Describing Computation within RDF

Chris Goad
The Behavior Engine Company
10 Sixth Street, Suite 108
Astoria, OR 97103
cg@ behaviorengine.com

Abstract. A programming language is described which is built within RDF. Its code, functions, and classes are formalized as RDF resources. Programs may be expressed directly using standard RDF syntax, or via a conventional JavaScript-based syntax. RDF constitutes not only the means of expression, but also the subject matter of programs: the native objects and classes of the language are RDF resources and DAML+OIL classes, respectively. The formalization of computation within RDF allows active content to be integrated seamlessly into RDF repositories, and provides a programming environment which simplifies the manipulation of RDF when compared to use of a conventional language via an API. The name of the language is "Fabl".

1. Introduction

Fabl[1] is a programming language which is built within RDF [1]. The constituents of the language - its code, functions, and classes - are formalized as RDF resources, as is the data over which computation takes place. This means that programs reside within the world of RDF content rather than being relegated to a separate realm connected to RDF via an API. The starting point for the formalization is DAML+OIL [2].

The language provides an efficient imperative programming framework for the RDF domain. Programs may be expressed as RDF objects using standard RDF syntax, or via a conventional syntax which might be described as JavaScript[2] enhanced with types and qualified property names. The language is designed to be easy to learn for programmers familiar with the conventional JavaScript/HTML/XML/DOM web-programming model. In fact, the conceptual cleanliness of RDF makes the language and its semantics far simpler than this conventional model. The initial implementation is similar in runtime efficiency to other scripting environments.

As a computational formalism for RDF, the neighboring points of comparison for Fabl are the RDF APIs (eg [3], [4]), in which computation is expressed in conventional ways, but the subject matter of the computation is expressed in RDF. Fabl has several advantages over APIs:

1. Simplicity of programming.
2. Functions and programs can be managed, inspected, manipulated, and annotated in the same manner as any other RDF resources; they are first-class citizens of the RDF world.

3. Fabl's type system exploits the RDF property-centric style. This yields a system of a kind different, and in some ways more expressive and flexible, than those found in the main thread of object-oriented type systems running from Simula through C++, Java, C#, and Curl.
4. Fabl programs are formalized within RDF in a manner that provides an open framework for extension of the language. The implementation of Fabl is, with the exception of a few low level utilities, written in Fabl itself. Further, the process by which programs are analyzed and converted into an efficiently executable form can be extended by addition of new RDF content. This means that extension of Fabl to include new language facilities, such as new control structures, new syntax, or new typing systems built on different principles can all be carried out in the RDF style: by extending the base of RDF files which describe the language.

Although Fabl defines a particular (albeit, extensible) textual format for programs on the one hand, and implements a particular byte-code and virtual machine for interpretation on the other, the core of the design is its formalism for describing imperative computation as RDF. This integrates computation into the RDF realm of distributed semantic description, decoupled from any particular source language and from any particular execution technique. Concretely, active entities, from simple spreadsheets to complex simulations, can be formalized in RDF, and made available to any agent that has a use for them, independent of the language (or graphical interface) from which they were created.

Whether or not the particular formalism introduced here is the right one, RDF can and should be used as a vehicle for standardizing computation as well as passive content. If nothing else, Fabl shows the practicality of this idea.

2. Application Scenarios

Close integration of computation with RDF can benefit both sides of the integration. Most trivially, RDF mechanisms can be used to annotate programs - for example by using the Dublin Core[5] to assert information about date, author, and publisher of code. With the development of simple computational ontologies, metadata about code of the sort useful to software engineeers can be asserted in RDF; examples include call trees, traces, and performance information. The openness of RDF, which allows continually evolving vocabularies and tools to be applied to preexisting data, should benefit the realm of programming as much as any other domain.

Beyond annotation, the formalization of functions and code as RDF resources is the first step in integrating algorithmic computation and inference in an RDF setting. The combination of inference and algorithmic computation might be applied to automatic assembly of programs from available components, and to problem solving which mixes inference and algorithmic computation (when a subproblem is inferred to be solvable by an available algorithm, the algorithm is invoked). This direction of work requires more complex computational ontologies which formalize the kinds of statements about computational objects needed to support useful inference.

Going in the other direction, thorough integration of computation with RDF facilitates the development of active RDF content. The initial application to which we are appying Fabl provides an example. We have defined relatively simple ontologies for geography (themed maps, as in GIS), and for events located in a geographical context. This geographical and historical information is depicted by interactive web-delivered maps in the Macromedia Flash format (see our web site[6] for examples). The active aspect of our RDF repository consists

primarily of handlers which generate interactive maps from the underlying geographical and historical information, and which maintain consistency between the data and its depiction as changes are made. The handlers are RDF resources and their relationship to other data is expressed by RDF statements. Regularities (eg all resources in this class have that handler) are asserted by DAML+OIL restrictions.

This application provides a template for a wide range of possible applications, wherein complex situations are represented in RDF, and where consistency constraints are automatically maintained by associated constraint propogation mechanisms which are at least partly algorithmic (rather than strictly deductive) in nature. The kind of complete integration proposed here is not the only possible approach to this kind of application, but we would argue that first-class status for computational entities in the RDF world removes a layer of indirectness and complexity that would otherwise be necessary.

3. An Example

Consider the the simplest of data structures, a point on the plane with two coordinates, which can be expressed in Java by:

```
public class  Point {
    double xc;
    double yc;
}
```

Here is an extract from a Fabl RDF file at http://purl.oclc.org/net/fabl/examples/geom defining the same structure:

```
<rdf:RDF
  xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:xsd ="http://www.w3.org/2000/10/XMLSchema#"
  xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns:fabl="http://purl.oclc.org/net/nurl/fabl/"
  xmlns:nurl="http://purl.oclc.org/net/nurl/"
>
<daml:DatatypeProperty rdf:ID="xc"/>
<daml:DatatypeProperty rdf:ID="yc"/>

<rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#xc"/>
      <daml:toClass
rdf:resource="http://www.w3.org/2000/10/XMLSchema#double"/>
      <daml:cardinality>1</daml:cardinality>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#yc"/>
      <daml:toClass
rdf:resource="http://www.w3.org/2000/10/XMLSchema#double"/>
      <daml:cardinality>1</daml:cardinality>
    </daml:Restriction>
</rdfs:Class>
```

The Fabl type system makes use of the March, 2001 version of DAML+OIL. The above RDF asserts that every member of Point has xc and yc properties, and that these properties each have exactly one value of type double. All of the examples in this paper use the name space declarations given just above, which will be abreviated in what follows by [standard-namespace-declarations]. Here is vector addition for points:

```
<rdf:RDF
[standard-namespace-declarations]
  xmlns:geom="http://purl.oclc.org/net/fabl/examples/geom#"
>

<fabl:code>
geom:Point function plus(geom:Point x,y)
  {
  var geom:Point rs;
  rs = new(geom:Point);
  rs . geom:xc = x.geom:xc + y.geom:xc;
  rs . geom:yc = x.geom:yc + y.geom:yc;
  return rs;
  }
</fabl:code>
</rdf:RDF>
```

The above text is not, of course, legal RDF. Rather, it represents the contents of a file intended for analysis by the Fabl processor, which converts it into RDF triples. The pseudo-tag <fabl:code> encloses Fabl source code; everything not enclosed by the tag should be legal RDF.

Note that the syntax resembles that of JavaScript, except that variables and functions are typed. Fabl types are RDF classes, and are named using XML qualified[7] or unqualified names (details below).

Here are the contents of the file http://purl.oclc.org/net/fabl/examples/color:

```
<rdf:RDF
[standard-namespace-declarations] >

<daml:Class rdf:ID="Color"/>
<Color rdf:ID="yellow"/>
<Color rdf:ID="blue"/>
<rdf:Property rdf:ID="colorOf">
  <rdfs:range rdf:resource="#Color"/>
<rdf:/Property>
```

The following fragment assigns a color to an existing Point: yellow if its x coordinate is positive, and blue otherwise:

```
<rdf:RDF
[standard-namespace-declarations]
  xmlns:geom="http://purl.oclc.org/net/fabl/examples/geom#"
  xmlns:color="http://purl.oclc.org/net/fabl/examples/color#"
>
```

```
<fabl:code>
  fabl:void function setColor(geom:Point x)
    {
    if (x . geom:xc > 0) x.color:colorOf = color:yellow;
    else x.color:colorOf = color:blue;
    }
  </fabl:code>
```

The expression fabl:void may only be used in a context where the return type of a function is indicated. It signifies that the function in question does not return a value. Note that fabl:void is not a class, and in particular is should not be identified with daml:Nothing. A function with return type daml:Nothing would indicate that the function returns a value belonging to daml:Nothing - an impossibility.

The setColor example illustrates the central difference between an RDF class and its counterparts in the object-oriented programming tradition. An RDF class is an assertion about properties possessed by a resource, which does not preclude the resource from having additional properties not mentioned in the class, nor from belonging to other classes, nor even from aquiring new properties and class memberships as time goes on. The progression of data types in programming languages exhibits growing freedom of type members: C or Pascal types exactly determine the structure of their members; C++ and Java classes determine the structure of members to a degree, but allow extension by subclasses; the RDF model leaves the structure of members free except as explicitly limited by the class definition.

Unless a property has been explicitly constrained to have only one value, Fabl interprets the value of a property selection:

```
x.P
```

as a bag. In the following example, the first function returns the number of colors assigned to an object, and the latter returns its unique color if it has only one, and a nul value otherwise.

```
xsd:int function numColors(daml:Thing x)
{
   return cardinality(x.color:colorOf);
}

color:Color function theColorOf(daml:Thing x)
{
   var BagOf(color:Color) cls;
   cls = x.color:colorOf;
   if (cardinality(cls)==1) return cls[0];
   else return fabl:undefined;
}
```

fabl:undefined is a special identifier which denotes no RDF value, but rather indicates the absence of any RDF value in the contexts where it appears.

4. RDF Computation in Fabl

RDF syntax and semantics can be viewed as having three layers: (1) a layer which assigns concrete syntax (usually XML) to RDF assertions, (2) the data model layer, in which RDF content is represented as a set of triples over URIs and literals, and (3) a semantic model, consisting of the objects and properties to which RDF assertions refer. DAML+OIL specifies

semantics[8] constraining the relationship between the data model and the semantic model.

The proper level of description for computation over RDF is the data model; the state of an RDF computation is a set of triples <subject,predicate,object>. This triple set in turn can be construed as a directed labeled graph whose nodes are URIs and literals, and whose arcs are labeled by the URIs of properties.

Fabl is executed by a virtual machine. An invocation of the Fabl VM creates an initial RDF graph which is in effect Fabl's own self description: the graph contains nodes for the basic functions and constants making up the Fabl language. Subsequent activity modifies the RDF graph maintained by the VM, called the "active graph". The Fabl interpreter can accept input from a command shell, or can be configured as a server in a manner appropriate to the application.

The universe of RDF files on the web plays the role of the persistent store for Fabl. The command

```
loadRdf(U)
```

adds the triple set described in the RDF file at URL U to the active graph.

The active graph is partitioned into pages. The data defining a page includes: (1) the external URL (if any) from which the page was loaded, (2) the set of RDF triples which the page contains, (3) a dictionary which maps the ids appearing in the page (as values assigned to the rdf:ID attribute) to the resources which they identify, and (4) a set of namespace definitions (bindings of URIs to namespace prefixes). Many pages are the internal representations of external RDF pages, but new pages can be created which are not yet stored externally.

```
saveRdf(x,U)
```

saves the page upon which x lies at the file U. The current implementation interacts with the external world of RDF via simple loading and saving of pages, but there are interesting additional possibilities involving distributed computation, which are outlined in a later section

A global variable or constant X with value V is represented by a daml:UniqueProperty named X whose value on the URI fabl:global is V. (It doesn't matter what values the property assumes when applied to other resources, nor does fabl:global play any other role.) For example, the following fragment defines the global pi:

```
  <daml:DatatypeProperty rdf:ID="pi">
    <rdf:type
rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
    <rdfs:range rdf:resource="http://www.w3.org/2000/10/XMLSchema#double"/>
  </daml:DatatypeProperty>

  <daml:Class rdf:about="http://purl.oclc.org/net/nurl/fabl/global">
    <pi>3.14159265358979323846 </pi>
  </daml:Class>
```

The values of global properties can be referred to directly by name in Fabl. For example, since http://purl.oclc.org/net/fabl/examples/geom includes the lines above defining pi, the following fragment illustrates reference to pi as a global:

```
  <rdf:RDF
  [standard-namespace-declarations]
    xmlns:geom="http://purl.oclc.org/net/fabl/examples/geom#"
  >
```

```
<fabl:code>
xsd:double function timesPi(xsd:double x){return x * geom:pi}
</fabl:code>
```

As indicated in the initial example above, basic manipulation of the active graph is accomplished via conventional property access syntax: If P is the qualified name of a property, and x evaluates to an object, then

```
x.P
```

returns a bag of the known values of P on x, that is, the set of values V such that the triple <x,P,V> is present in the active graph. However, if P is asserted to be univalued - if it was introduced as a UniqueProperty, or has a cardinality restriction to one value - then

```
x.P
```

evaluates to the unique value instead. The assignment

```
x.P = E
```

for an expression E adds the triple <x,P,value(E)> to the active graph, unless P has been asserted to be a univalued, in which case the new triple replaces the previous triple (if any) which assigned a value to P on x. The command:

```
var Type name;
```

is equivalent to:

```
<daml:UniqueProperty rdf:ID="name">
  <rdfs:range rdf:resource="Type"/>
</daml:UniqueProperty>
```

The function:

```
new(Type)
```

creates a new node N in the active graph, and adds the triple <N,rdf:type,Type>. Initially, nodes created with the new operator lack an associated URI. However, Fabl allows URIs to be accessed and set as if they were properties, via the pseudo-property uri.

```
x.uri
```

is the current URI of x if it has one, and fabl:undefined if not.

```
x.uri = newURI;
```

assigns a new URI to x. If newURI is already assigned to another node y in the active graph, x is merged with y. The merged node will possess the union of the properties possessed by x and y prior to the merge.

5. RDF Computation Via an API: A Comparison

The Java code below uses the Jena API[4] to implement the function presented at the beginning of section 3: vector addition of points. This sample is included to give the reader a concrete sense of the difference between Fabl code, which expresses elementary RDF operations directly as basic operations of the language, and code using an API, in which the same elementary operations must be expressed as explicit manipulations of a representation of RDF content in the host language (here, Java). This is the only purpose of the sample, and the details are not relevant to anything that appears later in this paper. Also, the points made here apply equally to other RDF APIs.

```
  import com.hp.hpl.mesa.rdf.jena.mem.ModelMem;
  import com.hp.hpl.mesa.rdf.jena.model.*;
  import com.hp.hpl.mesa.rdf.jena.vocabulary.*;

  // The class GeomResources initializes variables
  // xc, yc, and Point to RDF resources of the right kind.
  public class GeomResources {
      protected        static        final        String        URI        =
"http://purl.oclc.org/net/fabl/examples/geom#";
      public static String getURI(){return URI;}
      public static Property xc = null;
      public static Property yc = null;
      public static Resource Point = null;
      static {
          try {
              xc = new PropertyImpl(URI, "xc");
              yc = new PropertyImpl(URI, "yc");
              Point = new ResourceImpl(URI+"Point");
             } catch (Exception e) {
              System.out.println("exception: " + e);
          }
      }
  }

  public class GeomFunctions {
  // PointPlus is vector addition
      public static Resource PointPlus(Resource x,Resource y) {
             Resource rs = x.getModel().createResource();
             rs.addProperty(RDF.type, GeomResources.Point);
             rs.addProperty(GeomResources.xc,
                x.getProperty(GeomResources.xc).getDouble() +
                y.getProperty(GeomResources.xc).getDouble());
             rs.addProperty(GeomResources.yc,
                x.getProperty(GeomResources.yc).getDouble() +
                y.getProperty(GeomResources.yc).getDouble());
            return rs;
             }
  }
```

The Fabl implementation, we would argue, is easier to understand and easier to code. The difference is not due to any defect of the Jena API, but to the inherent indirectness of the API approach. Further, the direct expression of RDF primitives in Fabl is less than half the story

with regards to ease of use. More significant is the fact that Fabl types are DAML+OIL classes, and type checking and polymorphism at the RDF level are implemented within the language. When using an API, type checking at the RDF level is the user's responsibility. For example, Java will not complain at compile time (nor run time) if the method GeomFunctions.PointPlus is applied to resources which are not members of GeomResources.Point.

6. Nurls

In normal RDF usage, locators (that is URLs) are often used as URIs whether or not the entities they denote exist on the web. However, nothing prevents the use of URIs which are completely unrelated to any web location, for example:

```
<rdf:Description rdf:about= "my_green_sedan">
```

Identifying an entity in a manner which does not make use of a WWW locator has two advantages. First, the question of where to find information about the entity is decoupled from naming the entity, which allows all of the different varieties of information about the entity to evolve without disturbing the manner in which the entity is named. Among other things, this simplifies the versioning of RDF data. Second, use of non-locating URIs frees up the content of the URI for expressing hierarchy information about the entities described.

In the Fabl implementation, the triple

```
<X,fabl:describedBy,U>
```

means that U denotes an RDF file which provides information about X. (rdfs:isDefinedBy [9] has a closely related, but not quite identical intent; descriptions need not always qualify as definitions). U is also taken as relevant to any subject Y whose URI (regarded as a pathname, with "." and "/" as delimiters) extends that of X. For example, if my_green_sedan is described by U, then so are my_green_sedan.engine, and my_green_sedan/engine but not my_green_sedan_attenna. The Fabl command:

```
getRdf(Y);
```

loads the files known to describe the resource Y; that is those files F for which the triple <X,fabl:describedBy,F> is present in the active graph, and Y is an extension of X. A typical Fabl initialization sequence involves first loading a configuration file containing fabl:describedBy statements which indicate where to find information about basic resources. Then, as additional resources become relevant to computation, invocations of getRdf bring the needed data into the active graph. In future, lazy strategies may be implemented in which, for example, getRdf(X) is automatically invoked on the first access to a property of X. Also, nothing precludes future development of complex discovery technology for finding relevant RDF, rather than relying only on the simple describedBy mechanism.

It is desirable that non-locating URIs not conflict with URLs. For Fabl applications, we have reserved the URI http://purl.oclc.org/net/nurl/ as a root URI whose descendants will never serve as locators. This line appears in our standard namespace declaration:

```
xmlns:nurl="http://purl.olc.org/net/nurl"
```

Nurl stands for "Not a URL". The following fragment tells the FablVM where to find the description of the Fabl language, and illustrates the points just made:

```
<rdf:Description about= "http://purl.oclc.org/net/nurl/fabl">
   <fabl:describedBy
rdf:resource="http://purl.oclc.org/net/fabl/languageV0"/>
</rdf:Description>
```

To access a future release of the language which resides at .../languageV1, only the configuration file need change; RDF which mentions language primitives via the namespace prefix "fabl:" may be left unchanged.

If U is the source URL of a page in the active graph, and the triple <X,fabl:describedBy,U> is present in the active graph, then X is said to be a subject of the page. That is, the page has as its subject matter the part of the hierarchical URI name space rooted at X. A page may have more than one subject. When a new triple <A,P,B> is created in the course of computation, and the URI of A is an extension of the subject of a page, the new triple is allocated to that page. (Slightly more complex rules - not covered here - govern the case where the subjects of pages overlap.)

7. Identifiers

Identifiers in Fabl represent XML qualified or unqualified names. However, since the "." character is reserved for property selection, "." is replaced by "\" when an XML name is transcribed into Fabl. For example:

```
fablex:automobiles\ford
```

is the Fabl identifier which would have been rendered as

```
fablex:automobiles.ford
```

in XML. The interpretation of unqualified names is governed by the path and the home namespace. The path is a sequence of namespaces. When an unqualified name U is encountered, the Fabl interpreter searches through the path for a namespace N such that N:U represents a node already present in the active graph. When a new unqualified name U is encountered, it is interpreted as H:U, where H is the home namespace. Normally, the "fabl:" and "xsd:" namespaces are included in the path, enabling unqualified reference to Fabl language primitives and XML Schema datatypes[10].

8. Types

Any RDF type is a legal Fabl type.

```
X.rdf:type = T;
```

asserts that X belongs to the type T; that is it adds the triple <X,rdf:type,T> to the active graph. (This statement is legal only if T is a daml:Class, not an XML Schema datatype). Of course, a value may have many types.

Fabl also includes its own primitives for constructing new types, that is, for introducing new resources whose type is rdfs:Class. The following lines of Fabl introduce the type Point which was discussed earlier.

```
class('Point');
var xsd:double geom:xc;
var xsd:double geom:yc;
endClass();
```

The constructors BagOf(T), ListOf(T), SeqOf(T), and Function(O,I_0,...,I_N) generate parametric types denoting the set of all bags (resp. lists,sequences) with members in type T, and of all functions from input types I_0,.... I_N to output type O, respectively.

Except for the parametric types, any Fabl statement which introduces a type is equivalent to a set of DAML+OIL statements about the type. This was illustrated by the definition of Point which appeared earlier. Only a part of the DAML+OIL formalism is used for this purpose. A new Fabl class can be introduced by subclassing a daml:Restriction. Within the daml:Restriction, properties may be restricted either (1) by daml:hasValue, or (2) by daml:toClass with an optional daml:maxCardinality or daml:cardinality restriction with value 1. The effect is that properties may be assigned values, or may be assigned types. If a property is assigned a type, it may optionally be restricted to have either exactly one, or at most one value of that type. A new class may also be introduced as a daml:intersectionOf existing classes. Any DAML+OIL class may appear as a legal Fabl type, because any RDF type at all can so appear, but Fabl syntax will only generate types in the subset just described, and Fabl's type deduction mechanisms will not fully exploit available information in types outside the subset. Coverage of more of DAML+OIL can be implemented in future extensions of Fabl without disturbing the correctness of code written for the current subset.

Here are the details. A Fabl class definition starts with

```
class('classname');
```

and ends with

```
endClass();
```

Within the definition, statements of the form

```
var pathname = expression;
```

called an assignment and

```
var [qualifier] [type] pathname;
```

called an assertion may appear. The possible values of the optional qualifier are exists, optional, and multivalued (exists is the default). A pathname is a sequence of names of properties, separated by dots (".") and represents sequential selection of the values of properties along the path. The assertion:

```
var [qualifier] type pathname;
```

means that, for all elements X of the class being defined, if v is a value of X.pathname, then v belongs to type. The qualifier exists (resp. optional, multivalued) means that X.pathname must have exactly one value (resp. at most one value, any number of values).

```
var [qualifier] pathname;
```

makes no claim about the type of X.pathname, only about the cardinality of the set of values which it assumes (depending on the qualifier).

The assignment

```
var pathname = expression;
```

means that the value of the slot denoted by the pathname is initialized to the value of the expression at the time when the member X is created, or when the class is installed (see below). Here are examples:

```
class('Rectangle');
var Point geom:center;
var geom:width;  //already declared to be a xsd:double in geom:
var geom:height; //already declared to be a xsd:double in geom:
endClass();

class('RedObject');
var color:color = color:red;
endClass();

class('RedRectangleCenteredOnXaxis');
var Rectangle && RedObject this;
var geom:center.geom:xc = 0.0;
endClass();
```

In the last example, The && operator denotes conjunction, and the pathname this refers to members of the class being defined, so that

```
var class this;
```

means that the class within whose definition the statement appears is a subclass of class.

The translation of Fabl class definitions into DAML+OIL RDF is straightforward. Assertions translate into toClass restrictions, and their qualifiers to cardinality or maxCardinality restrictions. Assignments translate into hasValue restrictions. The only minor complication is that, when pathnames of length greater than one appear, helper classes are automatically generated which express the constraints on intermediate values in path traversal (details ommitted).

9. Dynamic Installation of Classes

Recall that

```
x.rdf:type = C;
```

asserts that x belongs to daml:Class C. Such statements can be executed at any time, thereby dynamically adding class memberships. The effect of the statement is not just to add

the triple asserting class membership, but also to apply the constraints which C imposes on its members. Consider, for example:

```
var rect = new(Rectangle);

rect.rdf:type = RedRectangleCenteredOnXaxis;
```

Recall that RedRectangleCenteredOnXaxis asserts constant values for slots geom:center geom:xc and color:color. Consequently, after the assertion that x belongs to this class, the following triples are added to the graph:

```
<x,rdf:type,RedRectangleCenteredOnXaxis>
<x,color:colorOf,color:red>,
<x,geom:center,center-uri>,
<center-uri,rdf:type,geom:Point>
<center-uri,geom:xc,0.0>
```

Here, center-uri represents an anonymous node which has been created to represent the value of the geom:center property of x.

10. Implementation of Parametric Types

Here is the definition of BagOf:

```
class('BagOf');
var daml:Class this;
var memberType;
endClass();

daml:Class function BagOf(rdfs:Class tp)
{
  var BagOf rs;
  rs = new(BagOf);
  rs . memberType = tp;
  rs . uri = 'nurl:fablParametricTypes/' + 'BagOf(' + uriEncode(tp.uri) +
')';
  return rs;
}
```

This definition appears within the Fabl language definition, where memberType has already been declared to be a UniqueProperty of type rdfs:Class. The operator uriEncode encodes reserved characters (such as ":" and "/") as described in the URI standard [11]. Note that BagOf implements a one-to-one map from the URIs of types T to the URIs of BagOf(T). The implementation of the other parametric types ListOF, SeqOf, and Function are analogous. Fabl programmers can introduce their own parametric types using the same strategy.

11. Types of Fabl Expressions

Fabl is not just a language in which types may be created and manipulated, but a typed language in the more usual sense that each Fabl expression E is assigned an rdfs:Class. Of

course, any particular Fabl value (ie node in the active graph) may have arbitrarily many types, but a Fabl expression is assigned one of the types which the values of the expression is expected to assume. Types of function applications are deduced in the usual way. If a Fabl function f is defined by

```
O function f(I0 a0,... IN aN)
{
...
}
```

then the type of f(i0,...iN) is O if i0,...iN have types I0 ... IN. Range assertions are exploited in type deduction concerning property selections. If the triple

```
<P,rdfs:range,T>
```

is present in the active graph for property P, the expression

```
x.P;
```

is given type Bag0 f(T), unless P is asserted to be univalued, in which case the type of x.P is T. (If more than one range type is assigned to P, this is equivalent to assigning the conjunction of the range types.)

```
E ~ T
```

performs a type cast of the expression E to type T. Type casts are checked at runtime: if the value of E does not lie in T when E~T is executed, an error is thrown. Simple coercion rules are also implemented; for example ints coerce to doubles, and conjunctions coerce to their conjuncts.

12. Functions and Methods

A function definition:

```
O function fname (I0 a0,... IN aN))
{
...
}
```

adds a function to the active graph under the decorated name

```
'f'+hash(uri_encode(I0.uri),...uri_encode(IN.uri),fname)+'_'+fname;
```

The purpose of decoration is to support polymorphism by assigning different URIs to functions whose input types differ. If the function definition appears within the scope of a class definition, the function is added beneath the URI of the class, and is invoked in the usual manner of methods: <object>.fname(...). If preceded by the optional keyword final a method cannot be overridden. The effect of a Java abstract method is obtained by including a property of functional type in a class definition. Overriding of methods takes place as a side effect of class installation when the class being installed assigns values to functional properties. This simple treatment of method overriding is more flexible than conventional treatments; for

example, dynamic installation of classes may change the set of methods installed in an object at any time, not only at object-creation time as in Java or C++. These points are illustrated by examples just below. The Fabl expression:

```
f[I0,...In]
```

denotes the variant of f with the given input types. For example, twice[Seq0 f(xsd:int)] denotes the variant of twice which takes sequences of ints as input. The Fabl operator:

```
supplyArguments(functionalValue,a0,...aN)
```

returns the function which results from fixing the first N arguments to functionalValue at the values of a_0,..a_N. Now, consider the following code:

```
class('Displayable');
var Function(fabl:void) Displayable\display;
...
endClass();
```

Note that by giving Displayable\display as the name of the functional property, we have allocated a URI for that property in the hierarchy beneath the URI for Displayable. This technique can be used in any context where a property which pertains to only one class is wanted. Consider also a concrete variant which displays rectangles:

```
fabl:void function display(Rectangle r)
{
....
}
```

Then, with

```
class('DisplayableRectangle');
var Displayable && Rectangle this;
var Displayable\display = supplyArguments(display[Rectangle],this);
endClass();
```

a class is defined which is a subclass of both Rectangle and Displayable, and which assigns concrete functions to the corresponding functional properties in the latter class. This is similar to what happens when a C++ or Java class contains a virtual method which is implemented by a method defined in a subclass. As noted earlier, the wiring of virtual methods to their implementations can only take place at object creation time in Java or C++, and cannot be undone thereafter, whereas Fabl allows wiring of functional properties to their implementations to take place at any time during a computation, via, for example

```
someRectanglePreviouslyUndisplayable.rdf:type = DisplayableRectangle;
```

Fabl supports assertion of constraints as part of class definitions - constraints which are applied to members at class installation time, and maintained thereafter by a constraint propagation mechanism. The constraint facility is beyond the scope of this paper.

13. Code as RDF

The foregoing discussion has described how Fabl data and types are rendered as sets of RDF triples. The remaining variety of Fabl entity which needs expression in RDF is code.

Code is represented by elements of the class fabl:Xob (Xob = "eXecutable object"). Xob has subclasses for representing the atoms of code (global variables, local variables, and constants), and for the supported control structures (blocks, if-else, loops, etc). Here is the class Xob:

```
class('Xob');
//atomic Xob classes such as Xlocal do not require flattening
var optional Function(Xob,Xob) Xob\flatten;
var rdfs:Class Xob\valueType;
endClass();
```

Subclasses of Xob include:

```
class('Xconstant'); //Constant appearing in code
var Xob this;
var Xconstant\value;
endClass();

class('Xlocal'); //Local variable
var Xob this;
var xsd:string Xlocal\name;
endClass();

class('Xif');
var Xob this;
var Xob Xif\condition;
var Xob Xif\true;
var optional Xob Xif\false;
endClass();

class('Xapply'); //application of a function to arguments
var Xob this;
var AnyFunction Xapply\function;
var SeqOf(Xob) Xapply\arguments;
endClass();
```

(The type AnyFunction represents the union of all of the function types Function(O,I_0...I_N)) The Fabl statement

```
if (test(x)) action(2);
```

translates to the Xob given by this RDF:

```
<fabl:Xif>
   <fabl:Xif.condition>
      <fabl:Xapply>
          <fabl:Xapply.function rdf:resource="#f001a0e6f_test"/>
          <fabl:Xapply.arguments>
              <rdf:seq>
                  <rdf:li>
                      <fabl:Xlocal>
                          <fabl:Xlocal.name>x</fabl:Xlocal.name>
                      </fabl:Xlocal>
                  </rdf:li>
              </rdf:seq>
          </fabl:Xapply.arguments>
      </fabl:Xapply>
   </fabl:Xif.condition>

<fabl:Xif.true>
      <fabl:Xapply>
          <fabl:Xapply.function rdf:resource="#f001a0e6f_action"/>
          <fabl:Xapply.arguments>
              <rdf:seq>
                  <rdf:li>
                      <fabl:Xconstant Xconstant.value=2/>
                  </rdf:li>
              </rdf:seq>
          </fabl:Xapply.arguments>
      </fabl:Xapply>
   </fabl:Xif.true>
<fabl:Xif>
```

f001a0e6f_action is the decorated name of the variant of action which takes an xsd:int as input. Verbose as this is, it omits the Xob properties. Correcting this omission for the Xlocal would add the following lines in the scope of the Xlocal element:

```
<rdf:type rdf:resource = "http://purl.oclc.org/net/nurl/fabl/Xob"/>
<fabl:Xob.valueType rdf:resource =
"http://www.w3.org/2000/10/XMLSchema:int"/>
```

(The Xob\flatten property does not appear because Xlocals do not require flattening). A full exposition of the set of all Xob classes is beyond the scope of this paper, but the above examples should indicate the simple and direct approach taken. The class

```
class('Xfunction');
var xsd:string Xfunction\name;
var rdfs:Class Xfunction\returnType;
var SeqOf(Xlocal) Xfunction\parameters;
var SeqOf(Xlocal) Xfunction\localVariables;
var Xob Xfunction\code;
var SeqOf(xsd:byte) Xfunction\byteCode;
endClass();
```

defines an implementation of a function. When a Fabl function is defined, the code is analyzed, producing an Xfunction as result. This Xfunction is assigned as the value of the decorated name of the function.

The following steps are involved in translating the source code of a Fabl function or command into an Xfunction:

```
Source code [Parser] ->
Parse tree   [Analyzer] ->
Type-analyzed form (Xob) [Flattener]->
Flattened form (Xob) [Assembler] ->
Byte Code (executed by the Fabl virtual machine)
```

All of these steps are implemented in Fabl. The parse tree is a hierarchical list structure in the Lisp tradition whose leaves are tokens; a token in turn is a literal annotated by its syntactic category. A flat Xob is one in which all control structures have been unwound, resulting in a flat block of code whose only control primitives are conditional and unconditional jumps. Separating out flattening as a separate step in analysis supports extensibility by new control structures, as will be seen in a moment.

The analysis step is table driven: it is implemented by an extensible collection of constructors for individual tokens. The constructor property of a token is a function of type Function(Xob,daml:List) which, when supplied with a parse tree whose operator is the token, returns the analysis of that tree. Here is the code for the constructor for if. The parse of an if statement is a list of the form (if <condition> <action>).

```
Xob function if_tf(daml:List x)
  {
  var Xob cnd,ift,Xif rs;
  cnd = analyze(x[1]); //the condition
  if (cnd.Xob\valueType!=xsd:boolean) error('Test in IF not boolean');
  ift = analyze(x[2]);
  rs = new(Xif);
  rs . Xif\condition = cnd;
  rs . Xif\true = ift; //no value need be assigned for Xif\false
  return rs;
  }
```

x[N] selects the Nth element of the list. Then, the statement

```
ifToken.constructor = if_tf[daml:List];
```

assigns this function as the constructor for the if token. More than one constructor may be assigned to a token; each is tried in turn until one succeeds.

The Xif class, like other non-primitive control structures, includes a method for flattening away occurences of the class into a pattern of jumps and gotos (details omitted). Constructors and flattening methods rely on a library of utilities for manipulating Xobs, such as the function metaApply, which constructs an application of a function to arguments, and metaCast which yields a Xob with a different type, but representing the same computation, as its argument.

This simple architecture implements the whole of the Fabl language. The crucial aspect of the architecture is that it is fully open to extension within RDF. New control structures, type constructors, parametrically polymophic operators, annotations for purposes such as aspect-oriented programming [12], and other varieties of language features can all be introduced by loading RDF files containing their descriptions. The core Fabl implementation itself comes into being when the default configuration file loads the relevant RDF; a different configuration file drawing on a different supply of RDF would yield another variant of the language. This is the sense in which the implementation provides an open framework for describing computation in RDF, rather than a fixed language.

Finally, note once again that Xobs provide a formalism for representing computation in RDF which does not depend for its definition on any particular source language nor on any particular method for execution. That is, it formalizes computation within RDF, as promised by the title of the paper, and can yield the benefits sketched in the introduction.

14. Implementation

The practicality of an RDF-based computational formalism is a central issue for this paper, so size and performance data for our initial implementation are relevant.

The implementation consists of a small kernel written in C. The size of the kernel as a WinTel executable is 120 kilobytes. The kernel includes the byte-code interpreter, a generation-scavenging garbage collector, and a loader for our binary format for RDF. The remainder of the implementation consists of Fabl's RDF self description, which consumes 700 kilobytes in our RDF binary format. A compressed self-installing version of the implementation, which includes the Fabl self description, consumes 310 kilobytes. Startup time (that is, load of the Fabl self description) is about one third of a second on a 400MHZ Pentium II. Primitive benchmarks show performance similar to scripting languages such as JavaScript (as implemented in Internet Explorer 5.0 by Jscript) and Python. However, further work on performance should yield much better results, since the language is strongly typed, and amenable to many of the same performance techniques as Java.

The full value of formalizing computation within RDF will be realized only by an open standard. We regard Fabl as a proof-of-concept for such a formalization. In the context of a standards effort, we would be willing to contribute as Open Source whatever part of Fabl's implementation is found to be relevant.

15. Future Work

The current Fabl implementation treats the external RDF world as a store of RDF triple sets, which are activated explicitly via loadRdf or getRdf. However, an interesting direction for future work is the definition of a remote invocation mechanism for RDF-based computation. Here is an outline of one possibility.

Values of the fabl:describedBy property might include URLs which denote servers as well as RDF pages. In this case, getRdf(U) would not load any RDF. Instead, a connection would be made to the server (or servers) S designated by the value(s) of fabl:describedBy on U. In this scenario, the responsibility of S is to evaluate properties, globals, and functions in the URI hierarchy rooted at U. Whenever a property E.P, a global G, or a function application F(x) is evaluated in the client C, and the URI of E, G, or F is an extension of U, a request is forwarded to the server S, which performs the needed computation, and returns the result. The communication protocol would itself be RDF-based, along the lines proposed on the www-rdf-interest mailing list[13]. Such an approach would provide simple and transparent access to distributed computational resources to the programmer, while retaining full decoupling of description of computation in RDF from choices about source language and implementation.

16. Other XML Descriptions of Computation

Imperative computational constructs appear in several XML languages. Two prominent examples are SMIL [14] and XSLT [15], in which, for example, conditional execution of statements is represented by the <switch> and <xsl:if> tags, respectively. The aims of these formalizations are limited to specialized varieties of computation which the languages target. Scripting languages encoded in XML include XML Script [16] and XFA Script [17].

Footnotes

1. Fabl™ is a trademark of The Behavior Engine Company, and is pronounced "fable".
2. The standardization of JavaScript is ECMAscript [18]

References

[1] W3C RDF Model and Syntax Working Group. Resource Description Framework (RDF) Model and Syntax Specification, http://www.w3.org/TR/REC-rdf-syntax/, February 1999

[2] Ian Horrocks, Frank van Harmelen, Peter Patel-Schneider, eds. DAML+OIL (March 2001), http://www.daml.org/2001/03/daml+oil-index, March 2001

[3] Chris Waterson. RDF: back-end architecture, http://www.mozilla.org/rdf/back-end-architecture.html, August 1999

[4] Brian McBride. Jena - A Java API for RDF, http://www-uk.hpl.hp.com/people/bwm/rdf/jena/index.htm, May 2001 (last update)

[5] Tim Bray, Dave Hollander, Andrew Layman, eds. Namespaces in XML, http://www.w3.org/TR/REC-xml-names/, January, 1999

[6] The Dublin Core Metadata Initiative, http://dublincore.org, July 2001 (last update)

[7] The Behavior Engine Company, http://www.behaviorengine.com, July 2001 (last update)

[8] Ian Horrocks, Frank van Harmelen, Peter Patel-Schneider, eds. A Model-Theoretic Semantics for DAML+OIL (March 2001), http://www.daml.org/2001/03/model-theoretic-semantics.html, March 2001

[9] Dan Brickley, R.V. Guha, eds. Resource Description Framework (RDF) Schema Specification 1.0, http://www.w3.org/TR/2000/CR-rdf-schema-20000327/, March 2000

[10] Paul V. Biron, Ashok Malhotra, eds. XML Schema Part 2: Datatypes, http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/, May 2001

[11] T. Berners-Lee. Uniform Resource Identifiers (URI): Generic Syntax, http://www.ietf.org/rfc/rfc2396.txt, August 1998

[12] Gregor Kiczales, John Lamping, Anurag Mendhekar and Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John Irwin. Aspect-Oriented Programming, 11th European Conference on Object-Oriented Programming, LNCS, vol. 1241, Springer Verlag, 1997

[13] Ken MacLeod, and respondents. Toying with an idea: RDF Protocol, http://lists.w3.org/Archives/Public/www-rdf-interest/2001Mar/0196.html, March 2001

[14] Jeff Ayars et al, eds. Synchronized Multimedia Integration Language (SMIL 2.0) Specification, http://www.w3.org/TR/2001/WD-smil20-20010301/, March 2001

[15] James Clark, ed. XSL Transformations (XSLT), http://www.w3.org/TR/xslt, November 1999

[16] DecisionSoft Limited, XML Script, http://www.xmlscript.org/ May 2001

[17] XML For All, Inc. XFA Script, http://www.xmlforall.com/cgi/xfa?XFAScript, May 2001

[18] ECMA. Standard ECMA-262 – ECMAScript Language Specification, http://www.ecma.ch/ecma1/stand/ecma-262.htm, December 1999