



UvA-DARE (Digital Academic Repository)

Logic programming for knowledge-intensive interactive applications

Wielemaker, J.

Publication date
2009

[Link to publication](#)

Citation for published version (APA):

Wielemaker, J. (2009). *Logic programming for knowledge-intensive interactive applications*. [Thesis, fully internal, Universiteit van Amsterdam].

General rights

It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations

If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: <https://uba.uva.nl/en/contact>, or a letter to: Library of the University of Amsterdam, Secretariat, P.O. Box 19185, 1000 GD Amsterdam, The Netherlands. You will be contacted as soon as possible.

Chapter 2

Using triples for implementation: the Triple20 ontology-manipulation tool

About this chapter This chapter has been published at the ISWC-05, Galway (Wielemaker et al. 2005) and introduces Triple20, an ontology editor and browser. The Triple20 application is described in the beginning of this thesis to clarify our position in knowledge representation for interactive applications, addressing research question 3c. The infrastructure needed to build Triple20 is described in the subsequent chapters, notably chapter 3 (the RDF database), chapter 6 (multi-threading) and chapter 5 (XPCE, connecting object oriented graphics libraries to Prolog).

Depending on the context, we refer to Triple20 as a tool, library, browser or editor. It can be used as a stand-alone editor. It can be loaded in—for example—ClioPatria (chapter 10) to explore (browse) the RDF for debugging purposes, while tOKo (Anjewierden and Efimova 2006; Anjewierden et al. 2004) uses Triple20 as a library.

Abstract Triple20 is an ontology manipulation and visualisation tool for languages built on top of the Semantic-Web RDF triple model. In this article we introduce a triple-centred application design and compare this design 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.

2.1 Introduction

Triples are at the very heart of the Semantic Web (Brickley and Guha 2000). RDF, and languages built on top of it such as OWL (Dean et al. 2004) 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. 2001 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. 2001 state that this enabled their team to quickly adapt 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 almost any domain. Our hypothesis is that a tool infrastructure using the triple data model at its core can profit from the shared understanding of the triple model. We also claim that, where the layering of Semantic Web languages provides 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 and browser that runs directly on a triple representation. First we introduce our triple store, followed by a description of how the model-view-controller design pattern (Krasner and Pope 1988, figure 2.1) can be extended to deal with the low level data model. In section 2.4.1 to section 2.6 we illustrate some of the Triple20 design decisions and functions, followed by some metrics, related work and discussion.

2.2 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` (chapter 3, Wielemaker et al. 2003b). Higher level primitives are defined on top of this using Prolog *backward chaining* rather than *transformation* of data structures. Here is a simple example that relates the title of an artwork with the name of the artist that created it:

```
artwork_created_by(Title, ArtistName) :-
    rdf(Work, vra:creator, Artist),
    rdf(Work, vra:title, literal(Title)),
    rdf(Artist, rdfs:label, literal(ArtistName)).
```

The RDF infrastructure is part of the Open Source SWI-Prolog system 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.

2.3 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 metadata format for expressing the basic entities managed by Protégé: classes, slots, etc. The GUI often follows the model-view-controller (MVC) architecture (Krasner and Pope 1988, figure 2.1). We would like to highlight two aspects of this design:

- All components in the tool set must conform to the same proprietary data model. This requirement complicates integrating tools designed in another environment. Also, changes to the requirements of the data model may pose serious maintainability problems.
- 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. This problem becomes apparent if the external data is represented in *extensible* formats such as XML or RDF.

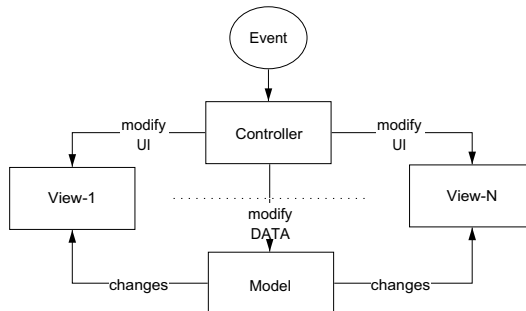


Figure 2.1: Model-View-Controller (MVC) design pattern. Controllers modify UI aspects of a *view* such as zooming, selection, etc. directly. During editing the *controller* modifies the *model* that in turn informs the views. Typically, the data structures of the *Model* are designed with the task of 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 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 of the RDF model 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 (figure 2.4, figure 2.5).

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.

2.4 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 concisely 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 (Miles 2001) thesaurus turns the triple set representing a thesaurus into an RDFS class hierarchy:¹

```
skos:narrower rdfs:subPropertyOf rdfs:subClassOf .
```

The widgets providing the ‘view’ have to be consistent with the data. In the example above, adding a single triple changes the semantics of each hierarchy relation in the thesaurus: changes to the triple set and changes to the view can be very indirect. We deal with this problem using *transactions* and *mediators* (Wiederhold 1992).

Both for journaling, 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*,² labelled 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

¹Whether this interpretation is desirable is not the issue here.

²The *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.

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 a direct representation of this resource. If an event involves a triple where the resource appears as *subject* or *object*, the widget is informed and typically refreshes itself. Because changes to the property hierarchy can change the interpretation of triples, all simple widgets are informed of such changes.

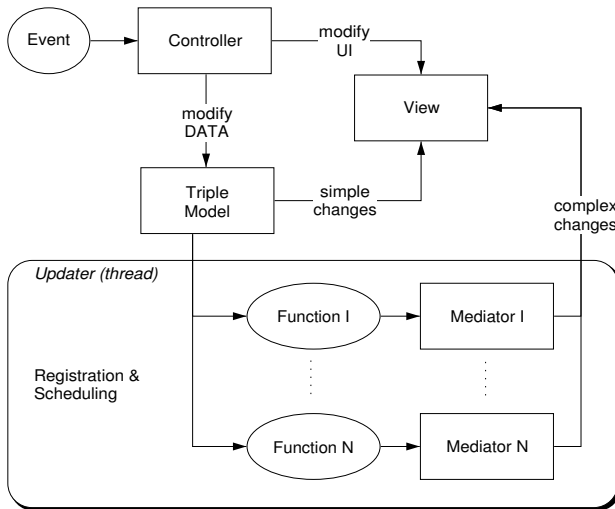


Figure 2.2: Introducing *mediators* to bridge the level of abstraction between triples and view. Update is performed in a different thread to avoid locking the UI.

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 term that is derived from the triple set through a defined function (see figure 2.2). For example, the mediator can be an ordered list of resources that appear as children of a particular node in the hierarchy, while the function is an OWL reasoner that computes the DL class hierarchy. Widgets register a mediator and accompanying function whenever real-time update is considered too expensive. 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 mediator. The function and its parameters are registered with the *updater*. The *updater* is running in a separate thread of execution

(chapter 6, Wielemaker 2003a), updating all mediators after each successfully committed transaction. This approach has several advantages.

- Because updating the mediators happens in a separate thread, the UI remains responsive during the update.
- 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.³
- One or multiple update threads can exploit multi-CPU (SMP) hardware as well as schedule updates over multiple threads to ensure that likely and cheap updates are not blocked for a long time by unlikely expensive updates.

2.4.1 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 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 widget decides which member widgets it creates and what 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.

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 of objects can define rules both for menu-items and the required operations at the data level that deal with the operation *delete*, deleting a single object from the set. Widgets inside the compound ‘inherit’ the menu item to their popup. This way, instances of a single widget class have different behaviour depending on its context in the interface.

Another example of using rules is shown in figure 2.3, where Triple20 is extended to show SKOS ‘part-of’ relations in the hierarchy widget using instances of the graphics class ‘rdf_part_node’, a subclass of ‘rdf_node’ that displays a label that indicates the part-of relation. The code fragment refines the rule for `child.cache/3`, a rule which defines the

³This has not yet been implemented in the current version.

mediator for generating the children of a node in the hierarchy window (shown on an example from another domain in the left frame of figure 2.5). 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 identifies the new rule set. The first rule defines the mediator for ‘parts’ of the current node, while the second creates the default mediator. The call to `rdf_cache/3` registers a mediator that is a list of all solutions of `V` of the `rdf/3` goal, where the solutions are sorted alphabetically on their label. `Cache` is an identifier that can be registered by a widget to receive notifications of changes to the mediator.

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

child_cache(R, Cache, rdf_part_node) :-
    rdf_cache(lsorted(V),
              rdf(V, skos:broaderPartitive, R), Cache).
child_cache(R, Cache, Class) :-
    super::child_cache(R, Cache, Class).

:- end_rules.
```

Figure 2.3: Redefining the hierarchy expansion to show SKOS part-of relations. This rule set can be loaded without changing anything to the tool.

Rule sets are translated into ordinary Prolog modules using the Prolog preprocessor.⁴ They can specify behaviour that is context sensitive. Simple refinement can be achieved by loading rules without defining new widgets. More complicated customisation 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.

2.5 An overview of the Triple20 user interface

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 of the RDF data.

- The *diagram* view (figure 2.4) provides a graph of resources. Resources can be shown as a label (*Noun*) or expanded to a frame (*cycle*). If an elements from a frame is dropped on the diagram a new frame that displays all properties of the element is shown. This tool simply navigates the RDF graph and works on any RDF document.

⁴Realised using `term_expansion/2`.

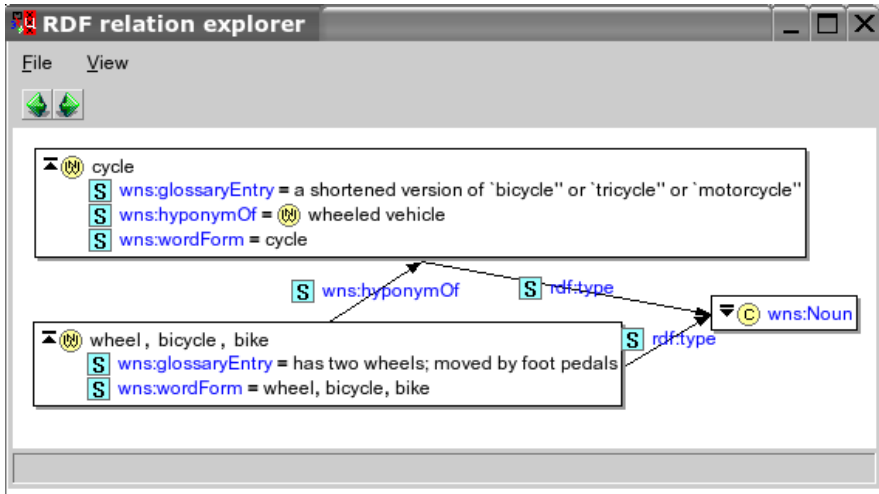


Figure 2.4: Triple20 graph diagram. Resources are shown using just their label or as a frame. Values or properties can be dragged from a frame to the window to expand them.

- The *hierarchy* view (figure 2.5, 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 section 2.4.1.
- A *tabular* view (figure 2.5, right window) allows for multiple resource specific representations. The base system provides an *instance* view and a *class* view on resources.

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 onto 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.

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 of the latter is shown in figure 2.6.

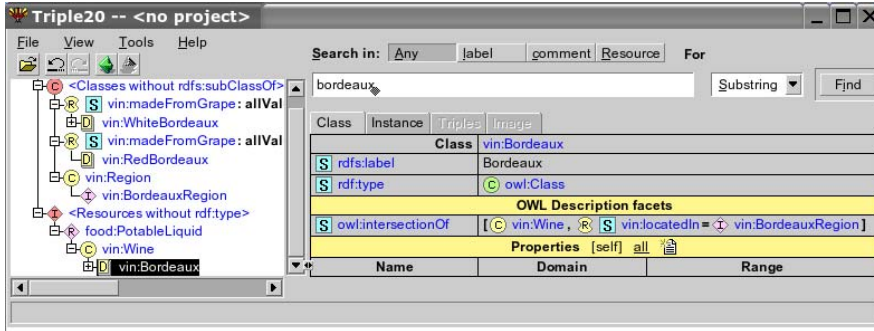


Figure 2.5: Triple20 main window after a search and select.

2.6 Implementation

2.6.1 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 named 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 added 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 figure 2.7.

Triple20 is designed to edit triples from multiple sources in one view as it is often desirable to keep each RDF document in its own file(s). If necessary, triples from one file can be inserted into another file.

2.7 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 (chapter 3, Wielemaker et al. 2003b) and performance is needed to deal with the low-level reasoning at the triple level. The maximum capacity of the triple store is approximately 20

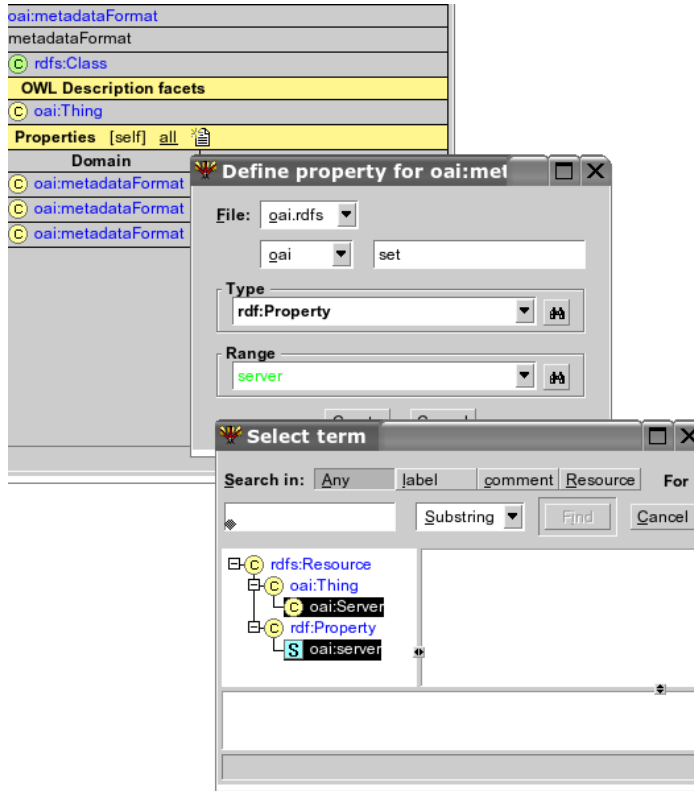


Figure 2.6: Create a property with name *set* and range *oai:Server*. While typing in the *Range* field, the style of the typed text is updated after each keystroke, where bold means ‘ok and unique’, red is ‘no resource with this prefix exists’ and green (showed) means ‘multiple resources match’. Clicking the *binocular* icon shows all matches in the hierarchy, allowing the user to select.

million triples on 32-bit hardware and virtually unlimited on 64-bit hardware.

We summarise some figures handling WordNet 1.6 (Miller 1995) in RDF as converted by Decker and Melnik. The measurements are taken on a dual AMD 1600+ machine with 2Gb memory running SuSE Linux. The 5 RDF files contain a total of 473,626 triples. The results are shown in table 2.1. 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

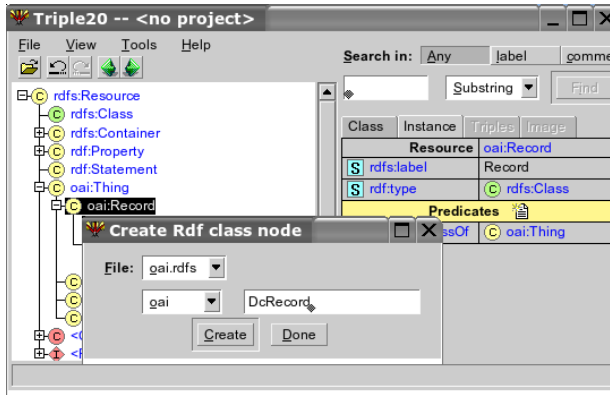


Figure 2.7: Create a new class *DcRecord*. The system proposes the file the class will be saved to (*oai.rdf*) as well as the namespace (*oai-*) based on the properties of the super class. Both can be changed.

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.

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

Table 2.1: Some figures handling WordNet on a dual AMD 1600+ machine. Loading time is proportional to the size of the data.

2.8 Related work

Protégé (Musen et al. 2000) is a landmark in the world of ontology editors. We have described how our design uses the RDF triple model as a basis, where Protégé uses a proprietary internal data model. As a consequence, we can accommodate any RDF document without information loss and we can handle multiple RDF sources as one document without physically merging the source material. Where Protégé is primarily designed as an *editor*, *browsing* is of great importance to Triple20. As a consequence, we have reduced the use of screen-space for controls to the bare minimum, using popup menus and drag-and-drop as primary inter-

action paradigm. Protégé has dedicated support for ontology engineering, which Triple20 lacks.

Miklos et al. 2005 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 to 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.

2.9 Discussion

We have realised an architecture for interactive tools that is based directly on the RDF triple model. Using the triples instead of an intermediate representation 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 facilitates exchange between applications. Interface components are only indirectly related to the underlying data model, which makes it difficult to apply the classical model-view-controller (MVC) design pattern for connecting the interface to the data. This can be remedied using *mediators*: intermediate data structures that reflect the interface more closely and are updated using background processing. Mediators are realised as Prolog predicates that derive a Prolog term from the triple database.

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 applications running on top of the Prolog triple store, such as ClioPatria (chapter 10).

The presented design is applicable to interactive applications based on knowledge stored as RDF triples (research question 3c). The overall design is language independent, although the natural fit of RDF onto Prolog makes it particularly attractive for our purposes.

Software availability

Triple20 is available under Open Source (LGPL) license from the SWI-Prolog website.⁵ SWI-Prolog with graphics runs on MS-Windows, MacOS X and almost all Unix/Linux versions, supporting both 32- and 64-bit hardware.

Acknowledgements

This work was partly supported by the ICES-KIS project “Multimedia Information Analysis” funded by the Dutch government. The Triple20 type-icons are partly taken from and partly inspired by the Protégé project.

⁵<http://www.swi-prolog.org/packages/Triple20>