Using triples for implementation: the Triple20 ontology-manipulation tool

Jan Wielemaker[1], Guus Schreiber[2], and Bob Wielinga[1]

[1] University of Amsterdam, Human Computer Studies (HCS),
Kruislaan 419, NL-1098 VA Amsterdam, The Netherlands,
{wielemak,wielinga}@science.uva.nl
[2] Free University Amsterdam, Computer Science
De Boelelaan 1081a, NL-1081 HV Amsterdam, The Netherlands
schreiber@cs.vu.nl

Abstract. Triple20 is a ontology manipulation and visualization tool for languages built on top of the Semantic-Web RDF triple model. In this article we explain how a triple-centered design compares to the use of a separate proprietary internal data model. We show how to deal with the problems of such a low-level data model and show that it offers advantages when dealing with inconsistent or incomplete data as well as for integrating tools.

1 Introduction

Triples are at the very heart of the Semantic Web [1]. RDF, and languages built on top of it such as OWL [2] are considered *exchange* languages: they allow exchanging knowledge between agents (and humans) on the Semantic Web through their *atomic* data model and well-defined semantics. The agents themselves often employ a data model that follows the design, task and history of the software. The advantages of a proprietary internal data model are explained in detail by Noy *et al.* [3] in the context of the Protégé design.

The main advantage of a proprietary internal data model is that it is neutral to external developments. Noy *et al.* [3] state that this enabled their team to quickly adopt Protégé to the Semantic Web as RDF became a standard. However, this assumes that all tool components commit to the internal data model and that this model is sufficiently flexible to accommodate new external developments. The RDF triple model and the higher level Semantic Web languages have two attractive properties. Firstly, the triple model is generic enough to represent *anything*. Secondly, the languages on top of it gradually increase the semantic commitment and are extensible to accommodate to almost any domain. Our hypothesis is that a tool infrastructure using the triple data model at its core can profit from the shared understanding when using the triple model for *exchange*. We also claim that, where the layering of Semantic Web languages provide different levels of understanding of the same document, the same will apply for tools operating on the triple model.

In this article we describe the design of Triple20, an ontology editor that runs directly on a triple representation. First we introduce our triple store, followed by a description on how the model-view-controller design can be extended to deal with the low level data model. In Sect. 4.1 to Sect. 6.2 we illustrate some of the Triple20 design decisions and functions, followed by some metrics, related work and discussion.

2 Core technology: Triples in Prolog

The core of our technology is Prolog-based. The triple-store is a memory-based extension to Prolog realising a compact and highly efficient implementation of **rdf/3** [4]. Higher level primitives are defined on top of this using Prolog *backward chaining* rather than *transformation* of data structures. A simple example:

```
class(Sub, Super) :-
    rdf(Sub, rdfs:subClassOf, Super),
    rdf(Sub, rdf:type, rdfs:'Class'),
    rdf(Super, rdf:type, rdfs:'Class').
```

The RDF infrastructure is part of the Open Source SWI-Prolog system[3] and used by many internal and external projects. Higher-order properties can be expressed easily and efficiently in terms of triples. Object manipulations, such as defining a class are also easily expressed in terms of adding and/or deleting triples. Operating on the same triple store, triples not only form a mechanism for *exchange* of data, but also for cooperation between *tools*. Semantic Web standards ensure consistent interpretation of the triples by independent tools.

3 Design Principles

Most tool infrastructures define a data model that is inspired by the tasks that have to be performed by the tool. For example, Protégé, defines a flexible meta-data format for expressing the basic entities managed by Protégé: classes, slots, etc. The GUI often follows the model-view-controller (MVC) architecture [5]. This design is illustrated in Fig. 1. There are some issues with this design we would like to highlight.

- All components in the tool set must conform to the same proprietary data model. This may harm maintainability and complicates integrating tools designed in another environment.
- Data is translated from/to external (file-)formats while loading/saving project data. This poses problems if the external format contains information that cannot be represented by the tool's data model.

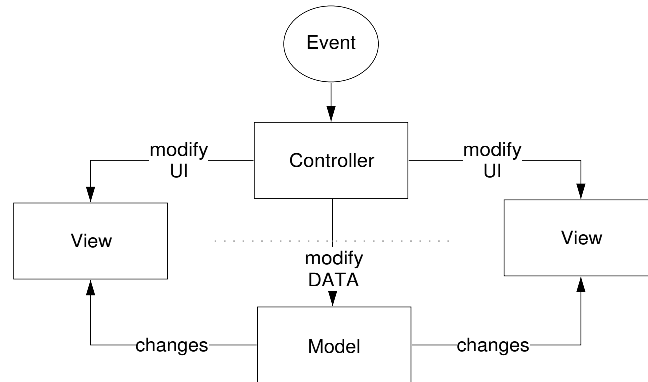

Fig. 1. Model-View-Controller (MVC) design pattern. Controllers modify UI aspects of a *view* such as zooming, selection, etc. directly. During editing the *controller* modifies the *model* that in turn informs the views. Typically, the data structures of the *Model* are designed with the task of the application in mind.

The MVC design pattern is commonly used and successful. In the context of the Semantic Web, there is an alternative to the proprietary tool data model provided by the stable RDF triple model. This model was designed as an *exchange* model, but the very same features that make it good for exchange also make it a good candidate for the internal tool data model. In particular, the *atomic* nature of the model with its standardised semantics ensure the cooperating tools have a sound basis.

In addition to providing a sound basis, the triple approach deals with some serious consistency problems related to more high-level data models. All Semantic Web data can be expressed precisely and without loss of information by the toolset, while each individual tool can deal with the data using its own way to view the world. For example, it allows an RDFS tool to work flawlessly with an OWL tool, although with limited understanding of the OWL semantics. Different tools can use different subsets of the triple set, possibly doing different types of reasoning. The overall semantics of the triple set however is dictated by stable standards and the atomic nature should minimise interoperability problems. Considering editing and browsing tools, different tools use different levels of abstractions, viewing the plain triples, viewing an RDF graph, viewing an RDFS frame-like representation or an OWL/DL view (Fig. 4, Fig. 5).

Finally, the minimalist data model simplifies general tool operations such as *undo*, save/load, client/server interaction protocols, etc.

In the following architecture section, we show how we deal with the low-level data model in the MVC architecture.

[3] http://www.swi-prolog.org

4 Architecture

Using a high-level data model that is inspired by the tasks performed by the tools, mapping actions to changes in the data model and mapping these changes back to the UI is relatively straightforward. Using the primitive RDF triple model, mapping changes to the triple store to the views becomes much harder for two reasons. First of all, it is difficult to define concise and efficiently which changes affect a particular view and second, often considerable reasoning is involved deducing the visual changes from the triples. For example, adding the triple below to a SKOS-based [6] thesaurus turns the triple set representing a thesaurus into a RDFS class hierarchy:[4]

skos:narrower rdfs:subPropertyOf rdfs:subClassOf .

The widgets providing the 'view' have to be consistent with the data. As we can see from the above the relation between changes to the triple set and changes to the view can be very indirect. We deal with this problem using *transactions* and *mediators* [7].

Both for journalling, undo management, exception handling and maintaining the consistency of views, we introduced transactions. A transaction is a sequence of elementary changes to the triple-base: *add*, *delete* and *update*,[5] labeled with an identifier and optional comments. The comments are used as a human-readable description of the operation (e.g. "Created class Wine"). Transactions can be nested. User interaction with a controller causes a transaction to be started, operations to be performed in the triple-store and finally the transaction to be committed. If anything unexpected happens during the transaction, the changes are discarded, providing protection against partial and inconsistent changes by malfunctioning controllers. A successful transaction results in an *event*.

Simple widgets whose representation depends on one or more direct properties of a resource (e.g., a label showing an icon and label-text for a resource) register themselves as simple representation of this resource. They will be informed if the resource appears in the *subject* or *object* of an affected triple or the `rdfs:subPropertyOf` hierarchy is modified in the committed transaction. In most cases this will cause the widget to do a simple refresh.

Complex widgets, such as a hierarchical view, cannot use this schema as they cannot easily define the changes in the database that will affect them and recomputing and refreshing the widget is too expensive for interactive use. It is here that we introduce *mediators*. A *mediator* is an arbitrary (Prolog Herbrandt-)term that is derived from the triple set through a defined function. For example, the term can be an ordered list of resources that appear as children of a particular node in the hierarchy which is computed using an OWL reasoner. Widgets register a mediator whenever real-time update is considered too expensive.

[4] Whether this interpretation is correct is not the issue here.

[5] The *update* change can of course be represented as a delete-and-add, but a separate primitive is more natural, requires less space in the journal and is easier to interpret while maintaining the view consistency.

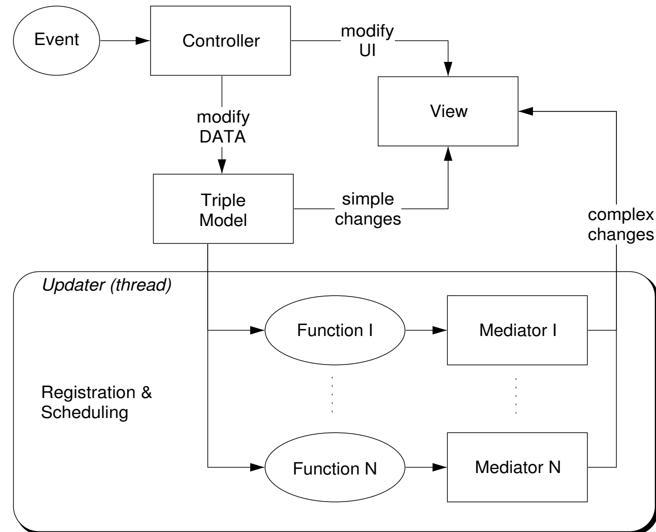

Fig. 2. Introducing *mediators* to bridge the level of abstraction between triples and view. Update is performed in a different thread to avoid locking the UI.

The function and its parameters are registered with the *updater*. The *updater* is running in a separate thread of execution, updating all mediators after each successfully committed transaction. If a mediator is different from the previous result, the controllers that registered the mediator are notified and will update using the high-level representation provided by the model term. This approach has several advantages.

- The UI remains responsive while updating the mediators.
- Updates can be aborted as soon as a new transaction is committed.
- Multiple widgets depending on the same mediator require only one computation.
- The updater can schedule on the basis of execution time measured last time, frequency of different results and relation of dependent widgets to the 'current' widget.[6]
- One or multiple update threads can exploit multi-cpu (SMP) hardware as well as schedule updates over multiple threads to ensure likely and cheap updates are not blocked for a long time by unlikely expensive updates.

4.1 Rules to define the GUI

The interface is composed of a hierarchy of widgets, most of them representing one or more resources. We have *compound* and *primitive* widgets. Each widget is responsible for maintaining a consistent view of the triple set as outlined in

[6] This has not yet been implemented in the current version.

the previous section. Triple20 widgets have small granularity. For example, most resources are represented by an icon and a textual label. This is represented as a compound widget which controls the icons and displays a primitive widget for the textual label.

In the conventional OO interface each compound widgets decides which member widgets it creates and what their their configuration should be, thus generating the widget hierarchy starting at the outermost widget, i.e. the toplevel window. We have modified this model by having context-sensitive rule sets that are called by widgets to decide on visual aspects as well as define context sensitive menus and perform actions. Rule sets are associated with widget classes. Rules are evaluated similar to OO methods, but following the part-of hierarchy of the interface rather than the subclass hierarchy. Once a rule is found, it may decide to wrap rules of the same name defined on containing widgets similar to sending messages to a superclass in traditional OO (Fig. 3).

The advantage of this approach is that widget behaviour can inherit from its containers as well as from the widget class hierarchy. For example, a compound widget representing a set can offer a *delete* menu-item as well as the method to handle deletion to contained widgets without any knowledge of these widgets.

Another example is shown in Fig. 3. In this context, Triple20 is used to view the results of transforming a XML Schema into RDF. XSD types are created as subclasses of `xsd:Type`, a subclass of `rdfs:Class`.[7] Normally, Triple20 does not show the instances of *meta-classes* in the hierarchy. As most schemas do not contain that many types and most types are not defined as a subtype of another type, expanding all XSD types as instances of the class is useful. The code fragment refines the rule for **child_cache/3**, a rule which defines the *mediator* for generating the children of a node in the hierarchy window (Fig. 5). The `display` argument says the rule is defined at the level of display, the outermost object in the widget part-of hierarchy and therefore acts as a default for the entire interface. The `part` argument simply identifies the new rule set. The first rule says the mediator for expanding a `xsd:Type` node is the set of resources linked to it using V `rdf:type` R, sorted by label name (**lsorted**(V)). The second rule simply calls the default behaviour.

Rule sets are translated into ordinary Prolog modules using the Prolog preprocessor.[8] They can specify behaviour that is context sensitive. Simple refinement can be achieved loading rules without defining new widgets. More complicated customization is achieved by defining new widgets, often as a refinement of existing ones, and modify the rules used by a particular compound widget to create its parts.

[7] That is, schema types are considered classes in a hierarchy of types.
[8] Realised using **term_expansion/2**.

```
:- begin_rules(display, part).

child_cache(R, Cache, rdf_node) :-
        rdfs_subclass_of(R, xsd:'Type'),
        rdf_cache(lsorted(V), rdf_has(V, rdf:type, R), Cache).
child_cache(R, Cache, Class) :-
        super::child_cache(R, Cache, Class).

:- end_rules.
```

Fig. 3. Redefining the hierarchy expansion for `xsd:Type`. This rule set can be loaded without changing anything to the tool.

5 User-interface principles

RDF documents can be viewed at different levels. Our tool is not a tool to support a particular language such as OWL, but to examine and edit arbitrary RDF documents. It provides several views, each highlighting a particular aspect:

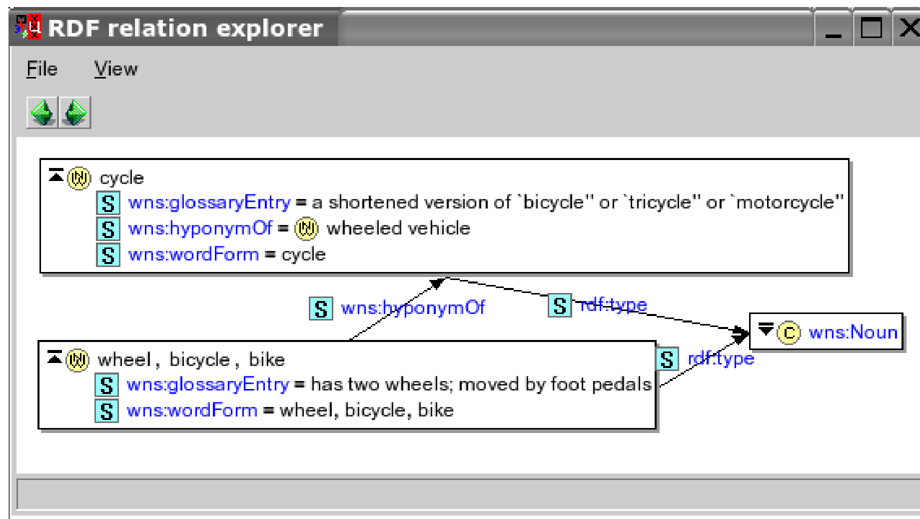

Fig. 4. Triple20 graph diagram. Resources are shown using just their label or as a frame. Values or properties can be dragged from a frame to the window to expand them.

– The *diagram* view (Fig. 4) provides a graph of resources. Resources can be shown as a label (*Noun*) or expanded to a frame (*cycle*). Elements from the frame can be dragged to the diagram as natural user-controlled mechanism

to expand the graph. This tool simply navigates the RDF graph and works on any RDF document.

- The *hierarchy* view (Fig. 5, left window) shows different hierarchies (class, property, individuals) in a single view. The type of expansion is indicated using icons. Expansion can be controlled using *rules* as explained in Sect. 4.1.
- A tabular window (Fig. 5, right window) allows for multiple resource specific representations. The base system provides an *instance* view and a *class* view on resources.

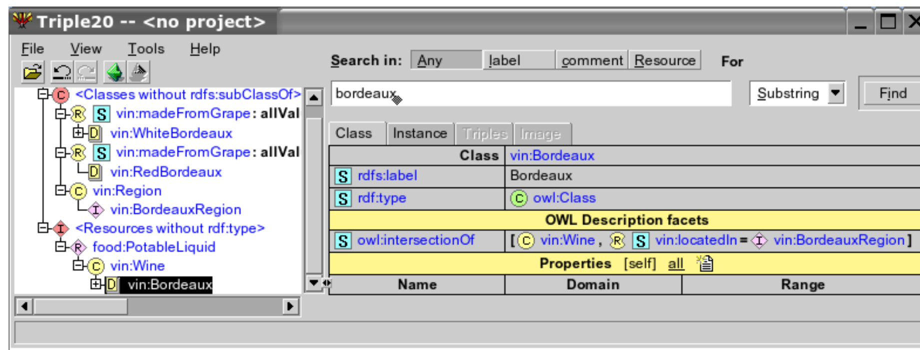

Fig. 5. Triple20 main window after a search and select.

Editing and browsing are as much as possible integrated in the same interface. This implies that most widgets building the graphical representation of the data are sensitive. Visual feedback of activation and details of the activated resource are provided. In general both menus and drag-and-drop are provided. Context-specific rules define the possible operations dropping one resource on another. Left-drop executes the default operation indicated in the status bar, while right-drop opens a menu for selecting the operation after the drop. For example, the default for dropping a resource from one place in a hierarchy on another node is to *move* the resource. A right-drop will also offer the option to associate an additional parent. Rules also provide context-sensitive menus on resources.

Drag-and-drop can generally be used to add or modify properties. Before one can drop an object it is required to be available on the screen. This is often impractical and therefore many widgets provide menus to modify or add a value. This interface allows for typing the value using completion, selecting from a hierarchy as well as search followed by selection. An example is shown in Fig. 6.

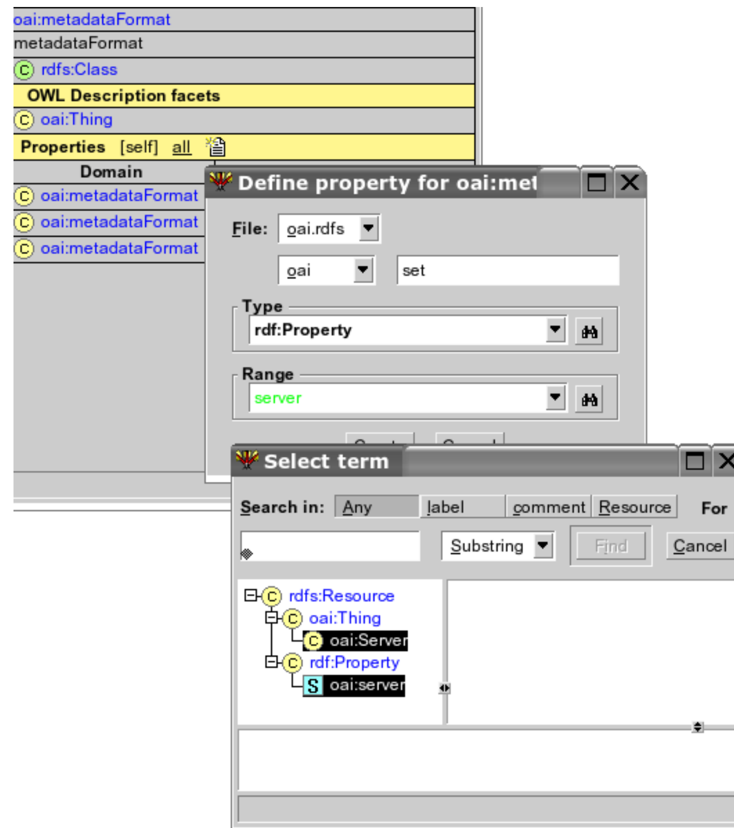

Fig. 6. Select a resource by typing, in this example *server*. The style indicates the status and is updated after each keystroke. Green (here) means there are multiple resources with this name. Hitting the *binocular* icon shows all matches in the hierarchy, allowing the user to select.

6 Implementation

6.1 The source of triples

Our RDF store is actually a quadruple store. The first three fields represent the RDF triple, while the last identifies the source or sub-graph it is related too. The source is maintained to be able to handle triples from multiple sources in one application, modify them and save the correct triples to the correct destination.

Triple20 includes a library of background ontologies, such as RDFS and OWL as well as some well-known public toplevel ontologies. When a document is loaded which references to one of these ontologies, the corresponding ontology is loaded and flagged 'read-only', meaning no new triples will be assigned to this source and it is not allowed to delete triples that are associated to it. This implies that trying to delete such a triple inside a transaction causes the operation to be aborted and the other operations inside the transaction to be discarded.

Other documents are initially flagged 'read-write' and new triples are associated to sources based on rules. Actions involving a dialog window normally allow the user to examine and override the system's choice, as illustrated in Fig. 7.

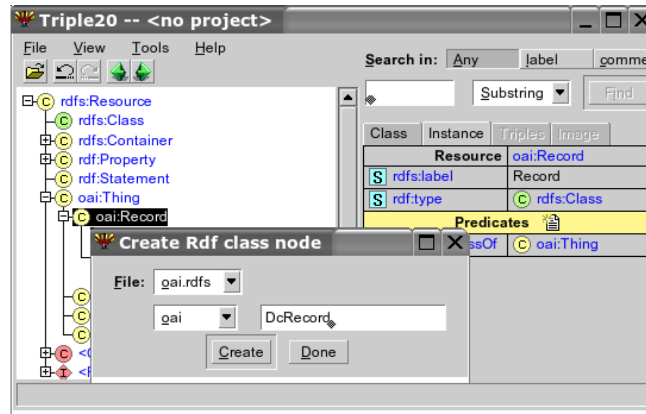

Fig. 7. Create a new class. The system proposes the file the class will be saved to as well as the namespace based on the properties of the super class. Both can be changed.

Although *referring* to other documents should be the dominant technique for reusing material on the Semantic Web, Triple20 allows for moving triples from one source to another realising reuse through copy, possibly followed by adjustment to the new context.

6.2 Projects

As an ontology editor, Triple20 is designed to operate in two modes. For simple browsing and minor editing of relatively small projects it can simply open

(load) a RDF document. It will automatically load referenced documents from its library, providing access to the document in its context. The ontology can be edited and saved similar to editing text documents with a word processor.

The above does not scale very well. It requires the relatively slow load and save from RDF/XML serialization and does not preserve specific settings of the editor related to the document, such as namespace abbreviations (e.g., rdfs for `http://www.w3.org/2000/01/rdf-schema#`), loading of related documents from locations outside the library, etc. For this reason, Triple20 provides *projects*. A project is simply a journal of all actions that can be reloaded by *replaying* it. Operations of committed transactions are simply appended to the project file. For documents that are loaded we save a snapshot with MD5 signature in the internal quick-load format, providing reliable and fast loading of the same triple set. The project approach has several advantages for dealing with the development of large documents.

- There is no need for saving intermediate 'safety' copies.
- The commented sequence of transactions allow for reviewing the changes, both for the author as a *change log* and for a reviewer that has to authorize changes for a central copy. We intend to add a mode for ontology maintenance, where each finished transaction will be annotated using the author, data and a motivation by the author.

7 Scalability

The aim of Triple20 and the underlying RDF store is to support large ontologies in memory. In-memory storage is much faster than what can be achieved using a persistent store [4], a requirement to deal with the low-level reasoning at the triple level. The maximum capacity of the triple store is approximately 40 million triples on 32-bit hardware and virtually unlimited on 64-bit hardware.

We summarise some figures handling WordNet [8] in RDF. The measurements are taken on a dual AMD 1600+ machine with 2GB memory running SuSE Linux. The 5 WordNet files contain a total of 473,626 triples. The results are shown in Tab. 7. For the last test, a small file is added that defines the `wns:hyponymOf` property as a sub property of `rdfs:subClassOf` and defines `wns:LexicalConcept` as a subclass of `rdfs:Class`. This reinterprets the WordNet hierarchy as an RDFS class hierarchy. Note that this work is done by the separate update thread recomputing the mediators and thus does not block the UI.

8 Related work

Protégé [9] is a landmark in the world of ontology editors. We have described how our design differs in Sect. 3. Where Protégé is primarily designed as an *editor*, Triple20 is primarily a *browser*. To avoid cluttering the view with controls,

Operation	**Time (sec)**
Load from RDF/XML	65.4
Load from cache	8.4
Re-interpret as class hierarchy	16.3

Table 1. Some figures handling WordNet on a dual AMD 1600+ machine. Loading time is proportional to the size of the data.

Triple20's widgets concentrate on popup menus, drag-and-drop and direct manipulation techniques. Protégé has dedicated support for ontology engineering, which Triple20 lacks.

OntoPlugin [10] is the plugin system of OntoEdit. The integration is not targeted at the data level, but at the tool level, dealing with integration of init, exit, menu options, etc. They aim at integrating larger components, making no commitment on a common data model.

JENA [11] is a Java-based environment for handling RDF data. The emphasis in this software lies on the RDF API and on the querying functionality, and not so much on ontology ontology editing, browsing and manipulation.

Similarly, the Sesame software [12] can be seen as complementary to Triple20, providing client/server-based access to RDF data repositories. Software for using our infrastructure and Sesame together is available from the SWI-Prolog website.

KAON [13] is an extensible ontology software environment. The main difference with Triple20 is that the KAON software is mainly aimed to provide middleware; the environment focuses on integrating distributed applications.

In [14], Miklós et al. describe how they reuse large ontologies by defining *views* using an F-logic based mapping. In a way our *mediators*, mapping the complex large triple store in a manageable structure using Prolog can be compared to this, although their purpose is to map one ontology into another, while our purpose is to create a manageable structure suitable for driving the visualisation.

9 Discussion

We believe the main weakness in our infrastructure is Prolog's poor support for declarative inferencing. We identify the following problems. Firstly, bad ordering in conjunctions may lead to poor performance. In another project[9] we have found that dynamic reordering is feasible and efficient. Secondly, frequent recomputation as well as commonly occurring loops in RDF graphs result in poor performance and complicated code to avoid loops. We plan to add tabling to SWI-Prolog to improve on this, in a similar way as tabling is realised in XSB Prolog [15].

[9] http://www.swi-prolog.org/packages/SeRQL/

We plan to study the possibility of adding external (DL) reasoners to the infrastructure. The can be handled elegantly as another type of *mediator*, connected through the SWI-Prolog XDIG [16] interface. We are afraid though that the communication overhead will be unacceptable for large triple stores.

We have realised a tool architecture that is based directly on the RDF triple model. The advantage of this approach over the use of a tool oriented intermediate model is that any Semantic Web document can be represented precisely and tools operating on the data can profit from established RDF-based standards on the same grounds as RDF supports exchange between applications. With Triple20, we have demonstrated that this design can realise good scalability, providing multiple consistent views (triples, graph, OWL) on the same triple store. Triple20 has been used successfully as a stand-alone ontology editor, as a component in other applications and as a debugging tool for other applications running on top of the Prolog triple store.

Software availability

Triple20 is available under Open Source (LGPL) license from the SWI-Prolog website.[10] SWI-Prolog with graphics runs on MS-Windows, MacOS X and almost all Unix/Linux versions, supporting both 32- and 64-bit hardware.

Acknowledgements

The Triple20 type-icons are partly taken from and partly inspired by the Protégé project. This work is partly supported by the Dutch BSIK project MultiMedian.

References

1. Brickley, D., Guha (Eds), R.V.: Resource description framework (RDF) schema specification 1.0. W3C Recommendation (2000) http://www.w3.org/TR/2000/CR-rdf-schema-20000327/.
2. Dean, M., Schreiber, A.T., Bechofer, S., van Harmelen, F., Hendler, J., Horrocks, I., MacGuinness, D., Patel-Schneider, P., Stein, L.A.: OWL Web Ontology Language Reference. W3C Recommendation, World Wide Web Consortium (2004) Latest version: http://www.w3.org/TR/owl-ref/.
3. Noy, N.F., Sintek, M., Decker, S., Crubezy, M., Fergerson, R.W., Musen, M.A.: Creating Semantic Web contents with protege-2000. IEEE Intelligent Systems **16** (2001) 60–71
4. Wielemaker, J., Schreiber, G., Wielinga, B.: Prolog-based infrastructure for RDF: performance and scalability. In Fensel, D., Sycara, K., Mylopoulos, J., eds.: The Semantic Web - Proceedings ISWC'03, Sanibel Island, Florida, Berlin, Germany, Springer Verlag (2003) 644–658 LNCS 2870.
5. Krasner, G.E., Pope, S.T.: A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. Technical report, Palo Alto (1988)
6. Miles, A.J.: Owl ontology for thesaurus data. Deliverable, SWAD-Europe (2001)

[10] http://www.swi-prolog.org/packages/Triple20

7. Wiederhold, G.: Mediators in the architecture of future information systems. IEEE Computer **25** (1992) 38–49
8. Miller, G.: WordNet: A lexical database for english. Comm. ACM **38** (1995)
9. Musen, M.A., Fergerson, R.W., Grosso, W.E., Noy, N.F., Crubézy, M., Gennari, J.H.: Componentbased support for building knowledge-acquisition systems. In: Conference on Intelligent Information Processing (IIP 2000), Beijing, China (2000) http://smi-web.stanford.edu/pubs/SMI_Abstracts/SMI-2000-0838.html.
10. Handschuh, S.: OntoPlugins a flexible component framework. Technical report, University of Karlsruhe (2001)
11. McBride, B.: Jena: Implementing the rdf model and syntax specification. In: Semantic Web Workshop, WWW 2001. (2001)
12. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: An architecture for storing and querying rdf and rdf schema. In: Proc. First International Semantic Web Conference ISWC 2002, Sardinia, Italy. Volume 2342 of LNCS., Springer-Verlag (2002) 54–68
13. Oberle, D., Volz, R., Motik, B., Staab, S.: An extensible ontology software environment. In Staab, S., Studer, R., eds.: Handbook on Ontologies. International Handbooks on Information Systems. Springer (2004) 311–333
14. Miklos, Z., Neumann, G., Zdun, U., Sintek, M.: Querying semantic web resources using triple views. In Kalfoglou, Y., Schorlemmer, M., Sheth, A., Staab, S., Uschold, M., eds.: Semantic Interoperability and Integration. Number 04391 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany (2005) <http://drops.dagstuhl.de/opus/volltexte/2005/47> [date of citation: 2005-01-01].
15. Freire, J., Warren, D.S., Sagonas, K., Rao, P., Swift, T.: XSB: A system for efficiently computing well-founded semantics. In: Proceedings of LPNMR 97, Berlin, Germany, Springer Verlag (1997) 430–440 LNCS 1265.
16. Huang, Z., Visser, C.: An extended dig description logic interface for prolog. Deliverable, SEKT (2003) http://wasp.cs.vu.nl/sekt/dig/.