From DAML-S to Executable Code

Sérgio Gaio
Communicating Intelligent Systems Group of ADETTI
Av. das Forças Armadas, Edifício ISCTE, 1600 Lisboa, Portugal
+351217801428
sergio.gaio@iscte.pt

António Lopes
Communicating Intelligent Systems Group of ADETTI
Av. das Forças Armadas, Edifício ISCTE, 1600 Lisboa, Portugal
+351217801428
antonio.luis@iscte.pt

Luis Botelho
Communicating Intelligent Systems Group of ADETTI
Av. das Forças Armadas, Edifício ISCTE, 1600 Lisboa, Portugal
+351217801428
luis.botelho@iscte.pt

ABSTRACT

This paper analyses the specification of agent control and agent knowledge in DAML-S and its conversion into executable code. Instead of the usual XML syntax, we propose a S-Expression syntax of DAML-S, which facilitates introducing two extensions in DAML-S specification: concrete definitions of concepts used in DAML-S service descriptions, which were not defined in the DAML-S specification; and logic-programming constructs that may be used in service description. Two approaches are discussed with respect to the generation of executable code. Both of them contain a first step in which the DAML-S service-description is parsed into an appropriate computer program data structure, called the DAML-S description parse tree. The first approach converts the DAML-S description parse tree into source files that must be compiled and linked in order to create the executable agent. The second approach relies on the run-time interpretation of the DAML-S description parse tree.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: Computer-Aided-Software-Engineering

General Terms

Algorithms, Design, Languages.

Keywords

Agent Specification, DAML-S and Agent Control

1. INTRODUCTION

The Pagoda of Creation [4][5] is a system, being developed within the Agentcities project [9], to help users create Personal Assistants for agent network applications. Personal Assistants are generated from a library of agent templates. Agent templates are high-level agent specifications consisting of four sections: the domain ontology, the agent control section, the decision knowledge section, and the agent interface definition section.

The domain ontology section describes the entities of the domain that the agent might have to deal with. The agent control section specifies the general flow of control of the execution. The control section contains decision points in which the agent must select one of a set of alternative courses of action. The decision knowledge section specifies the knowledge to be used by the agent in order to select one alternative course of action among the several specified in each decision point of the control section. Finally, the agent interface definition section specifies the interface between the user and the agent, which is based on the entities described in the domain ontology.

This paper presents the approach taken to represent the control and knowledge sections of the agent templates.

In [4], we proposed to use a textual form of AUML [8] diagrams to represent the agent control section, and Prolog to represent the knowledge section. However two reasons made us change our mind. First, the Agentcities project decided to use DAML-S [2] for service description and DAML+OIL for ontology representation. Secondly, we don't have a standard convenient textual notation for AUML diagrams – XMI would have to be modified since it is a textual notation for UML not AUML. Hence, we have decided to use DAML-S instead of AUML, to represent both, control and knowledge sections of the agent template. However, the experience reported in the paper shows that AUML would probably lead to higher-level specifications which would possibly constitute an advantage.

In order to use DAML-S in the agent control and knowledge sections of agent templates, DAML-S had to be extended. First, some elements of the DAML-S specification were under-specified. In the DAML-S original specification, the conditions (i.e., the guards) and the actions of the control constructs were unconstrained strings. However, the condition of a control construct must be a proposition or a Boolean expression (not any string); and the action of a control constraint must also be an action expression (not any string). Section 2.2 describes our proposal regarding conditions and actions of control-constructs.

Although very rich in terms of process control constructs (e.g., *while*, *if-then-else*, *split*, *sequence*), DAML-S adopted mainly a procedure-centred approach to software design. However, there are other important trends in the software industry, mainly in the agent engineering, notably the logic-programming paradigm and the object-oriented paradigm. In this view, we have extended the DAML-S to allow agent designers to adopt object-oriented and logic-programming approaches to software development, besides the procedure-centred and the concurrent paradigms already accommodated by the DAML-S original specification (see section 2.3).

Since the current proposal extends DAML-S with conditions and action descriptions, we have to define the most convenient way to express those new elements of the specification. First, although it has been proposed by others [6], mark-up languages such as XML are not suitable for logic expression representation

– they result in long difficult to read specifications. Secondly, the Agentcities project adopted the S-Expression syntax for agent communication. Third, there are already well-known concise and readable S-Expression representations of logic propositions and action descriptions. Given the above three reasons, we decided to propose a S-Expression syntax for the extended DAML-S. The adopted syntax is inspired in KIF [7] and FIPA-SL [3] (extended as in [1]) languages (see section 2.1).

2. EXTENDED DAML-S SPECIFICATION

DAML-S is a DAML description language, which is used to describe the properties and capabilities of Web services, using a mark-up language. The purpose of DAML-S is to facilitate the discovery, execution, composition and interoperation of WEB-based services.

Our approach uses DAML-S language to describe the agents control section creating a hierarchic structure that corresponds to the agent internal behaviour. Basically, a DAML-S specification is composed by 3 major objects: *ServiceProfile* - which describes what the service does; *ServiceModel* – which defines how the service works; and *ServiceGrounding* – which describes how we access the service. We use the class ServiceModel of DAML-S specifications and the hierarchy of its sub classes (e.g., simpleProcess, compositeProcess, If-Then-Else, atomicProcess) to describe the agent control section. In order to describe the knowledge section of the agent template, we have extended DAML-S with the capability to define predicates (see section 2.3).

2.1 S-Expression syntax of DAML-S

A DAML-S description is a set of related instances of DAML-S service description classes. Here, we briefly describe the representation of generic objects using the S-Expression syntax. The basic idea is to represent an object (i.e., an instance of a certain class), as a functional expression in which the functor is the name of the class, and the arguments are the names and values of the attributes of the object.

```
(If-Then-Else
  :if-condition (< n 10)
  :then (SimpleProcess
    :effect  (Print "Small"))
  :else (SimpleProcess
    :effect  (Print "Large")))
```

The above expression represents an instance of the DAML-S If-Then-Else class. This class has two mandatory attributes (*if-condition* and *then*), and a facultative attribute (*else*). The object described in the above expression has all the class attributes. The value of the *if-condition* attribute is the condition "(< n 10)". The value of the *then* attribute is an instance of the DAML-S *SimpleProcess* class. Finally, the value of the *then* attribute is another instance of the *SimpleProcess* DAML-S class.

In logic terms, an instance of a class such as the above description is a term. The values of the object attributes are also terms. Informally, it is easy to see that the above expression is the specification of a process that prints the string "Small" if n is less than 10; otherwise, it prints the string "Large".

2.2 Conditions and actions

Since the conditions and actions appearing in process control constructs are the values of attributes of the class representing the control constructs, syntactically they must be terms. Therefore, we need to represent conditions through Boolean functional terms. As a result, the conditions of control constructs have exactly the same structure of FIPA-SL propositions but, formally, they are represented by functional expressions. Consequently, we must treat logical connectives, quantifiers and relational operators as if they were Boolean functional symbols.

As discussed in [1] it is possible to represent object-oriented specifications using four new operators: *instance*, *value*, *apply*, and *execute*. Originally, *instance* is a relational operator. In this proposal it has to be a Boolean functional symbol. In the original proposal, *value* is a functional symbol therefore it does not have to be reified. *Apply* and *execute* are action operators therefore they are not used in the conditions of control constructs.

```
(and (> (value employee salary)  1000) (<
(value employee salary) 2000))
```

The above expression is a functional term representing the condition "the salary of the employee is between 1000 and 2000 Euros (European Union Currency)". (value *Object Attribute*) is a functional term representing the value of the specified attribute of the specified object. In the above case, it represents the salary of a specific employee.

In this proposal, action descriptions are represented by SL action designators, which have exactly the same structure as functional expressions. Besides application-dependent actions defined by the agent designer, we propose some domain independent action operators: *assign*, *print*, *read*, *apply* and *execute*. The last two were introduced in [1].

```
(sequence (simpleProcess :effect (assign n
(+ n 1))) (simpleProcess :effect (apply
(nth n messages) sendItself (sequence
receiver))))
```

The above expression is a sequence of two actions. First, the value of n is increased. Then the method *sendItself* is applied to the nth element of the list *messages* (which is a message) taking the receiver as an argument. The ontology of the domain (domain ontology section of the agent template) must specify the class *Message*, which, among other things, has the method *sendItself* taking the receiver as an argument. Notice that, although the method *sendItself* takes only one argument (the receiver of the message), the general specification of the *apply* operator is composed of the object to which the method is applied, the name of the method to be applied and a sequence of the arguments to be passed to the method.

2.3 Predicate definition and use

With the extended DAML-S language, it is possible to describe processes that define invocable predicates. This is the way we describe the knowledge section of the agent template. This can be done through the class *PredicateDefinition*. It allows creating a predicate with several arguments and a set of clauses representing facts and rules. An example will be explained in section 2.4.

In order to use predicates in the agent control structure, it is necessary to be capable of accessing all possible solutions of the predicate. Therefore, we decided to extend this DAML-S language with a mechanism that allows iterating through all possible solutions of a predicate invocation.

Three process control constructs were created to fulfil this objective: *next_solution, init_iterator* and *number_of_solutions*.

`next_solution` - returns *true* if it is possible to get a solution, and *false* otherwise. As a side effect, another solution of the predicate is provided, that is, the variables used in the interface with the predicate are instantiated with new values.

```
(next_solution
  :solution_generator <predicate
invocation>)
```

next_solution generates the next solution of a pre-initialised predicate. This initialisation is made through the use of the *init_iterator* operator:

```
(init_iterator
  :solution_generator <predicate
invocation>)
```

`number_of_solutions` – returns an Integer representing the number of solutions of an invocable predicate

```
(number_of_solutions
  :solution_generator <predicate
invocation>)
```

2.4 Specification Example

In this section a few examples will be given to explain the definition of predicates using the extended DAML-S. First, we will present a predicate definition with only a few facts, using the *PredicateDefinition* class.

Figure 1 represents an instance of the extended DAML-S class *PredicateDefinition*, contained in the knowledge section of the agent template, that defines an invocable predicate named *pub* with two arguments: *pubName* and *city*, both of type string. The value of parameter *instantiation* in both of those arguments is *any*, which means that they can be instantiated or not instantiated when they are passed on to the predicate.

```
(PredicateDefinition
  :name "pub"
  :invocable "True"
  :arguments (sequence
    (parameter
      :name pubName
      :restrictedTo string
      :instantiation any)
    (parameter
      :name city
      :restrictedTo string
      :instantiation any)
  )
  :clauses (set
      (pub "Charlie Shots" "Lisbon")
      (pub "Blue Lizard" "Lausanne")))
```

Figure 1 - "pub" Predicate Definition

This definition works more or less as a table in a relational database. In this case, it defines two instances of the predicate: one pub is named “Charlie Shots” and is located in Lisbon; the other is the famous “Blue Lizard” in Lausanne. It is worth noting that the data types and classes used in the definition must be specified in the domain ontology section of the agent template or else they must be pre defined types such as “string”.

```
(PredicateDefinition
  :name "LisbonNightPlace"
  :invocable "True"
  :arguments
    (parameter
     :name place
     :restrictedTo string
     :instantiation any)
  :clauses
    (forall ?x (implies (or
        (pub ?x "Lisbon")
        (fadoPlace ?x "Lisbon"))
      (LisbonNightPlace ?x))))
```

Figure 2 - "LisbonNightPlace" Predicate Definition

Figure 2 represents an instance of the class *PredicateDefinition* that describes another predicate, also contained in the knowledge section of the agent template, referring the *pub* predicate defined in Figure 1. In the above description, there is only one clause defining the predicate *LisbonNightPlace*. This clause is expressed in the syntax of the FIPA-SL content language. The informal reading is “pubs or *fado* places located in Lisbon are Lisbon night places”.

```
 (compositeProcess
  :name "PrintLisbonNightPlaces"
  :participants (set
    (parameter :name place :restrictedTo
string)
  )
  :invocable "True"
  :composedOf
   (sequence
    (SimpleProcess :effect (init_iterator
      :solution_generator
        (AtomicProcess
          :name LisbonNightPlace
          :parameter place
        )))
    (Repeat-While
      :while-Condition (next_solution
      :solution_generator
        (AtomicProcess
          :name LisbonNightPlace
          :parameter place
        ))
      :While-Process
        (simpleProcess :effect (print
place))
        )))
```

Figure 3 - "PrintLisbonNightPlaces" Procedure Definition

Figure 3 defines a procedure that uses the predicate *LisbonNightPlace* just defined. Since *LisbonNightPlace* is used within a procedural process, it must be used with a solution iterator. This procedure is placed in the control section of the agent template.

PrintLisbonNightPlaces uses the processes explained earlier: *init_iterator* and *next_solution*. The procedure is a sequence of two steps. The first step initialises the iterator with the predicate *LisbonNightPlace*, using the process *init_iterator*. The second step is a while loop that prints all Lisbon night places by using the process *next_solution*. Lisbon night places are the multiple solutions of the predicate *LisbonNightPlace*.

Since place has not been previously instantiated, it can be used to receive the predicate solutions. The atomic process *print* has also been added to the original DAML-S specification.

It is worth noting that our proposal creates a framework in which logic or declarative paradigm is integrated with other paradigms such as the procedure-centred, the object-oriented and the concurrent models of computation.

3. MAPPING DAML-S INTO COMPUTER CODE

Using the proposed DAML-S extension, it is possible to define a complete control section for an agent, which will then be converted into the control structure of a running agent. It is also possible to describe the knowledge section of an agent, by internally defining all the predicates that the agent will have access to.

This section considers two different approaches for the generation of the executable agent from the extended DAML-S specification. Both of them contain a first step in which the DAML-S service-description is parsed into an appropriate computer program data structure, called the DAML-S description parse tree.

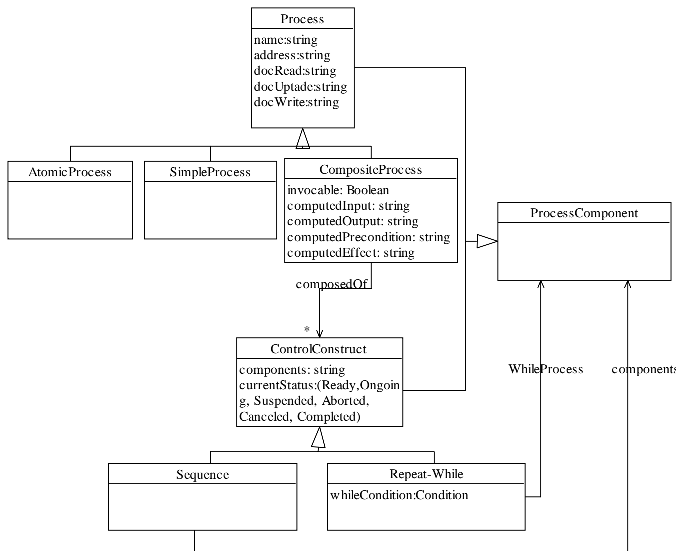

Figure 4 - DAML-S class diagram

The first approach converts the DAML-S description parse tree into a set of source files that must be compiled and linked in order to create the executable agent. The second approach relies on the run-time interpretation of the DAML-S specs parse tree.

We use the tools Lex and Bison, to parse DAML-S. It allows us to transform the text syntax into C++ objects. In order to do that we built a class diagram that has all the DAML-S classes together with the proposed extensions (see section 2). Figure 4 represents a subset of the actual DAML-S class diagram.

After parsing the extended DAML-S descriptions, it is possible to generate executable code in any language. The *pub* and *lisbonNightPlace* predicates described in section 2.4 can be converted into Prolog code as showed in Figure 5. Obviously, the same specification could, as easily, be converted into Java or C++ code.

```
pub('charlie Shot','Lisbon').
pub('Blue Lizard','Lausanne').
lisbonNightPlace(X) :-
   pub(X,'Lisbon'); fadoPlace(X,'Lisbon').
printLisbonNightPlaces:-
   lisbonNightPlace(X),
   write(X), nl, fail.
printLisbonNightPlaces.
```

Figure 5 - Example of generated code in Prolog

Instead of generating code that can be compiled and executed, another approach consists of using DAML-S as an interpreted programming language. We have decided not to use this last alternative because it would entail to create a fully implemented DAML-S interpreter capable of calling executable code from other programming languages, since there are already several agent building blocks in other programming languages.

4. CONCLUSIONS AND FUTURE WORK

We have presented an extension of DAML-S with new features that allow its use for service description following any of the most used paradigms of software development: the procedure-centred paradigm, the object-oriented paradigm, the logic-programming paradigm and the distributed paradigm. We have also described two alternative approaches to generating executable programs from extended DAML-S specifications.

Unfortunately, the approach has also an important disadvantage. We would have liked agent templates to be very high-level agent specifications. However, the extended DAML-S control descriptions are as low level as any programming language such as C++ or Java. Therefore, the next step is to improve the approach so that higher-level specifications can be used. The easiest way to go about this is by defining a library of more complex building blocks. The other possibility is to develop algorithms that can generate programs from action descriptions and goal specifications. This more sophisticated approach would enable totally declarative agent control and knowledge specifications.

We will evaluate the possibility of using the proposal presented in this paper as an abstract neutral programming language capable of generating code in several concrete programming languages, such as C++, Java, Lisp and Prolog. In order for this to be possible, it will suffice to develop a set of code generators

from the DAML-S Description Parse Tree to the different programming languages. The success of this future step will empower the software development capabilities of R&D teams, because it allows developing sharable, re-usable software, in spite of possible constraints regarding the specific programming language to be used in each project. This step will also enable furthering and evaluating the DAML-S specification.

5. ACKNOWLEDGEMENTS

The research described in this paper is partly supported by UNIDE/ISCTE and partly by the EC project Agentcities.RTD, reference IST-2000-28385. The opinions expressed in this paper are those of the authors and are not necessarily those of the Agentcities.RTD partners.

6. REFERENCES

[1] Botelho, L.M.; Antunes, N.; Ebrahim, M.S.; and Ramos, P.N. “Greeks and Trojans Together”. Proc. of the Workshop “Ontologies in Agent Systems” of the first Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS2002)

[2] DARPA Agent Markup Language. DAML-S 0.6 Draft Release. 2001. http://www.daml.org/services/daml-s/2001/10/

[3] Foundation for Intelligent Physical Agents. FIPA SL Content Language Specification. Specification Document XC00008G. 2001

[4] Lopes, A.L.; Gaio, S.; and Botelho, L.M. “Personal access to a worldwide agent network”. Proc. of the First International Joint Conference in Autonomous Agents and Multi-Agent Systems (AAMAS 2002). Forthcoming. 2002

[5] Lopes, A.L.; Gaio, S.; and Botelho, L.M. “The Pagoda of Creation”. ADETTI Internal Document. http://www.adetti-linha4.org/papers/

[6] McDermott, D.; and Dou, D. 2002. Representing Disjunction and Quantifiers in RDF. In Proc. of the Semantic Web Conference. Forthcoming. 2002

[7] National Committee for Information Technology Standards. Knowledge Interchange Format: Draft proposed American National Standards”. Technical Report NCITS.T2/98-004. 1998. http://logic.stanford.edu/kif/dpans.html

[8] Odell, J.J.; Parunak, H.D.; and Bauer, B. Representing Agent Interaction Protocols in UML. In Paolo Ciancarini and Michael Wooldridge (eds) Agent-Oriented Software Engineering. Springer, 2001. pp.121-140

[9] Willmott, S.; Dale, J.; Burg, B.; Charlton, P; and O'Brien, P. “Agentcities: a worldwide open agent network”. Agentlink News, 2001, 8:13-1