A Query Formulation Tool for Semantic Networks

Christos Asaridis

Thesis submitted in partial fulfilment of the requirements for the

Masters' of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, Heraklion, Crete,
GR-70013, Greece

Thesis Advisors:
Assistant Prof. *Yannis Tzitzikas*
Researcher *Martin Doerr*

This work was partially supported by the **Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).**

University of Crete
Computer Science Department

A Query Formulation Tool for Semantic Networks

Thesis submitted by
Christos Asaridis
in partial fulfilment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Christos Asaridis

Committee approvals: _____
Yannis Tzitzikas
Assistant Professor, Thesis Supervisor

_____
Martin Doerr
Researcher, Committee Member

_____
Dimitris Plexousakis
Professor, Committee Member

Departmental approval: _____
Angelos Bilas
Professor, Director of Graduate Studies

Heraklion, December 2013

Abstract

Over the last few years, there has been a trend of developing greater and greater metadata repositories, mostly based on RDF/S technology. A good example is the Europeana project in the field of cultural heritage. The international standard ISO (CIDOC Conceptual Reference Model) is a rich RDF schema able to integrate complex cultural data. Far more simplistic models (such as Dublin Core), which consist exclusively of core metadata (few classes-flat relationships), have only a limited potential to integrate knowledge and to apply reasoning to the schema. On the other hand, searching complex structures using query languages like SPARQL is a difficult and painful procedure because of the large amount of classes and properties in ontologies, such as the CIDOC-CRM.

In order to simplify the search process, we introduced a simpler model consisting of only a few fundamental classes and relationships. Fundamental relationships can be defined by using a "paths language" (over the CIDOC-CRM schema), which is designed for non-expert users. In order to take advantage of this paths' language, we have designed and implemented a graphical editor, which assists users in writing triple patterns and verifying their syntax. The graphical editor enables users to overview the declared paths and allows them to check the validity and the completeness of a query against the intended functionality. In addition, the editor allows users to easily modify, relax and restrict a query. The given functionalities help users to avoid errors, such as the misspelling of the name of classes (or properties), the omission of the use of predefined special characters for paths connection and the use of different kinds of brackets according to the context.

For the evaluation of the graphical editor, we asked non-expert users to test it and give us feedback of their usage experience. Users tested it in the CIDOC-CRM and the CIDOC-CRMdig, and the results of these tests were positive.

Περίληψη

Τα τελευταία χρόνια, επικρατεί η τάση για τη δημιουργία όλο και μεγαλύτερων αποθετηρίων μεταδεδομένων, συνήθως βασισμένων σε RDF/S τεχνολογίες. Ενδεικτικό παράδειγμα από το χώρο της πολιτισμικής κληρονομιάς, είναι το έργο Europeana. Το διεθνές πρότυπο ISO21127 (CIDOC Conceptual Reference Model) είναι ένα πλούσιο εννοιολογικό σχήμα ικανό να περιγράψει πολύπλοκα πολιτισμικά δεδομένα. Πιο απλοϊκά μοντέλα μεταδεδομένων (όπως το Dublin Core), τα οποία αποτελούνται αποκλειστικά από μεταδεδομένα πυρήνα (λίγες κλάσεις - οριζόντιες σχέσεις), έχουν περιορισμένες δυνατότητες στην ολοκλήρωση των πληροφοριών (integration) και στο συλλογισμό (reasoning) επί αυτών. Από την άλλη πλευρά, η αναζήτηση σε πολύπλοκες δομές με τη χρήση γλωσσών επερωτήσεων όπως η SPARQL, αποτελεί μια δύσκολη και επίπονη διαδικασία εξαιτίας του μεγάλου πλήθους κλάσεων και ιδιοτήτων των οντολογιών, όπως το CIDOC-CRM.

Για την διευκόλυνση της διαδικασίας αναζήτησης, προτείνεται ένα απλούστερο μοντέλο, αποτελούμενο από λίγες θεμελιώδεις κλάσεις και σχέσεις. Οι θεμελιώδεις σχέσεις μπορούν να οριστούν χρησιμοποιώντας μια 'γλώσσα μονοπατιών'' (επί του CIDOC-CRM) η οποία απευθύνεται στους μη έμπειρους χρήστες. Για την αξιοποίηση της γλώσσας μονοπατιών στην εργασία αυτή σχεδιάστηκε και αναπτύχθηκε ένας γραφικός συντάκτης ο οποίος επιτρέπει στο χρήση να συντάξει επερωτήσεις και να επιβεβαιώσει την ορθότητά τους. Ο γραφικός συντάκτης επιτρέπει την επισκόπηση των δηλωθέντων μονοπατιών, καθώς και τον έλεγχο εγκυρότητας και πληρότητας μιας επερώτησης. Επιπλέον, προσφέρει τη δυνατότητα τροποποίησης, 'χαλάρωσης'', και περιορισμού μιας επερώτησης. Οι προσφερόμενες λειτουργίες βοηθούν το χρήστη να αποφύγει τα λάθη, όπως ορθογραφικά λάθη στα ονόματα των κλάσεων (ή των ιδιοτήτων), παράλειψη των απαιτούμενων ειδικών χαρακτήρων για τη σύνδεση μονοπατιών, λάθος τύπος παρενθέσεων ανάλογα με την περίσταση, κ.α.

Για την αξιολόγηση του γραφικού συντάκτη ζητήθηκε από μη έμπειρους χρήστες να δοκιμάσουν και να αξιολογήσουν την εμπειρία χρήσης του. Οι χρήστες δοκίμασαν το λογισμικό στο CIDOC-CRM και στο CIDOC-CRMdig, και τα αποτελέσματα αυτών των δοκιμών ήταν θετικά.

Acknowledgements

Στο σημείο αυτό θα ήθελα να ευχαριστήσω τους επόπτες καθηγητές μου, κ. Ιωάννη Τζίτζικα και κ. Martin Doerr, τόσο για την εμπιστοσύνη που έδειξαν στο πρόσωπό μου, αναθέτοντάς μου τη συγκεκριμένη διπλωματική εργασία, όσο και για την άψογη συνεργασία μας τα δύο τελευταία χρόνια, καθώς και για την ουσιαστική τους καθοδήγηση και συμβολή στην ολοκλήρωση της παρούσας εργασίας. Θα ήθελα να εκφράσω, επίσης, τις ευχαριστίες μου στον καθηγητή κ. Δημήτρη Πλεξουσάκη για τη μεγάλη προθυμία του να συμμετέχει στην τριμελή επιτροπή.

Ακόμα, θα ήθελα να ευχαριστήσω το Ινστιτούτο Πληροφορικής του Ιδρύματος Τεχνολογίας και Έρευνας για την υποτροφία που μου προσέφερε καθ΄ όλη τη διάρκεια της μεταπτυχιακής μου εργασίας, καθώς και για την πολύτιμη υποστήριξη σε υλικοτεχνική υποδομή και τεχνογνωσία.

Πολλές ευχαριστίες θα ήθελα να εκφράσω επίσης στην Αθηνά, η οποία με αμέριστη υπομονή και κατανοήση ήταν εκεί να με υποστηρίζει και να με βοηθάει για το καλύτερο. Επίσης, στους φίλους μου Γιώργο, Σταύρο, Λευτέρη, Αντώνη, Μαρία, Κατερίνα, Σταμάτη και Μαριγιάννα για τη στήριξή τους και για όλες τις στιγμές που μοιραστήκαμε όλο αυτό τον καιρό. Ιδιαίτερες ευχαριστίες στην Κατερίνα Τζομπανάκη για την καθοδήγηση και την εποικοδομητική συνεργασία μας.

Τέλος, θα ήθελα να ευχαριστήσω ιδιαιτέρως τους γονείς μου Ιορδάνη και Μαρία, και την αδελφή μου Νικολέτα, για την υποστήριξη και την αγάπη που με περιέβαλαν.

στους γονείς μου

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The General Problem

Recent years have seen a remarkable surge in the daily information supply, a fact that in itself demonstrates the existence of sufficient data storage means. This phenomenon has led to the categorization of our data into different kinds of databases, and more specifically, has led to them having to be split into two distinct categories. This categorization is based on the nature of the information they store, that is, whether it is structured or unstructured. This differentiation in turn necessitates distinct means of information access. So, when in search of the most appropriate querying method we must also take into account the kind of database we are querying.

Unstructured database systems are those which are accessed by keyword-based search engines. Keyword-based searches require no prior knowledge of the structure of the database content. This is the main reason that the Web employs search engines which rely on this method in order to return as much data as possible. However, there are serious problems associated with this method such as high recall and mediocre to low precision rates. This happens because even though the main relevant pages are retrieved, with them are also retrieved thousands of documents which are only mildly relevant or completely irrelevant.

Structured database systems however, are based on a schema description, which is used for the formulation of queries by structured querying languages. Structured data is searchable by data type within its content. Furthermore, it is understood by

computers and is also efficiently organized for human readers. The most common form of structured data - or structured data records (SDR) - is a database where specific information is stored based on a methodology of columns and rows, a good example being, the kind of databases widely known as SQL[1]. These kinds of databases pose greater restrictions for the user, because the semantics of the schema (e.g. table and column names) must be known to the user in order for him to create a query. A plethora of such systems are based on the *closed-world assumption* (CWA).

In stark contrast, there are the semantic repositories which are structured to follow the *open-world assumption* (OWA)[1]. In 'Open World' systems, such as Digital Repositories, it is possible for information to be organized by different people, by using a schema in different ways, or even by using different schemata and languages. As a result, the information is by nature incomplete. The freedom offered by this assumption, is the root cause of a great challenge; the challenge to correlate different sources or even the sources themselves. The complexity grows if we broaden the assumption of semantic repositories to a larger scale and more specifically to a semantic network that consists of several schemata. Currently, the most popular search method in Open World systems is the keyword search which usually yields high recall rates but mediocre to low precision rates. But again, this requires the existence of the search term in the metadata in all correct fields.

The expectations of the Semantic Web are to solve through a variety of means, problems of recall and precision for data that has not been backed up, those of high redundancy, problems of resorting to rich formally structured metadata for documents of interest (and links between them), and difficulties encountered when combining data with general formal scenery knowledge. To achieve this purpose, Semantic Web Languages are used to formulate the data. More specifically, data is formulated under the *Resource Description Framework* (RDF) and the *Web Ontology Language* (OWL) schemata that are globally accessible via the Internet and can be combined to a certain degree. RDF Triple Store is the common name given to a database management system for RDF Data. This system provides data management and data access via Application Programming Interfaces (APIs) and query languages to RDF Data. Practically, many Triple Stores are in fact Quad Stores due to the need to maintain RDF Data provenance within the data manage-

[1] http://www.sql.org

ment system. Any Triple Store that supports Named Graph functionality is more than likely a Quad Store. The most advanced Digital Library systems, such as the Europeana[2] or cultureSampo[3], are based on this technology.

In systems like this, the CIDOC Conceptual Reference Model (ISO 21127) has become popular as a rich RDF schema able to integrate complex cultural data. Far more simplistic models, such as Dublin Core, which consist exclusively of core metadata (few classes-flat relationships), have only a limited potential to integrate knowledge and to apply reasoning to it. Some of the projects which use CIDOC-CRM are the German Digital Library[4], the ResearchSpace Project[5], the WISSKI Project[6], and the CLAROS Project[7].

The Semantic Web is an Open World system, which by definition means that users can know neither the exact content of the Semantic Web nor exactly in which terms things have been documented. Querying hundreds of different kinds of properties individually creates a huge recall gap compared to text retrieval, and querying a conjunction of even a few properties tends to frustrate the users with empty answers. Conversely, the idea of restricting the global semantic network to "core metadata" detracts from the reasoning and precision promised by the Semantic Web. Moreover, this idea constricts the needs of researchers and does not alleviate the problems intrinsic to incomplete knowledge.

1.2 The Proposed Framework

In our previous work [2], we had suggested a framework that would improve the effectiveness of searching in semantic networks. Such effectiveness related problems are caused by low recall rates. Most of the proposals regarding low recall rates in semantic networks focus on using smaller schemata with as little as possible information at the level of constructing the semantic web. A good example of this technique is the Dublin Core [8], in which the semantic network is formed as flatly as possible. Despite the effectiveness of this proposal in many cases, it

[2] http://www.europeana.eu/portal/
[3] http://www.kulttuurisampo.fi/index.shtml
[4] http://www.deutsche-digitale-bibliothek.de/
[5] http://www.researchspace.org
[6] http://wiss-ki.eu
[7] http://explore.clarosnet.org
[8] http://dublincore.org/documents/dces/

proves inefficient when we try to map complicated scientific data to small schemata with restricted semantic potential. Our suggestion was to simplify the schema for querying complex semantic networks, as opposed to focusing on simplifying the schema for the construction of the semantic network offered by the other proposals.

To achieve this, we had suggested a new promising querying system for semantic networks based on a few fundamental categories (FCs) and relationships. This system is based on modeling the world into five fundamental categories and defining certain generic fundamental relationships among them. Through this approach the size and the structure of the search model comes closer to core metadata that is more friendly to users. Furthermore, we can get higher recall rates if the simplified model subsumes the properties of the richer one. A possible problem is the reduction in precision rates because of the use of property propagation. However, this problem can be reduced by composing specializations of fundamental relationships or by expressing more restrictions over queries.

Subsequent to this theoretical approach to the effectiveness problem in searching semantic networks, the practical implementation of this proposed framework within the context of the CIDOC-CRM schema was begun. Fundamental relationships can be defined by using a "paths' language" over the CIDOC-CRM schema, which is designed to be easy to write and to comprehend by non-expert users. Through the use of this language, users are able to create complex paths in the CIDOC-CRM that contain all possible interpretations of the fundamental relationships in CIDOC-CRM. The design and the implementation of the "paths' language" was one of the two main challenges of this thesis.

The other main challenge was the construction of an integrated development environment (IDE) for the suggested paths' language. Since our target group is the non-expert user we decided to design and implement a Visual Query System (VQS), aiming at enlarging the communicational bandwidth of our paths' language, both by using a wider range of communicational media and by relying on human intuition as much as possible. To pursue this goal, we have developed a graphical editor which provides a visual query language counterpart to our paths' language, which assists in composing triplet patterns and verifying their syntax through the CIDOC-CRM. The development of the editor involved tackling issues from various perspectives such as layout, readability and interaction design.

The graphical editor provides an overview of declared paths that allows the

users to check the validity and the completeness of a query against its intended functionality. In addition, the editor allows users to easily modify, relax or restrict a query, with the overall purpose of alleviating the problem of errors. Such errors include misspelling the name of classes or properties during path composition, forgetting the use of some predefined special characters, like path connection characters, and the use of different kinds of brackets. Moreover, the graphical editor is able to read data from file. The imported data can be either in original software files, or in paths' language files. Besides saving the data at the end of the editing process, the graphical editor can export it back into the paths' language format.

1.3 Thesis Structure

In the next chapter we will extensively analyse the background of the problem that we are trying to solve and will subsequently explain our approach in solving it in meticulous detail. In chapter 3, we will make a linguistic analysis of our proposal where we describe our path expression language. In this analysis, we will demonstrate its syntax and expressive power. Chapter 4 describes our technical approach in two parts. The first regards the design and development process of path expression language grammar, while the second refers to our graphical editor, the process involving its design, all the critical decisions taken concerning human-computer interaction, and the editor's functionalities. Chapter 5 includes the validation process and its results, and finally, in chapter 6, our conclusions and suggestions for future work are presented.

Chapter 2

Motivation and Related Work

2.1 Background

Throughout this thesis, terminology pertaining to the Semantic Web will be encountered. In order to make it easier for the reader to track these terms, a concise description of the most used terms will be given in the following paragraphs.

Firstly, we are going to explain what the ontologies on the semantic web actually are. Ontologies are considered one of the pillars of the Semantic Web, although they do not have a universally accepted definition. A (Semantic Web) vocabulary can be considered a special form of (usually light-weight) ontology, or sometimes merely a collection of Uniform Resource Identifiers (URIs) with a (usually informally) described meaning [3]. On the Semantic Web, vocabularies define the concepts and relationships (also referred to as "terms") used to describe and represent an area of concern. Vocabularies are used to classify the terms that can be used in a particular application, characterize possible relationships, and define possible constraints to using those terms. In practice, vocabularies can be very complex (encompassing several thousands of terms) or very simple (describing one or two concepts only).

There is no clear division between what is referred to as "vocabularies" and "ontologies". The trend is to use the word "ontology" for a more complex, and possibly quite formal collection of terms, whereas "vocabulary" is used when such strict formalism is not required or is required only in a very loose sense. Vocabularies are the basic building blocks for inference techniques on the Semantic Web [4].

The Resource Description Framework (RDF) is a framework that represents information and encoding metadata and other knowledge on the Semantic Web [5]. The RDF has features that facilitate data merging even if the underlying schemas differ, and it specifically supports the evolution of schemas over time without requiring all the data consumers to be changed. It is an abstract model, a way to break down knowledge into discrete pieces, and while it is most popularly known for its RDF/XML syntax, RDF can be stored in a variety of formats. An abstract data model needs concrete syntax in order to be represented and transmitted, and RDF has been given syntax in XML.

Resource Description Framework Schema (RDF Schema, variously abbreviated as RDFS, RDF(S), RDF-S, or RDF/S) is a set of classes with certain properties using the RDF extensible knowledge representation language, providing basic elements for the description of ontologies, otherwise called RDF vocabularies, intended to structure RDF resources [6]. These resources can be saved in a triple-store to facilitate their accessibility within the query language SPARQL.

Ontology Web Language (OWL) is a set of mark-up languages which are designed for use by applications that need to process the content of information instead of just presenting information to humans [7]. OWL ontologies describe the hierarchical organization of ideas in a domain, in a way that can be parsed and understood by software. OWL has more facilities for expressing meaning and semantics than XML, RDF, and RDF-S, and thus OWL goes beyond these languages in its ability to represent machine interpretable content on the Web. It can be used to explicitly represent the meaning of terms in vocabularies and the relationships between those terms.

The Simple Protocol and RDF Query Language (SPARQL) defines a standard query language and data access protocol for use within the Resource Description Framework (RDF) data model [8]. It works for any data source that can be mapped to RDF. The SPARQL query language is based on matching graph patterns. The simplest graph pattern is the triple pattern, which is like an RDF triplet but with the possibility of a variable instead of an RDF term in the subject, predicate, or object positions. It is the most accepted querying language for RDF and is a W3C recommendation.

Visual Query Systems (VQSs) are defined as systems for querying databases that use a visual representation to depict the domain of interest and express related

requests [9]. Such systems take advantage of the well-known high bandwidth of the human-vision channel, allowing both recognition and handling of large quantities of information as well as the possibility of using visual feedback to improve the human-computer dialogue. VQSs provide both a language to express the queries in a visual format and a variety of functionalities to facilitate user-system interaction. As such, they are oriented towards a wide spectrum of users, especially novices who have limited computer expertise and generally ignore the inner structure of the accessed database.

2.2 Problem Statement

Currently, many attempts have been made to overcome the problem of unsatisfactory recall and precision rates which appear in semantic web queries. The most efficient solution to such problems is the construction of rich metadata as well as the use of associative queries [10]. Because the former solution, that of rich metadata structure schemata, is provided for, there is much scope for research and development of the latter, that of the use of associative queries. The greatest issue arising from the above is why associative queries display such low performance in Open World systems, more specifically, in information aggregation systems such as cultural metadata. This poor performance is due to various factors, some of which are analysed below.

One of these factors is that the relationship the user is looking for either has not been documented, or has been documented in a way unfamiliar to the user. For example, he may be searching for a thing which demonstrates the property "made of stone", but his search may not yield a result, because the thing has already been recorded in the system with the property of "has part stone", or demonstrates the property of "has type stone", which essentially is "made of stone". Working with ISO21127 we observed that it is practically impossible to normalize a global model for information integration into a unique representation of each property. Rather, in aggregation systems and the Semantic Web, one has to accept that properties are represented by reasonable alternatives that can be related through deductions. ISO21127 describes explicitly some of the most prominent alternatives as "shortcuts" (joins), which the precision librarians can only achieve - after extensive training - with their cataloguing rules which are restricted to very small

schemata (such as MARC [11]) and do not scale to the Semantic Web.

Another factor is that most queries the user makes are not isolated but comprise a sum of subqueries. That is, the user may for example be searching for things which are made of a specific material, come form a specific place, belong to a specific time period and so on. In such circumstances, the lack of recall for each of the subqueries, as a result of the absence of alternatives or as a result of incomplete knowledge, results in an abrupt decline in total recall rate. A successful solution to this problem is the use of an advanced search which provides the user with the ability to increase recall rates to the given subqueries. However, in order for the above technique to be efficient, the user must be informed of the recall rates of each subquery. In this way, the user is able to perform changes or even to completely remove the flawed conjunction in order to improve the total recall rate.

In addition, another difficulty the user faces in an attempt to compose adequate semantic queries, is the lack of facility inherent in the semantic querying languages themselves. A characteristic example is SPARQL, which is the language of choice among information technology experts. SPARQL is a language which has transferred the original relational paradigm to the graph structure of the semantic web. This, however, has resulted in an overly complex system which has created problems for even the expert user. Specifically, when SPARQL queries are essential to the representation of the proposed framework, it is difficult to verify that SPARQL will yield the results intended by a domain expert simply by reading it.

Despite the fact that the availability of semantic query languages is an important aspect of information retrieval capability, query developers are likely to benefit from the additional availability of tools that assist them with respect to the process of query formulation (i.e. the process of creating or editing a query). Ideally, query formulation tools should exploit the user interaction capabilities that contribute to the efficient design of accurate queries, and should take advantage of the power and expressivity provided by the constructs of the target query language to the maximum extent possible.

Most attempts to support the user with respect to query formulation have focused on graphical or visual techniques in the form of Visual Query Systems (VQSs) [9]. VQSs provide a number of advantages as opposed to simple text editors. Most conspicuously, such systems support the user in developing syntactically valid queries. They serve to constrain or guide editing actions so as to lessen the risk of

lexical or syntactical error. Other potential advantages include improved efficiency, understanding, and reduced training requirements.

2.3 Related Work

Conveniently and efficiently querying semantic networks is a problem of vital importance. In recognition of this, much research has been both undertaken and published. In this section, we note the most prominent, which we have separated into three categories. The first category is concerned with ways in which users can be aided in structuring their queries through facilities provided in the interface. The second category describes proposed solutions which assume that any simplification process must stem from the foundation of the network. They must focus on the employment of flat, simple metadata schemata. In the final category, reference is made to research already conducted towards simplifying the query model, and its commonalities to ours.

The creation of systems concerned with aiding the user to formulate semantic queries on the interface, are listed in the first category. Such formulations can be facilitated with menu-guided user interfaces employed to specify subject-property-object triplets, in combination with a look-ahead enhanced search, such as that in DBpedia [12]. In [13] a solution is proposed on query formulation based on both class and property browsing over plain or even fuzzy RDF/S. Nevertheless, so long as the end user understands the semantics of the classes and the linkage inherent in the schema, formulating queries by transitions on the RDF schema will succeed. Moreover, since a direct translation of the schema in terms of natural language is not always valid, employing it will probably lead to misnomers in the use of the schema. In addition, it would prove challenging for the end user, browsing on querying time, to formulate queries comprising long chains of triplets that depend considerably on property propagation through part/whole and derivation chains. However, this approach could be useful for formulating the paths which adhere to the Fundamental Relationships, an approach deserving of future consideration. There are, also, other systems, which incorporate a query formulation utilities with graph representations of the ontology. One of them are NiteLight [14]. The NITELIGHT enables users to create SPARQL queries using a set of graphical notations and GUI-based editing actions. The graphical notations supported

by NITELIGHT comprise a SPARQL-compliant Visual Query Language (VQL), called vSPARQL, which covers all syntactic elements of the SPARQL specification. The complexity of this VQL makes the tool largely unsuitable for users who have no prior experience with SPARQL; although this does not preclude the use of the tool in contexts where users are attempting to familiarize themselves with SPARQL-related capabilities.

Another one is the VIQUEN [15] is a graphical tool for semantic query construction, execution and visualization that is based on the IML data flow graph transformation language for manipulating RDF data. VIQUEN consists of three main parts. The

first part is a query-building environment in which high-level query operations are expressed by individual graphical components that are then chained together to represent the entire query. The second part is the execution environment, which presents the IML query that has been compiled from the graphical query components. In this environment, the user may examine and execute the compiled query, and obtain the resulting RDF data. The third part of VIQUEN consists of a visualization environment which depicts the query results as a graph consisting of nodes and edges. Another one is the Konduit VQB [16] is a tool for building visual workflows for RDF data within Nepomuk-KDE, allowing for a flexible access to the local RDF data as well as mashing up with web-based data. It features a visual programming environment and allows for various manipulations (merging, filtering, mashing up, creating visual workflows, etc.) as well as executing different actions (executing scripts, automatizing emails, etc.) using the queried RDF data. A query builder is used to generate SPARQL queries for querying components which act as data sources in the RDF workflows, producing data that is made use of further in the workflow. Also, there is the SPARQLViz [17] is a query editor centered around graphical query composition and natural language processing in an RDF visualization interface. This tool is an extension for IsaViz, a visual interaction tool for RDF. SPARQLViz implements graphical query composition by using a graphical user interface for generating SPARQL queries. The user has to click through different menus to compose a query. The tool demonstrates that it is possible to cover a great part of the SPARQL syntax with a simple user interface. However, no graphical query language is implemented, which makes the understanding of the relationships between different query parts difficult. Another one is the Optique[18]

project aims at providing an end-to-end solution for scalable Ontology-Based Data Access to Big Data integration, where end-users will formulate queries based on a familiar conceptualization of the underlying domain, that is, over an ontology. From user queries the Optique platform will automatically generate appropriate queries over the underlying integrated data, optimize and execute them. The key components in the Optique platform are the ontology and mappings that provide the relationships between the ontology and the underlying data. The disadvantage of all the above approaches is that they still require SPARQL knowledge or other intuitive for users techniques.

Yet another approach is the use of natural language queries, which are automatically mapped to associations of triplets of the implemented ontology by a built-in dictionary and some inference mechanism, such as the Power Aqua system [19]. This approach, though not necessitating the user know ontology terms off-hand, inherits all the well-known polysemy of natural language, which sacrifices precision, and often results in even worse recall than the explicit use of ontology terms. Other natural language search systems, such as Swoogle [14] and SemSearch [20] do not interface to a triple store.

The second category includes proposals that work towards reducing the complexity of the Semantic Network itself in order to reduce the accompanying complexity of querying it. The Dublin Core Metadata Elements [21], VRA Core [22] and other metadata standards for example, reduce the network to flat relationships. Currently, the most recognized model, the Dublin Core, proposes a set of 15 properties: contributor, coverage, creator, date, description, format, identifier, language, publisher, relation, rights, source, subject, title and type. Similarly, the Consortium of Interchange of Museum Information has proposed the metadata elements: who, what, when, where ("4W") as a domain of independent relations to four kinds of entities (person, thing, time, place), a kind of "faceted search" [23]. In short(explicitly), whatever relation a thing may have to a person is an answer to the question "who", whatever relation a thing may have to a thing is an answer to the question "what", whatever relation a thing may have to a time is an answer to the question "when" and whatever relation a thing may have to a place is an answer to the question "where". The above works relatively well for metadata restricted to describing only the history of objects. Otherwise, the ambiguity, for instance between history and subject, proves overwhelming. Systems like Arti-

facts Canada [1] provide an advanced search facility based on this paradigm (plus a "how"). However, the use of so few metadata properties is insufficient to allow for reasoning with integrated metadata [24] or query refinement. Imprecision in the primary documentation, in particular the lack of a defined concept of events, leads to difficulty in integrating related data from different sources, as demonstrated in [25]. Furthermore, deficiencies in reasoning capability result in the procedure of creating "simple" metadata a not so simple task. If the metadata are to be created individually for all elements of the complex correlation, graphs characteristic of history, those characteristic of interesting works of arts and those of e-science data, may need to be manually repeated hundreds to tens of thousands of times; rendering the procedure ineffective and liable to error. Let's consider, for instance, the over twenty implementations of the "Thinker" by Rodin [2] and all related artefacts. Because precise knowledge in the documentation is sacrificed for rapid recall, these systems cannot be scaled up precision-wise.

An interesting intermediary between extensive semantic networks and faceted search (combining distinctive properties of the two aforementioned categories) is the Finnish CultureSampo [26]. Nine facets are used instead of four, including material, events, and object types among others. For each facet, rich term hierarchies of inclusion or subsumption are used, and multiple explicit and direct relationships among facets are provided for, such as fifty kinds of social agent-agent relationships. CultureSampo overcomes the liable to error search for suitable properties to a query, provides deductions from term hierarchies, class and property subsumption, selection of valid query parameters, and a faceted search. It even provides a natural language search. Yet, it omits other deductions such as property propagation along part-whole relations and derivation chains.

The third category provides a summation of studies of techniques that could be used alternatively in place of the selected SPARQL querying language in order to implement the abstract model that this study proposes. The Research Space Project is one of these techniques, which uses OWLIM rules to interpret the corresponding Fundamental Relationships' paths that this work proposes. In such a way, each Fundamental Relationship is mapped to a new single relationship that materializes in the repository. Subsequently, the query of one Fundamental Rela-

[1] http://www.pro.rcip-chin.gc.ca/artefact/index-eng.jsp
[2] http://en.wikipedia.org/wiki/The_Thinker

tionship is reduced to the query of one single property. Another alternative, would be to use semantic views in order to represent each one of the Fundamental Relationships, like the vSPARQL [27] based on the SPARQL querying language or the RVL [28] based on the RQL [29] querying language, the latter not yet implementable. These alternatives much resemble the way Fundamental Relationships are treated in a view-like manner, but still require technical knowledge from the user when constructing the views or the rules that represent the query. Rather, a "paths' language", a simple method for building the path that represents the query, is provided. Additionally, as regards the rules proposal where rules materialize on the provided metadata, the mechanism should also take into account updates and modifications to the rules. This process would necessitate extra time and effort to update the respective relationships in the database, and could hence prove quite exasperating when other rules are similarly affected. Such an approach also introduces the problem of overloading the database with extraneous relationships, especially as far as already massive databases are concerned; a problem which is not applicable for non-materialized views.

In this study, we propose a method that tries to combine the best of the aforementioned approaches, and which surpasses them.

Chapter 3

Path Expressions Language

As we previously seen in [42], the Fundamental Relationships (FRs) framework is designed to be handled by non-technicians. Such users are people who are aware of the schema upon which the metadata is built, but who may also have little technical expertise. The usage of already existing semantic query languages, like SPARQL, requires that the user understand its complex semantics. Furthermore, coding in SPARQL is neither intuitive, nor easy to follow, especially for non-computer scientists [30] [31].

Consequently, a simpler language to formulate the query statements that correspond to each FR had to be designed. A pre-requirement for this language was that it be easy to write and read. Since the users are accustomed to triplet constructions, given that they are familiar with RDF ontologies, triplet constructions proved to be an effective option. Within these triplet constructions, we differentiate between the subject and the predicate by using "$--$ ", and between the predicate and the object by using "$->$":

$$\text{Triple:=subject}--\{\text{predicate}\}->\text{object}$$

An example triplet follows in which we have marked the classes in bold, in order to make it more comprehensible. According to the naming conventions of CIDOC-CRM, a term beginning with "E" denotes reference to a class within the schema, while a term beginning with "P" denotes reference to a property of the schema.

E18.Physical_Thing $--$ {P46F.is_composed_of} $->$**E18.Physical_Thing**

More complex triplet constructions would allow for reflexivity and transitivity in the predicate, indicated by [0,n]; *0* denoting that the predicate may not occur in the triplet, while *n* indicates that it may occur indefinitely. Transitivity is essential for the exploitation of property propagation, e.g. through part/whole or derivation chains:

Triple:=subject−− {(predicate)[0,n]}− >object

Consider the following triplet, in which a thing is connected to the entirety of the parts it comprises, not only to those directly stated as parts of it. Essentially, we infer that parts of parts are also parts of the thing.

E18.Physical_Thing −− {(P46F.is_composed_of)[0,n]} − >**E18.Physical_Thing**

This is translated to:

E18.Physical_Thing *(when predicate occurs 0 times)*

or

E18.Physical_Thing −− P46F.is_composed_of − >**E18.Physical_Thing** −− P46F.is_composed_of − >**E18.Physical_Thing** ... P46F.is_composed_of − > **E18.Physical_Thing** *(when predicate occurs n times)*
where "..." indicates the infinite repetition of the construction "E18.Physical_Thing −− P46F.is_composed_of − >E18.Physical_Thing". In this way the Thing is interconnected with all its subparts. Similarly, transitivity is used for copies of Things. If we have an instance where thing2 is a copy of thing1, and thing3 is a copy of thing2, then we infer through transitivity that thing3 is also a copy of thing1.

More complex triplet constructions that allow disjunction at the predicate level are supported. In this way alternative predicates can be defined in the same triplet construction, instead of having to be defined in multiple triplets, thus necessitating the use of one predicate per triplet. In this way the paths become more compact and easier read and update. The pattern to be followed in this case is:

Triple:=subject−− {predicate OR predicate ...} − >object

When the pattern includes reflexivity and transitivity in the predicates, the pattern to be followed is:

Triple :=subject−− {predicate[0,n] OR predicate[0,n] ...}− >object

For example, consider the following triplet:

E70.Thing −− {P92B.was_brought_into_existence_by OR P16B.was_used_for } − >**E7.Activity**

This triple indicates that we can move from the domain class 'E70.Thing' to the range class 'E7.Activity' in two ways (i.e. predicates): the "P92B.was_brought_into_existence_by" OR the "P16B.was_used_for". Instead of having to write two different triplets, the user is able to combine predicates that share the same domain and range. The remaining cases (which combine predicates that share the same domain but not the same range class) are to be described later in this section.

Patterns like the above can be combined to form paths. Paths are sequences of triplets. In their simplest form:

Path :=Triple:{Triple:{Triple:{Triple...}}}

The symbol ":" indicates that the current triplet (the triple before the ":") is followed by the path that is enclosed in the following opening-closing brackets "{" and "}". This syntax allows for any triplet to be followed by another triplet construction in the path. However, in order to ensure the semantic succinctness of the path, the object of the triplet before ":" must bear a sub-class relation (either way) to the subject of the triplet after ":". The following path is exemplary of this:

E24.Physical_Man-Made_Thing −− {P128F.carries} − >**E73.Information_Object**:

{**E73.Information_Object** −− {P67F.refers_to} − >**E70.Thing**}

In this path it is evident that because our goal is to retrieve things that refer to other things, we have a path beginning with the FC Thing and ending with the

FC Thing. In the middle, by employing intermediate classes, we can formulate a path that will lead to the desired result. Therefore in the example, we transition to the information objects carried by the thing in the domain and then to the things to which these information objects refer. To build this path, the object of the first triplet is conjoined with the subject of the second one, which can either belong to the same class (as in this example), or command a sub class relationship to it.

In the next example, we demonstrate an instance where the second subject of a series constitutes a super-class of the previous object. Here, a thing is created in an activity event (E7.Activity) and this activity event may form part of broader events (E5.Event) which may or may not be activities. Despite the fact that an E7.Activity is a E5.Event and the user could use an E5.Event instead of an E7.Activity, it is conceivable to write it in the way indicated in the following example:

E28.Conceptual_Object−− {P94B.was_created_by} − >**E7.Activity**:
{**E5.Event**−− {(P9B.forms_part_of)[0,n]} − >**E5.Event** }

More complex paths would also allow for splits in the path. Our path expressions provide for branching at the triplet level by employing the OR operator:

Path :=Triple:{ Triple:{Triple:{Triple...}} OR Triple:{Triple:{Triple...}} }

Up until this point two usages of the OR operator: among predicates and among triplets, have been referenced. OR among predicates can refer either to predicates that share the same domain and range classes, or to OR among triplets. This, however, leads to verbosity, because *OR among predicates* is the simplest of two ways to express the same thing. This verbosity can prove inconvenient because when the user wishes to change something he has composed he is obliged to change other parts of the path as well. The purpose of path patterns which contain OR at triplet level is to branch predicates which share the same domain class but not the same range class. This explains why in such cases we can not use OR among predicates.

As mentioned earlier, branching plays a significant role in our model implementation. Each Fundamental Relationship is mapped to different paths over the CIDOC-CRM schema. Although, these concatenated paths form the path of an

FR, begin from the same class and end in the same class, they may follow other routes using another property sequence. Essentially, the disparate paths followed are the different interpretations a Fundamental Relationship confers. For example, a Thing from Place may be interpreted as either a Thing located in Place OR a Thing created within Place. Sub-path is the name given to the different parts followed in order for the total FR path to be composed. Sub-paths are of vital importance when composing specializations for the FRs, especially when it is necessary to check if a path can be defined as sub-relationship of an FR.

An example is presented in the path which follows, where classes have been marked in bold. So as to identify things that are from a place, we include both those that are located in that place and those that were created at the place, and the things that are part of things that are located or were created at that place. To this end, we can either use the direct relationship P53F.has_forme_ or_current_location that leads to the place of the Thing, or we can retrieve the Place of the creation event of the Thing. Or in predicates may not be used because the P53F.has_former_or_current_location and the P94B.was_created_by do not share the same range class (E53.Place for the first, E5.Event for the second). Different 'triplet branches' must be created, and for this reason the aforementioned syntax is employed. As noted earlier, an issue was identified, regarding the linking of successive triplets. Here, the subject E24.Physical_Man-Made_Thing of the second triplet is a subclass of the object E70.Thing of the first triplet. Consequently, it is valid for it to be employed to continue the path. Hence, the path the user composes:

E70.Thing −− P46B.forms_part_of − >**E70.Thing**:
{**E24.Physical_Man-Made_Thing** −− P53F.has_former_or_current_location − >**E53.Place**
OR
E70.Thing −− P94B.was_created_by − >**E5.Event**:
{**E5.Event** −− P7F.took_place_at − >**E53.Place** }
}

This path can be split into two sub-paths, as presented below. This split is necessary in order to break a path into its constituent parts, so as to identify the

different interpretations that are given to an FR.

E70.Thing −− P46B.forms_part_of − >**E70.Thing**:
{**E70.Thing** −− P53F.has_former_or_current_location − >**E53.Place**}

and

E70.Thing −− P46B.forms_part_of − >**E70.Thing**:
{**E70.Thing** −− P94B.was_created_by − >**E5.Event**:
{**E5.Event** −− P7F.took_place_at − >**E53.Place** }
}

This introductory description has been presented in a simplified format of the paths' language we have created. In order to provide a more formal description of the language, the definition of the context-free grammar[1] is provided in the next section, where it is described in Extended Backus–Naur Form (EBNF)[2] notation.

3.1 Syntax

The formal structure of the syntax necessary in order to create path queries in the generated language is provided in this section as is the definition of a context free grammar. That is, a grammar in which production rules must be adhered to so that correct path queries can be composed.

DEFINITION 1:
An ontology schema (RDF schema) O is defined as the finite set:

$$O = (C, P)$$

where C is the set of classes and P is the set of properties of the schema. Classes are the nodes in the RDF graph which comprise the RDF schema, and Properties are the arcs which link the nodes. Essentially, an RDF graph is a set of RDF triplets, while a triplet consist of three parts:

[1] http://en.wikipedia.org/wiki/Context-free_grammar
[2] http://en.wikipedia.org/wiki/Ebnf

1. subject: a node (class) of the RDF graph
2. predicate: an arc (property) of the RDF graph
3. object: a node (class) of the RDF graph

DEFINITION 2:
$O = (O_1, O_2, \ldots, O_n), n \geq 1$ is the set of ontology schemata covered, where $O_n = (C_n, P_n)$ is defined as per Definition 1 (above). The set of the ontology schemata O can thus be re-defined as a set comprised of the set of all classes and in turn as the set of all properties within the individual schemata:

$$O = (C, P), where\ C = C_1 \cup C_2 \cup \cdots \cup C_n\ and\ P = P_1 \cup P_2 \cup \cdots \cup P_n$$

A context free grammar G is defined as the quadruplet:

$$G = (V, \Sigma, R, S)$$

where

V = {path, triple, predicate_expr, class, predicate, predicate_trans }, is the set of variables used in the grammar

Σ = { ‘c’ , ‘p’ , ‘:’ , ‘−−’ , ‘− >’ , ‘{’ , ‘}’ , ‘(’ , ‘)’ , ‘[0,n]’ , ‘OR’ }, are the symbols used in the grammar where $c \in C$ and $p \in P, C, P \in O$.

R is a finite relation from $VtoV \cup \Sigma(V \xrightarrow{R} V \cup \Sigma)$, represented by the rule set defined later in this section

S $= path \in V$, is the starting symbol

The set R consists of the following production rules:

path = triple (1)
path = triple, ‘:’ , ‘{’ , path, ‘}’ (2)
path = triple , ‘:’ , ‘{’ , path , ‘ OR ’ , path , ‘}’ (3)
triple = class , ‘−−’ , predicate_expr , ‘− >’ , class (4)
predicate_expr = ‘{’ , predicate | predicate , ‘ OR ’ , predicate , ‘}’ (5)
predicate = ‘p’ | predicate_trans (6)
predicate_trans = ‘(’ , ‘p’ , ‘)’ , ‘[0,n]’ (7)

class = ‘c’ (8)

DEFINITION 3:
We define paths’ language as the context free language of the grammar G. Presented is the set:

$$L(G) = \{w \in \Sigma^* : S^* \Rightarrow w\}$$

where the symbol $\overset{*}{\Rightarrow}$ indicates the repetitive rule application, as stipulated by the next formula:

$$\forall u, v \in (V \cup \Sigma)^*, u \overset{*}{\Rightarrow} v \text{ if } u_1, u_2, \ldots u_k \in (V \cup \Sigma)^*, k \geq 0$$
$$\text{such that } u \Rightarrow u_1 \Rightarrow u_2 \ldots \Rightarrow u_k$$

This definition indicates that by repetitively employing the given syntactical rules, we begin from a start symbol and end with a terminal symbol. Hence, the user can read or compose path queries using the class and property terms of the schema as well as the various terminal symbols defined in Σ.

Presented below is an example of grammar defined composition of path queries starting with the start symbol ‘path’ and using the 3rd rule:

path := triple ‘:’ ‘{’ path ‘OR’ path ‘}’ $\overset{(1)}{\Rightarrow}$
path := triple ‘:’ ‘{’ triple ‘OR’ path ‘}’ $\overset{(2),(4),(8)}{\Longrightarrow}$
path := ‘E70.Thing’ ‘−−’ predicate_expr ‘− >’ ‘E70.Thing’ ‘:’ ‘{’ ‘E24.Physical_Man-Made_Thing’ ‘−−’ predicate_expr ‘− >’ E53.Place ‘OR’ path‘}’ ‘}’ $\overset{(4),(5),(6),(7),(8)}{\Longrightarrow}$
path := ‘E70.Thing’ ‘−−’ ‘{’ ‘(P46B.forms_part_of)[0,n]’ ‘}’ ‘− >’ ‘E70.Thing’ ‘:’ ‘{’ ‘E24.Physical_Man-Made_Thing’ ‘−−’ ‘{’ ‘P53F.has_former_or_current_location’ ‘}’ ‘− >’ ‘E53.Place’ ‘OR’ ‘E70.Thing’ ‘−−’ ‘{’ ‘P94B.was_created_by’ ‘}’ ‘− >’ ‘E5.Event’ ‘:’ ‘{’ ‘E5.Event’ ‘−−’ ‘{’ ‘P7F.took_place_at’ ‘}’ ‘− >’ ‘E53.Place’ ‘}’ ‘}’

Consequently, by repetitively applying the aforementioned rules, we have composed a path query that essentially consists only of terminal symbols.

3.2 Expressive Power - Semantics

By expressive power of the paths' language, we mean the set of all paths expressible in that language, whereby the respective definition for querying language is adapted, as in [32]. In this section, the expressive power of our language is defined, where the available operators and other existing features that enhance the expressivity of the language is displayed. In addition to what the language can do, we also state what the language as yet cannot do, thereby illustrating existing limitations.

Explicit operators for union ('OR') among paths and among predicates and join (':') between successive triplets are provided for in our language. The possibility to define reflexive and transitive properties ('[0,n]') are also provided for. Each of these is examined in the following:

- **Union among predicates**

Union among predicates is a useful operation that enables the user to construct compact path queries, since the ability to add more than one predicate in a triplet, negates the obligation to compose multiple triplets.

Consider two predicates: $P_1, P_2 \in P\_expr$, where $P\_expr = P \cup P_r$ and P_r is the set of reflexive and transitive properties defined in the format provided by the grammar rules: $(P)[0, n]$. The union operation among the two predicates is:

$$\mathrm{P_1\ UNION\ P_2} = \{\mathrm{P_1\ OR\ P_2}\}$$

Semantics: D_1=dom(P_1), D_2=dom(P_2), R_1=ran(P_1) and R_2=ran(P_2) where dom(P) represents the domain class of P and ran(P) represents the range class of P. Also sub(D_1) is the sub-class set of D_1, sub(D_2) is the sub-class set of D_2, sub(R_1) is the sub-class set of R_1 and sub(R_2) is the sub-class set of R_2. Finally, $D \in$ sub(D). In order to ensure the valid usage of OR among predicates, it is required that:

- $D_1 \in \mathrm{sub}(D_2)$ || $D_2 \in \mathrm{sub}(D_1)$
- $R_1 \in \mathrm{sub}(R_2)$ || $R_2 \in \mathrm{sub}(R_1)$

These constraints ensure the validity of the triplet, since the subject and object will necessarily be the same for all predicates in the pattern.

- **Union among paths**

This feature enables the union of two or more paths. For example, the two paths: path_1, $\text{path}_2 \in V \cup \Sigma$. Then:

$$\text{path}_1 \text{ UNION path}_2 = \{\text{path}_1 \text{ OR path}_2\}$$

Semantics: path_1 and path_2 may be paths consisting of more paths. Object_1 is defined as the object of the final triplet of path_1, and as object_2 of path_2 accordingly. As before, $\text{sub}(\text{object}_1)$ is defined as the sub-class set of object_1, and $\text{sub}(\text{object}_2)$ as the sub-class set of object_2. The following constraint must be respected in order to ensure the valid use of the OR operator among the two paths:

- $\text{object}_1 \in \text{sub}(\text{object}_2) \mid\mid \text{object}_2 \in \text{sub}(\text{object}_1)$

This constraint requires that the two paths end up in the same "category", so that the path's final object can be either the super or sub class of the other.

- **Connection among triples**

Connection among triplets is utilized so as to generate sequences of triplets which form a path. Take two triplets: $t_1, t_2 \in V \cup \Sigma$. The connection of the two triplets is:

t_1 JOIN t_2 = t_1:{t_2} Semantics: Take object_1 as the object of triplet t_1 and subject_2 as the subject of the triple t_2. As before take $\text{sub}(\text{object}_1)$ as the sub-class set of object_1 and $\text{sub}(\text{subject}_2)$ as the sub-class set of subject_2 So as to ensure the continuity of the created path, we require that:

- $\text{object}_1 \in \text{sub}(\text{subject}_2\ ) \mid\mid \text{subject}_2 \in \text{sub}(\text{object}_1)$

This constraint ensures that triple_2 is a natural "follow on" of triple_1.

- **Reflexive and transitive properties**

The expressivity of the language is facilitated by including except for simple, transitive and reflexive predicates. Let property p $\in$ P. Its reflexive and transitive form: (p)[0,n]. Semantics: Let t=(subject, (p)[0,n], object) be a triplet which incorporates the reflexive and transitive property p. So as for property p to be defined as transitive, its domain and range must be of the same class. Hence:

- dom(p)=ran(p)

The transitivity of the property can not be constrained to a certain level; it extends as far as the metadata dictates and this defines *n* in the pattern "[0,n]". *0* represents reflexivity, and as such, the subject of t coincides with the object of t. The triplet t is thus interpreted as

1. subject
2. (subject, p, object)

Take for example the following path containing a reflexive and transitive triplet:

$$\text{Path=a} -- \{(\text{p})[0,\text{n}]\} -> \text{b:}\{\text{c} -- \{\text{l}\} -> \text{d}\}$$

This translates to

$$\text{Path=} \{\text{c} -- \text{l} -> \text{d OR a} -- \text{p} -> \text{b:}\{\text{c} -- \text{l} -> \text{d}\}\}$$

Upon examining the expressive potential of our language, we have concluded that certain operations have not yet been defined.

- **Negation**

Negation (NOT) has not as yet been defined in our language. No provision is made for:

- a negative triple: NOT {a −− p − > b}
- a negative path: NOT {{a −− p − > b:{c −− e − > f}}}
- a negative class: NOT {a}

- **Conjunction among paths**

Conjunction in our language is defined as connecting sequential triples through the use of the symbol ":". Although the semantics describing the use of the operator have been explained as "UNION among paths" in the previous lines, they have not as yet been defined as connecting paths. The following is an example which uses the AND operator:

```
E70.Thing -- {P46B.forms_part_of} -> E70.Thing: {
    E70.Thing -- {P53F.has_former_or_current_location} -> E53.Place
    AND
    E70.Thing -- {P94B.was_created_by} -> E65.Creation: {
        E65.Creation--{P7F.took_place_at}-> E53.Place
        }
    }
}
```

Chapter 4

Technical Approach

This chapter is going to provide the technical details on the implementation of our system. The implementation can be split into two main phases and a few sub phases.

1. Design and implementation of our proposed querying method. To complete this phase we have to separate it into three distinct sub phases:

    - the design of a new query language appropriate for the needs of our proposal. This language should be capable of being used for the formulation of the paths of the Fundamental Relationships and has to be both rich enough to express the needs of the users but also simple enough to be used by non-experts like curators.
    - design and implementation of a text based low level tool to aid the user in the process of constructing correct paths, «translating them to SPARQL and providing him with a number of other facilities»
    - implementation of the proposed model in the concept of the CIDOC-CRM schema.

2. Design and implementation of a graphical interface based tool, whose purpose is to enable the user to synthesize valid path expressions queries faster and more accurately. This graphical tool aims to eliminate any possible user lexical errors and confusions by using proposed techniques. The implementation of this phase is split into three sub phases:

- the design and development of an appropriate parsing tool, which aims to parse the paths of the Fundamental Relationships and convert them from text form to a visual graphic form
- the design and implementation of the graphical user interface for the end user, which enables the user to construct correct paths with the aid of visual techniques.
- the design and development of an extracting tool capable of exporting the created paths into different forms.

During the design of our system we have noted that it appeals to two kinds of users:

- the **administrative user**. Such users are able to use our path language and the appropriate configuration tools in order to facilitate the materialization of the proposed model in the concepts of the underlying schema - in our case the CIDOC-CRM schema.
- the **end-user**. These users are the ones who, though ignorant of the technologies described in this section, will be able to perform queries on the metadata using the proposed framework. However, they must knowledgeable of the proposed simplified querying model.

4.1 Parser Of Language

To implement a language, we have to build an application that reads sentences and reacts appropriately to the phrases and input symbols it discovers. A language is a set of valid sentences, a sentence is made up of phrases, and a phrase is made up of sub-phrases and vocabulary symbols. Broadly speaking, if an application computes or "executes" sentences, we call that application an interpreter. To react appropriately, the interpreter or translator has to recognize all of the valid sentences, phrases, and sub-phrases of a particular language. Recognizing a phrase means we can identify its various components and can differentiate it from other phrases [33] [34].

On our first attempts to develop the parser for our language, we used the Java Standard Edition library in combination with *RegEx* (Regular Expression) techniques. A regular expression is a sequence of characters that forms a search pattern,

mainly for use in pattern matching with strings, or string matching, i.e."find and replace"-like operations. This combination is very helpful and useful because it provides us with full access to the code of the parser. The parser also quickly interprets with the given string because of the purity of the code.

Despite the difficulties in development, arising from the plethora of the cases that the parser had to take into consideration, this version of the parser worked perfectly in the initial stages. The efficiency of our parser started to decline when the length and complexity of the path queries increased. This reduction in efficiency, which was caused by the huge number of nested paths, has to be isolated and analysed.

In an attempt to overcome the above problems, we focused on refining our code by reducing the complexity of our algorithms. This procedure helped us to solve many problems. However, this approach proved problematic with extended path queries. At that point, we had to solve problems which were not about the structure and the complexity of our algorithms, but concerned the Java Virtual Machine[1] (JVM) and how JVM manages system memory.

We considered these kinds of problems beyond our research field and we had no interest in solving them. Therefore, we conducted a survey in the field of new language recognition (the design and implementation of a language parser and lexical analyser) and we decided to use tools specialized in language recognition.

Parsers or *syntax analysers* are applications which recognize languages. The rules governing language structure are known as Syntax, which is used to construct a grammar which specifies path expressions language syntax. A grammar is simply a collection of rules that express the structure of a phrase. This kind of application translates grammars into parsers. It appears remarkably similar to what an experienced programmer might himself build manually. Grammars in turn follow the syntax of a language designed specifically for specifying other languages: tools' meta-language.

Parsing is much easier if it is separated into two similar but distinct tasks or stages. The separate stages simulate the way English text is understood by our brains. We do not read a clause character for character but rather perceive it as a stream of words. Subconsciously, the human brain collates character sequences into words and seeks the meaning of these words before recognizing grammatical

[1] http://docs.oracle.com/javase/7/docs/technotes/guides/vm/

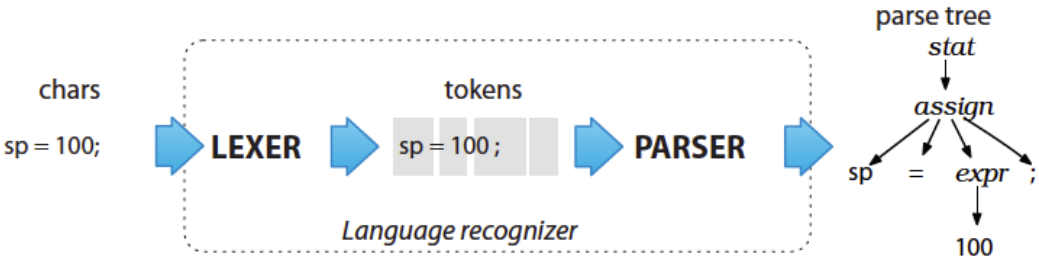


Figure 4.1: The basic data flow of a language recognizer

structure. This process is more obvious when we are interpreting Morse code because we convert the dots and dashes into characters prior to reading the message.

The process of grouping characters into words or symbols (*tokens*) is called *lexical analysis* or simply *tokenizing*. A program that tokenizes the input is known as a lexer. The lexer is in turn able to group related tokens into token classes, or token types, such as INT (integers), ID (identifiers), FLOAT (floating-point numbers), and so on. The lexer groups vocabulary symbols into types when the parser is concerned only about the type, not the individual symbols. Tokens are comprised of at least two pieces of information: the token type (identifying the lexical structure) and the text the lexer associates with that specific token.

The second stage is the actual parser which utilizes these tokens to recognize the sentence structure, in this case an assignment statement. By default, generated parsers build a data structure called a parse tree or syntax tree that records how the parser recognized the structure of the input sentence and its component phrases. The diagram in Figure 4.1 illustrates the basic data flow of a language recognizer.

The interior nodes of the parse tree are phrase names that group and identify their children. The root node is the most abstract phrase name, in this case stat (short for "statement"). The leaves of a parse tree are always the input tokens. Sentences, linear sequences of symbols, are really just serializations of parse trees we understand intuitively in hardware. In order to transfer an idea to someone, we need to create the same parse tree in their heads using a word stream.

By producing a parse tree, a parser delivers a convenient data structure to the rest of the application that contains complete information about how the parser grouped the symbols into phrases. It is easy for programmers to understand and process trees in consequential steps. The parser can even generate parse trees automatically.

4.1.1 Used Library

We decided to use an open source library called "ANother Tool for Language Recognition" (ANTLR) to implement our language. This library has specialised in the field since 1992[2] and seems to fit our requirements perfectly. The only disadvantage of choosing to move to an upper level language to implement our parser was that we had to learn and become familiar with the syntax of ANTLR. Despite our initial concerns, the structure of the ANTLR scripting language helped us to overcome the problem quickly.

The ANTLR tool generates recursive-descent parsers from grammar rules such as those presented in Figure 4.1. Recursive-descent parsers are essentially a collection of recursive methods, one per rule. The descent term refers to the fact that parsing begins at the root of a parse tree and proceeds toward the leaves (tokens). The first rule we apply, the start symbol, becomes the root of the parse tree. Another term for this kind of parsing is top-down parsing; recursive-descent parsers are just one kind of top-down parser implementation.

In order to gain a better understanding of what recursive-descent parsers look like, presented below is the (slightly cleaned up) method that ANTLR generates for rule assignment:

```
// assign : ID '=' expr ';' ;
void assign() { // method generated from rule assign
  match(ID); // compare ID to current input symbol then consume
  match('=');
  expr(); // match an expression by calling expr()
  match(';');
}
```

Among the many advantages to recursive-descent parsers is that the call graph which is drawn by invoking methods stat(), assign(), and expr() mirrors the interior parse tree nodes. The calls to match() correspond to the parse tree leaves. To build a parse tree manually in a hand-built parser, we would insert "add new sub-tree root" operations at the beginning of each rule method and a corresponding "add new leaf node" operation match().

[2]initial release

Method assign() simply checks to ensure that all necessary tokens are present and in the correct order. When the parser enters assign(), it need not choose between more than one alternative. An alternative is one of the choices to the right of a rule definition. Method stat() makes a parsing decision or prediction by examining the next input token. Parsing decisions predict which alternative will be successful. The term lookahead token is simply the next input token. A lookahead token is any token that the parser checks before matching it. Often, the parser requires a number of lookahead tokens to predict which alternative will succeed. It may even need to take into account all tokens comprising the file from the current position until the end.

4.1.2 Logic Explanation

In this section, we will analyse the way in which we designed and developed the parser for the paths' language. Having already defined the syntax of the language (Chapter 3) and making use of the Antlr library, we began to develop the parser. In order to abide by the structure of Antlr, we had to translate the syntax of the paths' language into the lexers and rules of Antlr.

Initially we defined all the lexers that were to be used during the composition of the rules. Most of these were isolated symbols, as for example "{" or "(", while others were combinations of characters, as for example the definition of the lexer for "white spaces" including space, new line etc. The conjunctions which were to be used within the triplet, or between triplets, as well as the parser constrained word "OR" were also to be defined as lexers.

Having defined all the lexers, we began to define the parser rules. Initially, we defined the simpler forms, such as what would be defined as a word, what would be defined as weight, and what would be defined as Subject, Predicate and Object. Ultimately, the more complex forms were addressed, such as what would be defined a triplet and how the nesting of the triplet would be defined.

A triplet is recognized by the parser as either an isolated Subject, or a triplet in its entirety: Subject-Predicate-Object (Figure 4.2).

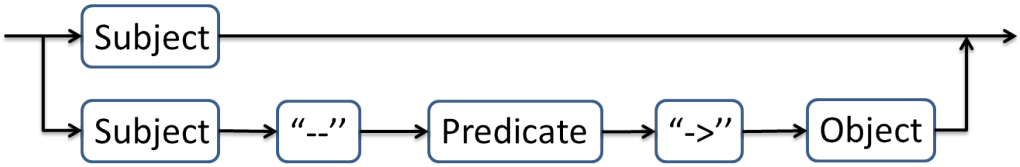


Figure 4.2: Triplet flowchart

Subsequently, since we had already defined what would be recognized as a triplet, we created the general form of the *expression* which defines the path query, as shown in Figure 4.3.

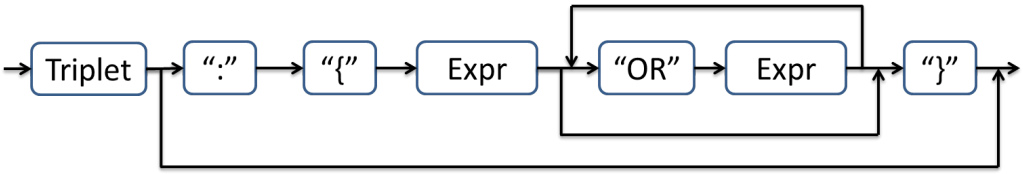


Figure 4.3: Expression flowchart

This flow chart describes how the parser recognizes the path query. As seen in the chart, the simplest structure which can be considered a path query is a triplet (and accordingly a single subject, as seen in Figure 4.2). Consequently, in an instance where the path query contains more than one triplet, a triplet connector (that is, ":") followed by a left hook, is provided for. At this point in the flow chart there is a variable "Expr" which represents the expression in its entirety. In this way, by recalling the expression itself, we ensure that the parser is not restricted in recognizing large and complex path queries. By exploiting the capabilities provided by the Antlr library, we managed to overcome any difficulties which had appeared during the development of the parser in native Java, arising from size of the path query. As a result, either nothing follows the variable "Expr", and the path query will contain a right hook, or, if it includes a disjunction, will have the reserved word "OR" followed by another "Expr" variable, and only then by the right hook. This combination of OR and Expr repeats itself according to the number of disjunctions.

By functioning in this way, the parser recognizes the paths' language. As a logical consequence, apart from being able to recognize path queries, the parser contains additional code, which involves both the management of recognized components, and the setting and recording of properties which describe these com-

ponents. Among the plethora of available properties are the order in which the triplets are conjoined, to which triplet the components belong, and whether they constitute the triplet's Subject, Predicate or Object.

Throughout the development stage, much time was devoted to the most detailed design and structural implementation of the parser, possible. This was required so that problems arising from the recognition of a large and complex path query could be avoided and so that the parser would be expandable, given that the paths' language is in its infancy and will continue to evolve. A characteristic example of this is the development of the graphical editor (to be analysed in the following section) which required the extension of the parser in order to convert text path queries to graph path queries.

4.2 Graphic Tool

Despite the usability of the 'Fundamental Relationship configuration' tool, we noticed that even though the user could compose paths faster, without much effort, he had to have at least a general idea of the schema in order to achieve the composition of fundamental relationship paths. The overview of the schema was needed because the Fundamental Relationship configuration tool did not provide support at the path composition stage, but returned useful information regarding syntax and logical errors after the compilation. Also, the user had to have minimum knowledge of, and be familiar with, writing in a computer language so as to avoid many syntactical errors related to the constraints of the language. Consequently, this disadvantage of our tool created a need for a more powerful and more user friendly configuration tool that overcame all that functional disadvantages of its predecessor.

Since our target group is non-expert users, we have decided to design and implement a Visual Query System (VQS), aiming at enlarging the communication bandwidth of our paths' language, using a wider range of communication media that rely as much as possible on the senses of the human being. To pursue this goal, we have developed a graphical editor which provides a visual query language counterpart to our paths' language, assisting in writing triple patterns and verifying their syntax through the CIDOC-CRM. This involves taking into consideration various perspectives like layout, readability and interaction design.

In this chapter, we will explain all the phases and steps followed during the process of designing and implementing our graphical interface based tool, whose purpose is to enable the user to synthesize valid path expressions queries faster and more accurately. This graphical tool aims to eliminate the possible user's lexical errors and confusions by using proposal techniques. As mentioned in the previous chapter, the implementation of this graphical tool was split into three sub phases:

- the design and development of an appropriate parsing tool, which aims to parse the paths of the Fundamental Relationships and convert them from text form to a visual graphic form
- the design and implementation of the graphical user interface for the end user, which enables the user to construct correct paths with the aid of visual techniques.
- the design and development of an extracting tool capable of exporting the created paths into different forms.

Because of the complexity of procedures, we had to make critical decisions about which libraries we were going to use. In the following sections we will explain all the decision making processes regarding the choice of libraries, how they were combined to give the expected result and, most importantly of all, which functionalities our program supports.

4.2.1 Description

In the first stage of development of the FRGE , we had to design and implement an import mechanism that gives the user the option to convert a text-based path of fundamental relationships to a graph. Therefore, when the user does not want to start from the scratch, he can import one of his previously edited path queries. We had already developed a parser, which made our suggested language functional, but we had to enrich it in order to support the conversion of the path from text to graph. In addition, we had to develop another parser, which supported the importation of paths saved in the original software file types. The files of this type are readable and editable only from the FRGE , contrary to the simple text version, which the non-graphical configuration tool uses to save the path queries.

Whether the user imports a fundamental relationship path query or he starts writing a graph path query from scratch, he will need an editor. In the second stage of the tool's development, we had to design and implement a graphical editor. The user interface should be easy to use, self explanatory and follow the general protocols of Human–Computer interaction (HCI). The FRGE provides an overview of declared paths, which allows the users to check the validity and the completeness of a query against the intended functionality. In addition, the editor allows users to easily modify, relax or restrict a query, with the overall purpose of alleviating the problem of errors. Such errors are misspelling the name of classes or properties during path writing, or forgetting the use of some predefined special characters, like path connection characters or the use of different kinds of brackets. All of these challenges were successfully met during the implementation of this stage.

All of these processes are based on a graph. This graph is an extended vector, which, in combination with other appropriately configured structures, like hash maps, keeps all the information of the paths. The most common tasks to be performed with graphs boil down to a handful of things:

- visualization
- (manual) creation
- editing
- automatic entity arrangement, i.e., automatic graph layout
- structural analysis

Visualization is the task of actually generating a visual representation of an existing graph. Note that this does not necessarily mean the creation of a static picture, but more generally describes forms of representation that could also allow for interaction. The items (manual) creation and editing both involve interacting with a graph, i.e., creating nodes, and connecting them with edges, or adding further elements to an existing graph, or removing them. Automatic entity arrangement is the creation of high-quality, easy-to-understand drawings of a graph to aid in communicating information more quickly. Structural analysis means solving advanced questions that relate to a graph structure, for instance, determining the shortest path between two nodes.

In the last stage of the development of the FRGE , we had to implement another mechanism responsible for the reverse processing of the first-stage parser. Informally, we will use the term *exporter* for this mechanism. The exporter has the ability to take the graph that the user has composed during the editing process, and convert it to any type the user wants. The exporter is capable of exporting the graph, either as an image (plenty of image types are supported), or as a text file of the native configuration tool file type. Practically, the exporter is nothing more than a parser, but instead of parsing text and giving it functionalities, it parses the drawn graph.

4.2.2 Technical Details-Prerequisites

The FRGE is a Java based component. We used the seventh version of Java for the development of the tool, which is the minimum acceptable version of Java that someone needs to run it. The fact that it is written in Java makes it independent of the operating system, meaning that the user has the ability to both process and transfer it to whichever computer he wishes.

The FRGE connects with a user-defined sesame repository and integrates with repository data. The graphical tool does not change anything in the repository data. In essence, the only interaction with the repositories is to read their data. The FRGE offers an embedded server in which the user can save and work their repositories.

The adding and removing of repositories can be achieved by means of any Internet browser because the server is accessible via the url of the local host. An essential pre-requisite in order for the user to operate in the embedded server is to have port 2525 open. Moreover, FRGE enables the user, if he so wishes , to work with repository data on any remote server. In this case, the user should know the url of the remote server as well as the name of the repository.

4.2.3 Used Library

For the development of the FRGE , a graphical library was necessary. The native graphical libraries of Java were not sufficient to meet the requirements of our project so we had to use an external library to satisfy them. There are currently a plethora of open source Java graphical libraries available, as are several commercial libraries.

After identifying and comparing these libraries, we decided to use the "yFiles for Java", a product of yWorks company[3].

yFile is an extensive Java class library that provides algorithms and components enabling the analysis, visualization, and the automatic layout of graphs, diagrams, and networks. This library is a good option for visualization, and also has a fairly good set of available graph algorithms, including several different mechanisms. It is highly stable and its complexity algorithms are of the quickest in the market. Other packages do not meet all of our needs. The main competitor of yFiles is the JGraph[4] . JGraph is open-source, and seems to get a lot of press from the open-source community. We have looked extensively at JGraph and have determined that it is not suitable for our needs. JGraph is oriented more towards design layout, like Visio, and does not meet our needs in terms of graph operations.

What is yFiles and how it works

yFiles is an extensive class library that provides algorithms and components for analyzing, viewing, and drawing graphs, diagrams, and networks. It is entirely written in the Java programming language and can be used to develop sophisticated applications to be deployed on any platform with a working Java Runtime Environment installation. In this section we will give an overview of the purposes the yFiles library serves. One of the main features of yFiles is the provision of sophisticated layout algorithms, which support the automatic generation of high-quality graph drawings.

yFiles Components

This section presents the different yFiles components and gives an overview of the packages and the functionality they provide. These yFiles library functionalities are divided into three parts:

- Basic, which serves as the "backbone" for the main part of the library
- Viewer for everything relating to user interaction
- Layout, which provides highly evolved automatic layout algorithms

[3]http://www.yworks.com/en/company.html
[4]http://www.jgraph.com/

yFiles Basic contains essential classes and data types for graph analysis tasks. It provides efficient implementations of advanced data types like graphs or priority queues. It furthermore makes a wide variety of graph and network algorithms available which form an indispensable tool-kit for any network analysis tasks.

yFiles Viewer builds upon yFiles Basic. It provides a powerful graph viewer component and other Swing-based GUI elements. The viewer component is showcased in the yEd graph editor application. Other notable features of yFiles Viewer are its support for diverse graph formats (e.g. GML, YGF, JPG, GIF, etc.), and its printing capabilities.

yFiles Layout builds upon yFiles Basic. It provides a perfect suite of graph layout algorithms, which offer unequalled opportunities. A multitude of layout styles like hierarchical, orthogonal, or circular are provided as easy-to-integrate components, which can be configured programmatically to suit most layout demands. Additionally, yFiles provides edge routing algorithms that make it possible to route edges into existing diagrams easily.

4.2.4 Functionality

In this section, we will explain all the functionalities of the FRGE . The functionalities will be fully described and we will cite figures or code snippets where it is considered necessary.

First view

In Figure 4.4 we see the initial environment the user comes across when starting FRGE . This environment is separated into four main parts. The first is the central part which constitutes the canvas onto which the user will work, compose, edit or process path queries. The remaining three parts function as auxiliaries and serve to facilitate the quickest and easiest interaction of the user with the FRGE . More specifically, the upper part consists of a toolbar, which contains various short-cuts as well as several general operations the editor offers.

The other two parts are found on the bottom part of the canvas and they too play an auxiliary role. On the left part, a navigational map is found: and below it various objects (nodes, connectors etc) appear, which the user can utilize in the composition of new triplets. These objects appear when the user enters

editing mode. On the other side, to the right of the canvas, another auxiliary panel exists which contains operations relating to the processing of the nodes, and more specifically contains lists through which the user can select the values of every node when entering or processing triplets. As with the previous panels, the content appears only in editing mode and more specifically according to the type of the node chosen (Subject/Object or Predicate). Analytically, the use and the functions these parts provide will follow.

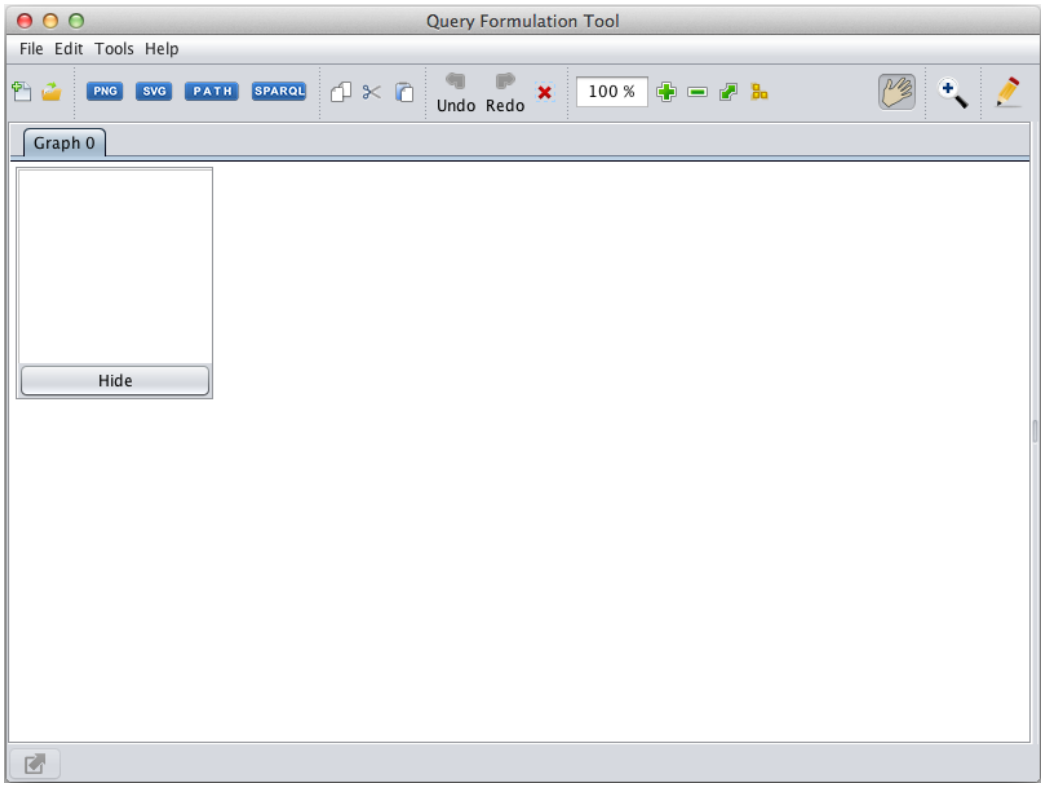


Figure 4.4: Graphical Configuration tool user interface

Background functionalities

Apart from the functions which are visible to the user and which he uses when interacting with the editor, it became necessary to construct various functions related to the core of the system in order to develop and to ensure the correct operation of FRGE .

Initially, it became necessary to embed an Apache Tomcat into the system. The reasons which dictated this were two: speed of communication and independence.

It was observed that with the use of an external server, the system slowed down considerably, and this was due to the speed of the connection with the server and the workload it bore. In order to ensure the processing of the path queries and the proper functioning of the proposal techniques, it became necessary for SPARQL queries to be executed in the repositories. Therefore, it became evident that if these repositories were to be found on a remote server, the speed of the system was adversely affected.

Apart from the problems associated with access speed and integration of repository data, the system became dependent upon the need for an Internet connection dictated by the use of an external server. By using an embedded server we negated this requirement and secured the independence of our system. This, together with the fact that our system is developed in Java, ensured a completely independent and ready for use application under any circumstances (plug and play).

Apart from creating path queries from scratch FRGE also had to support the import path queries from a file. No problems were experienced for path queries that had been created by the editor and had been saved in the editor's file form. However, problems began to arise when the import of path queries written in the paths' language had to be supported. Consequently, we considered it necessary to develop a parser to enable the paths' language to be understood by the editor, and to be converted to a graph so that the user could process it.

The parser which we had already created was used, and its capabilities extended, so that it could be functional and so that it could be integrated with FRGE . This improved version of the parser was now not only in a position to understand the paths' language, but was also capable of transforming the paths queries into graphs.

Having composed and processed the path queries, the user can save his work. Apart from this ability, FRGE provides an export option to various file types which can either be an image file or a text file especially transfigured for the paths' language or SPARQL query. For the successful conversion of path queries from graph form to text form, FRGE provides an export mechanism. The development process of this mechanism proved both demanding and complex. It was thus necessary to develop a complex algorithm which, apart from converting the graph, could also evaluate the logic of the path query. Essentially, it evaluated whether there were duplicate paths and removed them, thereby simplifying the path queries.

In the event that a user wishes to export the graph into a SPARQL query, the above mechanism goes one step further and converts the path query to a SPARQL query. In this SPARQL statement, the sought after variable is the $StartVar and with it any existing labels returned in the $Label variable. The parameter must be defined as an $Endvar variable.

The conversion of the user path query graph to a SPARQL query is performed by applying an algorithm that is designed to take the paths' language grammar into consideration. In general, connections among triplets (':') are translated to SPARQL joins ('.') while disjunctions ('OR') are translated to SPARQL disjunctions ('UNION'). The transliteration of a triplet into SPARQL involves the conversion of the triplet's domain and range classes into different variables in SPARQL. The predicate is used in the SPARQL only after placing the respective namespace prefix either "crm:" or "crmdig:" in front of it; in our case the CIDOC-CRM and CIDOC-CRMdig. In the interests of maintaining the continuity of the triplet chain, the variable used for the range in a triplet, will subsequently be used as the domain for the next triplet in the chain.

Nothing to type

One of the main functions of the FRGE is the use of proposal techniques which aid the user in the syntax and processing of path queries. The user need not be able (nor is he permitted) to manually define or change any value pertaining to the Subject, Predicate or Object. Each time he wishes to make a change, FRGE prompts the user to make a selection only within specific lists of available options, thereby eliminating the chance of his making any logical errors. These lists vary from case to case and their content is dynamically selectable. More specifically, these values arise from SPARQL queries to repository data which the editor executes, in search of all available values that the specific node can assume. This results in the user being able to compose logically correct path queries from the beginning, without needing to verify them at the end.

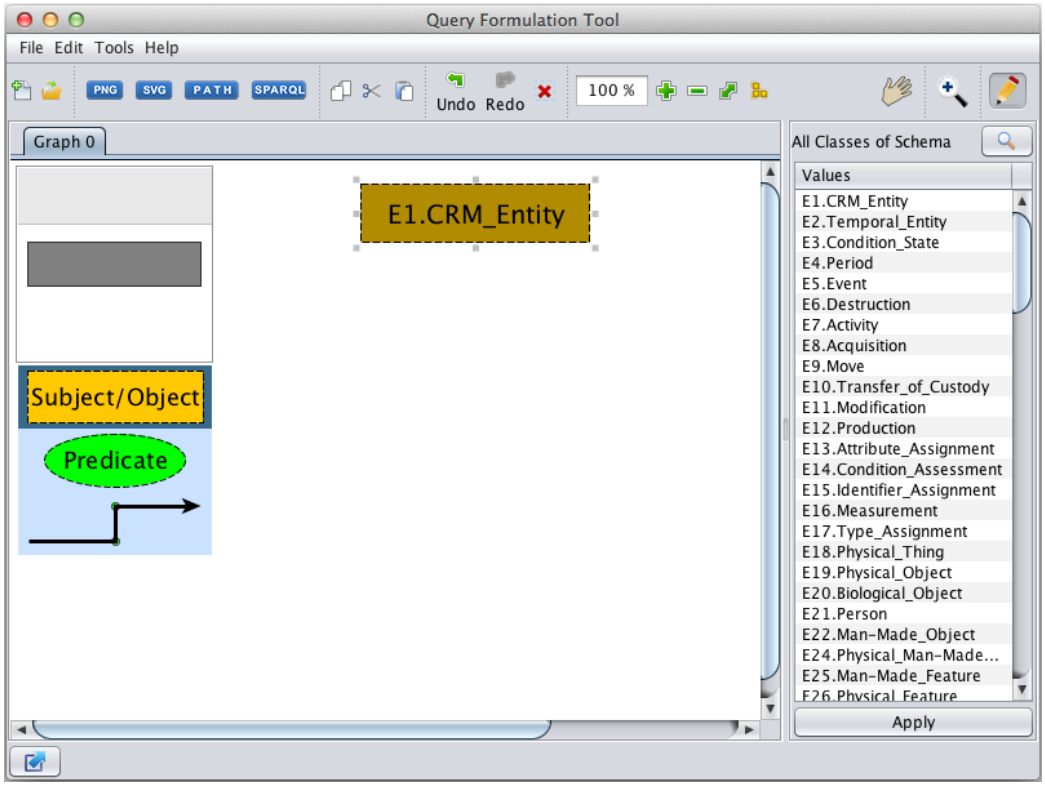


Figure 4.5: Nodes values set by selection, not by manually typing by user

Auto triplet completeness

FRGE , apart from allowing the user to select from suggested values, provides a logical organization to the composition of triplets. When the user wants to establish a new connection inside a triplet (Figure 4.6(a)), our tool automatically proposes the first possible value for the Predicate, and automatically creates and gives a value for the Object of that triplet (Figure 4.6(b)). Of course, the user can later change these values (as we have explained in the previous section)

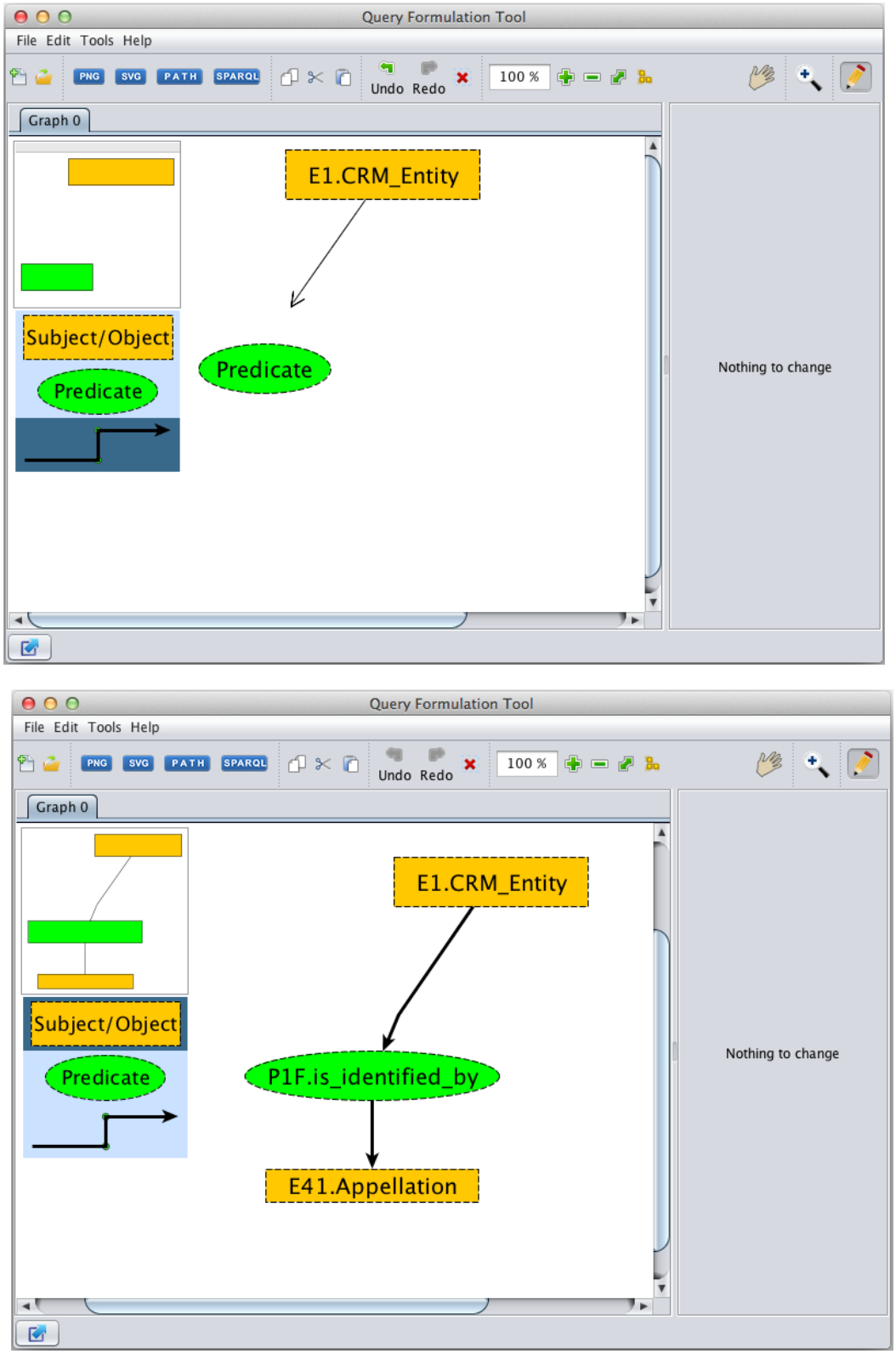
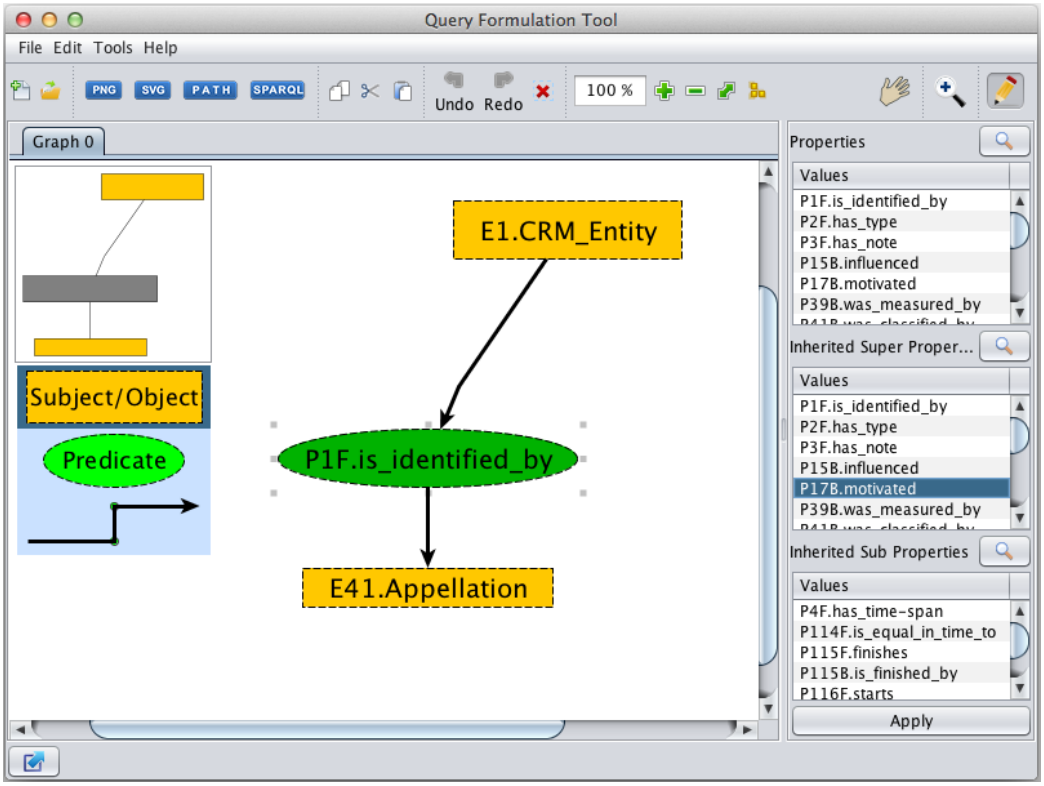


Figure 4.6: Complete automatically the triplet

Auto change Object values

The Object value is directly dependent upon the value of the Predicate in each triplet. As seen in the previous section, FRGE creates and provides a value for the Object node from the creation and definition of the value of the Predicate node. When the user chooses to change the value of the Predicate, the correspondent value of the Object node is given automatically.



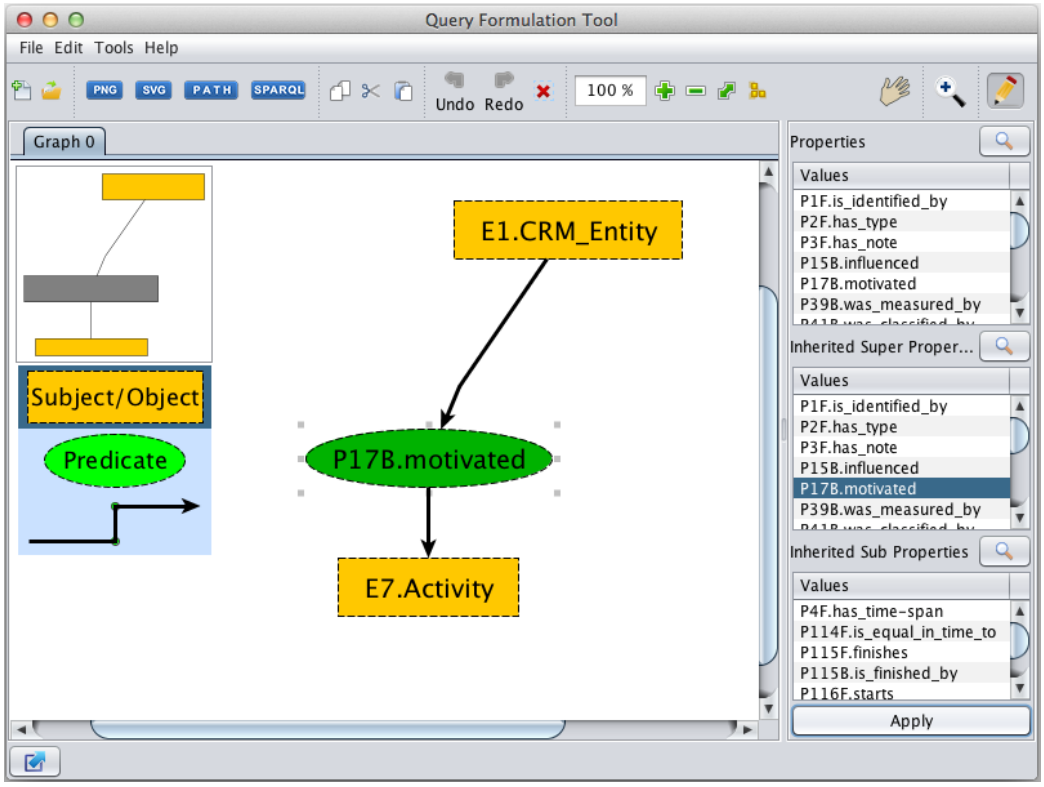


Figure 4.7: Auto change Object values

Auto path completeness

Another function of FRGE is the automatic creation of nodes missing from the created paths. When the user wants to establish a new connection between the Object of triplet A and the Predicate of triplet B, FRGE automatically creates the Subject and Object of triplet B and assigns to them the first possible value. The user can change these values afterwards if he so wishes.

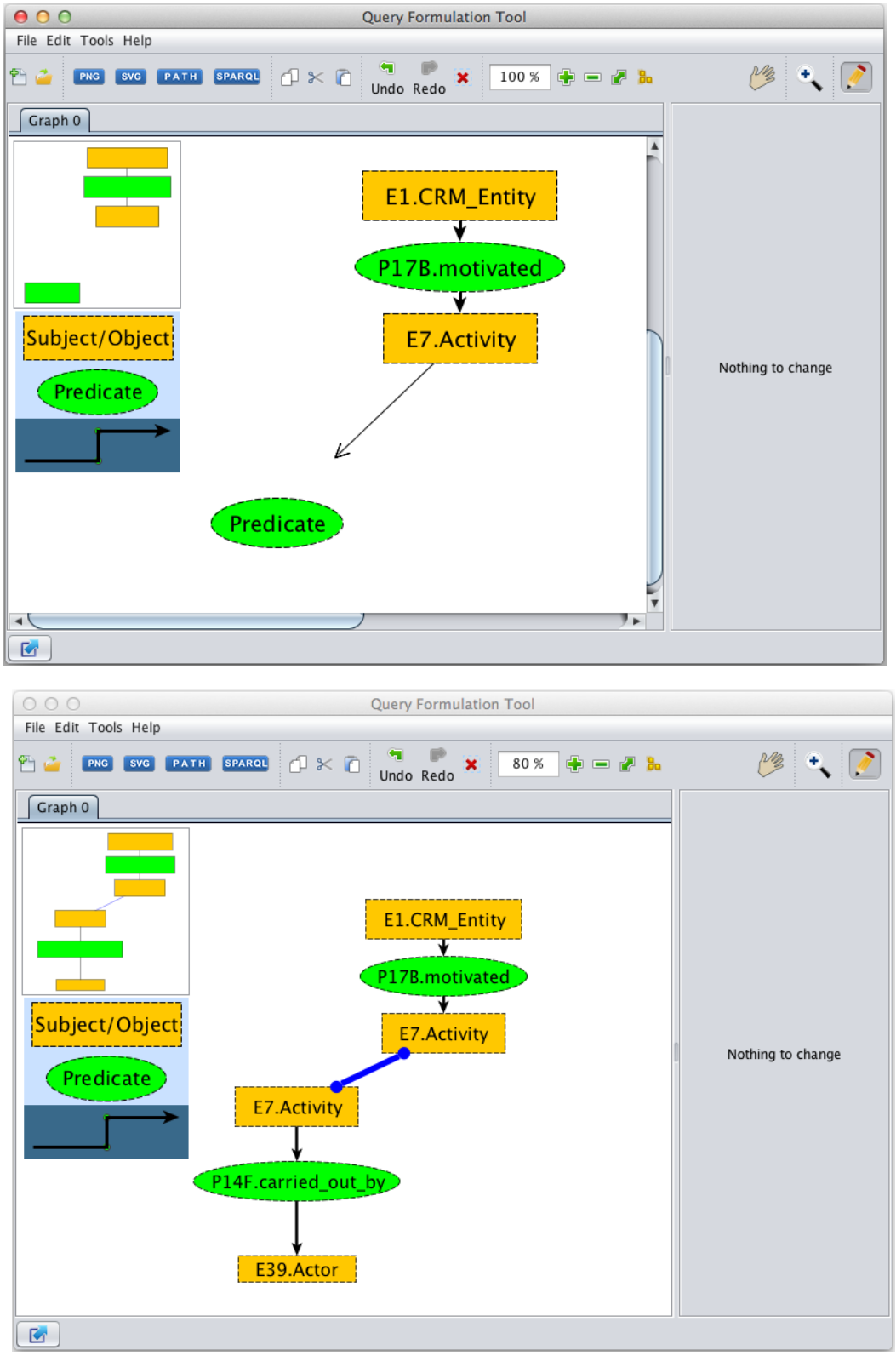
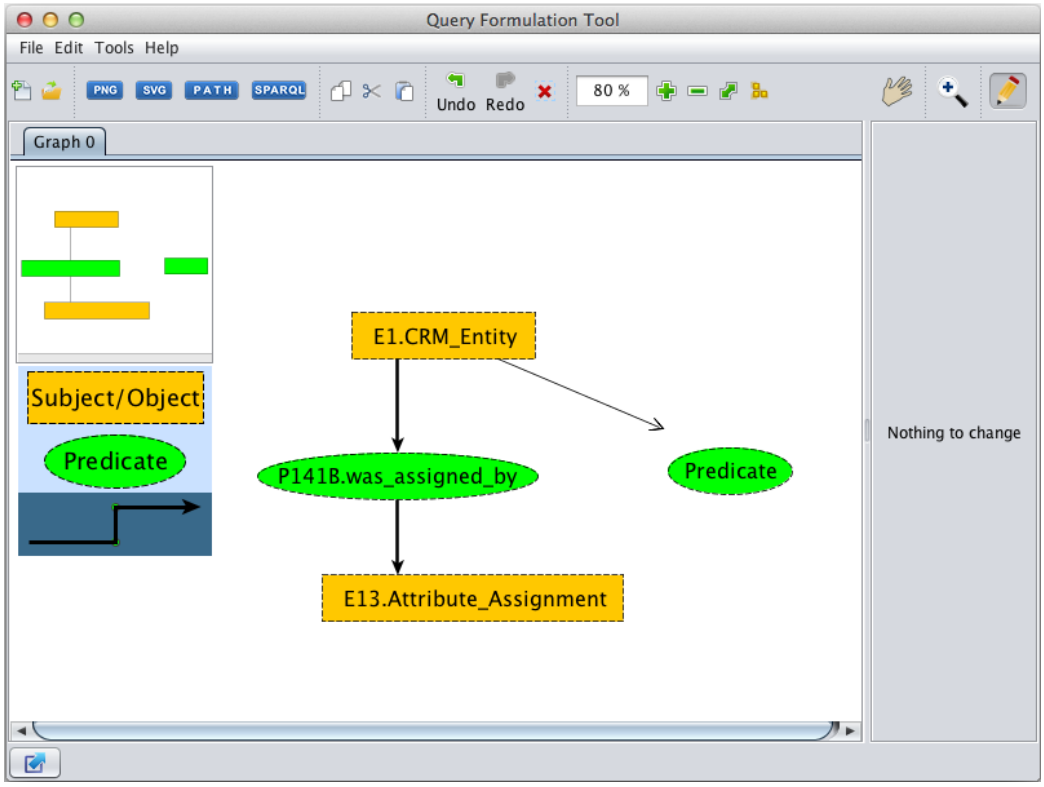


Figure 4.8: Auto path complete

Logic in auto complete

Another feature of FRGE is that apart from aiding the user in composing correct path queries, it guaranties that these paths are logically correct and as simple as possible. This results in the path queries being quicker and more efficient. In order to achieve this, FRGE checks if the auto created node of the Object already exists in the triplet, and if so, the editor connects it to the existing Object node. Thus, the graph remains "clean" and the user can easily interact with it.



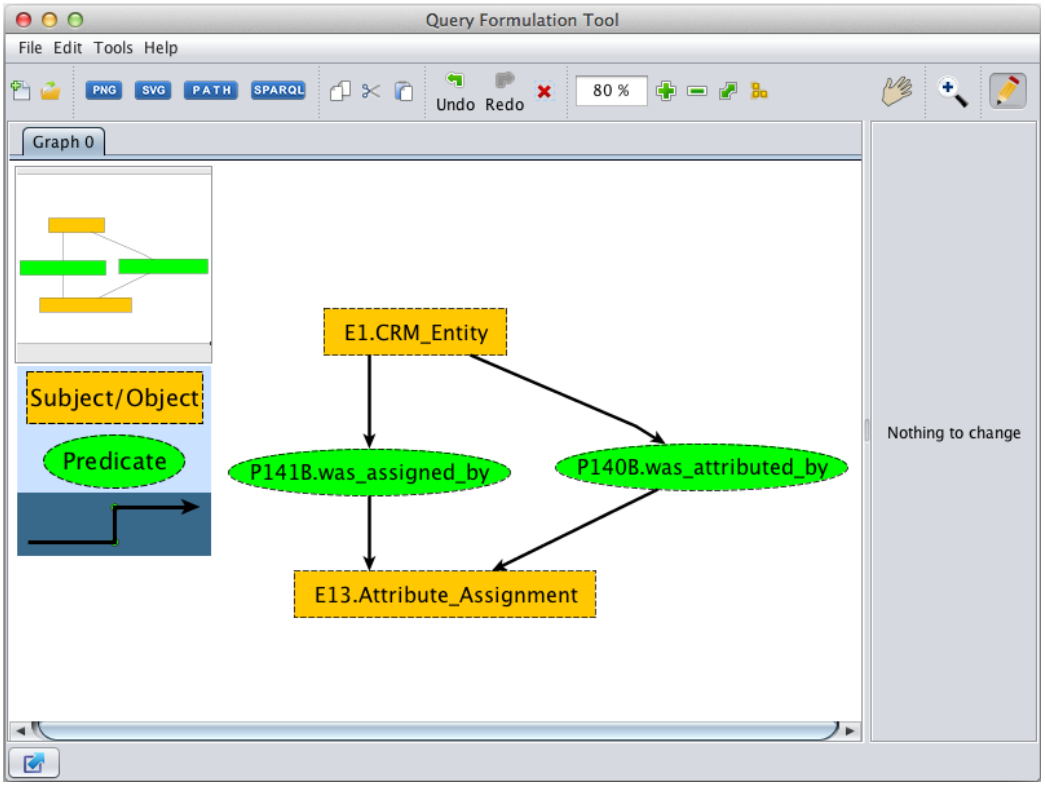


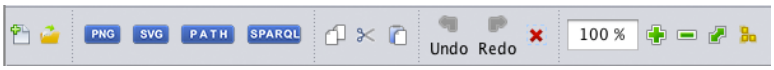
Figure 4.9: Logic in auto complete

Editing toolbar

As described beforehand, a toolbar is located on the upper part of the canvas(Figure 4.10(a)) whose aim is to effect the quickest and most efficient interaction with the editor. This toolbar is separated into two parts. The first part (Figure 4.10(b)) includes shortcuts related to the editing process and which concern various operations, such as deletion, undoing or redoing and applying layout.

The second part (Figure 4.10(c)) concerns alternating between modes, enabling the user to change from one of two system modes to the other. The former is navigation mode (hand icon) and the latter is editing mode (pencil icon). In navigation mode, the tool does not allow the user to make any changes or additions to the graph unless he wishes to navigate within it (scroll, zoom etc.). On the other hand, only when the user enters editing mode, does he have the ability to make changes he may wish to make to the graph.

(a) Toolbar



(b) Edit tools

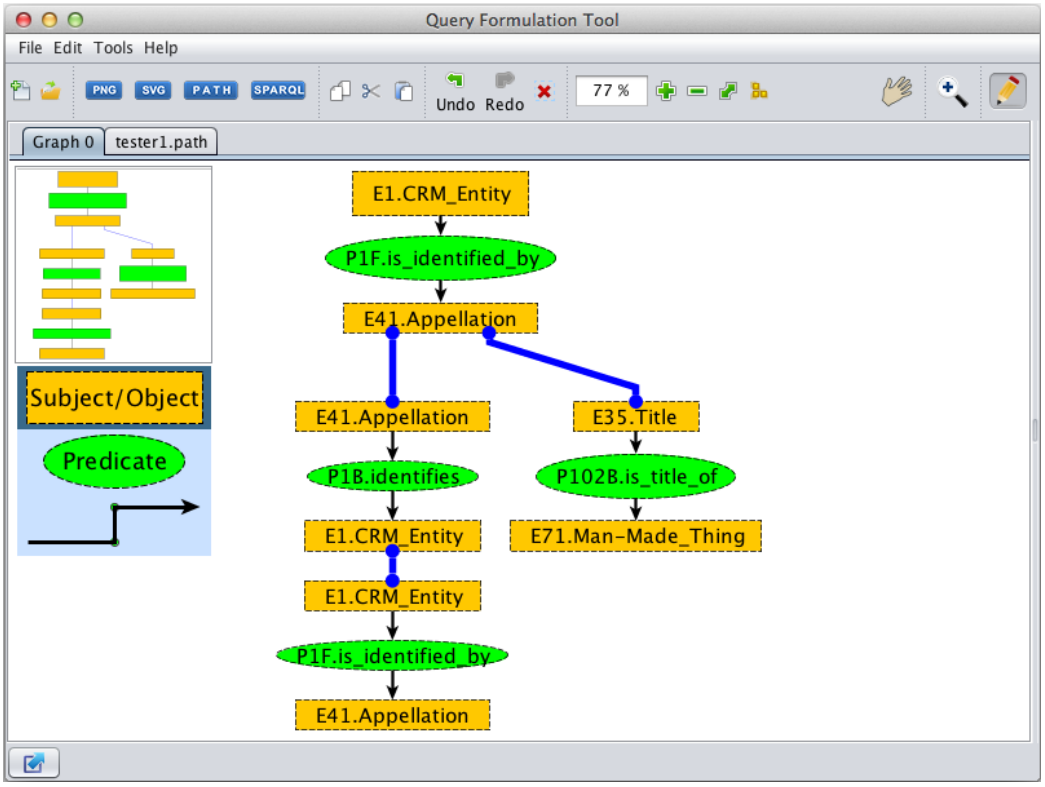
(c) Change modes

Figure 4.10: Editing Toolbar

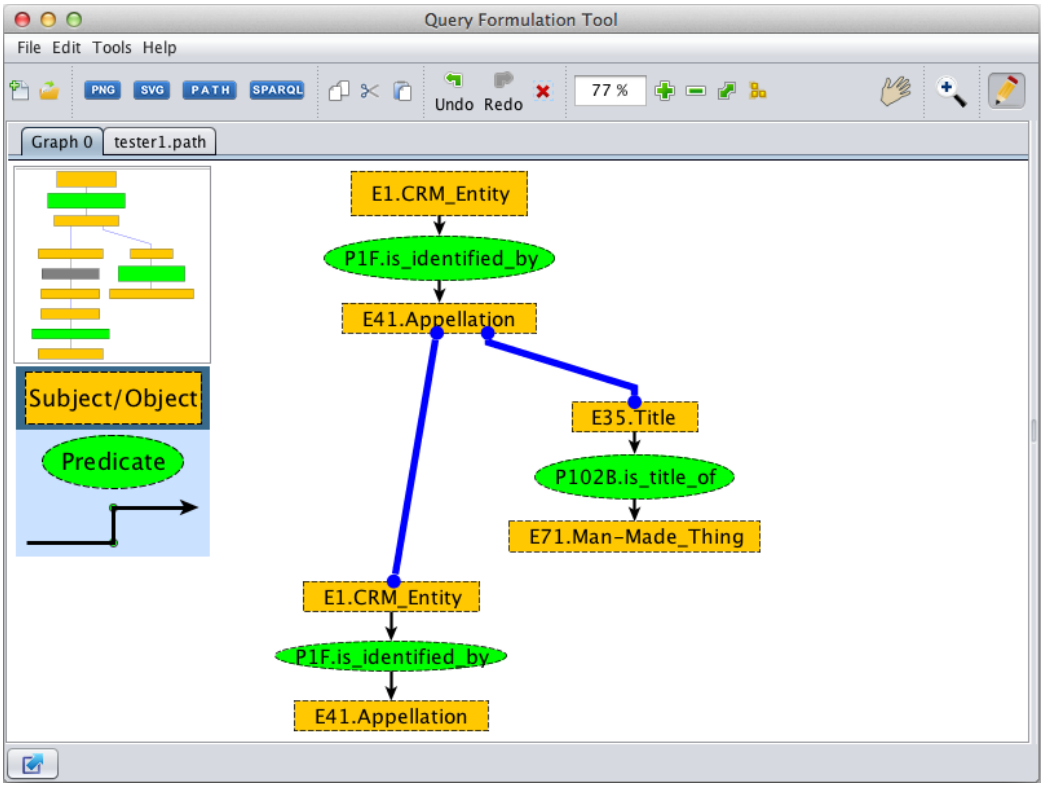
Graph editing features

FRGE supports the deletion of nodes or triplets. However, the inclusion of the deletion process within the software was not easy to effect, largely because any node deletion had to be checked in the interest of maintaining orthological graph structure. Consequently, FRGE checks if a triplet can be removed (checking whether it is permissible for a previous triplet to be connected with the following one) and if so, it removes the triplet and automatically constructs the connections between the two triplets(Figure 4.11).

Another processing tool the editor provides is the support of undo or redo commands. The significance of these commands is that no restriction is made as to the number of undo/redo commands that can be issued.



(a) Before delete action



(b) After delete action

Figure 4.11: Delete action

Edit values of nodes

Throughout the editing process the user need not type any value. Node values are selected from a selection of dynamic lists which contain all available values for the chosen node. These lists appear on a panel to the right of the canvas and as filled in differently each time depending upon the selected node (Figure 4.12). The possibility to enter values in ascending or descending order allows for the most efficient and simple user interaction with these lists because available values in a list may be considerable.

In addition, a search function is available within these lists. The user may either search for a complete value or a part thereof. The search mechanism does not seek values which constitute only the beginning of a string within the lists, but aims to find any values which contain the sought after string. This helps the user because it is not necessary for him to know the exact structure of the sought value but only a part of it. Finally, it is worth noting that the search is conducted as the user types, so he is able to immediately correct the short after string if he should find that the search results are not to his satisfaction.

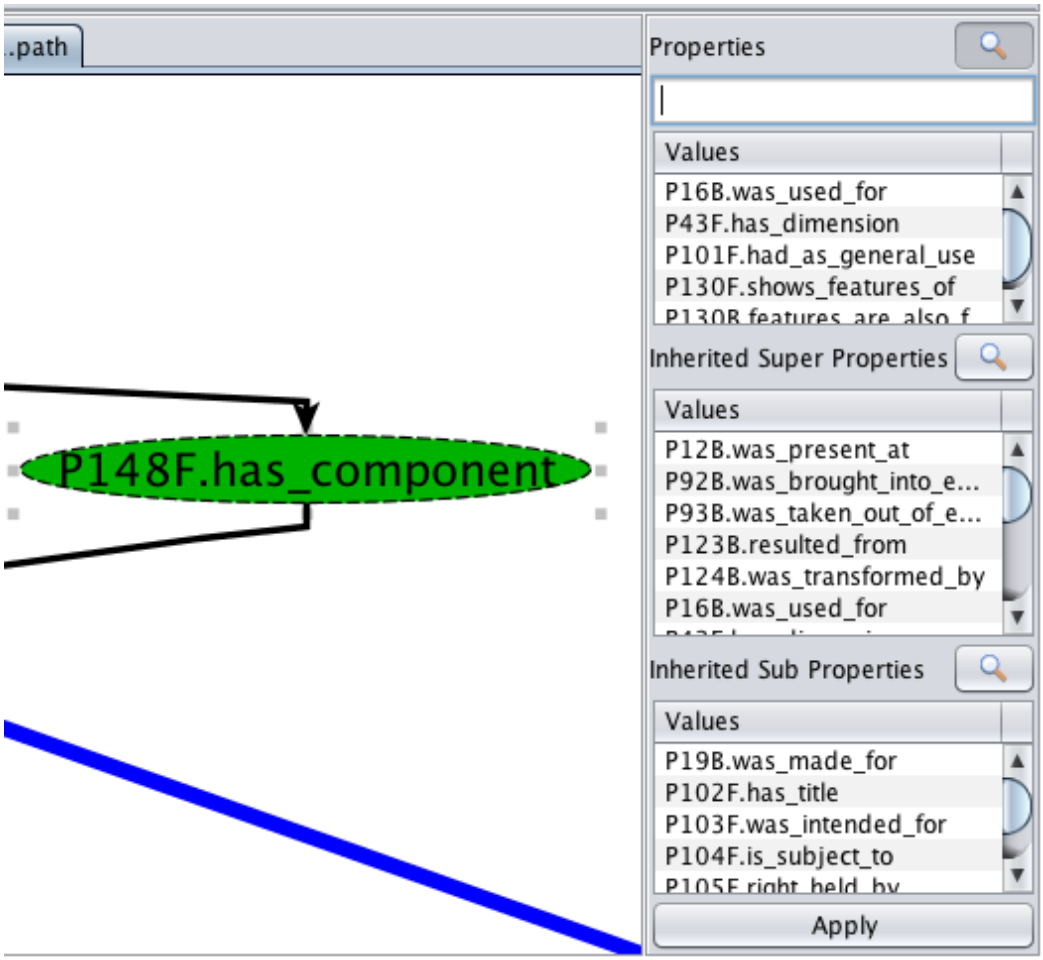


Figure 4.12: Changing nodes values

Multi-tab support

Another feature of FRGE is that it provides the user with the ability to compose or process parallel path queries. This ability is accelerated by the fact that the editor supports the simultaneous opening of multiple path queries in different tabs. In this way, the user has the ability to alternate quickly between paths, improving the efficiency of his interaction with the user interface.

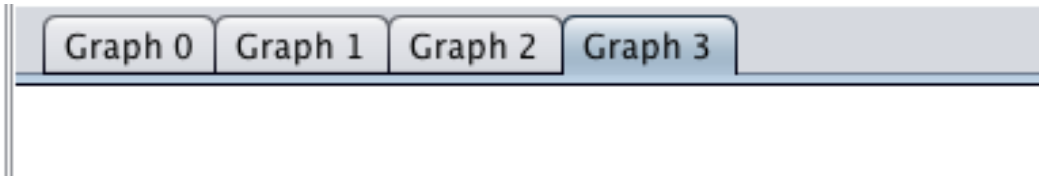


Figure 4.13: figure

User individualizations

FRGE provides the user with the ability, if he so wishes, to individualize his path queries. In this way, he may manage his own prefixes, multi-instantiations or disjoins (Figure 4.14, 4.15, 4.16, 4.17). The user is able to add, remove or even edit. These individualizations do not affect repository data directly since they are used and saved exclusively by the editor. Path queries are considered during composition, and dynamically appear wherever they are considered appropriate. It is stressed, that here too, the editor functions dynamically, ensuring that overlapping between multi-instantiations and disjoins does not occur (that is, it ensures that a multi-instantiation has not been recorded as a disjoin, and vice versa).

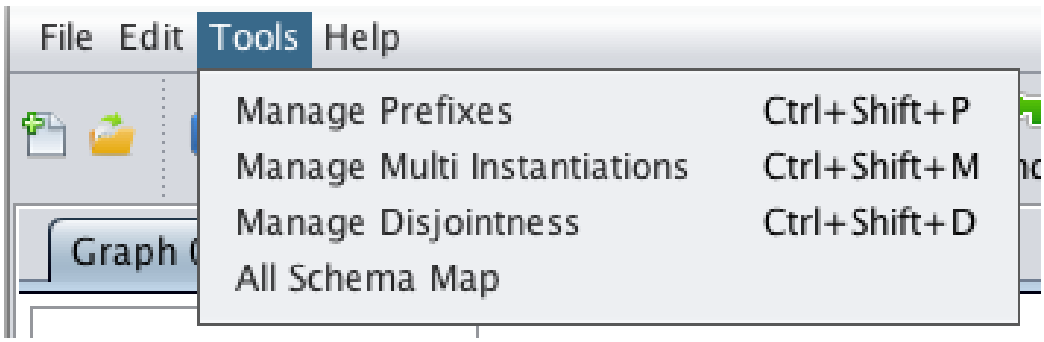


Figure 4.14: User specialization options

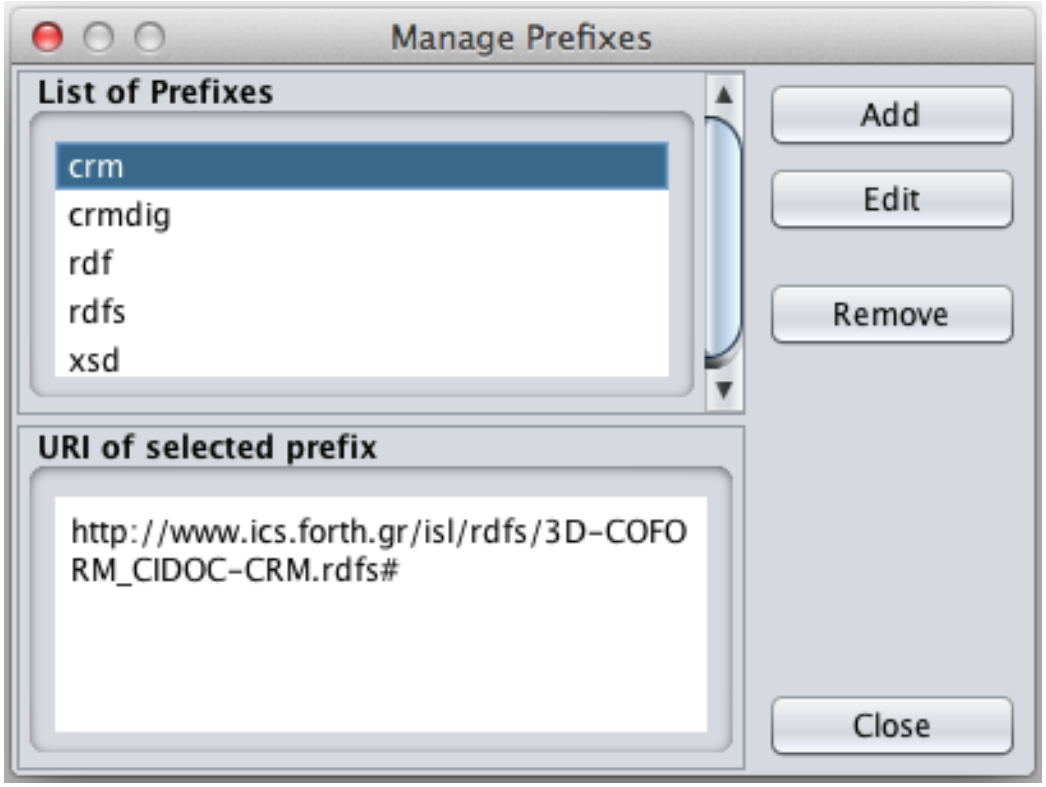


Figure 4.15: Prefixes Editor

By the term multiple instantiations, we refer to those instances which can be described either as one class of a schema or another. More specifically, the user has the ability to define whether the UUID of an object is an instance of more than one class and consequently inherits the properties of all of its superclasses (multiple inheritance). Multiple instantiation greatly simplifies the picture of how the world works, since we can omit the class hierarchy for certain cases there by reducing its complexity.

Some multiple instantiation cases have already been defined driven from the frequent use of the instances of the one as instances of the other. These are:

- E53.Place<->E19.Physical_Object, E39.Actors. Physical objects are often referred to as Places for example museums, organizations.

- E5.Event<->E52.Time-Span. Time spans can be considered as events for example 1821 frequently denotes the Greek revolution of 1821

- E52.Time-Span<->E5.Event. Often instead of specifying the time spans, we use instead the event for example we may say: "When Nikos was born, Thessaloniki was burned" meaning on 1917 and not the actual birth event of Nikos

- E7.Activity<->E6.Destruction. A destruction in the CIDOC-CRM schema is considered not to be an activity, thus can nott be performed by someone. Nevertheless, it is useful many times that we link it with an actor that responsible for it, so we may use it also as an activity. For example, Christo's cup was destroyed by "the breaking of Christo's cup by Maria", which was carried out by Maria. In this example, "the breaking of Christo's cup by Maria" is both, an instance of destruction and an instance of activity.

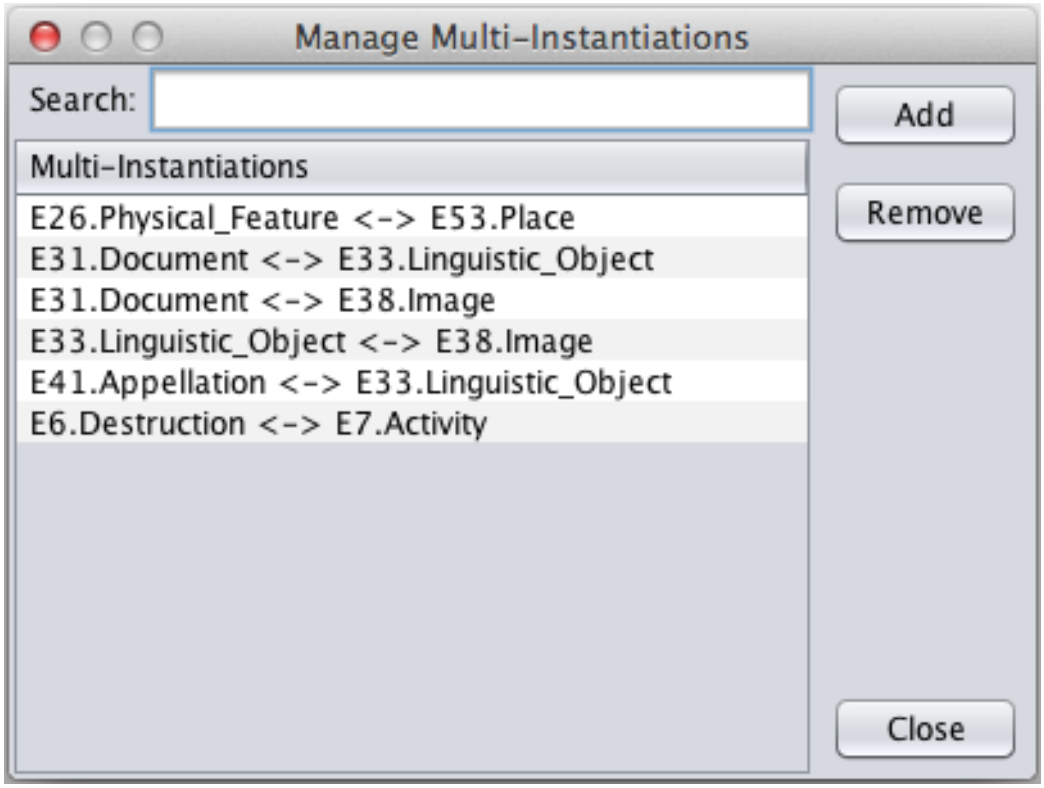


Figure 4.16: Multiple instantiations editor

As opposed to multiple-instantiation cases, when we refer to a disjoint case, we describe two classes that are not allowed to share the same instance. Moreover,

the disjoint property does not only affect the submitted classes, but also has a hierarchical and symmetrical affect, which means that it is inherited to the subclasses of the classes and is also valid vice versa. There is subsequently no need for there to exist duplicate entries in different orders for each disjoint case.

The following disjoint cases have already been defined for the FCs-FRs models. These have been derived from the CIDOC-CRM proposal for the declaration of all disjoint classes [5].

- E2.Temporal_Entity<-/->E77.Persistent_Item, E53.Place, E54.Dimension, E52.Time-Span, E59.Primitive_Value
- E77.Persistent_Item<-/->E2.Temporal_Entity, E53.Place, E54.Dimension, E52.Time-Span, E59.Primitive_Value
- E53.Place<-/->E54.Dimension, E52.Time-Span, E59.Primitive_Value, E77.Persistent_Item, E2.Temporal_Entity
- E54.Dimension<-/->E52.Time-Span, E59.Primitive_Value, E53.Place, E77.Persistent_Item, E2.Temporal_Entity
- E52.Time-Span<-/->E59.Primitive_Value, E2.Temporal_Entity, E77.Persistent_Item, E53.Place, E54.Dimension
- E39.Actor<-/->E24.Physical_Man-Made_Thing, E73.Information_Object, E41.Appellation, E51.Contact_Point, E55.Type
- E24.Physical_Man-Made_Thing<-/->E39.Actor
- E51.Contact_Point<-/->E39.Actor
- E73.Information_Object<-/->E39.Actor
- E41.Appellation<-/->E39.Actor
- E18.Physical_Thing<-/->E28.Conceptual_Object
- E28.Conceptual_Object<-/->E18.Physical_Thing
- E26.Physical_Feature<-/->E19.Physical_Object

[5] http://www.cidoc-crm.org/issues.php?id=92

- E19.Physical_Object<-/->E26.Physical_Feature
- E55.Type<-/->E30.Right,E39.Actor
- E30.Right<-/->E55.Type
- E56.Language<-/->E57.Material,E58.Measurement_Unit
- E58.Measurement_Unit<-/->E56.Language
- E57.Material<-/->E58.Measurement_Unit, E56.Language

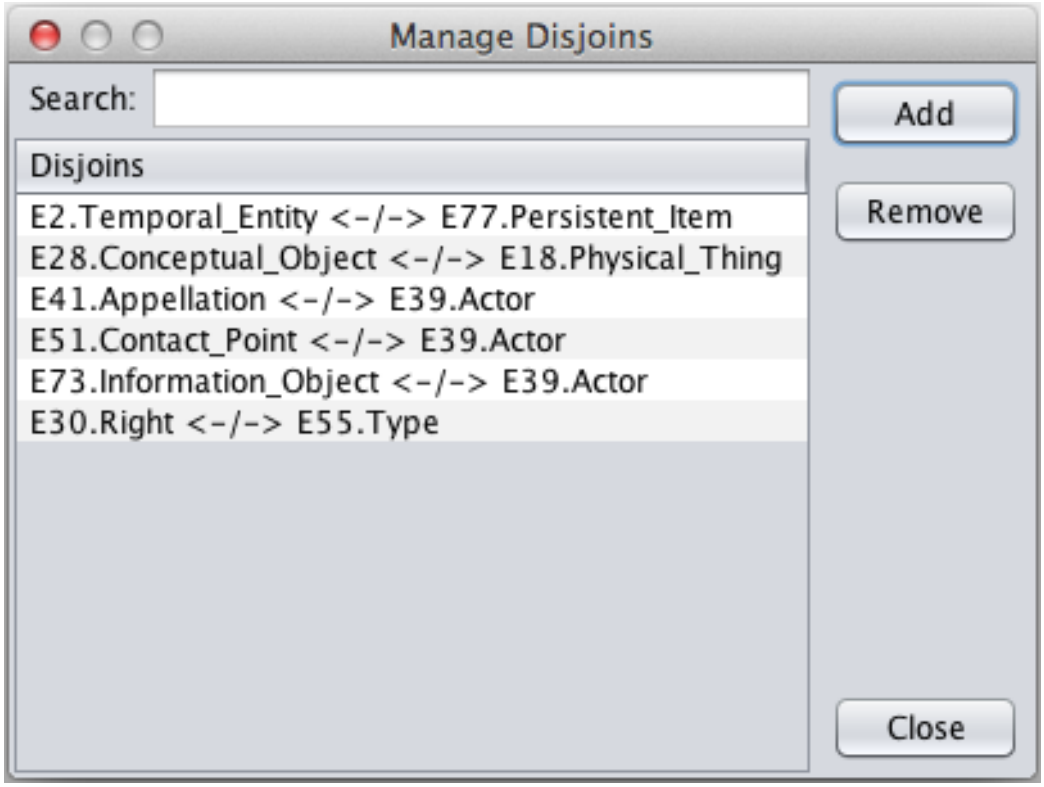


Figure 4.17: Disjointness Editor

Import text path queries

One of the most important functions of FRGE is the ability to import a path query which is in text form. As described earlier in section 4.2.4, by virtue of mechanism which we developed, the editor is able to convert path queries which are in text form (Figure 4.18) into graphs (Figure 4.19). In this way, the user is able to edit path queries in graph form.

```
tester1.path
E70.Thing -- { (P46F.is_composed_of) [0,n] OR (P106F.is_composed_of ) [0,n] OR (P148F.has_component) [0,n]}-> E70.Thing:
    {
    E70.Thing -- {P67B.is_referred_to_by OR P129B.is_subject_of} -> E89.Propositional_Object:
            {
                E73.Information_Object -- P94B.was_created_by -> E65.Creation:
                {
                    E65.Creation -- P7F.took_place_at ->E53.Place:
                    {
                        E53.Place  --( P89F.falls_within)[0,n]-> E53.Place
                    }
                }
                OR
                E73.Information_Object -- P128B.is_carried_by -> E24.Physical_Man-Made_Thing:
                {
                    E24.Physical_Man-Made_Thing -- (P46B.forms_part_of) [0,n] ->   E18.Physical_Thing:
                        {
                            E18.Physical_Thing -- P53F.has_former_or_current_location -> E53.Place :
                            {
                                E53.Place  --( P89F.falls_within)[0,n]-> E53.Place
                            }
                        }
                }
                OR
                E24.Physical_Man-Made_Thing -- P108B.was_produced_by -> E12.Production:
                {
                    E12.Production --(P9B.forms_part_of)[0,n] -> E5.Event:
                    {
                        E5.Event -- P7F.took_place_at ->E53.Place:
                        {
                            E53.Place  --( P89F.falls_within)[0,n]-> E53.Place
                        }
                        OR
                        E24.Physical_Man-Made_Thing -- P108B.was_produced_by -> E12.Production:
                        {
                            E12.Production --(P9B.forms_part_of)[0,n] -> E5.Event:
                            {
                                E5.Event -- P7F.took_place_at ->E53.Place:
                                {
                                    E53.Place  --( P89F.falls_within)[0,n]-> E53.Place
                                    OR
                                    E24.Physical_Man-Made_Thing -- P108B.was_produced_by -> E12.Production:
                                    {
                                        E12.Production --(P9B.forms_part_of)[0,n] -> E5.Event:
                                        {
                                            E5.Event -- P7F.took_place_at ->E53.Place:
                                            {
                                                E53.Place  --( P89F.falls_within)[0,n]-> E53.Place
                                                OR
                                                E24.Physical_Man-Made_Thing -- P108B.was_produced_by -> E12.Production:
                                                {
                                                    E12.Production --(P9B.forms_part_of)[0,n] -> E5.Event:
                                                    {
                                                        E5.Event -- P7F.took_place_at ->E53.Place:
                                                        {
                                                            E53.Place  --( P89F.falls_within)[0,n]-> E53.Place
Line 1, Column 1                                                                                    Spaces: 4    Plain Text
```

Figure 4.18: Path query in text form

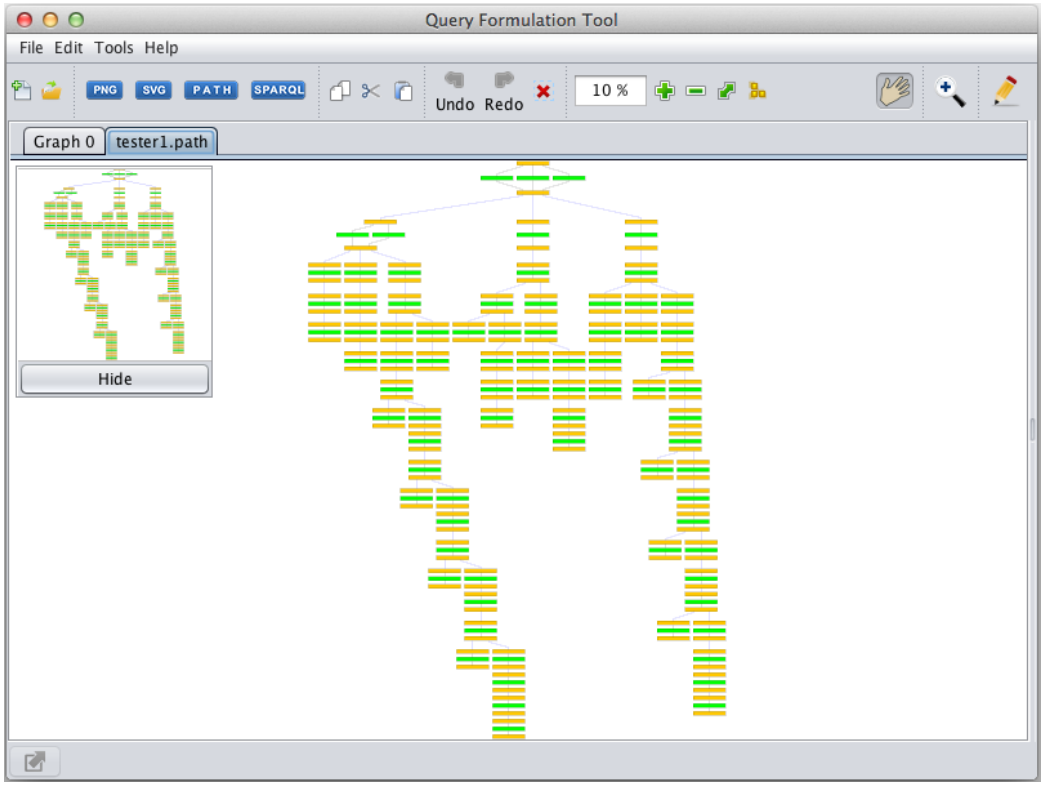
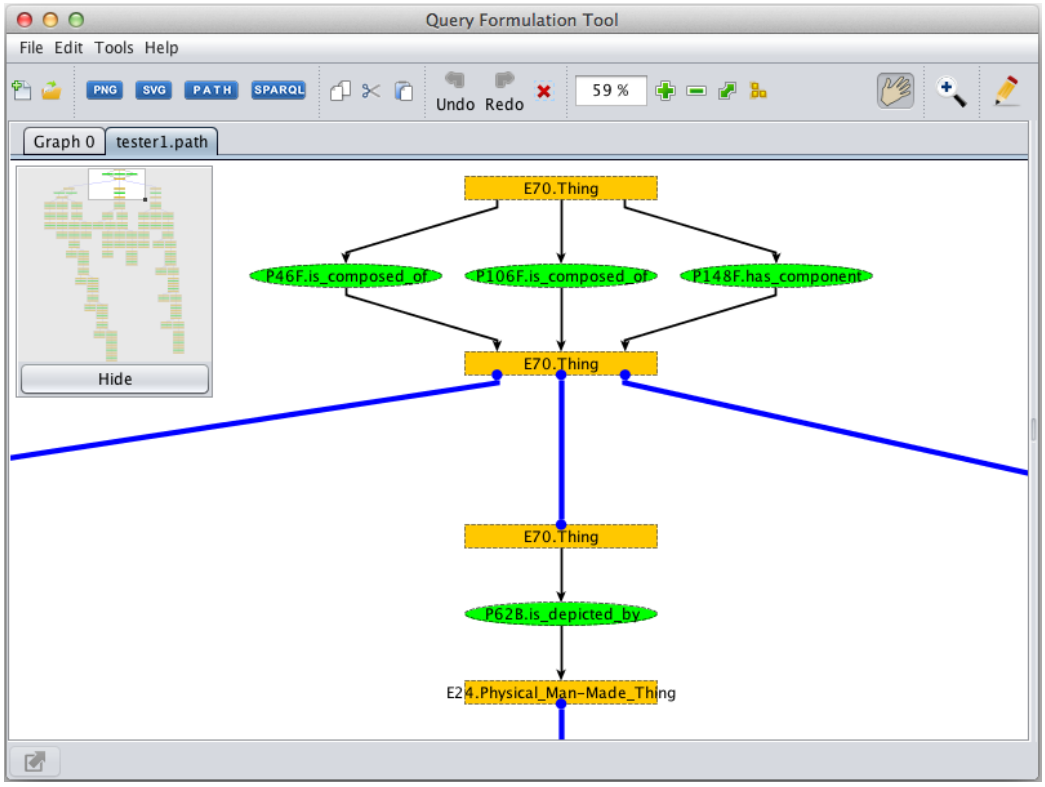


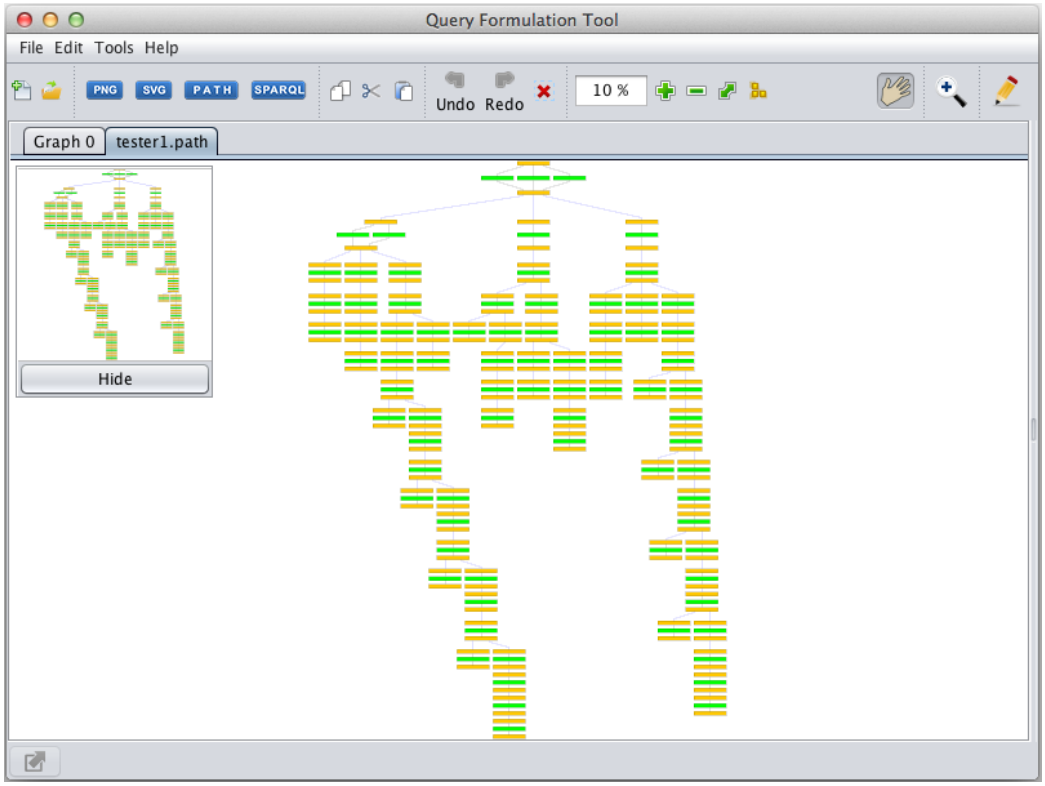
Figure 4.19: Imported path query in graph form

Browsing features

Apart from processing path queries, FRGE provides functions related to the ease of navigation within them and to a better understanding of them. Therefore, apart from access to navigation mode the user is able to zoom in and out, as seen in Figure 4.20. In this way, he is able to focus on any path query he wishes. Additionally, he is provided with an auxiliary map of the graph, which illustrates the structure of the path query in its entirety (Figure 4.21). This map shows the user the level of the zoom and its position relative to the graph as a whole, and in which part of the graph the user finds himself at any given moment.



(a) Zoom In



(b) Zoom Out

Figure 4.20: Zoom Features

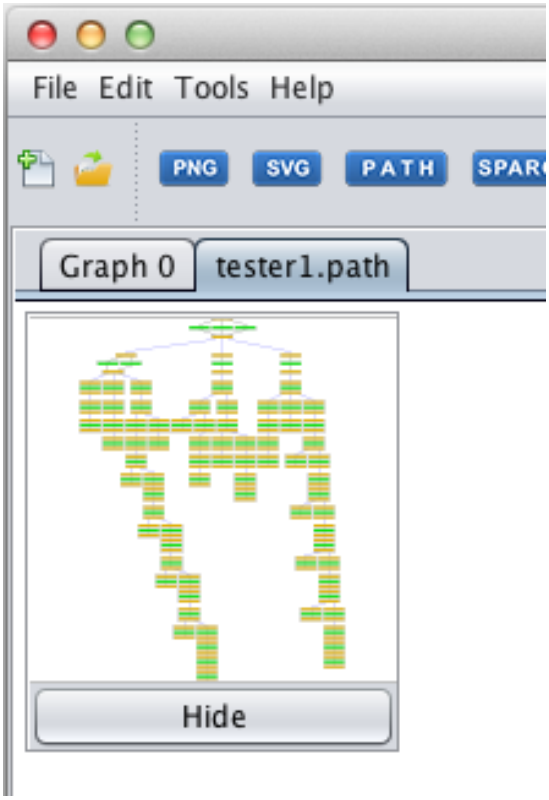


Figure 4.21: Graph map for easy navigation

Info panel

Another function which aids the user in better understanding and composing path queries is the inclusion of an auxiliary panel that contains information regarding the components of the graph. More specifically, information related to the selected node, such as information concerning differentiation between super and sub classes, as well as comments regarding class/property which the nodes represent within the schema, are included. It must be stressed, that the panel is not a permanent fixture, but it appears only when the user selects it.

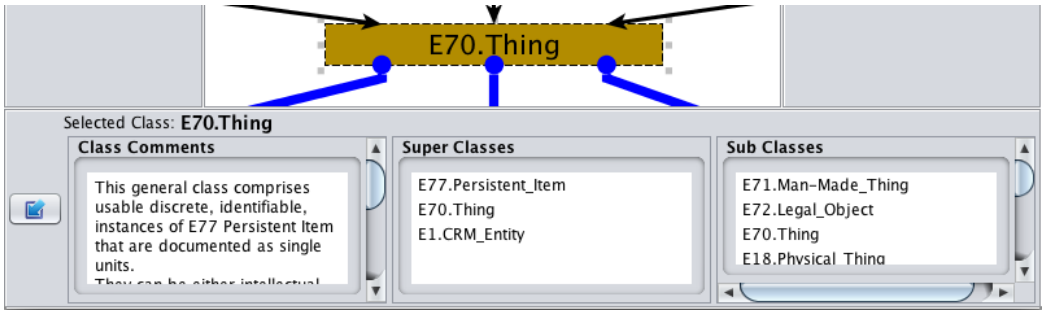


Figure 4.22: Info panel gives details about selected nodes

Schema overview

Moreover, another function provided by the editor is the ability to overview the whole schema in which the user works. In this functionality, the user has the ability to locate and to learn details about each class and their properties. In addition, he can overview all the relationships between the objects of the schema. More specifically, he can identify and specify how the classes are connected together (subclasses/superclasses relationships) and how they are connected through their properties.

Furthermore, when the user chooses any node in the graph, all the incoming and outgoing relationships to it (displayed as arrows) are coloured in differently in order to become more prominent. To the right side of the canvas there is a panel that contains all the schema's objects, separated into two lists, one for classes and one for properties. The user can chose one of these objects and the selected object will automatically be located and centralized in the canvas view. It should be noted that there is a search mechanism for both of these lists, hence finding the desired object proves easier. To the bottom of the canvas, there is another panel that displays much useful information about the selected object of the schema. If the user wishes, he can hide both of these panels (the one containing the lists and the other containing information), and thus enlarge the size of the canvas.

At the top of the canvas there is a toolbar, where the user can find several useful functionalities, such as the ability to change the shape of the graph by choosing a different layout. The available layouts are the Hierarchical, the Circular, the Organic and the Orthogonal. Another functionality is the ability to export the entire schema into a Scalable Vector Graphic (SVG).

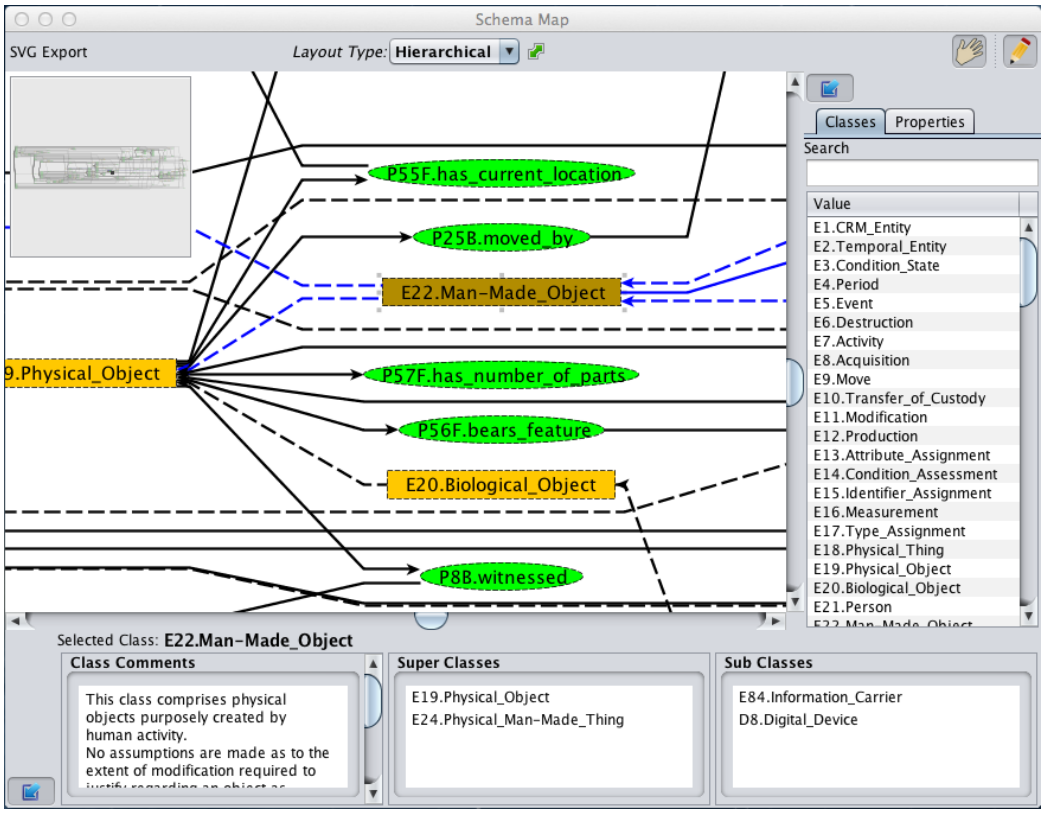


Figure 4.23: Schema overview

Other functionalities

Apart from the above mentioned functionalities FRGE provides a plethora of auxiliary functions which facilitate the quick and simple interaction between the editor and user. Through a series of pop up windows, the editor is instrumental in helping the user avoid errors both, during the composition of the path queries and during the available actions supported by the system (for example, the defining of user individualizations)

Additionally, the editor clearly presents the components of the graph by utilizing a variety of colours and shapes. In this way, the user can more easily understand the path queries and can subsequently more concisely interact with it.

Finally, the fact that the editor provides the user with the ability to shape the graph automatically by imposing a given layout, should be emphasized. The user has the ability to draw the graph with little care for specific positionings and with

little concern regarding its intelligibility, because at any time he wishes, he can ask of the editor to automatically format the components of the graph in tree layout.

Chapter 5

Experimental Evaluation

5.1 Evaluation

The four basic ways of evaluating user interfaces are: automatically (usability measures computed by running a user interface specification through some program), empirically (usability assessed by testing the interface with real users), formally (using exact models and formulas to calculate usability measures), and informally (based on rules of thumb and the general skill and experience of the evaluators). Under the current state of affairs, automatic methods do not work, and formal methods are very difficult to apply [35].

The main way to evaluate user interfaces is through empirical methods, with user testing admittedly being the most commonly used method. An inhibitory factor during the evaluation is the fact that searching for and finding real users turns out to be exceedingly difficult or too expensive in terms of recruiting users in satisfactory numbers in order to test all aspects of an evolving design. This fact leads us to the use of inspection as a way to "save users". Moreover, project schedules or budgets sometimes necessitate restrictions that make informal methods, like inspection, desirable as a "discount usability engineering" solution [36] [37] since they are highly cost-effective [38]. Usability inspection methods have been proven, by several studies, to provide the ability to find many usability problems that user testing would neglect to find. On the other hand, users are able to locate some problems that are overlooked by inspection, meaning that the best results can often be achieved by combining several methods [39] [40] [41].

In our case, for the evaluation process of our Graphical Fundamental Relationship configuration tool (FRGE), we asked non-expert users to test and give us a feedback of their usage experience. Users tested it in the CIDOC-CRM and the CIDOC-CRMdig, the two metadata schemas upon which our model was implemented. During the evaluation, we tested the efficiency and the usability of our new query system. We compare and contrast the users' ability to specify and interpret various types of queries in the fundamental relationships framework using FRGE versus the previous text-based configuration tool [42]. In order to provide better quality and a more thorough analysis of the results, we divided the evaluation process into the following main categories:

- Time to accomplish a task
- Correctness of queries
- Convenience of navigating and understanding the paths

We tested three level path expression queries of differing complexity in the above categories. The tested queries are the following:

1. **Short length query path**

```
E5.Event -- (P9F.consists_of)[0,n] -> E5.Event:
 { E63.Beginning_of_Existence -- P92F.brought_into_existence -> E70.Thing:
    { E70.Thing -- (F4F.is_composed_of)[0,n] -> E70.Thing }
 }
```

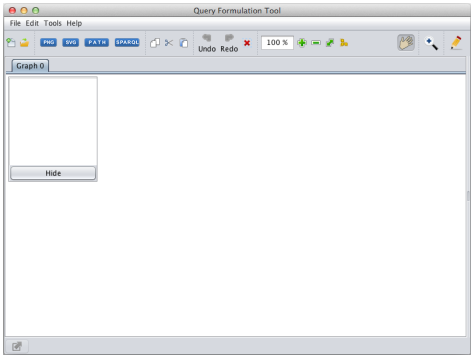
2. **Normal length query path**

```
E5.Event -- (P9B.forms_part_of)[0,n] -> E7.Activity:
 { E7.Activity--P14F.carried_out_by->E39.Actor:
     { E39.Actor--(P107B.is_current_or_former_member_of)[0,n]-> E39.Actor }
   OR
   E7.Activity -- P15F.was_influenced_by-> E39.Actor:
     { E39.Actor--(P107B.is_current_or_former_member_of)[0,n]->E39.Actor }
 }
```

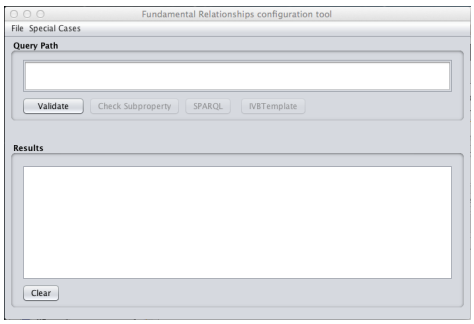
3. **Extended length query path**

```
E5.Event -- (P9F.consists_of)[0,n]-> E5.Event:
{ E63.Beginning_of_Existence --P92F.brought_into_existence ->E70.Thing:
  { E70.Thing --(F5F.consists_of_shows_features_of)[0,n]->E70.Thing:
    { E24.Physical_Man-Made_Thing -- P62F.depicts -> E70.Thing:
      { E70.Thing --(F4F.is_composed_of)[0,n]-> E70.Thing
      }
    OR
      E89.Propositional_Object --P67F.refers_to-> E70.Thing:
      { E70.Thing --(F4F.is_composed_of)[0,n]-> E70.Thing }
    OR
      E24.Physical_Man-Made_Thing -- P128F.carries -> E73.Information_Object:
      { E73.Information_Object -- P67F.refers_to -> E70.Thing:
        { E70.Thing --(F4F.is_composed_of)[0,n]-> E70.Thing }
      }
    }
  }
OR
 D2.Digitization_Process -- L1F.digitized ->E70.Thing:
 { E70.Thing--( F5F.consists_of_shows_features_of)[0,n]-> E70.Thing:
   { E70.Thing
   OR
     E24.Physical_Man-Made_Thing -- P62F.depicts -> E70.Thing:
     { E70.Thing -- (F4F.is_composed_of)[0,n]-> E70.Thing }
   OR
     E89.Propositional_Object --P67F.refers_to-> E70.Thing:
     { E70.Thing -- (F4F.is_composed_of)[0,n]-> E70.Thing }
   OR
     E24.Physical_Man-Made_Thing  -- P128F.carries -> E73.Information_Object:
     { E73.Information_Object --P67F.refers_to ->  E70.Thing:
       { E70.Thing -- (F4F.is_composed_of)[0,n]-> E70.Thing }
     }
    }
  }
}
```

We set measurement systems relative to each test and we judged the results based on the grades in each system. Before the tests, we explained the functionalities of each tool to the users and we encouraged them to explore each interface and to become familiar with them. The FRGE users took about 3-4 minutes on average to become familiar with it (Figure 5.1(a)), in contrast to the text-based tool (Figure 5.1(b)) users, who took on average about 1-2 minutes. This difference is attributed to the lack of interactive functionalities offered by the text-based tool.



(a) FRGE



(b) Text-based configuration tool

Figure 5.1: The two tools

The group of participants consisted of five user with different experience levels on the subject of query formulation. All of them were non-expert users and two of them were not even from the field of computer science. The last two participants were included in the tests because we wanted to examine our tool's level of user-friendliness, and to enforce our arguments in support of no need for previous technical knowledge.

Time to Accomplish a Task

During this test we asked the users to construct the given path expression queries using a different configuration tool each time. We used time scales to measure the results for these tests. The following table gives the average time required by the user to accomplish the requested task

These results indicate that, although the users of the graphical configuration tool spent more time learning its interface as opposed to that of the text-based configuration tool, the users of FRGE were able to use it more efficiently than

	Text-based Configuration Tool	Graphical Configuration Tool
Short query path	about 5 minutes	2 to 3 minutes
Normal query path	12 to 15 minutes	7 to 9 minutes
Extended query path	50 to 60 minutes	20 to 25 minutes

Table 5.1: Time to construct the paths queries using different configuration tool

the text-based interface in interpreting all types of queries. The greatest efficiency advantage of the graphical, over the text-based tool, is in specifying large and more complex path queries, with the users of the FRGE taking about half the time text-based tool users needed to specify these types of queries. The FRGE was especially designed to handle such complex queries and we thus expected the tool users to perform better on these types of queries than text-based tool users. The results confirmed our expectations and they show that the FRGE provides a 50%-60% (on average 53.89%) efficiency gain, in comparison to the text-based tool for such complex path queries.

Correctness of Queries

In this section, we repeated the previous experiment and we enumerated the errors users made during the procedure. This time, we compared FRGE users with text-based configuration tool users in terms of their accuracy and correctness. Unsurprisingly, the difference between the two tools was considerable.

The text-based users faced many troubles while trying to synthesize the path queries. There were two kinds of errors users ran into. The first one concerned the spelling of classes and properties, and the validity of the paths. Even though simplifying the schema for querying complex semantic networks improved the effectiveness of searching within them, text-based users faced problems in memorizing the names of the classes. Consequently, they encountered many spelling troubles in synthesizing query paths. The second one was the accuracy of the path queries. Remembering the complete picture of the schema and the relationships between classes and properties (identifying heredity) proved exceedingly difficult for users. Consequently, they could not be very accurate in their expressions. This failure to synthesize accurate query expressions paths led to low recall rates.

On the other hand, the FRGE users responded with great accuracy to our

demands. These positive results were expected because of our initial idea to design and implement a system to help and improve user interaction. The FRGE is designed not to let users type anything by default, and to ensure that every fundamental category or fundamental relationship used, has been chosen from a suggested list. Therefore, the avoidance of accuracy and correctness errors is another "by default" benefit. The graphical tool users were assisted by the auto-complete mechanism that the tool provides for them. These functionalities helped users to construct the path queries accurately and effortlessly, without having to face any problems in spelling the names of the classes or properties.

Convenience of Navigating and Understanding the Paths

In this category of tests, we asked the users to navigate through specific paths (from the top node to an end node) and we also asked them to try to understand their flow and make small changes correspondingly. Text-based tool users seemed to respond as well as the graphical tool users to the short length paths query. Both reacted easily to our demands without facing any remarkable difficulties.

In the second one, normal length path query, text-based users started to face problems in understanding the overall structure of the normal length path query. So, users had difficulties making changes to the path queries. On the contrary, FRGE users seemed to rapidly perceive the structure of the path query and were thus able to start editing it in almost zero time. This rapid user response coupled with the convenience of being able to interact with the path query, is attributed to the graph structure of the query and its graphical representation, which are both provided to users of the tool.

In the last path query (the extended one), the gap between the two configuration tools was remarkable. Text-based users seemed to be overtly confused with the overall architecture of the query, and it was almost impossible for them to isolate a triplet and to interact with it. On the other hand, the FRGE users seemed to fully understand the whole arrangement of the triplets despite the time needed to become familiar with the overall structure of the path query. Furthermore, they were able to isolate and edit sub paths or triplets at any query depth in almost zero time, without experiencing any confusion.

5.2 Conclusion

In this study, we used three criteria to compare and contrast the usability of Graphical Fundamental Relationship configuration tool with a text-based configuration tool: The time needed to accomplish specific tasks, the correctness (efficiency and accuracy) in specifying path expression queries, and the convenience of users being able to interact with the system.

The study demonstrated that despite FRGE exhibiting a slightly steeper learning curve than the text-based configuration tool, users were able to specify fundamental relationship queries more accurately and more efficiently (considerably more efficiently for extended path queries and paths using rare classes easily forgotten or misspelled). In general, the experimental evaluation demonstrated that as the path expression query became larger and more complex, the efficiency gap between the two tools grew, and the advantages of the graphic tool become more conspicuous.

Chapter 6

Conclusions and Future Work

6.1 Conclusion

In this thesis, in order to simplify the search process in metadata repositories(semantic networks), we dealt with the implementation of a simpler model consisting of only a few fundamental classes and relationships, which allows for an easier and more efficient way to search those kinds of repositories. Fundamental Relationships can be defined by using a "paths language" over the CIDOC-CRM schema, which is designed to be easy to write and to be comprehensible to non-expert users. Through this approach, the size and the structure of the search model comes closer to core metadata, which is more friendly to users. Furthermore, with this approach we can get higher recall rates if the simplified model subsumes the properties of the richer one. A possible problem is the reduction in precision rate because of the use of property propagation. However, improvement in precision rate can be attained by composing specializations of fundamental relationships or by expressing a greater number of restrictions over queries.

Our work was separated into two distinct parts. The first regarded the linguistic analysis of our paths expression language and the second concerned the implementation of a configuration tool for the proposed path language. In the first part, we defined the semantics of our language and the concepts within it, and we thought of some specific tasks and cases that someone would want to be able to perform with our language. Furthermore, we experimented with syntax ideas (the text of the language) for the above examples. All the aforementioned referred to

the design process of the language. Afterwards, we dealt with the implementation of our language and more specifically, we wrote out a formal grammar for the syntax. We also had to design and develop an interpreter for it whose implementation included the tasks of developing a front-end scanner and parser.

In the second part, our purpose was to design and implement an integrated development environment (IDE) for the paths expression language. This environment aims to facilitate the process of creating complex path queries in the CIDOC-CRM that contains all possible interpretations of the Fundamental Relationship in CIDOC-CRM terms. Since our target group is non-expert users we designed and implemented a Visual Query System (VQS), aiming at enlarging the communication bandwidth of our paths' language, using a wider range of communication media, relying as much as possible on the senses of the human being. To pursue this goal, we have developed a graphical editor, which provides a visual query language counterpart to our paths' language, assisting in writing triple patterns and verifying their syntax through the CIDOC-CRM. This involves issues from various perspectives like layout, readability and interaction design.

The graphical editor provides an overview of declared paths that allows the users to check the validity and the completeness of a query against its intended functionality. In addition, the editor allows users to easily modify, relax or restrict a query, with the overall purpose of alleviating the problem of errors. Such errors are the misspelling of the name of classes or properties during path writing, or the omission of the use of some predefined special characters, like path connection characters or the use of different kinds of brackets.

Moreover, the graphical editor is able to read data from a file. The imported data can be given either in original software file-type, or in the paths' language files. At the end of the user editing process, apart from saving the data, the graphical editor can export it back into the paths' language format.

Finally, for the evaluation of our graphical editor, we asked non-expert users to test it and give us feedback of their usage experience. Users tested it in the CIDOC-CRM and the CIDOC-CRMdig, the two metadata schemata on which our model was based and implemented. The results of these tests were positive.

6.2 Further Work

The way in which we selected the FCs and FRs is still mostly intuitive and leaves room for further work targeted at standardizing the model. Furthermore, since the model is basically implemented within the context of an on-going project, further testing and verification would be welcome. In our case, the model is implemented in two metadata schemata-the CIDOC-CRM and the CIDOC-CRMdig. Testing and verification could possibly be extended to a greater number of metadata repositories that employ the same or different schemata, of the same and/or different discourse.

Another area deserving of further scientific study would be a comparison in terms of time and memory allocation efficiency of the proposed SPARQL implementation versus the OWLIM rules implementation of the Research Space Project. Moreover, to further improve precision, alterations could be made to the paths? language in order to make it more expressive such as enabling the possibility of including negation (filters) in crucial positions of the FRs. As of yet, the model cannot use an FR to define other FRs, unless the FR is defined as schema property, e.g. through OWLIM rules.

On the other hand, the defining process of the paths expression language is sufficiently complete. Our research focused on making it as useful and extensive as possible, which is evidenced by our realization of all set objectives.

The Graphical Fundamental Relationship configuration tool , like any software, can accept many additional functionalities to improve it which could be included in future distributions. These functionalities can address issues regarding the interface and the interaction with the user, for example object representation and layouts, or can address issues concerning the core processes of the software; for example, the faster cooperation with the embedded server. Additionally, apart from system functionalities, tools which are not related to the procedure of composing path queries could be added which satisfy some secondary purpose, like a pre-existing tool which provides a schema overview and provides extra information related to the schema.

Future distributions of the FRGE could also focus on the development of an option to include an FR into the definition of another FR, instead of having to re-write the path for this FR into the new definition. As far as querying tools are concerned, along with the search results, a description on their provenance could

also be displayed. Put simply, an explanation as to why results have been returned. Another one extension of the FRGE will be the development of a feature that will give the ability to the user to collapse branches of the query paths. This feature will provide better visual representation and it will increase the usability of the query graph.

Bibliography

[1] K. Tzompanaki and M. Doerr, "A new framework for querying semantic networks," FORTH technical report TR419, Tech. Rep., 2011.

[2] ——, "Fundamental categories and relationships for intuitive querying cidoc-crm based repositories," FORTH technical report TR419, Tech. Rep., 2012.

[3] "Ontology - semanticweb.org." [Online]. Available: http://semanticweb.org/wiki/Ontology

[4] "Ontologies - W3C." [Online]. Available: http://www.w3.org/standards/semanticweb/ontology.html

[5] G. Klyne, J. J. Carroll, and B. McBride, "Resource description framework (rdf): Concepts and abstract syntax," *W3C recommendation*, vol. 10, 2004.

[6] J. Hendler, O. Lassila, and T. Berners-Lee, "The semantic web," *Scientific American*, vol. 284, no. 5, pp. 34–43, 2001.

[7] D. L. McGuinness, F. Van Harmelen *et al.*, "Owl web ontology language overview," *W3C recommendation*, vol. 10, no. 2004-03, p. 10, 2004.

[8] J. Pérez, M. Arenas, and C. Gutierrez, "Semantics and complexity of sparql," in *The Semantic Web-ISWC 2006*. Springer, 2006, pp. 30–43.

[9] T. Catarci, M. F. Costabile, S. Levialdi, and C. Batini, "Visual query systems for databases: A survey," *Journal of Visual Languages & Computing*, vol. 8, no. 2, pp. 215–260, 1997.

[10] D. Lee, J. Kwon, S. Yang, and S. Lee, "Improvement of the recall and the precision for semantic web services search," in *Computer and Information Science, 2007. ICIS 2007. 6th IEEE/ACIS International Conference on*. IEEE, 2007, pp. 763–768.

[11] "MARC STANDARDS (Network Development and MARC Standards Office, Library of Congress)." [Online]. Available: http://www.loc.gov/marc/

[12] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, "Dbpedia: A nucleus for a web of open data," in *The semantic web*. Springer, 2007, pp. 722–735.

[13] N. Manolis and Y. Tzitzikas, "Interactive exploration of fuzzy rdf knowledge bases," in *The Semantic Web: Research and Applications.* Springer, 2011, pp. 1–16.

[14] L. Ding, T. Finin, A. Joshi, R. Pan, R. S. Cost, Y. Peng, P. Reddivari, V. Doshi, and J. Sachs, "Swoogle: a search and metadata engine for the semantic web," in *Proceedings of the thirteenth ACM international conference on Information and knowledge management.* ACM, 2004, pp. 652–659.

[15] N. Dell, "Viquen: a visual query engine for rdf," *Seattle: University of Washington*, 2010.

[16] O. Ambrus, K. Möller, and S. Handschuh, "Konduit vqb: a visual query builder for sparql on the social semantic desktop," in *Workshop on Visual Interfaces to the Social and Semantic Web*, 2010.

[17] J. Borsje and H. Embregts, "Graphical query composition and natural language processing in an rdf visualization interface," *Erasmus School of Economics and Business Economics, Vol. Bachelor. Erasmus University, Rotterdam*, 2006.

[18] P. Haase, I. Horrocks, D. Hovland, T. Hubauer, E. Jimenez-Ruiz, E. Kharlamov, J. K. C. Pinkel, R. Rosati, V. Santarelli, A. Soylu *et al.*, "Optique system: Towards ontology and mapping management in obda solutions," in *Second International Workshop on Debugging Ontologies and Ontology Mappings-WoDOOM13*, 2013, p. 21.

[19] V. Lopez, E. Motta, and V. Uren, "Poweraqua: Fishing the semantic web," in *The Semantic Web: research and applications.* Springer, 2006, pp. 393–410.

[20] Y. Lei, V. Uren, and E. Motta, "Semsearch: A search engine for the semantic web," in *Managing Knowledge in a World of Networks.* Springer, 2006, pp. 238–245.

[21] "Using dublin core - the elements." [Online]. Available: http://dublincore.org/documents/usageguide/elements.shtml

[22] "Vra core 4.0 element description," 2007. [Online]. Available: http://www.loc.gov/standards/vracore/VRA_Core4_Element_Description.pdf

[23] G. M. Sacco and Y. Tzitzikas, "Dynamic taxonomies and faceted search: Theory," *Practice, and Experience. Springer, Germany*, 2009.

[24] C. Kakali, I. Lourdi, T. Stasinopoulou, L. Bountouri, C. Papatheodorou, M. Doerr, and M. Gergatsoulis, "Integrating dublin core metadata for cultural heritage collections using ontologies," in *International Conference on Dublin Core and Metadata Applications*, 2007, pp. pp–128.

[25] M. Doerr and D. Iorizzo, "The dream of a global knowledge network—a new approach," *Journal on Computing and Cultural Heritage (JOCCH)*, vol. 1, no. 1, p. 5, 2008.

[26] E. Hyvönen, T. Ruotsalo, T. Häggström, M. Salminen, M. Junnila, M. Virkkilä, M. Haaramo, E. Mäkelä, T. Kauppinen, and K. Viljanen, "Culturesampo–finnish culture on the semantic web: The vision and first results," in *Developments in Artificial Intelligence and the Semantic Web-Proceedings of the 12th Finnish AI Conference STeP*, 2006, pp. 26–27.

[27] M. Shaw, L. T. Detwiler, N. Noy, J. Brinkley, and D. Suciu, "vsparql: A view definition language for the semantic web," *Journal of Biomedical Informatics*, vol. 44, no. 1, pp. 102–117, 2011.

[28] A. Magkanaraki, V. Tannen, V. Christophides, and D. Plexousakis, "Viewing the semantic web through rvl lenses," in *The Semantic Web-ISWC 2003*. Springer, 2003, pp. 96–112.

[29] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl, "Rql: a declarative query language for rdf," in *Proceedings of the 11th international conference on World Wide Web*. ACM, 2002, pp. 592–603.

[30] L. McCarthy, B. Vandervalk, and M. Wilkinson, "Sparql assist language-neutral query composer," *BMC bioinformatics*, vol. 13, no. Suppl 1, p. S2, 2012.

[31] L. Han, T. Finin, and A. Joshi, "Gorelations: an intuitive query system for dbpedia," in *The Semantic Web*. Springer, 2012, pp. 334–341.

[32] R. Angles and C. Gutierrez, "The expressive power of sparql," in *The Semantic Web-ISWC 2008*. Springer, 2008, pp. 114–129.

[33] T. Parr, *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.

[34] ——, *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 2009.

[35] J. Nielsen, "Usability inspection methods," in *Conference companion on Human factors in computing systems*. ACM, 1994, pp. 413–414.

[36] ——, "Usability engineering at a discount," in *Proceedings of the third international conference on human-computer interaction on Designing and using human-computer interfaces and knowledge based systems (2nd ed.)*. New York, NY, USA: Elsevier Science Inc., 1989, pp. 394–401. [Online]. Available: http://dl.acm.org/citation.cfm?id=92449.92499

[37] ——, *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[38] R. Jeffries, J. R. Miller, C. Wharton, and K. Uyeda, "User interface evaluation in the real world: a comparison of four techniques," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '91. New York, NY, USA: ACM, 1991, pp. 119–124. [Online]. Available: http://doi.acm.org/10.1145/108844.108862

[39] J. Nielsen, "Usability inspection methods," in *Conference Companion on Human Factors in Computing Systems*, ser. CHI '94. New York, NY, USA: ACM, 1994, pp. 173–202. [Online]. Available: http://doi.acm.org/10.1145/259963.260531

[40] H. Desurvire, J. Kondziela, and M. E. Atwood, "What is gained and lost when using methods other than empirical testing," in *Posters and Short Talks of the 1992 SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '92. New York, NY, USA: ACM, 1992, pp. 125–126. [Online]. Available: http://doi.acm.org/10.1145/1125021.1125115

[41] C.-M. Karat, R. Campbell, and T. Fiegel, "Comparison of empirical testing and walkthrough methods in user interface evaluation," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '92. New York, NY, USA: ACM, 1992, pp. 397–404. [Online]. Available: http://doi.acm.org/10.1145/142750.142873

[42] A. Tzompanaki, "Design and implementation of a tool for formulating recall-oriented structured queries on semantic networks," Ph.D. dissertation, University of Crete, Crete, Sep. 2012.