Transforming information in RDF to rewriting logic*

Alberto Verdejo[1], Narciso Martí-Oliet[1], Tomás Robles[2], Joaquín Salvachúa[2], Luis Llana[1], and Margarita Bradley[1]

[1] Universidad Complutense de Madrid, {alberto,narciso,llana,bradley}@sip.ucm.es
[2] Technical University of Madrid, {robles,jsr}@dit.upm.es

Abstract. RDF looks like the first step to build the Semantic Web vision. Our long-term goal is to have a sound way to verify and validate the semantic web interactions that applications and agents may develop in a distributed environment. The first step for reaching this goal is to provide a useful semantic support to RDF itself. Based on this formal support, properties may be analyzed, as well as transformations and verifications can be performed. In this paper we propose an *intuitive* and *formal* semantics for RDF by means of a translation of RDF documents into *executable* object-oriented modules in the formal language Maude. This translation provides a semantics for RDF documents and allows programs managing them to be expressed in the same formalism, since Maude specifications are executable. Moreover, due to the reflective features of Maude, this translation can be implemented in Maude itself. Finally, translated RDF documents are integrated in an agent application written in Mobile Maude, that is, the same framework is used for both translating RDF documents and expressing the programs that manipulate them.
Keywords: RDF, Semantic Web, formal methods, rewriting logic, Maude.

1 Introduction

The current human-centered web is still largely encoded in HTML. Over the past few years, XML has been proposed as an alternative encoding which is intended also for efficient machine processing. It has become the standard for the exchange of information on the Internet. However, it is not a final solution because it only gives support for syntactic representation of information, but not for its meaning. RDF (Resource Description Framework) [12] and RDFS (RDF Schema) [5] represent an attempt to resolve these deficiencies by building on top of XML, although they are still a bit limited for knowledge representation.

* Research supported by MCyT projects *AMEVA: Desarrollo Formal de Sistemas Basados en Agentes Móviles* (TIC2000–0701–C02–01) and *MIDAS: Metalenguajes para el diseño y análisis integrado de sistemas móviles y distribuidos* (TIC2003–01000).

Tim Berners-Lee conceives the Semantic Web as a layered architecture [3]. At the lowest level RDF provides a simple data model and a standardized syntax for *metadata* (data about data) about web resources by providing the language for writing down factual statements. The next layer is the *schema* layer where the definition of concrete vocabularies is given by means of the RDF Schema language. The final layer is the *logical* layer given by a formal knowledge representation language. It is important that each layer is an extension of RDF.

Our long-term goal is to have a sound way to verify and validate the Semantic Web *interactions* that service agents may develop in a distributed environment, for example as part of web services management. This will enable the possibility of reasoning about the information that is being exchanged allowing all the involved partners to have a common understanding. One first step for reaching this goal is to provide a *formal*, *intuitive*, and *executable* semantics to RDF and RDFS. The "official" model-theoretic semantics for RDF and RDFS is presented in [11] (more on this at the end of next section).

We propose in this paper an alternative semantic support to RDF by means of Maude, which is a formal language based on a first-order rewriting logic [7, 16] with well-defined syntax, formal models, and corresponding soundness and completeness theorems. Maude provides an executable language integrated in a global framework including functional elements and concurrency facilities. Using these facilities mobile agents and other advanced elements may be managed as natural elements into the logic of Maude. Maude includes in the same declarative framework both *logic* and *control*, which is a key difference with respect to other logic-based languages. We use the language Maude for:

- giving semantics to RDF documents by translating them into executable object-oriented Maude modules;
- implementing this translation; and
- implementing the applications that make use of the translated documents.

Under this formalized approach, RDF documents can be easily translated into Maude modules and therefore they may be data for Maude applications, as we show in Sections 4 and 5. Translating RDF documents into Maude allows their integration with web agents also defined in Maude. Hence, the development of web agents and their behavior is fully integrated and formally defined in a simple framework, as we will see in Section 5. One of the key features of our approach is that it is object-oriented, so the full power of object-orientation is supported, including inheritance.

In Section 2 RDF and RDFS are briefly introduced. In Section 3 the language Maude is presented by showing its syntax and key features. We pay special attention to object-oriented Maude modules. In Section 4 we describe the proposed translation that provides a semantics for RDF and RDFS documents, and how this translation is implemented by using Maude itself. A case study is presented in Section 5, where the translation is used by agents in a mobile system. We conclude with some comments on future work in Section 6.

2 RDF and RDFS: syntax and semantics

The Resource Description Framework (RDF) [19] is a general-purpose language for representing information in the World Wide Web. It provides a common framework for expressing this information in such a way that it can be exchanged between applications without loss of meaning, by providing a simple way to state properties of *web resources*, that is, objects that are uniquely identifiable by a Uniform Resource Identifier (URI) [18].

RDF is based on the idea that the things we want to describe have properties which have values (which can be literals or other resources), and that resources can be described by making statements that specify those properties and values. A statement has three components: a specific resource (subject), a property (predicate), and the value of this property for that resource (object). A collection of these statements that refers to the same resource is called a *description.* A concrete machine-readable syntax using XML is defined in [12]. For example, the following RDF document describes a laser printer:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:ps="http://printers.org/schema/">
  <ps:LaserPrinter about="http://HPprinters/HPLaserJet1100" >
     <ps:PrinterTechnology>Laser Jet</ps:PrinterTechnology>
     <ps:PrinterResolution>600 dpi</ps:PrinterResolution>
     <ps:Price>399</ps:Price>
  </ps:LaserPrinter>
</rdf:RDF>
```

In order to uniquely identify properties and their meaning, RDF uses the XML namespaces mechanism [4]. Meaning in RDF is expressed through reference to a *schema* (see below). For example, in the previous example the resource `LaserPrinter` and the property `Price` are imported from `http://printers.org/schema`. The statements in a `Description` refer to the resource determined from the `about` attribute (interpreted as a URI).

Two important RDF concepts are *containers*, used to hold collections of resources, and *reification*, used for making statements about other statements (for a more detailed explanation of these two concepts we refer to [12]).

RDF user communities require the ability to say certain things about certain kinds of resources. The declaration of these properties (attributes) and kinds of resources (classes) is done by means of an *RDF schema* (RDFS) [5]. This mechanism provides a basic *type system* for use in RDF models. Instead of defining a class in terms of the properties its instances may have, an RDF schema will define properties in terms of the classes of resources to which they apply.

The following RDFS document[1] describes a class of printers with a subclass of laser printers, and a printer property, namely, its price (the rest of properties could be defined in the same way):

[1] We use `&rdfsns;` as an abbreviation of `http://www.w3.org/2000/01/rdf-schema`.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
  <rdfs:Class rdf:ID="Printer">
    <rdfs:subClassOf rdf:resource="&rdfsns;#Resource"/>
  </rdfs:Class>
  <rdfs:Class rdf:ID="LaserPrinter">
    <rdfs:subClassOf rdf:resource="#Printer"/>
  </rdfs:Class>
  <rdfs:Property rdf:ID="Price">
    <rdfs:domain rdf:resource="#Printer"/>
    <rdfs:range rdf:resource="&rdfsns;#Literal"/>
  </rdfs:Property>
  ...
</rdf:RDF>
```

The `domain` property is used to indicate the class on whose members a property can be used. The `range` property is used to indicate the class that the values of a property must be members of.

In [11] a model-theoretic semantics for RDF and RDFS is presented. The semantic definition translates an RDF graph into a logical expression "with the same meaning." Basically, a graph arc is mapped to an atomic assertion and the complete graph is mapped to the existential closure of the conjunction of the translations of all the arcs in the graph. Also a notion of entailment in RDF is studied. A similar approach is followed in [10] where an axiomatization for RDF is provided by specifiying a mapping of a set of descriptions into a logical theory expressed in first-order predicate calculus. This translation not only specifies the intended meaning of the descriptions, but also produces a representation of the descriptions from which inferences can automatically be made using traditional automatic theorem provers and problem solvers. Although these approaches have different important strengths of their own, they are not well suited for directly executing a system.

In our approach we translate into a *formal language*, but where the translations can be *executed*. So we gain both the advantages of moving into a formal world where properties can be formally verified, and the advantages of being able to implement *prototypes* with which we get confidence of our systems specifications and implementations. The fact of having executable specifications is important not from the point of view of RDF documents that specify data, but from using the same framework both for translating RDF documents and for expressing the programs that manipulate them.

3 Rewriting logic and Maude

Maude [7] is a high level, general purpose language and high performance system based on rewriting logic [16], a logic *of change* in which deduction directly corresponds to the change [13]. Among the advantages of rewriting logic, we may emphasize the following:

- *It has a simple formalism*, with only a few rules of deduction that are easy to understand and justify;
- *It is very flexible and expressive*, capable of representing change in systems with very different structure;
- *It allows user-definable syntax*, with complete freedom to choose the operators and structural properties appropriate for each problem;
- *It is intrinsically concurrent*, representing concurrent change and supporting reasoning about such change;
- *It supports modelling of concurrent object-oriented systems* in a simple and direct way;
- *It has a semantics based on initial models* that support a "no junk, no confusion" version of the closed world assumption;
- *It is realizable in a wide spectrum logical language* (*Maude*) supporting executable specification and programming.

In rewriting logic the state of a system is formally specified as an algebraic data type by means of an equational specification. In this kind of specifications we can define new types (by means of keyword `sort(s)`); subtype relations between types (`subsort`); operators (`op`) for building values of these types, giving the types of their arguments and result, and which may have attributes such as being associative (`assoc`) or commutative (`comm`), for example; and equations (`eq`) that identify terms built with these operators. The following *functional* module (with syntax `fmod...endfm`) defines the natural numbers with an addition operation:

```
fmod NAT is
  sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op _+_ : Nat Nat -> Nat [assoc comm].
  vars N M : Nat .
  eq 0 + N = N .
  eq s(N) + M = s(N + M) .
endfm
```

Equations are assumed to be confluent and terminating, that is, we can use the equations to reduce a term t to a unique, canonical form t' that is equivalent to t (they represent the same value). The Maude system does not check these properties of equational specifications, but there are related tools that can be used for that purpose.

The *dynamic* behavior of such a distributed system is then specified by rewrite rules of the form $t \longrightarrow t'$, that describe the local, concurrent transitions of the system. That is, when a part of a system matches the pattern t, it can be transformed into the corresponding instance of the pattern t'. Rewrite rules are included in *system* modules (with syntax `mod...endm`). For example, the next module defines nondeterministic natural numbers and nondeterministic choice. A module can import, or include, the definitions of another module by means of keyword `inc`.

```
mod ND-NAT is
  inc NAT .
  sort NdNat .
  subsort Nat < NdNat .
  op _?_ : NdNat NdNat -> NdNat [assoc comm].
  var N : Nat .  var ND : NdNat .
  eq N ? N = N .
  rl [choice] : N ? ND => N .
endm
```

A set of natural numbers is regarded as a nondeterministic natural number of sort `NdNat`, that is, a number that could be anyone of those in the set. The operation `_?_` denotes the union of nondeterministic natural numbers, which is associative and commutative, and obeys also an idempotence equation. The `choice` rule provides nondeterministic choice.

Rewriting logic has revealed itself to be a general and flexible logical and semantic framework [14], in which many different logics, models of computation, and a wide range of languages can be represented, can be given a precise semantics, and can be executed. In this paper we claim that it can also be used to give semantics to metadata description frameworks such as RDF.

One of the main properties of Maude (and rewriting logic) is that it is reflective, that is, Maude can be represented into itself in such a way that a program (or module) M in Maude may be data for another Maude program, which can modify M, obtain information about it, or ask to *execute* it.

In Maude, key functionality of this reflective power has been efficiently implemented in the functional module `META-LEVEL`, where Maude terms are reified as elements of a data type `Term`, Maude modules are reified as terms in a data type `Module`, the process of reducing a term to normal form is reified by a function `metaReduce`, and the process of rewriting (executing) a term by applying the rewrite rules of a module is reified by a function `metaRewrite` [7]. We use these features in the implementation of the translation from RDF into object-oriented Maude modules, and when the translation is used in an example about a mobile agent system in Section 5.2.

3.1 Object-oriented specification in Maude

In an object-oriented Maude module (a special kind of system module, with syntax `omod...endom`) classes are declared with the syntax `class` C `|` $a_1 : S_1$`,...,` $a_n : S_n$, where C is the class name, a_i is an attribute identifier, and S_i is the sort of the values this attribute can have. An *object* in a given state is represented as a term `<` O `:` C `|` a_1 `:` v_1`,` `...,` a_n `:` v_n `>`, where O is the object's name (belonging to a set `Oid` of object identifiers), and the v_i's are the current values of its attributes. *Messages* are defined by the user for each application (introduced with syntax `msg`). Subclass relations can also be defined, with syntax `subclass`.

In a concurrent object-oriented system the concurrent state, which is called a *configuration*, has the structure of a multiset made up of objects and messages that evolves by concurrent rewriting (modulo the multiset structural axioms of

RDF/RDFS	Maude
RDF document	Object-oriented module
Class	Class
Resource	Object
Property	Attribute
Container	Abstract data type
URI	Object identifier

Table 1. RDF concepts translated into Maude

associativity, commutativity, and identity) using rules that describe the effects of *communication events* between some objects and messages. The rewrite rules in the module specify in a declarative way the behavior associated with the messages. The general form of such rules is

$$\begin{aligned}
&M_1 \dots M_n \ \langle O_1 : F_1 \mid atts_1 \rangle \dots \langle O_m : F_m \mid atts_m \rangle \\
&\longrightarrow \langle O_{i_1} : F'_{i_1} \mid atts'_{i_1} \rangle \dots \langle O_{i_k} : F'_{i_k} \mid atts'_{i_k} \rangle \\
&\quad\ \langle Q_1 : D_1 \mid atts''_1 \rangle \dots \langle Q_p : D_p \mid atts''_p \rangle \\
&\quad\ M'_1 \dots M'_q \quad \mathit{if}\ C
\end{aligned}$$

where $k, p, q \geq 0$, the M_s are message expressions, $i_1, \dots, i_k$ are different numbers among the original $1, \dots, m$, and C is a rule condition. The result of applying a rewrite rule is that the messages $M_1, \dots, M_n$ disappear; the state and possibly the class of the objects $O_{i_1}, \dots, O_{i_k}$ may change; all the other objects O_j vanish; new objects $Q_1, \dots, Q_p$ are created; and new messages $M'_1, \dots, M'_q$ are sent.

We will use this kind of system modules to provide the semantics for RDF documents and to implement the examples.

Later, in Section 5.2 we will integrate the translated RDF documents in an agent application written using Mobile Maude. This extension of Maude provides some new concepts related with mobility (mobile objects and processes) that are expressed in Maude itself, as explained in Section 5.1.

4 RDF/RDFS translation into Maude

The main pieces in an RDF document are resources, properties, containers, URIs, and classes. We identified which elements of a Maude module could correspond naturally to these RDF pieces. The principal result is that Maude object-oriented modules are a good choice to represent RDF documents in Maude by giving them the natural, intuitive meaning. Table 1 shows the correspondence between RDF pieces and Maude elements.

In this section we describe the translation of RDF (including reification and containers) and RDFS documents into object-oriented modules in Maude. The driving idea is that an RDF description of a resource will be translated into an object in Maude.

Maude modules are used for describing RDF schemas. Those modules will be included in the translation of any particular RDF document using the predefined vocabulary. The following module defines the basic vocabulary for RDFS. It defines a data type for URI references and declares that they can be used as object identifiers (`Oid`). A class for resources is defined with several attributes. Every described resource will be an instance of this class, although we have used the same relaxed idea used by RDF of what an instance is. An object O is an instance of class C if it is declared as belonging to this class and it only has attributes defined for this class (or any of its superclasses), but not all the attributes have to be initialized. Apart from this consideration, all the power of object-orientation is supported, including inheritance (as explained in Section 5.2). URI references and instances of the class of resources are put together in a general type `Resource`. A class for properties is defined, and it is declared as a subclass of resources. There is also a data type for representing literals, which uses the predefined type of quoted identifiers (`Qid`).

```
omod http://www.w3.org/2000/01/rdf-schema is
  inc QID .
  sorts URI Resource .  subsort URI < Oid .
  op uri : Qid -> URI .
  class ResourceClass | comment : Literal,  label : Literal,
                        seeAlso : Resource, isDefinedBy : Resource .
  subsorts URI ResourceClass < Resource .
  class Property .
  subclass Property < ResourceClass .
  sort Literal .
  op literal : Qid -> Literal .
endom
```

There is another module defining the predefined vocabulary for RDF. A class `Statement` is declared for representing RDF statements, that is, a *reified statement* will be represented as an instance of this class. The class has three attributes: `subject`, `predicate`, and `object`. The module also declares classes for the different RDF containers by giving precise definitions of what they mean. For example, there is a class for bag containers which are described in [12] as "unordered lists of resources or literals." In Maude, we can define what this exactly *means* by defining a data type for *multisets* of resources and literals, with a constant operator `mt` for the empty multiset and a union operator `_&_` which is declared to be associative, commutative, and with the empty multiset as identity element. There are similar classes for sequences and alternatives, although in each case the union operator is defined in a different way. For example, the union operator for sequences is declared as associative and with identity the empty sequence, but it is not declared as commutative, because a sequence is "an ordered list of resources or literals" [12].

```
omod http://www.w3.org/1999/02/22-rdf-syntax-ns is
  inc http://www.w3.org/2000/01/rdf-schema .
  class Statement | subject : Resource, predicate : Property,
                  object : Resource .
  *** containers
  class Container .  subclass Container < Resource .
  class Bag | val : BVal .  subclass Bag < Container .
  sort BVal .  subsorts Literal Resource < BVal .
  op mt : -> BVal .
  op _&_ : BVal BVal -> BVal [assoc comm id: mt] .
  ...
endom
```

The translation of a user-defined RDF document into an object-oriented module of Maude is summarized in Table 1. Let us see some examples for illustrating the translation of user-defined RDF documents. The RDF document describing a laser printer in Section 2 is translated into the following object-oriented module in Maude:

```
omod example is
  inc http://www.w3.org/1999/02/22-rdf-syntax-ns .
  inc http://printers.org/schema .
  op http://HPprinters/HPLaserJet1100 : -> Object .
  eq http://HPprinters/HPLaserJet1100 =
   < uri('http://HPprinters/HPLaserJet1100) : LaserPrinter |
     PrinterTechnology : literal('Laser`Jet),
     PrinterResolution : literal('600`dpi),
     Price : literal('399) > .
endom
```

The two namespaces used in the RDF document have been translated to module inclusions. The resource has been translated to an object constant and one equation defining it. This object has three attributes whose values are literals.

Anonymous resources are also supported and translated to object constants as above, although instead of a URI we use a local identifier to name them. Container descriptions are translated to objects of a class like the class `Bag` commented above, and the enumerated items are included in its attribute as a value built by using the corresponding union operator.

The RDFS description of printers in Section 2 is translated as follows:

```
omod Printers is
  inc http://www.w3.org/1999/02/22-rdf-syntax-ns .
  inc http://www.w3.org/2000/01/rdf-schema .
  class Printer | Price : Literal,
                  PrinterTechnology : Literal,
                  PrinterResolution : Literal .
  subclass Printer < ResourceClass .
  class LaserPrinter .
  subclass LaserPrinter < Printer .
endom
```

The namespaces have been translated into module inclusions, as above. The RDFS class declarations have been translated into Maude class declarations, and the subclass properties have been translated into subclass declarations in the Maude module. When a subclass relation is declared as `subclass` C `<` C', the class C is the subclass and the class C' is the superclass. The effect of a subclass declaration is that the attributes, messages, and rules of the superclass are inherited by the subclass. The property `Price`, with domain `Printer` and range `Literal`, has been translated into an attribute declaration of class `Printer` whose values can be of sort `Literal`. The other two properties are translated similarly.

By using the reflective features of rewriting logic and Maude, and moving up to the metalevel, where Maude modules become data that can be manipulated, we can equationally define operations that perform the translation described above in an *automatic* way. These operations traverse the elements of an `RDF` value and build the module step by step by including the translation of each element, as explained by means of examples above. The complete Maude code implementing this translation together with the predefined modules described above can all be found in `http://www.ucm.es/sip/alberto/semantic-web`.

5 Case study

In this section we present a simple application of the translation process. The proposed translation has been used in an example where a buyer agent visits several sellers which give him their printers information in RDF. The buyer keeps the price of the cheapest printer. The example has been implemented using Mobile Maude, a Maude extension that supports mobile computation.

5.1 Mobile Maude

The flexibility of rewriting logic for representing very different styles of communication, either synchronous or asynchronous, its facility for supporting distributed, concurrent object-oriented systems, and its reflective capabilities for supporting metaprogramming and dynamic reconfiguration, make it a very suitable formalism for the specification of distributed systems based on mobile agents, on which the proof of properties about security, correctness, and performance, can be based.

Mobile Maude [8] is an extension of the Maude language supporting mobile computation. It is appropriate for the specification and prototyping of distributed systems based on mobile agents, where data, states, and programs can be moved. Moreover, it has a *formal basis* for the development of security models and the verification of properties for such models. The key entities in Mobile Maude are *processes* and *mobile objects*. Both are defined as classes in Maude. Processes are computational environments where mobile objects evolve and communicate with each other. Mobile objects are created inside a process, they can move to another process, they can operate inside a process, and they can send

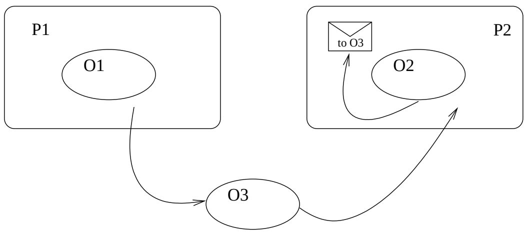

Fig. 1. A view of processes and mobile objects.

(receive) messages to (from) other mobile objects in the same process or in other processes.[2] This is illustrated in Figure 1, where there are two processes (P1 and P2) and three mobile objects (O1, O2, and O3). Mobile object O3 is moving from process P1 to process P2, while the object O2 has just sent a message addressed to the moving object O3.

Processes and mobile objects are defined in Maude as classes `P` and `MO`, respectively. The class `P` of processes is declared as follows:

```
class P | cnt : Nat, cf : Configuration, guests : Set[Mid],
          forward : PFun[Nat, Tuple[Pid, Nat]] .
```

The main attribute is `cf`, the configuration of guest mobile objects. The attribute `guests` is the set of identifiers of mobile objects that currently reside in the process; `cnt` is the counter of mobile objects created in the process; and `forward` is a function used to locate the mobile objects created in the process. The names of processes range over the sort `Pid`, whereas the names of mobile objects range over the sort `Mid` and have the form `o(PI,N)`, where `PI` is the name of the object's parent process, that is, the process where it was created, and `N` is a number that distinguishes the children of `PI`.

The class `MO` of mobile objects is defined as follows:

```
class MO | mod : Module, s : Term, p : Pid,
           hops : Nat, gas : Nat, mode : Mode .
```

The mobile object's module that defines a mobile object behavior must be object-oriented, and `mod` is the metarepresentation of that module. The term `s` is the metarepresentation of the actual configuration of the mobile object; this configuration has the following form: `C & C'`, where `C'` is the outgoing messages tray (a multiset of outgoing messages) and `C` contains the state of the mobile object (as defined in the module `mod`) and a multiset of unprocessed incoming messages. The rest of the attributes in the class are: `p`, the identifier of the process where the mobile object currently is; `hops`, a natural number indicating the number of hops between processes that the object has performed so far; `gas`,

[2] Some mobile agents languages, as Cardelli's Ambient Calculus [6], forbid this last kind of communication, allowing only communications inside the same process.

a natural number that limits the rewrite steps that the object can do; and `mode`, that indicates if the process is active or not.

Messages in the configuration and in the module may be of any form, but those being pulled in or out of the mobile object must have a specific form. In particular, messages getting in and out of mobile objects must be of one of the following forms:

- `to MID : MSG`, to send the message content `MSG` to the object whose identifier is `MID`. The `MSG` part is built with user-defined syntax.
- `go(PID)`, to go to the process whose identifier is `PID`.
- `go-find(MID, PID)`, to go to the process where the object `MID` is, trying as first alternative the process `PID`.
- `newo(MOD, OBJ, OID)`, to create a mobile object where `MOD` is the metarepresentation of the module where the object is defined, `OBJ` is the initial state of the object to be created, and `OID` is its identifier.

When a mobile object wants to deliver messages of this kind it puts them in its outgoing messages tray.

The complete Mobile Maude *system code*, plus some related information, can be found in `http://maude.cs.uiuc.edu/maude1/mobile-maude`.

The code describing the behavior of mobile objects is called *application code.* In the next section we will present several examples of this kind of code.

In [9] a case study using Mobile Maude is presented, and it is shown how an object-oriented specification in Maude can be made mobile. An ambitious wide area application, namely the reviewing system for a conference, going from its announcement to the edition of the proceedings, is specified and implemented. Such example was proposed by Cardelli in [6] as a challenge for any wide area language to demonstrate its usability, although it was previously used by different authors. Mobile Maude was used successfully to implement this system. Moreover, the Maude formal specification of Mobile Maude was used to execute the example. This case study and the possibility of executing it allowed us to test different alternatives both in the language and in the specification of the system. Although in the actual specification RDF documents are not used (different agents communicate with each other with a pre-established small vocabulary), it can be easily modified in such a way that agents communicate by means of RDF documents. Then the translation presented in this paper could be used by the agents to translate and understand the received information.

5.2 Buying printers

In this example we have two different classes of mobile objects: sellers and buyers. Although in the simple example described here sellers do not move, they have to be mobile objects because they communicate with other mobile objects, so they have to be recognized as mobile objects by the Mobile Maude system. There is another class of objects, the comparers, that are used by buyers to compare printers. These are not mobile objects, as described below. A buyer visits several sellers. The buyer asks each seller he visits for the description of the seller's printer. The seller sends this description in RDF format, which the buyer translates and gives to his comparer, that keeps the price of the cheapest printer.

First we define the sellers. They are static agents whose behavior is defined in the following module. The class `Seller` has an attribute `description` with the RDF description of the printer it sells, using the schema in Section 2. When a seller receives a description request, it sends the description in RDF form.

```
omod SELLER is
  inc RDF-SYNTAX .
  class Seller | description : RDF .
  op get-printer-description : Oid -> Contents .
  op printer-description : RDF -> Contents .
  vars S B : Oid .  var D : RDF .
  rl [get-des] : (to S : get-printer-description(B))
       < S : Seller | description : D > & none
    => < S : Seller | > & (to B : printer-description(D)) .
endom
```

Note how the seller's state is described in rule `get-des` by means of the `_&_` operator in order to separate the inner state and incoming messages from the outgoing messages. Due to that we can use this module to build mobile objects in Mobile Maude.

Before defining the buyers, we define the class `Comparer` whose instances are able to compare different printers, keeping the price of the cheapest printer. When a comparer is near a printer, it looks the price of the printer, and compares it with the best printer it knows, updating its knowledge if necessary. Note that the printer object disappears, because it does not represent a real printer, but a printer information, that is useless after the comparer has looked up its information.

```
omod COMPARER is
  inc Printers .
  inc DEFAULT[Nat] .
  class Comparer | best : Default[Nat] .
  var P C : Oid .  var Q : Qid .  var N : Nat .
  var Atts Atts' : AttributeSet .
  rl [compare] : < P : Printer | Price : literal(Q), Atts >
       < C : Comparer | best : N, Atts' >
    => < C : Comparer | best : if (convert(Q) < N) then
                        convert(Q) else N fi, Atts' > .
endom
```

A comparer is not a mobile object of Mobile Maude. It does not move independently, and cannot send or receive messages from other mobile objects. It is a Maude object that will travel inside a buyer's attribute, as we will see below.

Note how the variable `Atts` of sort `AttributeSet` is used in the printer object. By using this variable, the rule can be applied to any printer with *at* *least* an attribute `Price`; if the printer has more attributes, they will be caught by the variable `Atts`.

This style of programming is quite useful for the Semantic Web. If a seller has defined its own RDF schema, extending the one presented in Section 2 by defining a subclass of printers with new properties which are important for him, it will send printer descriptions with some properties unknown for our comparer. But the above implementation will also be useful in this case, because the extra properties (attributes) will be caught by the variable `Atts`.

Finally, we define the buyers. The module `BUYER` describes the behavior of a buyer agent. It has a list `IPs` with the addresses of the sellers. It has to visit all the sellers, asking each one for the description of the printers. The buyer has an attribute `app-state` with the current state of its comparer, metarepresented. It has to be metarepresented because the buyer wants to be able to execute the comparer. Each time it receives a new description, it translates the RDF description into a Maude module M with a `Printer` object. It puts this object together with the current state of its comparer, and asks to rewrite them (by using `metaRewrite`, see Section 3) in the Maude module obtained by joining M with the module containing the comparer code.

```
omod BUYER is
  inc RDF-Translation .
  sort Status .
  ops onArrival asking done : -> Status .
  class Buyer | IPs : List[Oid], status : Status, app-state : Term .
  op get-printer-description : Oid -> Contents .
  op printer-description : RDF -> Contents .
  var PD : RDF .  var Ss : List[Oid] .  vars B S : Oid .
  var PI : Pid .  var N : Nat .  var T : Term .
  rl [move] : < B : Buyer | IPs : o(PI,N) + Ss, status : done > & none
    => < B : Buyer | status : onArrival > & go-find(o(PI,N),PI) .
  rl [onArrival] : < B : Buyer | IPs : S + Ss, status : onArrival > & none
    => < B : Buyer | status : asking > &
       (to S : get-printer-description(B)) .
  rl [new-des] : (to B : printer-description(PD))
       < B : Buyer | IPs : S + Ss, app-state : T, status : asking >
    => < B : Buyer | IPs : Ss, app-state : metaRewrite(
          addDecls(up(COMPARER), translate('Printer,PD)),
          '__[T,extractResources(translate('MOD,PD))],0), status : done > .
endom
```

The first rewrite rule, `move`, handles the travels of the buyer: if it has finished in the current process (its status is `done`) and there is at least one seller name in the `IPs` attribute, it asks the system to take it to the host where the seller is. On arrival, the buyer asks the seller for the printer description, giving the seller's name. When the RDF description arrives, the buyer translates it to Maude, extracts the resource corresponding to the printer description, puts it together with the comparer, and asks to rewrite the result in the module with the comparer's behavior, which will change the comparer's state.

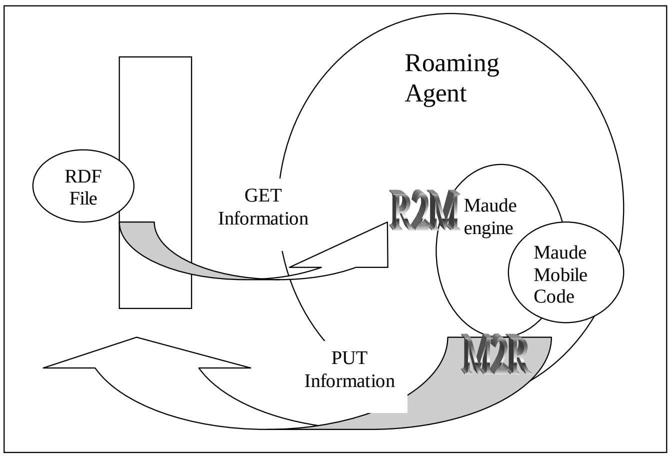

Fig. 2. RDF $\leftrightarrow$ Maude translations

The full code of this example can be found in `http://www.ucm.es/sip/alberto/semantic-web`.

6 Conclusions and future work

In this paper we presented the first results of a translation from RDF/RDFS to the language Maude. This translation produces a formal version of the original RDF/RDFS data, without requiring any extra information in the original RDF/RDFS documents, thus preserving compatibility with other approaches. We think this approach offers a sound way for formalizing the Semantic Web.

A key point for success of our translation model is to integrate the approach using Maude with the real Web, as described in Figure 2, where an agent with Mobile Maude code interacts with usual Internet services by getting the RDF file that contains all the required information. Then an RDF2Maude translation will be carried out. After operating, one or more result files will be produced. These files will be translated into RDF by a Maude2RDF translation. This approach will allow the formalized service to interact with the usual Web applications and services. Maude has already been integrated with real Web applications in [1, 2].

The work presented here is the first step to allow a formal model for Web services. This promising area will be enriched with the Semantic Web services approach by enriching not only the services with a semantic definition over RDF or OWL-S [15], but also allowing to access a specification of the dynamic semantics of the operations carried out by this service.

We plan to carry out further research in order to integrate our work with the OWL specification [17] and specially with OWL-S, to extend it on a formal way in order to enable a possible wide future of new formal Semantic Web services.

References

1. A. Albarrán, F. Durán, and A. Vallecillo. From Maude specifications to SOAP distributed implementations: a smooth transition. In *Proceedings VI Jornadas de Ingeniería del Software y Bases de Datos, JISBD 2001, Almagro, Spain*, 2001.
2. A. Albarrán, F. Durán, and A. Vallecillo. Maude meets CORBA. In *Proceedings 2nd Argentine Symposium on Software Engineering*, Argentina, 2001.
3. T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 2001.
4. T. Bray, D. Hollander, and A. Layman. Namespaces in XML, 1999. `http://www.w3.org/TR/REC-xml-names`.
5. D. Brickley and R. Guha. RDF vocabulary description language 1.0: RDF Schema. W3C Recommendation, 10 February, 2004. `http://www.w3.org/TR/rdf-schema`.
6. L. Cardelli. Abstractions for mobile computations. In J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, LNCS 1603, pages 51–94. Springer, 1999.
7. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.
8. F. Durán, S. Eker, P. Lincoln, and J. Meseguer. Principles of Mobile Maude. In D. Kotz and F. Mattern, eds., *Agent Systems, Mobile Agents, and Applications, ASA/MA 2000*, LNCS 1882. Springer, 2000.
9. F. Durán and A. Verdejo. A conference reviewing system in Mobile Maude. In F. Gadducci and U. Montanari, eds., *Proceedings Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002*, ENTCS 71, pages 79–95. Elsevier, 2002.
10. R. Fikes and D. MacGuinness. An axiomatic semantics for RDF, RDF-S, and DAML+OIL. W3C Note, 2001. `http://www.w3.org/TR/daml+oil-axioms`.
11. P. Hayes. RDF semantics. W3C Recommendation, 10 February, 2004. `http://www.w3.org/TR/rdf-mt`.
12. F. Manola and E. Miller. RDF primer. W3C Recommendation, 10 February, 2004. `http://www.w3.org/TR/rdf-primer`.
13. N. Martí-Oliet and J. Meseguer. Action and change in rewriting logic. In *Dynamic Worlds: From the Frame Problem to Knowledge Management*, pages 1–53. Kluwer Academic Publishers, 1999.
14. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Second Edition, Volume 9*, pages 1–87. Kluwer Academic Publishers, 2002.
15. D. Martin, editor. OWL-S: Semantic Markup for Web Services. `http://www.daml.org/services/owl-s/1.1/overview`.
16. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
17. D. McGuinness and F. van Harmelen. OWL Web Ontology Language. W3C Recommendation, 10 February, 2004. `http://www.w3.org/TR/owl-features`.
18. IETF Uniform Resource Identifiers (URI) Working Group, 2000. `http://ftp.ics.uci.edu/pub/ietf/uri/`.
19. Resource Description Framework (RDF) / W3C Semantic Web Activity. `http://www.w3.org/RDF`.