplicity and the presence or absence of an *ordered* constraint (which only applies to association ends in UML at present, not attributes). Currently it is assumed that the only primitive types in the UML model are the OCL types Boolean, Integer, Real and String, and these are mapped to the corresponding Java class types (with Real mapping to Double) instead of primitive types. This allows the possibility

[1]To be precise, the stylesheet is applied not to an encoding of the diagram itself, but to the model—the declarations of classes and the relationships between them—that is encoded in the XMI document.

of a null value for fields representing attributes or association ends with a multiplicity range that includes zero. Operations are declared in the corresponding interfaces and classes, and an empty body is generated within classes.

Of course, Java does not allow multiple inheritance and the stylesheet does not do any restructuring to avoid this — it passes multiple inheritance through for the Java compiler to detect.

Although Figures 1 and 2 do not show this, class diagrams often include *role names* on association ends. These are used as Java field names if they exist, otherwise default names are generated based on the OCL conventions for specifying navigation paths in class diagrams. In particular, the name of the class next to an association end is used as a default role name, with the initial letter in lowercase. Any ambiguity due to the lack of role names (e.g. in the case of unlabelled reflexive associations) is not detected and will be caught by the Java compiler when declarations of multiple fields with the same name are encountered. The visibility of attributes and association ends specified in the UML model is respected, but for the purposes of modelling agent languages it is expected that the designer will make these public.

Finally, the generated class, interface or field names are checked against a list of Java reserved words and an underscore character is prepended if necessary to avoid a Java compilation error.

A parameter to the stylesheet specifies whether class constructors should be generated. If this option is turned on, the generated constructors will contain parameters corresponding to all attributes and composition links related to the class (either inherited or declared in the class). Figure 4 illustrates the use of the generated constructors to build a message object within an agent, as well as the use of the marshalling method described in Section 6.

The stylesheet currently handles association classes but not qualified associations or attributes, associations or operations with class scope. Enumeration types are not supported, and there is also no mapping of UML namespaces to Java packages.

Figure 4 illustrates some advantages and disadvantages of the generated Java code for constructing messages. The example code presented shows how a message can be created and serialised using the generated Java classes. This is certainly longer than using a string-based message creation function with the FIPA ACL expression shown in the upper part of Figure 3 provided as a parameter. Also, the order of arguments is dictated by the mapping process. A fuller ACL model would include many optional message parameters, each of which would have a corresponding argument in the constructor. This would have to be given a null value if not required. These problems could be solved by providing an alternative keyword–value version of the constructor, and by providing additional input to the mapping process in the form of a binding schema, as is being investigated for the Java XML Data Binding facility under development by the Java Community Process [19].

The main advantage of the generated application programmer interface (API) used to create the message in Figure 4 is that it *is* generated automatically. It provides a strongly typed interface for constructing messages and thus helps to reduce errors. Also, messages are not necessarily composed in one step as depicted in this simple example in Figure 4. They may be constructed incrementally during an agent's computations, which would be more con-

```
x = new Variable("x");
m = new InformRef(
          "agent1", "agent2", // Sender and receiver
          Arrays.asList(new String[] { "Family", "Professions", "People" }), // Ontologies
          new DefiniteDescription(
                x,
                new Conjunction(
                      Arrays.asList(
                        new Wff[] { new AtomicFormula(
                                    new Predicate("parent", "Family", 2),
                                    Arrays.asList(
                                      new Term[] { new Constant("Mary", "People"), x })),
                                  new AtomicFormula(
                                    new Predicate("profession", "Professions", 2),
                                    Arrays.asList(
                                      new Term[] { x, new Constant("Mavis", "People")}))
                                }))),
          new Constant("Mavis", "People"));
ByteArrayOutputStream messageStream = new ByteArrayOutputStream();
MarshalHelper.marshalObjects(Collections.singleton(m), // All objects to be marshalled
                             m,                        // Root object
                             "http://some.namespace/for/message#",
                             messageStream);
```

Figure 4: Using the generated Java code to create and serialise a message

veniently and safely done using this API than by piecing together strings. In addition, a structured representation would be required as the output of a string-based parser anyway. This technique provides a starting point on top of which it should be possible to define more readable string-based representations. An interesting direction for future research is to investigate ways of mapping from a UML model to a string representation of a language, and to formally relate this to the corresponding object-oriented API.

5. XMI TO RDF SCHEMA

The XMI format was designed to allow the interchange of UML models. As such, it is based on an XML document type definition encoding concepts from the UML meta model, such as the concepts of class and association, and the instances in an XMI document correspond to particular classes and associations. As UML includes object diagrams, it would be possible to serialise objects within an XMI document, but the relationships between objects would be expressed as separate though interrelated elements describing objects, links and link ends. To achieve a more compact serialisation it is necessary to generate a document schema that is specialised to the particular model in a UML document. One option that has been investigated elsewhere is to generate an XML DTD from a UML class diagram [17] (although many features of class diagrams, including generalisation relationships, were not supported). Work has also been done on producing DTDs [5] or XML Schemas [9] from models expressed in ontology modelling languages, but experience from the latter work showed that the XML Schema notion of type inheritance does not correspond well to inheritance in object-oriented models.

For the marshalling framework described in this paper it was decided that mapping class diagrams to RDF schema specifications would be a simpler approach than using XML schema. The Resource Description Framework (RDF) [23] is a simple entity–attribute–value model designed for expressing metadata about resources. The RDF Schema is a set of predefined resources (entities with uniform resource identifiers) and relationships between them that define a simple meta model, including concepts of classes, properties, subclass and subproperty relationships, a primitive type *Literal*, bag, set and sequence types and domain and range constraints on properties.

In mapping from an XMI document to an RDF schema, it was not a goal to express all details of the model, only enough to facilitate serialisation of model instances. For example, the generated schema does not distinguish composition relationships from associations — this would require extensions to RDF Schema and is not required because knowledge of the relationship type is built into the marshalling code. If an agent needs access to a full description of a language it can access the XMI file directly using one of the available or forthcoming Java APIs for XMI [12, 18].

The XMI to RDFS stylesheet was developed by adapting and simplifying the one for XMI to Java. One issue that required addressing was the problem that RDF properties are first class entities—they are not defined relative to a class. Therefore a given property cannot be defined to have a particular range when applied to objects of one class and another range when applied to objects of a different class[2]. The option chosen in this work was to create properties with names of the form *Classname.FieldName* so that these were unique for each class.

Another issue was that RDFS has no notion of an interface. Instead, interfaces are modelled as classes, and realisation relationships between classes and interfaces are modelled as subclass relationships between classes.

The appendix shows how the example message from Figure 3 can

[2]Various solutions to this have been discussed in the www-rdf-interest mailing list (see thread beginning with the message http://lists.w3.org/Archives/Public/www-rdf-interest/2000Feb/0157.html).

be serialised with reference to the RDF schemas corresponding to the ACL and ContentLanguage models. Admittedly this is rather verbose compared to the original ACL message, but the RDF/XMI representation is designed for easy machine parsing rather than human readability.

Note that there is an XML namespace (e.g. http://nzdis.otago.ac.nz/0_1/ACL#) associated with each generated schema, and this must be provided as a parameter to the stylesheet.

Further discussion and examples of the mapping from UML to RDFS can be found in Reference 1.

6. MARSHALLING MESSAGES

Once the XMI to RDFS mapping was defined, the ability to generate marshalling and unmarshalling methods was added to the XMI to Java stylesheet. The aim in this code was to avoid the need to reflect on the model (by Java reflection or accessing the XMI or RDFS files). Instead the marshalling methods explicitly marshal each field in the class that corresponds to an attribute as well as fields corresponding to composition relationships. The actual serialisation to and from RDF is performed by a utility class that uses an existing Java RDF API [10].

The use of the serialisation mechanism is illustrated in Figure 4. The class `MarshalHelper` has a static method `marshalObjects` that takes four arguments: a collection of objects to include in the serialised object diagram, the object that is considered to be the 'root' or reference point of the diagram, an XML namespace for the RDF resources that will be defined in the RDF serialisation, and the output stream to which the serialised objects should be written. The method returns the Uniform Resource Identifier (URI) for the root object in the resulting XML document (this return value is not used in Figure 4).

7. CONCLUSION

This paper has focused on the use of UML to define agent communication and content languages, building on previous work investigating the use of UML as an ontology modelling language [2–4]. The use of a common modelling language for the ACL and content languages has helped to clarify the relationships between expressions in these two types of language. In particular, the FIPA SL notion of "entry points" to the SL grammar can be explained in terms of a requirement that content languages include concepts that are declared to implement 'marker' interfaces representing the key concepts of proposition, action description, definite description and reference in the ACL model. Further research is needed to study the relationship between content languages and ontologies. In particular, previously it has been proposed that ontology-specific content languages could be generated automatically to provide convenient and compact serialisation formats for encoding descriptions of domain objects within content language expressions [2]. It is important that any mechanism for this is well founded and based on a clear understanding of the relationships between the ACL, the ontology-specific content language and the ontology itself. It is intended to investigate this issue using a meta-modelling approach [4].

Further work is needed to provide a more convenient form of constructor for the generated Java classes. It would also be beneficial to provide a mechanism for defining mappings between an abstract language specification in UML and a string-based grammar. Such a mapping would require additional inputs defining formatting conventions for particular types of objects, sets and lists, etc. This is an important topic for further research as it would allow a single structured language specification to be used for both XML-based and traditional string-based encodings.

It is our hope to continue the present work in cooperation and with suggestions from FIPA on abstract syntaxes for the full FIPA ACL. The goal is also to prepare appropriate mappings for all content languages already defined by FIPA, in particular all SL-family languages. Once the abstract syntax specifications are finished, it would be desirable to publish all abstract models expressed as UML diagrams together with other related FIPA specifications. It would be feasible then to integrate the reference implementation discussed in this paper with existing FIPA platforms, thus providing developers with a flexible framework to experiment with possible extensions to the current ACL and content languages.

8. REFERENCES

[1] S. Cranefield. Networked knowledge representation and exchange using UML and RDF. *Journal of Digital Information*, 1(8), 2001. http://jodi.ecs.soton.ac.uk/.

[2] S. Cranefield and M. Purvis. UML as an ontology modelling language. In *Proceedings of the Workshop on Intelligent Information Integration, 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, 1999. http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-23/cranefield-ijcai99-iii.pdf.

[3] S. Cranefield and M. Purvis. Extending agent messaging to enable OO information exchange. In R. Trappl, editor, *Proceedings of the 2nd International Symposium "From Agent Theory to Agent Implementation" (AT2AI-2) at the 5th European Meeting on Cybernetics and Systems Research (EMCSR 2000)*, Vienna, 2000. Austrian Society for Cybernetic Studies. Published under the title "Cybernetics and Systems 2000". An earlier version is available at http://www.otago.ac.nz/informationscience/publctns/complete/papers/dp2000-07.pdf.gz.

[4] S. Cranefield, M. Purvis, and M. Nowostawski. Is it an ontology or an abstract syntax? – Modelling objects, knowledge and agent messages. In *Proceedings of the Workshop on Applications of Ontologies and Problem-Solving Methods*, pages 16.1–16.4, 2000. http://delicias.dia.fi.upm.es/WORKSHOP/ECAI00/16.pdf.

[5] M. Erdmann and R. Studer. Ontologies as conceptual models for XML documents. In *Proceedings of the 12th Workshop on Knowledge Acquisition, Modeling and Management (KAW'99)*. Knowledge Science Institute, University of Calgary, 1999. http://sern.ucalgary.ca/KSI/KAW/KAW99/papers/Erdmann1/erdmann.pdf.

[6] T. Finin, Y. Labrou, and J. Mayfield. KQML as an agent communication language. In J. M. Bradshaw, editor, *Software Agents*. MIT Press, 1997. Also available at http://www.cs.umbc.edu/kqml/papers/kqmlacl.pdf.

[7] FIPA. Foundation for Intelligent Physical Agents Web pages. http://www.fipa.org/, 2001.

[8] M. R. Genesereth and S. P. Ketchpel. Software agents. *Communications of the ACM*, 37(7), July 1994.

[9] M. Klein, D. Fensel, F. van Harmelen, and I. Horrocks. The Relation between Ontologies and Schema-languages: Translating OIL-specifications in XML-Schema. In *Proceedings of the Workshop on Applications of Ontologies and Problem-Solving Methods*, 2000. http://delicias.dia.fi.upm.es/WORKSHOP/ECAI00/7.pdf.

[10] S. Melnik. RDF Java API project Web page. http://www-db.stanford.edu/~melnik/rdf/api.html, 2000.

[11] National Committee for Information Technology Standards. Draft proposed American national standard for Knowledge Interchange Format. http://logic.stanford.edu/kif/dpans.html, 1998.

[12] Novosoft. Novosoft UML Library for Java. http://sourceforge.net/projects/nsuml/, 2000.

[13] OASIS. The XML Cover pages. Organization for the Advancement of Structured Information Standards (OASIS) Web site at http://www.oasis-open.org/cover/xml.html, 2000.

[14] Object Management Group. OMG Unified Modeling Language Specification, version 1.3. http://www.omg.org/technology/documents/formal/unified_modeling_language.htm, 2000.

[15] Object Management Group. XML Metadata Interchange (XMI) Specification. http://www.omg.org/technology/documents/formal/xml_metadata_interchange.htm, 2000.

[16] Objects by Design. Transforming XMI to HTML. Web site at http://www.objectsbydesign.com/projects/xmi_to_html.html, 2000.

[17] D. Skogan. UML as a schema language for XML based data interchange. In *Proceedings of the 2nd International Conference on The Unified Modeling Language (UML'99)*, 1999. http://www.ifi.uio.no/~davids/papers/Uml2Xml.pdf.

[18] Sun Microsystems. JSR #000040: Metadata API specification. Java Community Process JSR, 1999. http://java.sun.com/aboutJava/communityprocess/jsr/jsr_040_mof.html.

[19] T. Sundsted. Adelard, one year later. Available online at http://www.javaworld.com/javaworld/javaone00/j1-00-adelard.html, 2000.

[20] World Wide Web Consortium. XSL Transformations (XSLT) specification version 1.0. http://www.w3.org/TR/xslt, 1999.

[21] World Wide Web Consortium. Resource Description Framework (RDF) Schema Specification 1.0. http://www.w3.org/TR/2000/CR-rdf-schema-20000327/, 2000.

[22] World Wide Web Consortium. Extensible Markup Language (XML) Web pages. http://www.w3c.org/xml, 2001.

[23] World Wide Web Consortium. Resource Description Framework (RDF) Web pages. http://www.w3c.org/RDF/, 2001.

APPENDIX

The following is an encoding of the example message shown in Figure 3 in terms of the RDF schemas generated from the class diagrams in Figures 1 and 2. This is not precisely the format produced by the marshaller (via Melnik's RDF API [10]), but has been converted into a more compact form for ease of reading. However, this is equivalent to the original apart from the omission of URIs for all the objects.

```
<rdf:RDF xml:lang="en"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:acl="http://nzdis.otago.ac.nz/0_1/ACL#"
  xmlns="http://nzdis.otago.ac.nz/0_1/SL#">

<acl:InformRef>
  <acl:Message.sender>agent1</acl:Message.sender>
  <acl:Message.receiver>agent2</acl:Message.receiver>
  <acl:Message.ontology>
    <rdf:Seq>
      <rdf:li>Family</rdf:li>
      <rdf:li>Professions</rdf:li>
      <rdf:li>People</rdf:li>
    </rdf:Seq>
  </acl:Message.ontology>
  <acl:InformRef.definiteDescription>
    <DefiniteDescription>
      <DefiniteDescription.variable>
        <Variable Variable.name="x"/>
      </DefiniteDescription.variable>
      <DefiniteDescription.wff>
        <Conjunction>
          <Conjunction.wff>
            <rdf:Seq>
              <rdf:li>
                <AtomicFormula>
                  <AtomicFormula.predicate>
                    <Predicate
                      Predicate.name="parent"
                      Predicate.ontology="Family"
                      Predicate.arity="2"/>
                  </AtomicFormula.predicate>
                  <AtomicFormula.term>
                    <rdf:Seq>
                      <rdf:li>
                        <Constant
                          Constant.name="Mary"
                          Constant.ontology="People"/>
                      </rdf:li>
                      <rdf:li>
                        <Variable Variable.name="x"/>
                      </rdf:li>
                    </rdf:Seq>
                  </AtomicFormula.term>
                </AtomicFormula>
              </rdf:li>
              <rdf:li>
                <AtomicFormula>
                  <AtomicFormula.predicate>
                    <Predicate
                      Predicate.name="profession"
                      Predicate.ontology="Professions"
                      Predicate.arity="2"/>
                  </AtomicFormula.predicate>
                  <AtomicFormula.term>
                    <rdf:Seq>
                      <rdf:li>
                        <Variable Variable.name="x"/>
                      </rdf:li>
                      <rdf:li>
                        <Constant
                          Constant.name="doctor"
                          Constant.ontology="Professions"/>
                      </rdf:li>
                    </rdf:Seq>
                  </AtomicFormula.term>
                </AtomicFormula>
              </rdf:li>
            </rdf:Seq>
          </Conjunction.wff>
        </Conjunction>
      </DefiniteDescription.wff>
    </DefiniteDescription>
  </acl:InformRef.definiteDescription>
  <acl:InformRef.reference>
    <Constant
      Constant.name="Mavis"
      Constant.ontology="People"/>
  </acl:InformRef.reference>
</acl:InformRef>

</rdf:RDF>
```