Ruby on Semantic Web

Vadim Eisenberg, Yaron Kanza
Computer Science Department, Technion
Haifa, Israel
{eisenv,kanza}@cs.technion.ac.il

***Abstract*—The *impedance mismatch problem* that occurs when relational data is being processed by object-oriented (OO) programs, also occurs when OO programs process RDF data, on the Semantic Web. The impedance mismatch problem stems from the inherent differences between RDF and the data model of OO languages. In this paper, we illustrate a solution to this problem. Essentially, we modify an OO language so that RDF individuals become first-class citizens in the language, and objects of the language become first-class citizens in RDF. Three important benefits that follow from this modification are: (1) it becomes natural to use the language as a persistent programming language, (2) the language supports implicit integration of data from multiple data sources, and (3) SPARQL queries and inference can be applied to objects during the run of a program. This demo presents such a modified programming language, namely *Ruby on Semantic Web*, which is an extension of the Ruby programming language. The demo includes a system, where users can run applications, written in *Ruby on Semantic Web*, over multiple data sources. In the demo we run code examples. The effects of the execution on the data sources and on the state of the objects in memory are presented visually, in real time.**

I. Introduction

The *Semantic Web* is a collection of technologies for managing linked data on the World-Wide Web [1]. Its vision is to transform the Web into one global database, using URIs and common data models. A more modest goal is to use the Semantic Web technologies for Enterprise Information Integration (see [2]), by considering all the data of an enterprise, or of a closed community, as one global database, creating a *Corporate Semantic Web*.

In the Semantic Web, data are represented and accessed using the RDF data model. In RDF, the data items are called *individuals*, they have *properties* attached to them, and they can be linked one to another using the properties, forming a graph. Thus, applications in Object-Oriented (OO) programming languages over the Semantic Web, need to cope with RDF individuals. However, there are inherent differences between objects of OO programming languages and RDF individuals. Essentially, differently from objects, RDF individuals have URIs, properties and they can be members of more than one class. Differently from RDF individuals, objects in OO languages have methods and they are members of a single direct (declared) class. These differences cause an *impedance mismatch problem*, similar to the impedance mismatch between OO languages and relational databases.

In this demo, we illustrate a solution to the impedance mismatch problem. The solution is based on using a unified data model for both RDF and the OO languages. In the unified model, (1) individuals can have methods (2) objects must have URIs and properties, and (3) an object can be a member of more than one class. In a programming language over the unified model, RDF individuals become first-class citizens of the language while maintaining all the capabilities of object-oriented programming. Objects of the OO language become first-class citizens of RDF query languages and reasoning tools.

For comparison, the traditional approach for processing RDF in OO applications is by applying an RDF-to-OO mapping that exchanges the RDF individuals to objects, e.g., ActiveRDF [3]. In such systems, the RDF individuals are mapped to "proxy" objects that represent them during runtime, without handling the impedance mismatch problem. Our approach has the following advantages over a mapping.

First, treating RDF individuals as first-class citizens of a programming language allows compilers, and other language tools, to exploit special features of RDF for increasing programming productivity and for improving performance, e.g., using these features for compiler optimizations, syntax checks and refactoring. Adding methods to RDF enriches the RDF framework, in the same way that stored procedures enrich relational databases in an RDBMS.

Second, by using the unified model the language becomes a *persistent programming language*—any object can be defined as persistent and can be stored implicitly, relieving the programmer from the burden of explicitly managing persistence (see [4], [5]). This is because RDF and related Semantic-Web technologies are designed for representing and storing data on the Web. Thus, by considering the objects of an application as RDF individuals, it becomes natural to store them on the Web. Furthermore, adding URIs to the objects of a programming language enables associating between objects and data sources. Such associations facilitate data integration by representing in a natural way the mapping of objects to data sources.

A third advantage of our approach is the ability to directly apply in the language Semantic-Web tools for querying RDF and for reasoning over the data, as will be illustrated later.

Due to the above reasons, our solution is not another mapping between RDF and OO. It is a programming language with a unified model—a hybrid of RDF and OO—where artifacts from both models are first class citizens of the language. In the proposed language, reasoning and querying can be applied over both the objects of the language and RDF individuals. Furthermore, the same code that is used for processing in-memory objects can also be used for processing RDF individuals on the Web. The model also provides a synergy between language tools, e.g., SPARQL queries and OWL ontologies can include references to methods, and methods can embed SPARQL queries and apply reasoning, implicitly.

In order to demonstrate our approach, we implemented an extension to the Ruby programming language, called *Ruby on Semantic Web*. In this demo, we present the extended language and its capabilities.

In the following section we present the principles of a programming language that uses the unified data model. In Section III, we provide an example that illustrates the advantages of the unified model. We describe the main implementation details and the architecture of *Ruby on Semantic Web* in Section IV. Finally, in Section V, we present our demo, using the scenario described in Section III.

II. Principles of the Hybrid Model

In this section we present the principles of the hybrid RDF-OO data model. These principles guarantee that objects will be first-class citizens in the Semantic Web and RDF individuals will be first-class citizens of OO programming languages.

URIs and Access Transparency. Every object has a URI, similar to RDF individuals. Any object of the language and any RDF individual can be accessed by a URI, providing *access transparency*, as defined in [6]. The URIs provide direct access to objects, as if they are data items in a database. A method in the language can either access an object directly by a reference to it (a pointer) or using a URI. For instance, a program may store a list of object URIs on a disk, and later read this list for accessing these objects.

Property Attachment. RDF properties can be added to the Ruby objects. Note that properties are being added dynamically, during runtime, not just during the construction of the objects.

Procedural Attachment. Every class, including RDF classes, can have methods. The methods are called on the instances of the class. Procedural attachment is mentioned as a useful feature in [7]. All the methods have URIs, similar to properties in RDF, and they are serializable, conforming to an RDF format. Thus, it is possible to call methods using their URIs, directly, bypassing the regular dispatch mechanisms of the language.

Multiple Class Membership. Every object can be a member of multiple classes, and the classes can be changed dynamically, during runtime, as in the Semantic Web. Message dispatch is being done as in languages with multiple inheritance.

Orthogonal Persistence. Adding URIs to objects makes the language *orthogonally persistent* in the sense that persistence is implemented as an intrinsic property of the execution environment (see [8]). The URI specifies how to access an object that is not in memory and how to reflect changes to objects that are in memory.

Location and Migration Transparency. For every persistent object, the storage location of its instance can be determined based on the URI of the object. This yields a mapping logic that is separated from the program, providing *location transparency*, as defined in [6]. Programmers manipulate the data using only URIs, and they are relieved from specifying the locations of the data. A program is written as if it works over the entire World-Wide Web, while the boundaries of this virtual world are defined by some mapping between URIs and data sources. The data sources and the mapping can be modified without changing anything in the application logic, which is known as *migration transparency* (see [6]). The program is written as if all the data being accessed are in RDF, while in reality data reside in different sources, are represented in different models and are stored in various formats. To that end, data are mapped on-the-fly to RDF.

Persistence Independence. According to [8], a beneficial design principle of a persistent programming language is *persistence independence*, where the programmer should be oblivious to whether objects are persistent or not, i.e., the code should operate in the same manner on both persistent and transient objects. In our implementation, the values of properties can be retrieved and updated by applying "." (dot) and "="operators of Ruby, using a syntax that is similar to method calls and assignments. This syntactical similarity makes it possible to call methods as if they are RDF properties. Consequently, in our prototype it is possible to pose SPARQL queries over all the objects of the language, both the in-memory objects and the ones representing persistent data. This is similar to the LINQ technology [9] in which SQL-like queries can be posed over the native objects and arrays of a programming language. Being able to pose SPARQL queries on all the objects of the language makes these objects "equal citizens" in this aspect, and satisfies the *persistence independence* property.

Language-Integrated Queries. We implemented embedding of SPARQL and SPARQL/Update queries in Ruby on Semantic Web. This makes the queries *language-integrated* (first-class citizens of the language), as in LINQ.

Logical Inference is one of the important and powerful features of the Semantic Web, and it can be naturally applied in our proposed unified model. This means that once a Semantic-Web reasoner is "plugged into" the language environment, the operations of the language are executed while considering the inferred information.

The synergy of all the features listed above should provide a significant impact on the productivity of writing data-processing applications, especially when processing integrated data, using Semantic-Web technologies.

III. Illustrating the Approach

We illustrate the advantages of the hybrid data model by an example.

Setting: Consider an information system in a made-up company. The company is divided into departments and each department is divided into groups. Each employee can be: (1) a regular employee that belongs to a group, (2) a group manager, (3) a department manager, or (4) the CEO of the company. The *direct manager* is defined according to the structure of the company, i.e., for a regular employee it is his group manager, for a group manager it is her department manager, etc. *Indirect manager* is defined as the transitive closure of the direct-manager relationship.

The data in the company is distributed in the following way. There are two relational databases and an RDF triple store. The first relational database contains a table EMPLOYEES. The second relational database contains two tables: GROUPS and DEPARTMENTS with information about the managers and the structure of the company. An RDF triple store contains information about the groups and the employees in each group.

Since direct managers of employees are implied from the hierarchy in the company, they are not explicitly represented in the data, in order to simplify changes in the data, and to prevent inconsistencies and data duplication. The definition of *indirect manager* is codified in an OWL ontology.

In addition to the persistent data, the company monitors entrances of employees into its premises, e.g., using a card reader that for each entry creates an object of class *EnteredPerson*, with the ID of the entered person and the entrance time as its fields. The object is discarded when the person leaves.

Task: *The CEO of the company is interested in receiving an absence report showing the percentage of absent employees per their* <u>*indirect*</u> *manager.*

The task involves four subtasks: (1) finding the direct manager of each employee, (2) computing all the indirectly-managed employees, for every manager, (3) counting the number of the indirectly-managed employees, and (4) querying which of the indirectly-managed employees are represented by the objects of class *EnteredPerson* (meaning they are present).

Accomplishing the task by the means of SPARQL alone is **not feasible**, because a *count* aggregation function is not part of standard SPARQL,[1] yet it is required for the computation in Subtask 3. Furthermore, Subtask 4 requires to pose the SPARQL queries on the native objects of an OO language, and this cannot be done by Semantic-Web tools, without employing the hybrid data model.

A different approach is to use an OO language together with an RDF-to-OO mapping. In this solution, data about employees, groups and departments are mapped to "proxy" objects. Ad-hoc reasoning must be programmed for calculating the indirectly-managed employees (Subtask 2). In addition, it is impossible to run SPARQL queries on the the objects of the language, so a regular code, instead of queries, must be written to accomplish subtasks 1, 2 and 4.

In a programming language with the hybrid model, the task can be accomplished as follows. A method *hasManager* is added to the RDFS class *Employee*, so it can be used in SPARQL queries, in OWL ontologies and can be subjected

[1]The lack of count is merely an example. Even if count will be added to the SPARQL standard, there will be other functions we would like to express in an ordinary programming language, when querying the data.

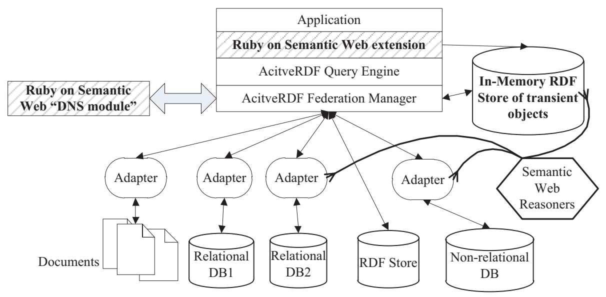


Fig. 1. System architecture

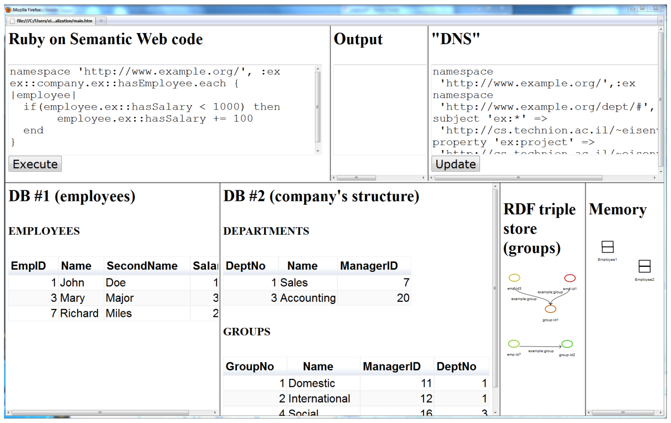
to reasoning. The method itself is written in our programming language, so it can use SPARQL queries for accomplishing its goal. The *manages* relationship and the *indirectly-manages* relationship can be computed by using a SPARQL query and applying a standard OWL reasoner over the ontology of the company. Subtask 4 can be accomplished by posing a SPARQL query over both the RDF data and the in-memory objects of class *EnteredPerson*. A *count* method, implemented in the OO language, can be called over the objects that represent the present employees, and over the RDF individuals that represent all the employees.

Our approach facilitates handling tasks such as the one we presented. In our model, the subtasks can be accomplished in a natural and succinct way, by using RDF methods, OWL ontologies, SPARQL queries and standard OWL reasoners. Without these tools it is less productive to perform the subtasks—some ad-hoc code must be written instead of using general reasoners and query engines, the code is more verbose and the application is more sensitive to possible changes in the company definitions.

Note that in our language, programmers access the data as if all the data reside in one source and they are all in RDF format. This is achieved by federating data sources and using adapters to RDF (see Section IV). The approach provides flexibility where programmers do not discern between persistent and transient entities. For instance, making the objects of class *EnteredPerson* persistent does not require changing the code, only specifying a mapping to a data source. Similarly, changing the definition of the *indirectly-manages* relationship can be done by changing the ontology, rather than by changing the code.

IV. Architecture

In order to demonstrate programming with the hybrid model, we implemented an extension of the Ruby programming language [10], namely *Ruby on Semantic Web.* We decided not to develop a new programming language from scratch, but to graft the desired features on an existing OO programming language, in our case, on Ruby. We chose Ruby since it is an Object-Oriented dynamic programming language with high metaprogramming capabilities and flexible syntax. We built our prototype on top of ActiveRDF [3]—a Ruby library for manipulating RDF data, and we used implementation ideas from [3].



Ruby on Semantic Web code

```
namespace 'http://www.example.org/', :ex
ex::company.ex::hasEmployee.each {
|employee|
  if(employee.ex::hasSalary < 1000) then
       employee.ex::hasSalary += 100
  end
}
```

Output

"DNS"

```
namespace
 'http://www.example.org/',:ex
namespace
 'http://www.example.org/dept/#',
subject 'ex:*' =>
 'http://cs.technion.ac.il/~eisen
property 'ex:project' =>
```

DB #1 (employees)

EMPLOYEES

EmpID	Name	SecondName	Sala
1	John	Doe	1
3	Mary	Major	3
7	Richard	Miles	2

DB #2 (company's structure)

DEPARTMENTS

DeptNo	Name	ManagerID
1	Sales	7
3	Accounting	20

GROUPS

GroupNo	Name	ManagerID	DeptNo
1	Domestic	11	1
2	International	12	1
4	Social	16	3

Fig. 2. Screenshot of the application visualizing Ruby on Semantic Web

The architecture of Ruby on Semantic Web is depicted in Figure 1. The prototype is built on top of the ActiveRDF architecture [3]. The parts that we added appear hatched.

In the figure, the enterprise data reside in several data sources—in documents, in several relational databases, in an RDF Store and in some non-relational database. There are adapters for transforming SPARQL queries and SPARQL/Update statements into queries/statements suitable to each data source, for example to SQL in case of a relational database. We extended a popular RDB to RDF mapping platform—D2RQ [11], to handle SPARQL/Update statements. The adapters provide virtual RDF views on all the data.

The adapters are connected to the ActiveRDF Federation Manager—the layer that is responsible for distributing queries among multiple sources and aggregating the results. The arrows are bidirectional, meaning that both querying and updating of data sources is enabled. Ruby on Semantic Web uses the ActiveRDF Query Engine to execute queries.

The system includes a "DNS" module that maps URIs of the objects to data sources, in order to determine which data source new data should be written to. While reading values of properties of individuals and running SPARQL queries can be done on a collection of federated sources, without specifying in which data source each piece of information resides, writing properties and running SPARQL/Update statements requires specifying where the new pieces of information must be added.

An additional data source—an in-memory RDF store—is used to represent and to query information about the native Ruby objects during runtime. The type of every new Ruby object and all the RDF properties attached to it, are added to the in-memory RDF store during runtime.

Semantic-Web reasoners can be plugged into the in-memory RDF store and into the adapters of other data sources, in order to enable logical inference over the data.

An application that runs on top of Ruby on Semantic Web is unaware of all the lower layers of the system. The application works with URIs and is oblivious to where the data reside, what are the formats of the data, which data items are stored and which are inferred by the reasoners. The application can access any object by URI and can run SPARQL queries on all the objects. All these provide implicit data integration, orthogonal persistence, persistence independence, and also access, location and migration transparency.

V. Demo Scenario

In the demo we present a simplified scenario of processing data from different data sources in a made-up organization, using Ruby on Semantic Web. The goals of the demo are (1) to illustrate the simplicity of programming in Ruby on Semantic Web (2) to illustrate implicit data integration in our system, and (3) to show how persistence is implicit in our system.

In the demo, the data in the made-up company, described in the section III is visualized. In addition we present code examples of Ruby on Semantic Web, to demonstrate the following capabilities of the language: (1) Reading and updating data of different sources, implicitly—without specifying the location of the data. (2) Changing the data configuration in the organization by changing the rules in the "DNS" module. (3) Running SPARQL queries, integrated in the code, on the data in the sources and on the objects in memory (i.e. the non-persistent objects). (4) Applying inference, in code statements and in queries, over the data. (5) Making in-memory objects persistent by changing "DNS" rules.

The demo is a web application that visualizes the persistent entities—those in the data sources, and the in-memory objects (see Figure 2). It contains a text area for entering code and "DNS" rules. The output produced by the code execution appears in a text area for the output. The changes in the data sources or in memory are shown in separate frames.

Acknowledgment

This work was partially supported by the Israeli Ministry of Science and Technology, Grant 3/6472.

References

[1] T. Berners-Lee, J. A. Hendler, and O. Lassila, "The Semantic Web," *Scientific American*, vol. 284, no. 5, May 2001.
[2] A. Y. Halevy *et al.*, "Enterprise Information Integration: successes, challenges and controversies," in *SIGMOD '05*.
[3] E. Oren *et al.*, "ActiveRDF: Object-Oriented Semantic Web Programming," in *WWW '07*.
[4] M. Atkinson, "Persistence and java - a balancing act," in *Objects and Databases*, ser. Lecture Notes in Computer Science, vol. 1944, 2001.
[5] M. Atkinson *et al.*, "The Object-Oriented Database System manifesto," in *Proceedings of the 1st Intl. Conf. on Deductive and Object-Oriented Databases*, 1989.
[6] W. Emmerich, *Engineering Distributed Objects*. John Wiley & Sons, 2000.
[7] "OWL Web Ontology Language Use Cases and Requirements," W3C, Tech. Rep., Feb. 2004.
[8] M. P. Atkinson *et al.*, "An orthogonally persistent java," *SIGMOD Rec.*, vol. 25, no. 4, 1996.
[9] E. Meijer, B. Beckman, and G. Bierman, "LINQ: Reconciling object, relations and XML in the .NET framework," in *PODS 2006*.
[10] D. Flanagan and Y. Matsumoto, *The Ruby Programming Language*. O'Reilly, 2008.
[11] C. Bizer and A. Seaborne, "D2RQ - treating non-RDF databases as virtual RDF graphs," in *In Proceedings of the 3rd International Semantic Web Conference (ISWC2004)*, 2004.