Abstract

As the final outcome of our work in the context of the Wisdom project, a prototype for query formulation has been developed in order to prove the validity of the M-FIRE framework [3, 5, 4], and the feasibility of our approach to knowledge representation and exploration. The prototype includes a client module (written in C#) and a server module (written in Java) which communicate through a state-based TCP connection. Many clients can connect to a single M-FIRE server and deliver requests to it. In a typical scenario, clients would ask for a representation of some part of the available knowledge, and the server would reply by returning semantically annotated XHTML pages (the M-FIRE framework can actually support other representation formats than XHTML); after rendering is completed, actions made by the user on such documents will trigger new requests, which are eventually translated into SPARQL queries; the results of the generated queries are processed by the M-FIRE server and translated into a new XHTML document, which is then sent back to the client for rendering.
Abstract

As the final outcome of our work in the context of the Wisdom project, a prototype for query formulation has been developed in order to prove the validity of the M-FIRE framework [3, 5, 4], and the feasibility of our approach to knowledge representation and exploration. The prototype includes a client module (written in C#) and a server module (written in Java) which communicate through a state-based TCP connection. Many clients can connect to a single M-FIRE server and deliver requests to it. In a typical scenario, clients would ask for a representation of some part of the available knowledge, and the server would reply by returning semantically annotated XHTML pages (the M-FIRE framework can actually support other representation formats than XHTML); after rendering is completed, actions made by the user on such documents will trigger new requests, which are eventually translated into SPARQL queries; the results of the generated queries are processed by the M-FIRE server and translated into a new XHTML document, which is then sent back to the client for rendering.

1 Introduction

M-FIRE (Metaphor-based Framework for Information Representation and Exploration) is a flexible framework for delivering intuitive, domain-specific representations of RDF documents [3, 5, 4]. In short, M-FIRE allows to instantiate a customized representation and navigation system by declaratively specifying (1) how some pieces of information in a given (RDF) source document shall be represented (through so-called representation rules), and (2) what query shall be triggered by a user action on the provided representation (through so-called navigation rules) in order to obtain a new source document. Representation rules contain a set of directives that are processed by a generic representation engine, whose output is a semantically annotated document to be rendered on the user’s screen by a suited visualization program. Likewise, navigation rules drive the translation of events produced by the user on the delivered representation into queries to be executed on the underlying knowledge base (KB). The new source document obtained as a result of such queries is then processed again by the representation engine, and the newly generated page is sent back to the user’s client program for rendering. Then, the process is repeated (see Section 2 for a sample use case).

The selection of both the representation rule(s) that shall be used to generate a graphical rendering for a given source document and the navigation rule(s) according to which user actions shall be translated into queries may happen in two ways: either by somehow parsing the content of the source document and finding out the most suited rules to handle a specific set of resources, possibly taking into consideration run-time parameters like user preferences, user device, and so on (dynamic rule selection), or by following a finite state automaton specification provided by the system administrator, that we would call a metaphor, possibly chosen by the user from an archive of available metaphors (metaphor-based rule selection). Since the metaphor specifies which representation rule(s) must be followed in order to obtain a graphical rendering for some information, and which navigation rule(s) will drive the translation of actions into queries for obtaining new information to be rendered, one could say that the metaphor actually determines
the sequence of pages that are shown to the user throughout her work session. Section 4 will document the metaphors and rules that we designed for supporting user interaction in Wisdom.

Though M-FIRE supports any rule selection algorithm, the metaphor-based one is the default, built-in mechanism for specifying the application behavior; furthermore, the metaphor-based rule selection is the only selection mechanism supported by the prototype we have built.

The prototype includes a client module (written in C#) and a server module (written in Java), which communicate through a state-based TCP connection – see Section 3 for an in-depth design view of both the client and the server component. The UML deployment diagram shown in Figure 1 illustrates the basic architecture: as one can guess, many clients may concurrently connect to the same server and forward requests to it.

![Figure 1: UML deployment diagram of the M-FIRE prototype](image)

M-FIRE does not place any constraint on the location of both the KB and the metaphor archive: in particular, it is not a requirement that the KB and the metaphor archive reside on the same computer as the M-FIRE server component, provided that an interface for performing remote SPARQL queries is available (metaphors are just text documents obeying a particular grammar, and could be retrieved via HTTP from anywhere).

Figure 2 illustrates the Wisdom setting, where a network of nodes called super-peers is envisaged, each one covering a sub-network of peers that provide instance-level information about some application domain.

![Figure 2: UML deployment diagram of the top-level Wisdom components](image)

Each super-peer defines a schema according to which instances retrieved from its sub-peers are integrated and exposed to other super-peers (and a set of inter-peer concept mappings to enable query rewriting and distributed query processing). Each super-peer features its own
(virtual) KB and hosts an instance of the $P^3PQ$ distributed query processor [2]. The $P^3PQ$ distributed query processor is capable of (1) accepting queries (and returning their results) formulated on the local schema of the hosting super-peer through a Web Service interface, (2) rewriting queries so that they can be forwarded to other super-peers for which a proper set of mappings exists, and (3) integrate the results obtained from many super-peers.

This gives rise to a four-tier architecture in which the two topmost layers (M-FIRE client, M-FIRE server) deal with graphical representation and query formulation, while the lower ones deal with query processing, query routing, instance retrieval and object fusion. The M-FIRE client component allows the user to select:

1. The KB to be explored (i.e., the super-peer to connect to);
2. The metaphor to be used for exploring the selected KB (by choosing among a set of available metaphors).

On the other hand, the server component will manage the back-end issues involved in the process of representation and navigation, and will extract RDF documents from the selected KB by forwarding SPARQL(-like) queries to the $P^3PQ$ module hosted by the super-peer exposing that KB. Though M-FIRE is not bound to any graphical format when generating a representation for a given RDF document, our prototype is only capable of producing table-based XHTML encodings.

2 Use case

As described in Section 1, the prototype we developed includes two components: one client module and one server module, communicating through a state-based TCP connection. In this section, we are going to discuss a use case in order to give a clue on how M-FIRE and our prototype work. A typical interaction sequence between the user and the system gives rise to the following communication pattern between the client and the server:

1. After having established a connection to the M-FIRE server, the user selects the KB (s)he wants to explore and the metaphor according to which that KB shall be explored. The client immediately informs the server about the user’s selection. The text box in the lower part of the GUI shows the raw communication log between the client and the server.

Figure 3: After having connected to the M-FIRE server at the specified network address in (a), the user selects the desired KB and metaphor from the provided listboxes in (b)
2. When the user clicks on the START button, the client sends a HOME request to the server in order to obtain a first “home page” representation as the starting point to navigate the KB. After receiving a HOME request, the server executes a default query (specified by the metaphor) on the KB to extract the RDF source document to be presented to the user; then, it represents that document through the representation rule(s) specified in the metaphor, and sends back the outcome to the client for rendering. The server also selects the navigation metaphor that will be later used to drive the translation of the user actions into queries, again by following the specification contained in the metaphor;

Figure 4: After the user presses the START button in (a), the M-FIRE server executes a default query, represents the resulting source document through the rule specified in the metaphor, and returns the outcome to the client; when receiving a reply from the server, the client renders the document in (b)

3. The the user acts on the obtained representation (e.g., by double clicking on a drawing, doing drag&drop on some visual items, and so on), and the client generates a description of the event – including the semantic annotations of the object(s) on which the action was performed – which it sends to the server as the payload of an ACTION request;

Figure 5: Upon a mouse click by the user on some part of the generated drawing, the client collects the semantic annotations (lower part of the screen) attached to that item and includes them in an ACTION request to the server

4. By following the directives contained in the navigation metaphor selected at step 2, the server processes the event description and generates one or more queries. RDF documents resulting from those queries are then merged together into a new RDF document;
5. A set of representation rules is selected, by following the finite state automaton specification provided by the selected metaphor, to drive the representation of the RDF document obtained at step 4. Then, the representation process is carried out and a new page is obtained to be sent back to the client; also, a set of navigation rules is selected (determined by the metaphor) that will later drive the interpretation of events produced by the user on the obtained page.

6. Then document obtained at step 5 is forwarded to the client for rendering, and step 3 is repeated, except that further user actions will be interpreted according to the navigation rule(s) selected at step 5.

![Figure 6: In (a), a new page is received and rendered by the client. Again, the user acts on it by clicking on a visual item. The client produces a new event description and sends it to the server. The server translates this action into a set of queries, whose merged results form a new source document to be represented. The outcome is then sent back to the client and rendered as in (b). This cycle may be repeated.](image)

We refer to the above interaction cycle as semantic browsing. Semantic browsing sometimes involves a more articulated communication between the client and the server, particularly when handling errors or when accomplishing two-way requests (e.g., the client sending an ACTION request to the server, and the server subsequently requesting the client for parameter filling). In those cases, it is not always true that a single user action is directly translated into a query over the current KB: sometimes, a query may be incrementally constructed as the result of a sequence of actions and selections by the user on the delivered representations. In WISDOM, this feature is used to allow for generic conjunctive queries to be formulated on an ontology schema (see Subsection 4.2 for details).

3 Details

3.1 The client module

Our M-FIRE client is a .NET executable written in C#, and is capable of rendering HTML documents by reusing the Microsoft WebBrowser® control; this is enough for our purposes, because XHTML is the only encoding format supported by our prototype (see Subsection 3.2).

As discussed in Section 1, M-FIRE clients establish a state-based TCP connection with the server in order to initiate the communication. A special connection and selection form located near the upper-right corner of the main window (see Figure 7(c)) is provided for this purpose: the user first specifies the network address and the remote port of the service in the above text boxes, then (s)he presses the small green button on the right to attempt a connection.
Figure 7: The knowledge base viewer, shown in (a), highlights the currently selected KB; the metaphor tab, shown in (b), highlights the currently selected metaphor; the connection and selection form, shown in (c), reports both the currently selected KB and the currently selected metaphor, and shall be used to specify which M-FIRE server the client should connect to.

Once the connection is established, the client retrieves from the server, among other things, the list of the published KBs and the list of the available metaphors for their presentation, and populates the list boxes shown respectively in Figure 7(a) and in Figure 7(b). After selecting one KB and one metaphor, the user may press the start button appearing in Figure 7(c) to get the home representation and start exploring the selected KB.

The main controls in the user interface are the knowledge encoding viewer (see Figure 8(a)) and the source document viewer (see Figure 8(b)). The knowledge encoding viewer embeds the Microsoft WebBrowser® control and shows the XHTML encoding document returned by the server – properly modified to support event handling, as explained later on. The source document viewer reports the set of RDF triples from which the currently delivered representation was obtained (i.e., the source document).

Figure 8: The knowledge encoding tab, shown in (a), displays the currently delivered representation; instead, the source document tab shown in (b) displays the RDF triples of the source document for which that representation was obtained.

Apart from rendering, the client is also responsible for capturing events generated by the user, detecting the visual items on which the action originating the event was performed, extracting the semantic annotations attached to those visual items, and assembling a description of the generated event.

Since the .NET wrapper for the WebBrowser® control does not allow to retrieve the HTML element located at a given pair of screen coordinates, event handling must necessarily be performed from within the browser’s process: some javascript code is perfectly suited for this purpose. The client injects a javascript code snippet in the HEAD section of every document received from the server. This code detects the XHTML element at which a given mouse event (e.g., a left click) was targeted as soon as it occurs, extracts the RDF text of the associated semantic annotation (see Subsection 3.2 for details on how semantic annotations are encoded in XHTML documents produced by the server), builds a description of the occurred event, and overwrites the current page URL so that it includes the event description as a substring. Then, whenever the page URL is rewritten, a custom hook routine in the client is invoked which extracts the event description from the URL and sends an ACTION request to the server.

The entire communication between the client and the server is logged by the client and listed
in the special text box. The text box is contained inside the log tab, located in the lower part of the main window and shown in Figure 9. Messages longer than 100 characters are truncated to fit the size of the window.

![Figure 9: The log tab shows the low-level communication between the client and the server](image)

One of the most important forms, especially for debugging representation and navigation rules, is the annotation tab. The annotation tab is located in the lower part of the main window, next to the log tab, and is shown in Figure 10. Whenever the user moves the mouse over a visual item in the currently rendered XHTML encoding, the semantic annotation tag will be updated with the semantic annotations associated to that visual item.

![Figure 10: The annotation tab shows the semantic annotations associated to the currently highlighted visual item(s) in the delivered representation](image)

We conclude the overview of the user interface of our M-FIRE client prototype by illustrating the query tab. The query tab, shown in Figure 11 and located next to the annotation tab in the lower part of the main window, allows the user to check the low-level text of the query currently built by the M-FIRE server. In M-FIRE, queries may either be generated at a single stroke, as the consequence of an event triggered by the user, or they may be assembled incrementally throughout a sequence of many user’s actions on the delivered representation. If this is not the case, the query tab will not be active and its content will be empty.

![Figure 11: The query tab shows the partial query (or queries) incrementally built by the server throughout a sequence of actions by the user](image)

### 3.2 The server module

The M-FIRE server component of our prototype acts much like a Web server: it keeps listening to a particular TCP port until an inbound connection requests is signalled. When that happens, the server creates a new thread for handling request sent through that connection. Concurrent requests are allowed and are handled by different threads, which can only access their local data. A few global data are also maintained by the server (for instance, the number of active connections), but they are properly protected from concurrent modification. Once a connection is established, the client may forward any of the following kinds of request to the server:

- **HOME request** When receiving a HOME request, the server executes the default query specified by the selected metaphor in order to extract an initial RDF source document. This
document is then processed by the representation engine, and the outcome is sent back to
the client as a “home page” for exploring the KB.

**ACTION request** ACTION requests are sent to the server whenever an event is generated by
the user on the delivered representation (e.g., a mouse click or a drag&drop). ACTION
requests include a description of the occurred event, and the semantic annotation of the
visual items involved in the action.

**SOURCE request** The server replies to a SOURCE request by sending back to the client the
RDF source document for which the currently displayed representation was obtained. The
M-FIRE client in our prototype sends a SOURCE request whenever the user clicks on the
source document tab shown in Figure 8(b).

**RULE request** A client can forward a RULE request to the server in order to retrieve both the set
of representation rules that drove the generation of the currently displayed representation,
and the set of navigation rules according to which the user’s actions on that representation
will be translated into queries.

**ENCODER request** A client can forward a ENCODER request to the server in order to retrieve
the encoding format (i.e., HTML, SVG, GraphML, etc.) of the currently displayed repre-
sentation. In our M-FIRE prototype, XHTML is the only supported format. As a further
limitation, tables, text boxes, and images are the only graphical constructs (with styling
directives, of course) that may be used by representation rules. For a more in-depth ex-
planation of the encoding process and a detailed discussion of the relationship between
representation rules and the encoding format, please refer to [3, 5].

**LIST request** LIST requests can be sent to retrieve the list of one or more of the following:
the available metaphors, the published knowledge bases, the available representation and
navigation rules, and the available encoders.

**QUERY request** Not all user actions in M-FIRE are directly translated into queries. Some-
times, queries can be constructed incrementally as the consequence of many actions and
selections (see Section 3, step 6). When this is the case, the client may retrieve the cur-
rently assembled query by sending a QUERY request to the server. Our M-FIRE client
prototype will forward such a request to the server every time the user clicks on the query
tab shown in Figure 11.

**CLOSE request** A CLOSE request is received by the server when a client is about to discon-
nect. Whenever that happens, the server frees the associated resources and stops the
responding request handler thread.

The M-FIRE server is written in Java and implements the M-FIRE representation and nav-
igation engines. About representation, as explained in [3, 5], M-FIRE allows to first specify
an abstract description of the representation to be generated, not bound to any specific graph-
ical format, through representation rules. Only in a subsequent step, the abstract description
is translated into a concrete target format (e.g., SVG, HTML, or GraphML) by a dedicated
program, so to accomplish what we call encoding. In principle, different encodings are possible
for the same abstract representation document: however, it is not always the case that every
encoding is suitable to convey a given abstract representation. For instance, vector graphics,
graph nodes with curve edges, and many other visual items or effects are not encompassed by
HTML. In our prototype, we have only developed an XHTML encoder: consequently, represen-
tations defined by our rules can only feature tables, images, text areas, and styling instructions
supported by XHTML and CSS.
Semantic annotations for a given representation object are embedded in its XHTML encoding by creating an invisible DIV element associated to the outermost XHTML tag contributing to that object’s representation. The association is established by referring the (automatically generated) identifier of the DIV element inside the annotated tag’s ID attribute. The text content of the DIV element includes the set of RDF triples which constitute the meaning of the referring tag. This allows the M-FIRE client to extract the formal description of a visual item the user acted on (see Subsection 3.1).

About navigation, M-FIRE does not specify how to access RDF data: triples could be either retrieved from a text file, stored in a persistent database, or managed in-memory. The M-FIRE framework relies on the presence of an RDF provider exposing an interface for query execution; in our prototype, we have plugged an RDF provider proxy into the framework, remotely communicating with the $P^3PQ$ component and blindly forwarding queries to it. The UML deployment diagram shown in Figure 12 provides a refined view of the architecture illustrated in Figure 2.

Figure 12: Refined UML deployment diagram of the M-FIRE prototype in WISDOM

The $P^3PQ$ component exposes its API through a Web Service interface and basically implements two main operations:

1. executeQuery: runs a SPARQL-like query with preferences (see [6] for the query language grammar) over the virtual KB exposed by the specified peer, forwarding a properly rewritten version of the query to other peers in the network and integrating the results. Results are returned in the form of a Jena ResultSet [1], which is then converted into an equivalent RDF document by the M-FIRE server before being processed for representation.

2. getOntology: returns and OWL document describing the schema of the ontology of a specified peer. This operation will be invoked when the user presses the start button to obtain a “home page” representation as the starting point to interact with the KB.

M-FIRE supports queries over any kind of RDF document: thus, RDF providers on which the M-FIRE framework may rely are supposed to implement a generic executeQuery operation. However, in WISDOM, the schema and instance level are well separated, and queries can only be formulated on the ontology schema and can only have instance-level data as a result. This is why a special getOntology operation, different from executeQuery, is required.

To be compliant with the M-FIRE framework, our proxy adapter only implements one generic executeQuery function: the implementation of executeQuery by our adapter forwards a getOntology request to the $P^3PQ$ query processor whenever it infers from the query pattern that solutions will have bindings to resources in the ontology schema; on the other hand, a executeQuery request is issued when the query result will contain only instance-level data.

4 The metaphors

In the context of the WISDOM project, two metaphors have been designed expressing alternative ways to represent an ontology schema, as detailed in the following subsections. Both metaphors share the same navigation rules, thus allowing to uniformly formulate conjunctive queries over
the current representation of the ontology classes and properties, disregarding the way those classes and properties are represented.

Furthermore, both representation metaphors adopt the same representation rules for representing query solutions as a vertical list of tables displaying a set of variable bindings (name-value pairs). All representations delivered by our M-FIRE server prototype are XHTML documents in which images and pieces of text are arranged according to a table-based layout.

4.1 Representation

Two sets of representation rules have been designed that differentiate how an ontology schema is represented by the two available metaphors. Both sets of rules are meant to provide a graphical representation for some classes of statements found in the source RDF document, like instantiations of rdfs:Class, instantiations of owl:ObjectProperty and owl:DatatypeProperty, and triples having rdfs:domain and rdfs:range as predicate.

In both cases, a table-based layout is produced, each class being represented by a frame that contains the class name and a picture (reference is made to Figure 13); the picture is meant to give a clue about the real-life entity denoted by the corresponding class.

![Figure 13: Two alternative views of the same ontology schema, as provided by the two available metaphors. Such different views are obtained by letting different sets of representation rules drive the representation process.](image)

As shown in Figure 13(a), the first metaphor renders each class by displaying its associated icon on the left side of the frame (together with her local name), and by listing on the right side the local names of those properties whose domain includes that class. Within such list, datatype properties come first and feature a light yellow background and a black font, while object properties are written in brown bold characters on a dark yellow background. Moreover, for each object property, the icons of every class included in the property range are also displayed on the left of the property’s local name.

On the other hand, Figure 13(b) shows how the second metaphor would represent the same piece of information: the local name and the icon of the class are now placed side by side on the top part of the frame, and properties are listed right below. Object properties come first and are written in bold blue characters on a light blue background, while datatype properties are drawn in black characters on a cyan background. Finally, for each object property and datatype property, the property’s local name is put on the left side of the row, while the right side contains the local names of the classes included in the property range.

As discussed in Subsection 4.2, both metaphors adopt a single set of navigation rules. Such rules specify how to translate the actions performed by the user on the currently represented classes and properties into queries; notably, this translation does not depend on how those
classes and properties are represented. Once a conjunctive query has been formulated by the user through a sequence of actions on the schema representation, a set of query solutions will be returned (properly encoded as an RDF graph), and shall be delivered to the user. For this purpose, a rudimental representation is given through a further set of representation rules, shared by both metaphors, that produces a result similar to the one shown in Figure 14.

Figure 14: Query solutions are presented by both metaphors as a vertically arranged list of variable binding sets.

As can be noticed, each query solution is represented by a table, whose rows contain the representation of the variable bindings defining that solution. Each variable binding is a pair associating a variable name (printed on the left half of the corresponding row) to a literal value (appearing on the right half).

4.2 Navigation

For what concerns navigation, a set of navigation rules have been conceived allowing to express conjunctive queries on an ontology schema. The way such queries are formulated does not depend on the adopted metaphor, at least conceptually. Suppose for instance, that given any representation of the Booking ontology schema, the user wants to retrieve the name and the address of all hotels in the KB: then, the user shall left-click on the text representing the two properties to be selected for output (in whatever order), and then click on a prearranged button to launch the query. Where the property name is located inside the frame representing the class, and which font is used to write it, have no influence on the sequence of object selections that the user has to perform in order to formulate a given query.

The $P^3PQ$ query processor supports a SPARQL-like query language whose syntax is defined in [2]: basically, SPARQL SELECT queries are supported with some constraints on the FILTER statements and on the use of variables, extended with an optional PREFERENCES clause allowing to express complex combinations of user preferences. Furthermore, variables included in the SELECT clause may only be bound to literal values: therefore, only datatype properties may be selected for output by clicking on their representation.

The navigation rules allow to formulate a query through the following actions (the order is irrelevant):

- A left-click on a datatype property selects that property for output;
- A right-click on a datatype property allows to specify a filtering expression (a comparison operator and a value) for that property. Figure 15 shows the dialog window popped up by the client module when this action is performed;
- A left-click on an object property uses that property to express a join between two classes (in our prototype, a class cannot play more than one role in a query).
As discussed at the end of Section 2, semantic browsing in M-FIRE is realized through two navigation mechanisms: in the simplest case, a query may be triggered by a single user action and return a new source document to be represented. However, this paradigm is not powerful enough to allow for more expressive navigation primitives. In particular, it is not enough to allow for formulation of conjunctive queries. M-FIRE supports the latter kind of navigation by allowing queries to be assembled incrementally through a sequence of user actions: besides triggering the execution of a query, in fact, navigation rules may also instruct the M-FIRE server to create new session variables or to update the value of an existing session variables.

The navigation rules that we designed for our prototype follow precisely this interaction model: when a left-click is performed by the user on a datatype property in order to select it for output, the navigation rule handling that event will append a string to the session variable used for storing the text of the \texttt{SELECT} clause; likewise, whenever a right-click is performed on a datatype property, the navigation rule handling that event will append a statement filter to the session variable that stores the text of the \texttt{WHERE} clause; finally, whenever a left-click is performed on an object property, a join statement is properly set up to link the variables associated to the classes in the domain and the range of the selected property.

When the query is launched (by pressing a dedicated button), the navigation rule handling that event will eventually put together the separate parts and forward the query to the M-FIRE server. In turn, the M-FIRE server will deliver the query to the P\textsuperscript{3}PQ module through the API illustrated in Section 3.2.

Once the instance-level result is obtained and represented, no more navigation is possible, and the interaction cycle therefore comes to an end point. This is due to the fact that instance-level query solutions (i.e., variable bindings) are not semantically annotated. In order to formulate a new query, the user has to return back to the ontology schema, by pressing the \texttt{start} button shown in Figure 7(c).

5 Conclusions

We have developed a prototype of the M-FIRE framework in the context of the \textsc{Wisdom} project, and we showed how M-FIRE could be used to deploy a graphical interface for formulating conjunctive queries over an ontology schema. The possibility to adopt alternative metaphors for exploring a given ontologies proves the flexibility of the tool, while the use of the same navigation rules for both metaphor in spite of different representation rules is a proof that modularity and reusability have been addressed in the desing of the framework.
References


