A Flexible Integration Framework for Semantic Web 2.0 Applications

Eyal Oren, Armin Haller, Manfred Hauswirth, Benjamin Heitmann, and Stefan Decker, *National University of Ireland, Galway*

Cédric Mesnage, *University of Lugano*

The Semantic Web application framework (SWAF) extends Ruby on Rails to enable rapid development of integrated Semantic Web mash-ups.

Current Web applications are mostly database driven. Developers design a database schema and then construct the application logic (which generates Web pages for user interaction) on top of the schema. These applications are centralized and rely on their own relational database, limiting the possibilities for data integration. *Mash-ups* (often called Web 2.0 applications) are an emerging Web development paradigm that combines functionality from different Web applications.

You could consider mash-ups as dynamic software compositions that combine components into a larger functionality. Whereas traditional software composition spans both functional and data composition, the challenges of composing Web applications are primarily in integrating their data sources.

Consider, for example, the development of a social networking application on the Web. Such an application would let users define their personal profile (such as their name, address, interests, and education), and then search for and add people to their list of friends. Other users could browse the social network, discover links between people, and calculate degrees of separation.

The Semantic Web is a web of data that can be processed by machines, enabling data reuse and integration on top of the existing Web.[1] Building a social networking application on top of Semantic Web data rather than on a traditional database would offer several advantages:

- Users could publish their personal profiles online using, for example, the friend-of-a-friend (http://foaf-project.org) vocabulary.
- Users could express their favorite music, hobbies, pets, and so on, using different vocabularies, unrestricted by the schema used by the Web application.
- In addition to information about the user and the user's social network, profiles could contain information about other people. For example, Benjamin can list Cédric's workplace and Armin's friends, which can differ (deliberately or not) from the information in Cédric's or Armin's profile.
- Semantic Web data, with its global identifiers and graph-based model, supports data integration by following the links in

0740-7459/07/$25.00 © 2007 IEEE

the graph and fetching additional data on demand from online sources.

Composing data sources on demand requires features typical of dynamically typed languages[2] such as late binding and dynamic typing, which alleviate the need for a priori agreement on data structures and programmatic interfaces. Our Semantic Web application framework (SWAF) largely automates the composition of online data sources, reducing the application development cost. It offers a fast, cheap, and flexible integration of interesting, semantically rich data. The framework accommodates changes in data sources, schemas, and instance data.

Programming with Semantic Web data

The Semantic Web uses the Resource Description Framework[3] as the representation model. Statements in RDF are triples, consisting of a subject, a predicate, and an object. A triple asserts that a subject has a property with some value. For example, the triple `Cedric rdf:type foaf:Person` states that Cédric is a person. RDF Schema[4] is the schema language for the Semantic Web. It lets you describe vocabularies in terms of classes and properties. For example, the triple `foaf:name rdfs:domain foaf:Person` states that `name` is a property of a `person`. For readability, the combination of RDF and RDF Schema is often referred to as RDF(S).

Decentralized data

Although the Semantic Web is data oriented and the World Wide Web is document oriented, both are fundamentally decentralized, heterogeneous, and open. The Semantic Web isn't a global database, centralized in one location with one agreed-upon schema and one meaning. Instead, anyone can make any statement at any location, using any vocabulary or structure. In contrast, as table 1 shows, traditional database-driven Web applications are centralized, with a fixed schema, vocabulary, and data source.

Conceptual and physical decentralization leads to

- naming differences, because one person might describe a book's "author," a second the book's "writer," and a third its "creator";
- differences in data structures, because some users might describe their language skills in their personal profiles while others describe their pets or their favorite colors; and
- federated storage, because statements can be published to any Web location without central registration.

Table 1

Traditional and Semantic Web applications.

Web applications	Semantic Web applications
Centralized	Decentralized
One fixed schema	Semistructured
One fixed vocabulary	Arbitrary vocabulary
Centralized publishing	Publish anywhere
One data source	Many distributed data sources
Closed systems	Open systems

The Semantic Web has no central data repository, no central agreement on meaning, and no central policy on terminology or structure.

Incompatible models

Developing Semantic Web applications requires handling the RDF(S) data model in a programming language. Although most software development follows the object-oriented paradigm, programming in RDF is triple-based. An object-oriented API for Semantic Web data would map RDF Schema classes to programming classes, RDF resources to programming objects, and RDF predicates to methods on those objects. Such an API lifts data elements into first-class citizens in the language (for example, `http://xmlns.com/foaf/0.1/firstName => Person.firstName`), thus making it easier to handle them.

The semantics of classes and instances in RDF Schema assume an open-world model and are based on description logic. Static object-oriented type systems, on the other hand, typically assume a closed-world model and so are constraint-based. As table 2 demonstrates, this fundamental semantic difference affects Web application frameworks that, based on the relational model, automatically map database tuples to programming objects.

Dynamic general-purpose languages (such as Smalltalk, Perl, Python, and Ruby) are dynamically typed, typically have higher-order constructs (for example, passing methods to

Table 2

Mismatches among class and instance semantics.

Capability	Relational model	Object-oriented languages (for example, Java and C#)	RDF Schema
Class membership	Each tuple belongs to exactly one table and thus to one class.	Each object is a member of exactly one class.	Resources can belong to multiple types.
Inheritance	No inheritance is possible.	Classes can only inherit from at most one superclass (although you can use interfaces to model type inheritance)	Classes can inherit from multiple superclasses.
Object conformance	Each tuple must strictly conform to the table definition.	Instances' structure must exactly follow their class definitions.	Class definitions aren't exhaustive and don't constrain their instances' structure.
Semistructured data	Each tuple must belong to exactly one table and must follow that table's schema.	Class definitions are required for all instances (the class defines the instances' properties).	Instances might deviate from the class definition or appear without any schema information. Properties can have multiple values of multiple types.
Runtime evolution	Database schemas don't usually change during runtime.	Static object-oriented systems don't typically let class definitions evolve during runtime.	RDF integrates heterogeneous data with varying structures and from varying sources, where both schema and data evolve during runtime.

methods), and support complete reflection—both introspection (obtaining information on objects at runtime) and intercession (modifying objects at runtime). Such dynamically typed languages address the mismatches in table 2 as follows:

- *Class membership*. These languages' dynamic typing doesn't require that we define object classes statically, but lets us determine them at runtime by the object's capabilities. Dynamic typing maps well to RDF(S) class membership, which can also change dynamically during runtime. Although objects in most dynamically typed languages can have no more than one class at a time, we can change that behavior at runtime (using intercession).
- *Inheritance*. Similarly, most dynamically typed languages don't support multiple inheritance. However, you can add this behavior through reflection because, in the absence of behavioral definitions, inheritance in RDF(S) only affects object typing.
- *Object conformance*. Dynamically typed languages typically don't require objects to strictly conform to their class definitions, but instead let them deviate from their classes by, for example, specifying a different behavior (method implementation) for several objects of the same class.
- *Semistructured data*. These languages' flexibility often removes the need for attribute definitions in classes and lets instance attributes have various values of various types without problems.
- *Runtime evolution*. Because dynamically typed languages are interpreted and don't rely on strictly predefined classes, they're well suited for flexible environments in which both data and schema can evolve. Their introspection features let programs interrogate the schemas and domain vocabulary at runtime.

Dynamic languages' flexible capabilities imply that type safety isn't guaranteed and that runtime errors can occur. However, we tolerate these drawbacks to accommodate our decentralized integration scenario.

Challenges for existing frameworks

Frameworks such as Struts (http://struts.apache.org), Ruby on Rails (http://rubyonrails.org), and Django (www.djangoproject.com) are a popular way to develop Web applications such as our example social networking application (see also the "Related Work in Semantic Web Application Development" sidebar). These frameworks overcome a common problem with dynamically typed Web applications—that is, the implementation of business logic tends to be interleaved with the markup of presentation templates and database operations. These frameworks address these issues by using

the model-view-controller (MVC) pattern, which separates an application into three parts:[5]

- the *application model* manages data representation and business logic,
- the *views* present the data and manage user interaction, and
- the *controller* handles control flow.

Interaction pattern for traditional applications

The basic interaction pattern in Web application frameworks is *c*reate, *r*ead, *u*pdate, and *d*elete data from the database. The CRUD pattern maps directly to basic database operations in SQL. The user interface exposes data, usually through templates rendered with values from the database. The application translates all manipulations on the user interface into database queries.

Interaction pattern for the Semantic Web

Resource creation doesn't impose additional requirements in Semantic Web applications. Reading information about resources occurs in two steps:

1. The application fetches information from referenced data sources, such as the online FOAF profile of Cédric's friends (through the *knows* property in our example).
2. The application must then integrate the collected information with existing information. The graph-based data model and the global URIs facilitate this task; however, determining whether the same resources are referred to differently still requires object consolidation strategies.

Because Semantic Web applications include both local and remote data sources, you update and delete information differently from how you would in a relational scenario. Because remote data sources typically offer read-only access, you need to store updates locally. Integrating remote data sources and letting users update information requires that you track provenance and versioning. Provenance lets human readers assess information's trustworthiness. For example, Cédric might decide to see only Armin's own information, whereas Benjamin might decide to also include what Cédric said about Armin.

Related Work in Semantic Web Application Development

Several other projects provide methods for developing Semantic Web applications.

The Semantic Hypermedia Design Method (SHDM) is a complete design methodology and accompanying framework for Semantic Web applications. It's based on the theory of navigational sets.[1] Hera is a comparable methodology that supports the design and engineering of Semantic Web information systems.[2] Oscar Corcho and his colleagues introduce a Semantic Web portal using the model-view-controller (MVC) design pattern but don't integrate it with an existing framework.[3]

However, these approaches don't address the specific problem we discuss: the semantic differences between relational and Semantic Web data.

Researchers have also attempted to develop an object-oriented API in Java. RDFReactor (http://rdfreactor.ontoware.org), Elmo (http://openrdf.org/doc/elmo), and Jastor (http://jastor.sourceforge.net) are three such projects. These approaches don't account for the flexible and semistructured nature of RDF data, relying instead on RDF Schema to generate corresponding classes. They also assume the schema's stability, requiring manual regeneration and recompilation if the schema changes. Finally, they assume the RDF data's conformance to such a schema by not allowing objects whose structure differs from their class definition.

References

1. F. Lima and D. Schwabe, "Application Modeling for the Semantic Web," *Proc. Latin-American Web Congress* (LA-Web), IEEE CS Press, 2006, pp. 93–102.
2. R. Vdovjak et al., "Engineering Semantic Web Information Systems in Hera," *J. Web Eng.*, vol. 2, no. 1–2, 2003, pp. 3–26.
3. O. Corcho, A. López-Cima, and A. Gómez-Pérez, "A Platform for the Development of Semantic Web Portals," *Proc. Int'l Conf. Web Eng.*, ACM Press, 2006, pp. 145–152.

A Semantic Web application framework

Our SWAF extends Ruby on Rails, a powerful framework for rapid Web application development using the dynamic and flexible Ruby language. We chose Ruby because of its dynamic typing and strong reflection features, which lets us address the mismatches noted earlier. Additionally, by connecting to an existing Web development community, we can leverage its existing functionality and ecospace of plug-ins and extensions, and thus reduce the adoption barrier.

Figure 1 shows the SWAF architecture, which consists of two parts:

- a plug-in that provides a generator for controllers and views, and
- the ActiveRDF library, which maps RDF(S) resources to Ruby objects.

The plug-in automatically generates the views and controllers, which the application

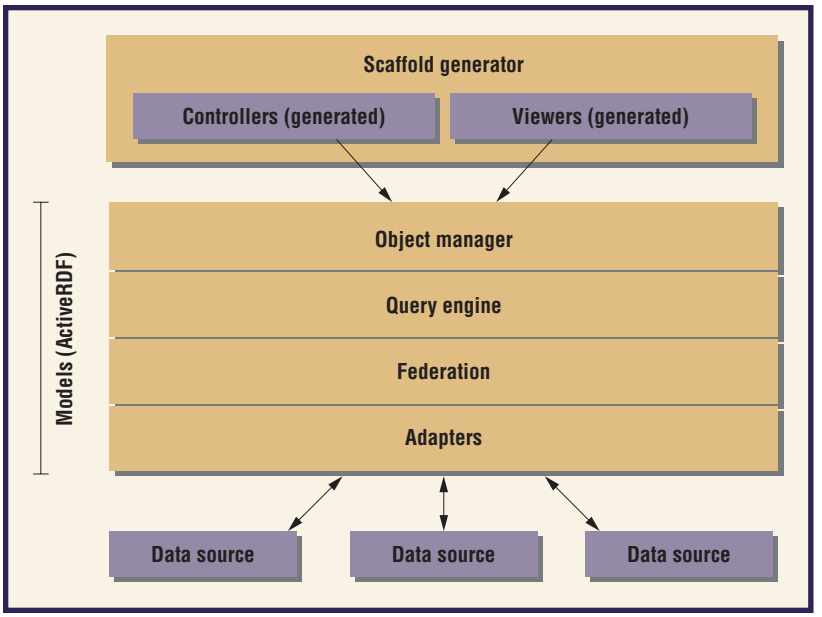


Figure 1. The Semantic Web application framework architecture.

Figure 2. An interface showing Cédric's friend-of-a-friend profile.

developer can then customize. The ActiveRDF library provides the model. Its four layers incrementally abstract RDF data into objects.

Generating controllers and views

The SWAF generators automatically scaffold functional code for classes of RDF(S) data. We customize the scaffold generators for specific vocabularies, letting developers customize their application's behavior for these vocabularies, but they also include default behavior.

To implement our social networking application, we create the scaffolding for FOAF profiles with the command: "`./script/generate swaf:scaffold person foaf http://xmlns.com/foaf/0.1/`," which results in a skeleton application with basic CRUD actions to manipulate profiles, and *fetch*, *list*, and *search* actions to manage remote data sources.

Figure 2 is a screenshot of the generated interface. As you can see in the bottom of the figure, the application automatically shows the provenance and version history of all statements. *RDF reification* (a technique for expressing statements about statements in RDF) tracks versions internally.[6]

None of the generated views rely on the schema; rather, they use the actual instance data. For example, because Cédric's profile contains the `openid` property (top of figure 2), this property is displayed and can be edited, even though it isn't described in the FOAF schema (lacking a schema description, however, it also lacks a human-readable label).

Figure 3 shows the generated files: a controller for `people`, some helpers, a `person` model, a shared `people` layout, and several views for displaying the various actions. (The underscored files are partial views used in AJAX actions.)

Figure 4a shows the generated `show` action in the controller.

SWAF generates similar code for the actions `list` (which lists all known people), `search`, and `edit` (which lets you update a profile). The `show` code creates a new `foaf:Person` identified by the given URI. This person is a proxy object representing the RDF(S) resource. The `show` code then uses the SWAF library to find all known statements about this person. By default, the `find-all-stats` call looks into the local store of crawled statements. If no information is available, it tries to fetch more data from that resource's URI.

Figure 4b shows the generated fetching code. It checks whether information is already known about the resource and fetches the RDF otherwise. RDF documents can contain `seeAlso` statements pointing to more information. We automatically follow these pointers when encountered.

SWAF generates all described behavior automatically. The application developer can then inspect and modify all code by changing actions and views or by adding new actions and views.

Modeling in SWAF: ActiveRDF

The generated SWAF controllers and views manipulate RDF data using our ActiveRDF mapping. ActiveRDF provides virtual models according to the MVC pattern. The mapping

architecture's general principle is to represent RDF resources through transparent proxy objects.[7] Each proxy object represents one RDF resource but doesn't contain any state. The proxy translates all methods (manipulations) on the proxy object into (read or write) queries related to the proxy's RDF resource.

Object manager. The object manager is the library entry point and provides all the mapping functionality. It provides the domain model with its manipulation and generic search methods. It isn't a generated API (hence the name *virtual*), but uses reflection to catch and respond to unhandled method calls (such as `Cedric.firstName`).

The object manager maps RDF data to objects and data manipulation to methods. For example, when the application calls a `find` method or creates a new person, the mapping layer translates this operation into a query on the data source. The object manager also creates object-oriented classes from RDF Schema classes if schema information is available.

In RDF(S), every object can have multiple types. (RDF(S) doesn't specify whether the object should also inherit behavior from all of these types. The question only arises in an object-oriented setting, because types in RDF(S) don't actually exhibit any behavior. The possibility of multiple behavioral inheritance is an open issue in ActiveRDF.) Because Ruby doesn't allow such multiple membership, we override all built-in methods that use an object's class to rely on the `rdf:type` defined in the data source.

Mapping resources. The object manager maps all RDF Schema classes to Ruby classes, all RDF resources to Ruby objects, and all RDF properties to attributes on the Ruby objects. All RDF resources are by default created as Ruby objects of class `RDFS::Resource` (the top-level concept in RDF Schema).

The object manager offers a virtual API to manipulate RDF. Domain-specific methods such as `Cedric.age` or `Cedric.name` aren't generated but are provided virtually. The object manager catches their invocation and translates the method call into a query (caching such a translation could improve performance). Without the object manager's interference, Ruby would throw a `MethodNotFound` error. Such reflection caters to flexibility. Because we don't generate the API but simulate it based on the data available at runtime, we don't need to recompile or regenerate the API when the data changes. We can divide this virtual object manager API into two sections: class-level methods for searching resources and instance-level methods for manipulating resources.

Figure 3. Generated application scaffolding files.

```
# shows details of one resource (its properties and
values)
def show
       @person = FOAF::Person.new(params[:uri])
       @statements =
SWAF::Statement.find_all_about(@person)
end
```

(a)

```
def fetch
       @person = RDFS::Resource.new(params[:uri])

       known = Query.new.ask.where(@person,:p,:o).execute
       unless known
          $db.fetch statements(@person.uri)
          $db.fetch statements(@person.rdfs::seeAlso) if
@person.rdfs::seeAlso
       end
       redirect_to :action => 'show', :uri => @person.uri
end
```

(b)

Figure 4. Code generated by the controller: (a) `show` action and (b) `fetch` action.

Searching resources. If the application programmer knows the resource's URI, he or she creates a proxy object, as in the `show` action

You can manipulate resources depending on the source's data access permissions and capabilities.

described earlier. If the URI is unknown, ActiveRDF offers two ways to search for it: with dynamic finders in the object manager, or through the query API.

The following two examples demonstrate how we use dynamic finders to implement a search functionality on our social networking application. The first shows a search for all resources named Cédric; the second for all 30-year-olds named Cédric:

```
FOAF::Person.find_by_foaf::name(
'Cedric')
FOAF::Person.find_by_foaf::name_
and_foaf::age('Cedric',30)
```

These queries aren't programmed into ActiveRDF. The generic translation from method invocations to queries is programmed, but the resulting query depends purely on the available data and the given method invocation. For example, the object manager would translate the second dynamic finder into the following query:

```
Query.new.distinct(:s).where(:s,
rdf:type, foaf:Person).
  where(:s, foaf:name,
  "Cedric").where(:s, foaf:age,
  30)
```

Manipulating resources. You can manipulate resources depending on the source's data access permissions and capabilities. The following example uses a standard Ruby closure to traverse all of Cédric's friends and print each friend's name. The application could use such example code to show each user's social network:

```
puts Cedric.name
puts "Cedric␣knows:␣" +
Cedric.knows.each do |friend|
 puts friend.name
end
```

In this example, the object manager transparently catches the methods `Cedric.name`, `Cedric.knows`, and `friend.name` and translates each into a query. It similarly handles invocations that change attribute values but generates update queries instead of read-only queries. For example, the first method call invokes the dynamically generated query:

```
Query.new.select(:o).where
(Cedric.uri, foaf.name, :o)
```

Query engine. The query engine provides an abstract query API that's independent of a specific data source and query language. The object manager uses the query engine internally to construct queries for each object manipulation. The application developer can also use it to execute complex queries on the data sources. The current implementation supports conjunctive datalog with select, distinct, and arbitrary where clauses; counts; and full-text search.

The following are some typical queries. The first query counts Cédric's friends, and the second finds all resources (people) that mention "Ireland" in a property:

```
Query.new.count.distinct(:o).
where(cedric, foaf:knows, :o)
Query.new.distinct(:s).where(:s,
:p, :o).where(:o, :keyword,
'Ireland')
```

Federation manager. To accommodate the distributed nature of Semantic Web data, the federation manager oversees the collection of available data sources, distributes the queries over all registered data sources, and aggregates the results. The current implementation achieves query distribution by simply querying all data sources sequentially. It aggregates the query results through a union of individual results (using set union for distinct queries or bag union for nondistinct queries).

Adapters. Adapters provide access to a specific type of Semantic Web data, typically an online data store or RDF file. The adapters translate generic RDF operations to a store-specific API. We need such RDF data-store-specific adapters because RDF stores currently lack a standardized query language with CRUD access. To wrap a data source, adapters must conform to a standard interface that includes methods to `query`, `add`, `delete`, and `load` data.

We've implemented adapters for end points using the standard RDF query language, SPARQL, and to several types of RDF data stores. We've also developed *rdflite*, a simple and lightweight RDF store and adapter that allows prototyping without installing a full-fledged RDF store.

SWAF, like any middleware solution, raises the abstraction level and reduces the amount of application code necessary, because we've factored repetitive tasks into the framework's libraries. As such, it increases productivity because developers write less application code, and it facilitates application maintenance.

We've implemented several applications using SWAF. A faceted metadata browser for arbitrary RDF data is available at www.browserdf.org, and lets users explore arbitrary RDF data through a technique called "faceted browsing." The SIOC browser for online social communities is available at www.activerdf.org/sioc. This portal aggregates online community data, such as forums, weblogs, or IRC chats that export their information in SIOC (an RDF vocabulary for online communities). Users can browse this aggregated data and see, for example, contributions written by the same people across different forum systems and weblog engines. In terms of development effort, in both applications the models are automatically provided, the controllers contain around 100 (SIOC browser) and 300 (faceted browser) lines of code, and the views contain around 110 (SIOC browser) and 200 (faceted browser) total lines of HTML, Ruby, and JavaScript code.

We've shown elsewhere that for typical queries, the ActiveRDF mapping adds only a little overhead, negligible compared to the query execution time on the data store itself.[7]

We're continuing to work on SWAF and ActiveRDF. Mostly, we're extending and improving its functionality based on feedback from developers that use it in building real Semantic Web applications.

Acknowledgments

The Science Foundation Ireland supported the work presented in this article under grants SFI/02/CE1/I131 and SFI/04/BR/CS0694.

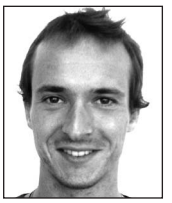
References

1. T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," *Scientific American*, vol. 284, no. 5, May 2001, pp. 34–43.
2. O. Nierstrasz et al., "On the Revival of Dynamic Languages," *Proc. Software Composition*, LNCS 3628, Springer, 2005, pp. 1–13.
3. G. Klyne and J.J. Carroll, eds., *Resource Description Framework: Concepts and Abstract Syntax*, World Wide Web Consortium (W3C) recommendation, Feb. 2004; www.w3.org/TR/rdf-concepts.
4. D. Brickley and R. Guha, eds., *RDF Vocabulary Description Language 1.0: RDF Schema*, World Wide Web Consortium (W3C) recommendation, Feb. 2004; www.w3.org/TR/rdf-schema.
5. T. Reenskaug, *Models, Views, Controllers*, tech. report, Xerox PARC, 1979.
6. P. Hayes, ed., *RDF Semantics*, World Wide Web Consortium (W3C) recommendation, Feb. 2004; www.w3.org/TR/rdf-mt.
7. E. Oren et al., "ActiveRDF: Object-Oriented Semantic Web Programming," *Proc. Int'l World-Wide Web Conf.*, ACM Press, 2007, pp. 817–824.

About the Authors

Eyal Oren is a PhD student in computer science at the Digital Enterprise Research Institute. His research interests include techniques for manipulating, analyzing, and using Semantic Web data. He received his MSc in computer science from Delft University of Technology. He's the creator of ActiveRDF, BrowseRDF, and SemperWiki. He's a member of the ACM. Contact him at DERI, National Univ. of Ireland, IDA Business Park, Lower Dangan, Galway, Ireland; eyal.oren@deri.org.

Armin Haller is a PhD student in computer science at the Digital Enterprise Research Institute. He is funded by the Science Foundation Ireland under the m3pe project and by the European Union through the IP project SUPER. His research interests include workflow interoperability and the merging of Semantic Web ideas with traditional workflow technologies. He received his MA in business information systems from the University of Applied Sciences Wiener Neustadt. Contact him at DERI, National Univ. of Ireland, IDA Business Park, Lower Dangan, Galway, Ireland; armin.haller@deri.org.



Cédric Mesnage is a PhD student in computer science at the University of Lugano, where he works in the Nepomuk European Project focusing on the social semantic desktop's architecture. His research interests include Web development, the Semantic Web, collaborative tagging, social networks, and relations between computer science and social sciences. He received his MSc in computer science from the University of Caen, France, with specialization in algorithmics and information modeling. Contact him at the University of Lugano, Via G. Buffi 13, 6960 Lugano, Switzerland; cedric.mesnage@lu.unisi.ch.

Manfred Hauswirth is vice-director of the Digital Enterprise Research Institute and a professor at the National University of Ireland, Galway. His research interests include semantic distributed information systems and applications, including Semantic Web services, sensor network infrastructures, and peer-to-peer systems. He received his PhD in computer science from Technical University Vienna. Contact him at DERI, National Univ. of Ireland, IDA Business Park, Lower Dangan, Galway, Ireland; manfred.hauswirth@deri.org; www.manfredhauswirth.org.

Benjamin Heitmann is a research intern in the Digital Enterprise Research Institute. His main research interests are Web applications and semantic publishing. He holds a BSc in computer science from the University of Karlsruhe in Germany. Contact him at DERI, National Univ. of Ireland, IDA Business Park, Lower Dangan, Galway, Ireland; benjamin.heitmann@deri.org.

Stefan Decker is a professor at the National University of Ireland, Galway, director of the Digital Enterprise Research Institute, and leader of the institute's Semantic Web Cluster. He received his PhD in computer science from the University of Karlsruhe. Contact him at DERI, National Univ. of Ireland, IDA Business Park, Lower Dangan, Galway, Ireland, stefan.decker@deri.org.