Visualization Techniques for Rule-based Reasoning in Uncertain Knowledge Bases

submitted by

Timm Meiser

submitted

Saarbrücken, July 26, 2010

angefertigt unter der Leitung von / supervised by

Dr. Martin Theobald

begutachtet von / reviewers

Dr. Martin Theobald
Prof. Dr. Gerhard Weikum
Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, ................................................ ........................................ (Datum/Date) (Unterschrift/Signature)
Abstract

In recent years, several projects have built large semantic knowledge bases, with the help of information extraction techniques. By applying these techniques to unstructured (or loosely structured) web sites like Wikipedia, the received knowledge bases may contain uncertainty or even inconsistency to some extend. To tackle the problem of this potential data uncertainty and inconsistency, the Max-Planck Institute Saarbrücken has developed URDF. URDF is an efficient reasoning framework for graph-based RDF knowledge bases. Thereby, URDF uses a SPARQL-like query model. Moreover, URDF augments first-order reasoning by a combination of soft and hard rules. In addition, URDF applies a novel approximation algorithm for a generalized version of the Weighted MAX-SAT problem to resolve inconsistencies between the underlying knowledge base and the inferencing rules at query time. The knowledge base currently used by URDF to answer user-given queries is YAGO. Thereby, URDF produces potentially complex lineage information during its reasoning process. These produced reasoning data, which are a valuable source of information for the user, pose some tough challenges for a suitable visualization.

In this thesis, we present UViz (URDF Visualization), a complete visualization system, using URDF as reasoning backend. UViz is built in a client-server fashion. Thereby, UViFace (UViz Visualization InterFace), the visualization interface of UViz, uses Adobe Flex and Flash Player to run as a RIA (Rich Internet Application) inside a common web browser. Thereby, URDF runs on the server. Moreover, URDF applies the Flex-specific data service BlazeDS to guarantee fast data exchange between the visualization on the client and the URDF on the server. Finally, UViz integrates the Flare visualization toolkit to provide a dynamic and visually appealing graph visualization. This way, UViz is able to visualize the information produced by URDF in an intuitive and meaningful way. Thereby, UViFace supports three different operation modes to explore the visualization, examine the lineage information and compare query results with and without rule changes. Moreover, UViFace applies several state-of-the-art visualization techniques to support the user in working with the visualized data. Finally, UViFace provides a visualization that allows the user to accomplish URDF-related user interface tasks intuitively. This is demonstrated in this thesis.
Contents

Abstract v

1 Introduction 1
  1.1 Motivation ................................................. 1
    1.1.1 Background ........................................... 2
    1.1.2 Problem Definition ................................. 5
  1.2 System Overview and Outline of the Thesis ............... 9
    1.2.1 System Overview ..................................... 9
    1.2.2 Outline of the Thesis ............................... 10

2 Related Work 13

3 URDF Reasoning Framework 19
  3.1 Data and Representation Model ........................................ 21
    3.1.1 Soft Rules ........................................... 21
    3.1.2 Hard Rules ........................................... 23
    3.1.3 Queries ............................................... 24
  3.2 Reasoning Engine ............................................... 25
    3.2.1 Dependency Graph ..................................... 25
    3.2.2 SLD Resolution ....................................... 26
    3.2.3 Weighted MAX-SAT Algorithm ........................... 29
  3.3 Summary ..................................................... 29

4 Information Visualization Technologies 31
  4.1 Web Application Design ....................................... 31
    4.1.1 Client-Server Architectures ........................... 32
    4.1.2 State-of-the-Art Web Application Frameworks ......... 37
  4.2 Adobe Flex .................................................. 39
    4.2.1 Flex Programming Languages .......................... 42
    4.2.2 Flash Player Runtime .................................. 43
    4.2.3 BlazeDS - The Flex Data Service .................... 45
  4.3 Flare Visualization Toolkit .................................. 47
    4.3.1 Background ............................................ 47
    4.3.2 The Flare API ........................................ 48
## CONTENTS

4.3.3 Example Applications .................................. 51  
4.4 Summary .................................................. 52  

5 **UViz - The URDF Visualization System** 53  
5.1 Server-Side Architecture .................................. 55  
5.1.1 Data Processing and Managing ......................... 55  
5.1.2 URDF API .............................................. 57  
5.2 Client-Side Architecture .................................. 59  
5.3 System Communication .................................... 60  
5.4 Application Work-Flow .................................... 63  
5.5 Performance Measurements ................................. 66  
5.6 Summary .................................................... 67  

6 **UViFace - The UViz Visualization Interface** 69  
6.1 Information Visualization ................................. 71  
6.2 UViFace Graph Visualization ............................ 74  
6.2.1 Default Mode ........................................... 74  
6.2.2 Explanation Mode ...................................... 80  
6.2.3 Comparison Mode ...................................... 82  
6.3 Control Panels ............................................. 84  
6.3.1 Statistics Panel and Options Panel .................. 85  
6.3.2 Rule Panel .............................................. 86  
6.3.3 Query Panel ............................................. 87  
6.4 User Interface Tasks ...................................... 88  
6.4.1 Graph Exploration ..................................... 89  
6.4.2 Reasoning and Explanation ............................ 91  
6.4.3 Comparing Query Results ............................. 93  
6.4.4 Comparing Rule Changed Results ..................... 95  
6.5 Summary .................................................... 97  

7 **Discussion** 99  
7.1 Conclusion ................................................ 99  
7.2 Limitations ............................................... 101  
7.3 Outlook .................................................... 102  

8 **Appendix** 105  

**Bibliography** 110
Chapter 1

Introduction

1.1 Motivation

This Master thesis was written at the Max Planck Institute for Computer Science (located in Saarbrücken). The context of the thesis is the URDF system, which has been developed at this institute. URDF is a graph-based RDF reasoning framework for SPARQL-like queries. URDF currently uses YAGO [SKW08] as underlying knowledge base to use for its reasoning. Thereby, URDF augments first-order logic by the application of two different types of rules. The first type are soft rules with Datalog-style (potentially recursive) Horn clauses. This rule type is used by URDF to infer additional facts not present in the knowledge base. The second type of rules are hard rules in the shape of mutually exclusive sets of facts. These hard rules are used by URDF to denote that only one of the facts of such a set can be valid at the time. The hard rule in shape of bornIn(Al_Gore,x) could be used to state that all potential birth places of the entity Al Gore (possible birth places can be bound to x during the grounding process over the knowledge base) are mutually exclusive (see [The10]). Knowledge bases like YAGO can exceed a certain degree of uncertainty or even inconsistency. This means, inaccurate or contradictory information from the information source of such a knowledge base could have been integrated into the respective knowledge base. But the inconsistencies are often not obvious at first glance and can only be detected by querying the knowledge base. URDF is specifically targeted at resolving inconsistencies occurring between the knowledge base and the inferencing rules. To resolve the inconsistencies, URDF therefore applies its hard rules and a novel approximation algorithm for a generalized version of the Weighted MAX-SAT (Maximum Satisfiability) problem. This is done at query time. Consequently, URDF does not have to be changed, in case the underlying knowledge base changes.

The application of the inferencing rules and the MAX-SAT solver produces potentially complex derivation information for the query results. Con-
CHAPTER 1. INTRODUCTION

Consequently, presenting these complex information to the user in a suitable way could help the user to understand the produced results. Moreover, a suitable visualization of the result information could support the user in finding new and interesting information to be further explored. Keeping these important points in mind, the main objective of the remaining part of this chapter is to motivate the reasons leading to this thesis and to outline the remaining structure of this thesis.

1.1.1 Background

Recent projects like YAGO [SKW08], DBPedia [ABK\textsuperscript{+}07], Intelligence in Wikipedia [WA\textsuperscript{+}08] and KnowItAll [ECD\textsuperscript{+}04] built large semantic knowledge bases from resources like the Internet. Using information extraction techniques, these projects extract information from web sites like Wikipedia. These information resources are at most loosely structured (but mostly unstructured). Using information extraction techniques usually leads to uncertainty among the extracted data. Web sites like Wikipedia can have multiple different authors. Consequently, these information resources can contain false, incomplete or contradictory information. This data uncertainty and inconsistency can often not be discovered directly but only through potentially expensive inference steps (see [The10]). Therefore, projects like YAGO apply confidence values to their extracted information (in the case of YAGO in the form of RDF triples). All these factors lead to intricate relationships among the information stored in such a knowledge base. On top of that, the unstructured origin of the source data makes constructing a relational schema out of this data a hard task. Consequently, this circumstance calls for a schema-free and graph-based data model, which can be found in RDF.

The information representation language RDF (Ressource Description Framework) [RDFb] exists for more than a decade. Originally, it was designed as light-weight knowledge representation for the Semantic Web (see [NW10]). But recently, RDF has gained more and more momentum in the context of the Semantic Web and even in commercial software systems. There exist E-science data repositories which support the RDF data model as an import/export format. One of these repositories is RDFizer [RDFc]. In addition, even some online-community-related Web 2.0 platforms take the RDF data model into consideration, e.g., as exchange format. To name an example, UniProt [Uni] is such a project.

The data entities in RDF have all the same simple triple structure \textit{(subject, predicate, object)}. Assume information about the car “Ferrari Testarossa”, built by Ferrari. Among the data triples stored in a knowledge base could possibly be (see also [NW10]):
In RDF, as it is shown above, entity attributes and relations between different entities are represented by predicates. Consequently, all the RDF triples contained in a RDF repository are linked through their relationships, defined by their used predicates. This linking structure essentially represents a directed, labeled graph. Thereby, the edges represent the named connection between two entities, which are represented by the nodes of the graph. The resulting graph view of the RDF data is the easiest possible logical representation for RDF. This kind of representation is often used in visual explanations for RDF. The special features of RDF require a distinct way of querying the RDF data.

The current state-of-the-art query language to search in RDF repositories is SPARQL [SPA]. If we want to know all car manufacturers that reside in Italy, we can define the following SPARQL query:

```
Select ?n Where {
?x <hasName> ?n . ?x <hasResidence> "Italy" .
?o <hasType> "car" . ?o <isBuiltBy> ?x .
}
```

In their most basic form, SPARQL queries look like the example query above. Thereby, SPARQL queries usually have a “Select” and a “Where” statement. The selection specifies variables to extract from the result. The “Where” statement specifies the join patterns to be used for the query. Thereby, a pattern consists of the same triplet form as the RDF data triples. The constituents of such a pattern are either variables denoted by a leading “?” or constants denoted by quotation marks. The “dots” represent conjunctions of the patterns and therefore potential joins between the distinct results of these patterns in the queried database.

Finally, having such a huge RDF knowledge base like YAGO, there is now an urgent need for special systems that use SPARQL as query language.
to “search” for information in these RDF repositories. Some of such systems only try to match the patterns specified in the query with the RDF facts in the knowledge base. This way, only RDF facts existing in the knowledge base are used for the query result. But there are also more sophisticated systems, which apply logical reasoning to infer RDF facts not known to the knowledge base. In logical reasoning, rules with a precondition and an implied conclusion are applied to a set of base facts. This way, new information not known before can be inferred. Logical reasoning is not a new topic. Systems which apply logical reasoning have been used for many years. But nowadays, these systems are more than ever in the focus of the computer science world. This is due to the emerging interest in in Semantic Web-like reasoning. An example of a recent system of that kind is Sesame [BKvH03]. This system incorporates uncertainty among the data in its data processing. Another recent system dealing with uncertainty is the already mentioned URDF.

URDF is an RDF reasoning framework for SPARQL-like queries. URDF relies on ontological background knowledge, currently delivered by the YAGO knowledge base [SKW08]. YAGO consists of information gathered from the category system and the info-boxes of Wikipedia. By applying WordNet to the YAGO information, the resulting ontology represents a highly accurate source of knowledge. The YAGO data consists of entities like Al_Gore and Tipper_Gore, as well as relations between those entities, e.g. isMarriedTo. The entities and relations are combined into facts of the form isMarriedTo(Al_Gore,Tipper_Gore).

If a new query is submitted to URDF, URDF uses its reasoning engine to compute the query result. Thereby, URDF applies several techniques to dynamically extend the knowledge base and detect and resolve inconsistencies among produced facts. URDF uses two distinct processing phases to accomplish this. In the first phase, the grounding phase, URDF applies its soft and hard rules. The potentially recursive soft inferencing rules with their Datalog-style implications are used to infer new facts not present in the knowledge base. To detect inconsistencies among base facts and inferred facts, URDF uses its hard rules (hard mutual-exclusion constraints) in the form of “competitor sets”. As already mentioned, these hard rules allow to express that only one of the facts contained in such a set is allowed to be valid at the time. Both rule types are applied during the grounding phase, while the grounding is done in a top-down manner by an algorithm called SLD resolution. Thereby, all facts that are relevant for answering the query are integrated into the dependency graph, which is a subset of the knowledge base. After the grounding phase, URDF uses a distinct reasoning phase to resolve detected inconsistencies and produce the final query result. Therefore, URDF constructs a Boolean formula in CNF (Conjunctive Normal Form),

\[1\text{http://en.wikipedia.org/wiki/Main\_Page}\]
\[2\text{http://wordnet.princeton.edu/}\]
containing all grounded base facts, as well as all grounded soft and hard rules. The task of the MAX-SAT solver of URDF now is to evaluate this formula. Thereby, the algorithm has to assign truth values to the facts contained in the formula, while maximizing the sum of the weights for the satisfied clauses of the formula. The exact computation of an optimal solution for that problem is not feasible at query time. Therefore, URDF uses an approximation algorithm for the MAX-SAT problem.

The final query results computed by URDF contain all these information produced during the execution of the reasoning engine. More precisely, every fact inferred contains lineage information, which shows how the fact was derived by URDF. Moreover, every fact is attached with a confidence value as well as a truth value from the reasoning phase. The lineage information of the facts represent valuable information for a user of URDF because it explains the derivation of the query result. But the lineage information can be rather complex, depending on how many soft rules hard rules could be applied to infer the fact. Consequently, the question arises how this information can be visualized in an easy to understand and visually appealing way. How can we tackle this challenge?

This thesis presents UViz (URDF Visualization) as a solution to this challenge. UViz is a graph-based visualization system, which is built around URDF. Using current state-of-the-art technologies and techniques, UViz supports URDF in presenting and explaining the results of the reasoning process. The specific reasons for the development of UViz and the key objectives in the realization of UViz are discussed in the following subsection.

1.1.2 Problem Definition

In this subsection, we want to discuss the challenges for a good visualization, which we tackle with our UViz system, in more detail. Thereby, we differentiate between general and reasoning-related challenges. We start with the general challenges for a good visualization.

General Challenges

The first design decision concerning a new system is the type of deployment for the final application. Different types like desktop or web applications are possible. The system should be available to a large audience and be easily upgradable and maintainable. A feasible way to guarantee this is the common web browser as distribution platform and “visualization frame” for our UViz system.

Following the idea of efficiency for the visualization system, the execution environment for the visualization should be fast and highly efficient. The programming languages of the visualization component should be easily con-
CHAPTER 1. INTRODUCTION

Figure 1.1: YAGO provides only concrete answers. In a), the query *Where does Al Gore live?* is specified and delivers no answer. In b), the query *To whom is Al Gore married* delivers the one and only answer *Tipper Gore*.

nectable by APIs (Application Programming Interface) to the Java-based URDF system.

Moreover, the communication between the visualization component and the URDF reasoner has to be as efficient as possible. And, to guarantee a visually pleasing and dynamic user interface and visualization, the technologies used for the visualization should provide a rich set of graphical capabilities.

Apart from these more general challenges for the design of UViz, there exist several challenges regarding the URDF component, which will be integrated into UViz. These reasoning-related issues, which mainly drove the development of the UViz system, will be discussed next.

**Reasoning-Related Challenges**

To see the challenges in an easy way, let us have a look at Figure 1.1. This is an already existing simple user interface to query YAGO. What is visible in Figure 1.1? In a), the user wanted to know: *Where does Al Gore live in?* The answer to his query is: *There were no results.* In b), the user asked: *To whom is Al Gore married?* This time, the answer is: *Tipper Gore*. This is his current wife. Why did the YAGO interface return these results. In the first example query, YAGO did not find any fact about where Al Gore lives. In the second case, YAGO found a fact about to whom Al Gore is
CHAPTER 1. INTRODUCTION

married. Depending on the query and the information contained in YAGO, there can be more than one answer to a given query. The important fact is the circumstance that the YAGO interface will always only return an answer if the variables in the query patterns could be bound to entities of the knowledge base. This means, if RDF triples matching the query patterns are contained in YAGO, the interface will return one or more answers. If not, then the interface does not return anything. Consequently, relations among entities not directly reflected by contained RDF triples will not be detected. Thus, no new knowledge in form of new facts can be inferred from the existing ones. This, however, can be accomplished by using URDF.

In Figure 1.2, the URDF result of the same example query from Figure 1.1 a) is shown. This time, the query result contains many more result information waiting to be examined by the user. This is due to the usage of soft and hard rules. The application of the rules within URDF can produce facts not present in the knowledge base. Thereby, the lineage information produced during the reasoning are available to be used by the user as well. These circumstances have to be captured appropriately by a visualization component and lead to the following concrete reasoning-related objectives for UViz.

- First, the visualization should support the user in finding the information he or she is looking for.
- Second, the user should be able to explore the knowledge base over the visualization interface, and be able to learn interesting new information this way.
- Third, the visualization should be capable of explaining the returned query answers and their derivation (“lineage”) to the user in an easy to understand way.

Apart from these three objectives, further goals existed for the development of UViz. Having the dynamic aspect of URDF in mind, the visualization interface should provide a way to change the rules used by URDF. Moreover, the user should be able to compare results of queries before and after changes to the rules of URDF. Visual comparisons of results of different queries should be provided as well. Finally, the visualization should apply current state-of-the-art techniques to support the user in working with the visualization. Useful techniques for a graph-based visualization, like zooming, highlighting of data, or collapsing of nodes are interesting techniques to use for a visualization. Having a potentially large amount of information returned by URDF, this could support the user in examining the query results more efficiently.

Summarizing these issues just discussed, we conclude this section with the concrete questions that motivated this thesis:
CHAPTER 1. INTRODUCTION

Figure 1.2: In a), the query result of the query: *Where does Al Gore live?* is shown. In b), an excerpt from the produced lineage data to the aforementioned query is presented. The pictures are screenshots, captured from an execution of URDF in the Eclipse IDE.

- How should the visualization system architecture look like? And how should the system communication work, to guarantee a fast data exchange between URDF and the visualization?

- How does the visualization component have to be designed, and which visualization techniques can be applied to support the user?

- How can the visualization provide the exploration of the knowledge
base and the configuration of the URDF rules?

- How can the UViz visualize differences between query results in general, and between query results after changes to the URDF rules?

- And finally, how can we visualize the URDF-produced data in a meaningful and visually pleasing way?

Having discussed the background of UViz and the reasons for the development of UViz, we now want to give an overview of how we tried to tackle the aforementioned challenges with UViz. Therefore, we will already present an overview of UViz and its applied technologies. Moreover, we will give the outline for the remaining chapters of this thesis.

1.2 System Overview and Outline of the Thesis

1.2.1 System Overview

After an intensive analysis of possible technologies to use for our UViz system, we decided to enable the user to access UViz via a common web browser. To achieve this goal, we use a client-server architecture and Adobe Flex to provide the visualization frontend of UViz, called UViFace (UViz Visualization InterFace), as a RIA (Rich Internet Application). This enables the visualization component of UViz to run in the Flash Player inside a common web browser. Moreover, we use BlazeDS for the communication between the visualization frontend and the URDF backend application. To provide a visually appealing and interactive visualization, we integrate the Flare visualization toolkit into our visualization interface UViFace. To get a better idea of the UViz system, we now want to introduce all the used technologies and the system architecture of UViz.

Architecture

Client-Server Model: UViz is designed according to the client-server model. More specifically, UViz uses the rich client model (see Subsection 4.1.1). In this model, a distinct runtime environment is used as browser plugin to execute the RIA (see above) in the web browser. Thereby, the runtime that executes UViFace, the visualization frontend of our UViz application, is Flash Player.

Server: We use URDF as backend of UViz on the server. While accessing the YAGO knowledge base, managed in an Oracle database, URDF processes the user given queries and returns the computed results back to the visualization frontend of UViz.
CHAPTER 1. INTRODUCTION

Client: The frontend component of UViz, named UViFace, runs as a Flex application in Flash Player inside a common web browser. Consequently, UViFace represents the client side of UViz. Moreover, UViFace provides the user and visualization interface of UViz.

Following this rich client model, the visualization frontend is executed in Flash Player on the client side, while URDF runs as backend of UViz on the server side. This way, the visualization can run autonomically and has to call the server and URDF only to submit queries and reload data. Following, the technologies used in UViz are presented.

Used Technologies

Flex: Flex is a framework to build RIAs and is provided by the Adobe. The Flex SDK (Software Development Kit)\(^3\), which contains the Flex software framework, is thereby available as free closed source and as open-source version. Moreover, we used the Flex Builder (the Adobe development environment based on Eclipse) to develop UViz.

BlazeDS: The Adobe BlazeDS data service provides several components to easily connect Flex client applications to Java backend applications. In addition, the compact binary transmission format AMF is used by BlazeDS. By applying these technologies, we are able to provide an efficient communication within UViz.

Flare: The Flare visualization toolkit is a pure ActionScript library for graph-based visualizations that run in the Flash Player. We use this library within our visualization component UViFace.

1.2.2 Outline of the Thesis

Finally, we want to present the outline for the remaining chapters of this thesis:

- In Chapter 2, approaches and systems related to UViz will be presented.
- After presenting related work, we will examine the URDF reasoning framework in detail in Chapter 3.
- In Chapter 4, we will discuss the technologies used to realize UViz.
- In Chapter 5, we will discuss the overall architecture of UViz. Moreover, the communication within the system, as well as the implemented URDF API, will be examined.

\(^3\)http://www.adobe.com/cfusion/entitlement/index.cfm?e=flex4sdk
- Chapter 6 will present the visualization frontend of UViz, called UViFace. In this chapter, we will discuss the visualization itself. Moreover, we will introduce the user interface control panels and show several user interface tasks to exemplify the usage of UViz.

- In Chapter 7, we will conclude this thesis and give an outlook for future plans regarding UViz.

- Finally, in the appendix (chapter 8), we will present the soft and hard rules used by URDF and some queries we use for performance measurements in Section 5.5.
Chapter 2

Related Work

As the RDF data model receives more attention in the Web 2.0 community now, there is a need for suitable RDF data visualizations. There are numerous approaches of visualizing this type of data, reaching from textual representations to sophisticated graph representations. Having our own visualization in mind, we will focus on graph visualization approaches.

One system that visualizes RDF data is the IsaViz system [Isa]. IsaViz is a graphical framework for browsing and authoring of RDF models. These models can be loaded and then be represented as a graph. IsaViz uses the Jena Semantic Toolkit\(^1\) [Jen] [McB02], Java, and the visualization library GraphViz [Gra]. IsaViz provides a 2.5D user interface, zooming, and a basic graph navigation. Moreover, graphs can be built and edited, and RDF and XML data can be imported and exported. In comparison to UViz, neither are rules applied, nor can data be dynamically loaded. Essentially, this is a more simple visualization system, which only shows RDF models.

Another related system that visualizes RDF data is BrownSauce [Bro]. BrownSauce is an attempt to write a generic RDF browser similar to IsaViz, but much simpler. It was developed by Damian Steer, a former Hewlett-Packard employee. BrownSauce can be executed as a standalone application or as a web application within a server. Again, Jena is used as semantic component.

Welkin [Wel] is another RDF visualization tool. It is a graph-based RDF visualizer, which provides several features to adjust the graph. But again, there are no rules involved in the data and the visualization. Thereby, Welkin runs as a Java Web Applet.

One more RDF Visualization tool is the RDF Gravity tool [RDFa]. RDF Gravity can be used to visualize RDF/OWL Graphs and ontologies. It is a more complex system. Besides a general graph visualization, RDF Gravity provides several graph layouts. Moreover, graph interaction and navigation

---

\(^1\)Jena is implemented in Java and provides an RDF API, the aforementioned import and export functionality for IsaViz, an OWL API, and a SPARQL query engine.
is supported by several mouse events (e.g., clicking and dragging of nodes and edges of a graph) and zooming of the visualized graph. In addition, the system provides a rich set of filters to allow specific graph views. A full text search and the visualization of multiple RDF files are supported as well. Thereby, RDF Gravity uses the JUNG graph library [Jun] and Jena.

The next visualization system for RDF data graphs we want to introduce is Visual Browser [Vis]. Visual Browser is a Java application, which is executable as a desktop application or a Java Web Applet. Thereby, Visual Browser is able to visualize RDF schema data. Here, the RDF data is visualized in an entity-relationship graph. Visual Browser uses the Jena framework for its semantic knowledge gathering as well. The visualization itself is powered by an adjusted TouchGraph engine [Tou]. Visual Browser uses a forced-based layout (see Subsection 6.1). Additionally, space efficient collapsing and expanding of nodes is provided by Visual Browser.

Another system we want to present is the Paged Graph Visualization (PGV) system [DKS07] [Pag]. This visualization toolkit is written in Java, as many of the visualization systems today. In contrast to many other approaches, PGV is designed to run as a Web Applet. The underlying graph drawing engine is again GraphViz. The special feature of PGV is the way the graph is visualized. Instead of showing the whole graph and then refining the visualization to only show a small part of the graph, PGV starts with a small part of the graph and then expands this graph according to the user needs.

This closes the enumeration of simpler graph-based RDF visualization systems. Most of these systems are implemented in Java and are rather targeted at the desktop than the web browser. Moreover, the visualizations are simpler, because of the lack of visualization components for reasoning data, as it is the case for UViz. Additionally, these system do not provide dynamic data reloading to explore a certain knowledge base entity.

The next types of systems are more sophisticated in their way of visualizing the data and regarding their provided algorithms and features. Now, there are usually inferencing rules involved in the processing and the visualization as well.

An often used sophisticated framework to visualize knowledge bases is Protégé [SRND04]. Protégé is an ontology editor and knowledge-base framework, which is free and open-source. Protégé supports the Protégé-Frames and Protégé-OWL editors to build and edit ontologies. These ontologies can also be exported to several formats, e.g., RDF(S) or OWL. Protégé is written in Java, is extensible and supports plugin development. The Protégé-OWL editor is an extension of Protégé and supports the Web Ontology Language (OWL). Among the various features of the Protégé-OWL editor are the loading and saving of OWL and RDF ontologies and the integration with the already mentioned Jena toolkit. More importantly, the Protégé-OWL editor
allows to edit and visualize classes, properties, and SWRL$^2$ rules [SWR]. Moreover, the execution of reasoners (e.g., description logic classifiers) is possible with the Protégé-OWL editor. In addition to that, the editor can be easily extended. One example of such an extension is the OWLViz plugin [OWLb]. OWLViz is used with the Protégé-OWL plugin. The class hierarchies of an OWL ontology can be viewed and navigated, and the comparison of asserted and inferred class hierarchies are allowed. Thereby, inconsistent concepts are colored in red. Apart from the also well-known plugin Jam-balaya [Jam], there exist several other plugins, which show class instances, relations between them and other useful information. Among these plugins are OWLPropViz [OWLa], OntoGraf [Onta], OntoViz [Ontb], SOVA [SOV], and TGViz [TGV].

There exist also approaches to visualize Prolog$^3$ and Java programs, e.g., the Visur/Rar system mentioned in [HSGF], [SHH03] and [HSvG03], and the one explained in [Gra99].

The latter system can visualize Prolog and even Java programs and belongs to the field of Software Visualization. The system is able to show graphical representations of conjunctions and disjunctions among the predicates of applied Prolog rules. By using the layout types tree and block diagram, the system can visualize inference rules and the dependencies among the predicates contained in these rules.

The former system Visur/Rar has been used to visualize procedural and declarative programs written in Prolog and Java. These programs have to be stored in XML files to be visualized by Visur/Rar. Recently, the system has also been used to visualize knowledge bases stored in XML files [SBH04]. Thereby, the system is able to visualize the certainty of data items inferred from given rules. This way, prooftrees of given queries can be visualized.

Another visualization system for rule-based knowledge bases is the system discussed in [ZB06]. They present a system that visualizes knowledge bases and their provided rules. Thereby, the system uses the already mentioned JUNG graphics library, as well as the Fruchtermann-Reingold algorithm for force-based graph layouts [FR91]. But this system follows a special way of doing this. By submitting a set of queries, the system is able to present a graph visualization of the dependencies among the inference rules used to produce the query results. This approach is similar to our UVis system, but only the dependencies among the used rules are visualized in the system mentioned in [ZB06]. There is no entity-relationship graph present. Moreover, there is no uncertainty involved in the computation and the visualization.

---

$^2$SWRL stands for Semantic Web Rule Language.
$^3$Prolog is a general purpose logic programming language and is often used in the field of artificial intelligence and computational linguistics.
$^4$http://www.swi-prolog.org/
$^5$http://www.gprolog.org/
The last three systems we want to introduce are RDF-ListGraph [HZ08], ChainGraph [HL09] and RelFinder [HLS10] [LHSZ10] [HHL+09]. In all three projects, Philipp Heim was or has been involved. As in the UViz system, Adobe Flex is used in all three projects. In RDF-ListGraph and ChainGraph, Heim combines two visualization approaches. The idea is to provide a list representation of the information returned by the execution of SPARQL queries. In this visualization mode, the user is able to specify three different filters. These are the class filter, the property filter, and the display filter. While the class filter determines from which classes instances should be selected, the property filter selects the properties to display. Finally, the display filter applies visual filters to hide, show and highlight specific information in the visualization. If the user wants to see the output of applied filters, he can switch to a graph view. Here, nodes represent instances and edges represent the properties shared by these instances. The used layout algorithm is again the Fruchtermann-Reingold algorithm [FR91]. One special feature of RDF-ListGraph is the reduction of the visual data. Only properties that are shared by at least two instances are displayed. In addition, to get a cleaner layout, the shared properties build distinct nodes. Consequently, the number of nodes increases and the number of edges doubles, as now two instances are connected by two join edges and a property node in between. RelFinder, on the other hand, pursues a slightly other approach. The main idea here is that the user is provided with an interface to define two objects present in the knowledge base. These objects represent a left and a right start point. From these start points, an algorithm tries to find objects and relations between the stated objects. This produces a dependency graph between the previously defined objects. In addition, if homonyms, polysemes, or incomplete user input is present, regarding the specified start objects, a disambiguation process is started. A special SPARQL query is then sent to the underlying knowledge base and returns knowledge base entities potentially matching the initial objects. This is done while the user states the objects. Moreover, several graph visualization related features are provided.

Having discussed all of these system related to our UViz system, we want to point out the differences again. Many of these systems are implemented as desktop applications, whereas UViz uses a client-server architecture to provide the visualization frontend as rich client application via the Flash Player in a web browser. Most of these systems use Java graph libraries while UViz uses Flex, to leverage the Flash Player capabilities. Only some of these system query the underlying information in a dynamic way. We provide a query interface to state even complex queries consisting of several constituents. Moreover, our UViz system supports features like dynamical loading of data related to a certain instance. In addition, UViz provides a graph visualization of two mixed granularities, an entity-relationship graph and a fact-lineage tree. Additionally, we deal with uncertainty and inconsistency in the used
data. Moreover, we reflect this uncertainty and inconsistency in our visualization in an understandable way.

To the best of our knowledge, none of the systems mentioned in this chapter deals with uncertainty and inconsistency of its used data in its visualization. Additionally, only some of the systems provide the set of features to navigate and explore the visualized graph the way we do. In the following chapter, we will discuss in more detail our underlying reasoning framework URDF.
Chapter 3

URDF Reasoning Framework

In this chapter, we want to discuss in more detail the URDF reasoning framework.

URDF is an efficient reasoning framework for graph-based RDF knowledge bases. Thereby, queries in URDF are stated in a SPARQL-like fashion (see Subsection 1.1.1). To augment first-order reasoning, URDF applies two different types of rules to its reasoning. The first type of rules are soft rules. These rules contain Datalog-style (recursive) implications and have a weight attached to them. This weight reflects the “importance” of the rule. The second type are hard rules. These types of rules form sets of mutually exclusive facts (see Section 1.1). To produce results to a given query, URDF accesses an underlying knowledge base, which is currently YAGO [SKW08].

URDF could use its expressive rule formulations to compute all possible derivable facts in advance. But the framework would not remain dynamic, regarding updates to the knowledge base. The system would have to bottom-up compute all possible facts every time the knowledge base is updated. This is computationally very expensive. Instead, URDF uses its set of different rules to return the best-possible results in an efficient top-down manner at query time.

Thereby, one of the main objectives for the development of URDF was to be able to resolve potential inconsistencies arising between the knowledge base and the inferencing rules. The inconsistency stems from noisy rules and the often noisy information that is produced by knowledge extraction techniques applied to sources like the Internet. Because disambiguation problems can produce false assertions, and statements on web pages can be inaccurate, the knowledge stored in databases can contain a severe degree of uncertainty or even inconsistency. Therefore, URDF applies a novel approximation algorithm for a generalized version of the Weighted MAX-SAT problem (see Subsection 3.2.3). This algorithm integrates the results of the application of the soft and hard rules during the reasoning into its computation.
As already mentioned, the ontological knowledge base used by URDF is the YAGO knowledge base [SKW08]. It is an RDF knowledge base built from Wikipedia and WordNet. To demonstrate the previous inconsistency statement and to show the way the graph-based RDF data looks, we will now present a small knowledge base example (see Figure 3.1). This example will be used throughout the whole chapter to demonstrate the capabilities of URDF.

In Figure 3.1, we can see different information about computer scientists. More precisely, among other things, we identify the following information:

- Ullman, Chaudhuri and Maier are computer scientists,
- Ullman works at Stanford,
- Ullman supervised Chaudhuri and Maier,
- and with a certain possibility (the confidence in the particular knowledge base fact, originating from the data extraction process), we know Maier could have graduated from Princeton, and Chaudhuri may have graduated from Princeton and Stanford.

With the application of the soft and the hard rules, URDF could derive further facts about the presented entities and distinctively state which facts
are actually valid. Without further restricting constraints, there is no formal inconsistency\(^1\) in the example knowledge base so far. This means, there is no contradiction present in the knowledge base. We will go into more detail about the data inconsistency and how URDF tries to resolve it in the next sections of this chapter. To understand how URDF tries to achieve this goal and how URDF computes the final possible answers, we follow this outline for the remaining sections:

- In Section 3.1, we will talk about the data and representation model. To be more specific, the query model and the two kinds of different rules are discussed here, namely the soft rules and the hard rules.

- In Section 3.2, the reasoning steps and the resolution of inconsistencies will be discussed.

- We conclude this chapter with a small summary in Section 3.3.

In the following section, the model for the soft rules, hard rules, and the queries will be presented.

### 3.1 Data and Representation Model

Before we can go into more detail regarding the queries and rules, we first have to define several terms. Following the definitions from [The10], we state:

A knowledge base \( KB = \{ F, C, S \} \) is a triple that consists of RDF base facts \( F \), soft clauses \( C \) and hard rules \( S \). An RDF graph is a labeled multigraph, which is directed. Its nodes are entities (e.g., individuals and literals), and the labeled edges represent the relationships between these nodes (entities). Consequently, we can have the two nodes (entities) \( Ulman \) and \( Stanford \), which are connected by the labeled edge (relation) \( worksAt \). Formally written, let's assume we have a finite set of relations \( Rel \) and a finite set of entities \( Ent \supseteq Rel \). Then, the resulting RDF graph is a set of triplets (facts) \( F \subset (Rel \times Ent \times Ent) \).

Let us now discuss the model for the soft rules in URDF.

#### 3.1.1 Soft Rules

Soft rules represent potentially recursive first-order logic rules over RDF facts. This kind of rules adds advanced capabilities like transitivity and type inference to the reasoning engine. Soft rules do not have to hold for all facts, the rules can be violated. We define several terms to explain the model of the soft rules (see [The10]).

\(^1\)http://en.wikipedia.org/wiki/Inconsistency
A soft rule is called grounded soft rule over a set $F$ of RDF facts, if the rule is a set $C \subseteq F$ of facts and each atomic fact $f \in C$ is marked as either positive or negative and thus represents a literal. Such grounded soft rules represent clauses, which are disjunctions of literals. Using the possible worlds semantic, we have the following situation. A clause is called satisfiable if we have a set of facts $F$ and a possible world $p$ with the total function $p: F \rightarrow \{true, false\}$, such that none of the hard rules is violated and the following holds:

$$\exists f \in C, \quad p(f) = true \quad \text{or} \quad \bar{f} \in C, \quad p(\bar{f}) = false$$ (3.1)

Consequently, ungrounded soft rules do not fulfill all of these conditions. Ungrounded soft rules are Horn rules where at most one positive literal is allowed. URDF allows simple arithmetic predicates that are “closed” within the rule. Thus, these predicates can be evaluated (grounded) from the given query bindings at rule processing time. An ungrounded soft rule $C'$ over $F$ implicitly represents all grounded soft rules $C$ that have their variables substituted by entities (see [The10]). Ungrounded soft rules are solely used for readability reasons, only grounded versions are used as input for the MAX-SAT algorithm of URDF (more details in Subsection 3.2.3).

Horn clauses with only one positive literal can be transformed into rules with implications, having a head literal and a body consisting of several conjunct literals. Thereby, these body literals implicate the head literal. Thus, we obtain the following rule versions (using the knowledge base from Figure 3.1) (see [The10]):

$$C'_{\text{example}} : \text{hasAcademicAdvisor}(a, b) \land \text{worksAt}(b, c) \quad \rightarrow \quad \text{graduatedFrom}(a, c) [0,4]$$

$$C_{\text{example}_1} : \neg\text{hasAcademicAdvisor}(\text{Chaudhuri, Ullman}) \lor \neg\text{worksAt}(\text{Ullman, Stanford}) \lor \text{graduatedFrom}(\text{Chaudhuri, Stanford}) [0,4]$$

$$C_{\text{example}_2} : \neg\text{hasAcademicAdvisor}(\text{Maier, Ullman}) \lor \neg\text{worksAt}(\text{Ullman, Stanford}) \lor \text{graduatedFrom}(\text{Maier, Stanford}) [0,4]$$

The first rule formulation represents an ungrounded soft rule. The second and third rule formulations show the two possible (regarding the given knowledge base) grounded versions of the first rule. Thereby, the variables
CHAPTER 3. URDF REASONING FRAMEWORK

$a, b$ and $c$ are replaced by their bound entities from the knowledge base. How are these variable-entity bindings derived?

For every query literal or body literal of a soft rule $C'$, URDF tries to find matching facts (literals in RDF triple form that only contain constants). These facts are either received from the knowledge base directly, or they are inferred by evaluating the soft rules through a recursive algorithm called \textit{SLD resolution} (see Subsection 3.2.2) in Prolog and Datalog. Thereby, body literals of a soft rule $C'$ are only replaced by soft rule inferred facts if a respective body literal can be matched with the head literal of another soft rule $C'_1$.

As we can see from the rules above, each soft rule is extended by a non-negative, real-valued weight, which reflects the “importance” of that respective soft rule. How are these weights derived? URDF computes the weights by considering the proportion of right cases over false cases for soft rules (see [The10]). This means, URDF computes the conditional probability that a soft rule is true, given that its body is true. This way, we receive a weight in the range of $[0, 1]$, which reflects the probability that a distinct soft rule is correct in the knowledge base. The soft rule weights are propagated to all grounded versions of a particular rule.

Summarizing this subsection, soft rules can help to infer new facts. But what happens if there are two contradicting facts? How can this problem be solved? Then, we need additional constraints. These constraints exist in URDF. They are called \textit{hard rules} and are discussed in the next subsection.

3.1.2 Hard Rules

In a knowledge base like YAGO, it is possible that contradictory facts about entities are contained. A person could be born on more than one particular day or in more than one birthplace. The uniqueness of certain relations for an entity have to be always satisfied. This means, there should be a mechanism to guarantee that certain relations, e.g., the \textit{isMarried} relation, are not valid for a set of facts containing such a relation and the same entity more than once. Consequently, a person should not be born on two different days.

That is the reason UDRF introduces \textit{hard rules} to restrict the reasoning engine to logically correct answers. Hard rules $S$ represent clauses that form mutually exclusive sets of facts, so-called \textit{competitor sets}. These rules are called hard rules, because they are “hard” constrains and may not be violated during the reasoning process. Consequently, they have no weight attached to them. As it is the case for the soft rules, the hard rules can be differentiated into grounded and ungrounded versions as well. Again, only the grounded versions are used as an input to the MAX-SAT solving algorithm of URDF. An ungrounded hard rule could look like $\textit{graduatedFrom}(\text{Maier}, y)$. This ungrounded hard rule represents all grounded atoms that contain a uni-
versity where Maier could have graduated from. But only one of these atoms is allowed to be valid at the time (can be assigned \textit{true}). Consequently, using the knowledge base from Figure 3.1 and its base facts and derived facts, the resulting hard rules have the following form (see \cite{The10}):

\begin{align*}
S_{example_1} & : \{ \text{graduatedFrom(Chaudhuri,Stanford)}, \\
& \quad \text{graduatedFrom(Chaudhuri,Princeton)} \} \\
S_{example_2} & : \{ \text{graduatedFrom(Maier,Stanford)}, \\
& \quad \text{graduatedFrom(Maier,Princeton)} \}
\end{align*}

They can also be written as conjunctions of binary Horn clauses that consist of only negated literals (this is omitted here).

After having presented the models for the soft and hard rules, we want to introduce the query model of URDF now.

### 3.1.3 Queries

The general structure of SPARQL-like queries was discussed in Subsection 1.1.1. URDF uses this kind of conjunctive queries to process the information from the knowledge base and return query answers to the user. In Figure 3.2, we can see the query graph of the example query: \textit{Which computer scientist was supervised by Ullman, and where did he or she graduate from?}

As we will learn later on, queries in URDF have to contain at least one constant. This is a restriction to reduce the amount of data the reasoning engine has to investigate to find the query answers. In the example query from Figure 3.2, the char \$ denotes a variable. Having stated the query, URDF now has to find isomorphic embeddings of such a query graph in the knowledge base. In doing so, each of these embeddings delivers distinct bindings to the contained variables of the query. In the example from Figure 3.2, we would obtain two results for the two possible computer scientists having been advised by Ullman.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{query_graph.png}
\caption{An example query graph (see \cite{The10}).}
\end{figure}
To add expressiveness to the processing of queries and to resolve inconsistencies among the data, URDF then applies the aforementioned soft and hard rules. After introducing the data model of URDF, we will now focus on the way URDF processes information and applies its rules and MAX-SAT solver to infer new facts and resolve inconsistencies.

### 3.2 Reasoning Engine

The URDF reasoning framework consists of several components, which provide their functionality to different processing steps (see [The10]). The core system conforms to a common SPARQL engine. This engine is extended by first-order soft and hard rules (see Subsections 3.1.1 and 3.1.2). To bind the variables contained in the query literals, the reasoning engine applies a top-down inference algorithm called SLD Resolution (see [The10]). In addition, URDF uses a novel approximation algorithm for a generalized version of the Weighted MAX-SAT problem. This is done to resolve inconsistencies and will be explained in Subsection 3.2.3. Moreover, to reduce the amount of information to process, URDF considers only a small fraction of the possible facts. This is accomplished by building a connected and directed subgraph of the knowledge base. This subgraph, called *dependency graph*, contains only the facts necessary for solving the given query. In the following subsection, we will therefore define the *dependency graph*.

#### 3.2.1 Dependency Graph

Before we can define the dependency graph, we have to discuss the reasons URDF constructs this graph. The used knowledge base can be potentially huge. Consequently, the inclusion of all base facts in the final reasoning step would negatively influence the execution time of the reasoning engine. As a consequence, the user would have to wait for answers considerably longer. To provide reasoning at query time, URDF considers only this small subgraph of the knowledge base. The dependency graph is the input for the later applied approximation algorithm for the Weighted MAX-SAT problem. The question now is: “How is the dependency graph constructed?”

The *dependency graph* $D$ emerges from the repeated application of the potentially recursive rules, as a predicate can occur in the head literal, as well as in the body literal of a distinct soft rule. In addition, soft rules can also produce result facts as input for other soft rules. Thereby, the formal definition of the dependency graph looks as follows (see [The10]):

**DEFINITION 1.** Dependency Graph. Given a knowledge base $\mathcal{KB} = \{\mathcal{F}, C, S\}$ and a conjunctive query $Q$, then:

- All possible groundings $f \in \mathcal{F}$ of the query atoms $q \in Q$ are facts in the dependency graph $D$. 

• If a grounded fact $f_n$ is in $\mathcal{D}$, then all grounded facts $f_1, ..., f_{n-1}$ of all grounded soft rules $C \in \mathcal{C}$, in which $f_n$ is the head, are also in $\mathcal{D}$.

• If a grounded fact $f$ is in $\mathcal{D}$, then all grounded facts $f_1, ..., f_k$ of the grounded hard rule $S \in \mathcal{S}$, which are mutually exclusive to $f$, are also in $\mathcal{D}$.

Definition 1 defines all relevant aspects of the dependency graph and also presents the recursive construction of the dependency graph. This construction is accomplished by the already mentioned recursive algorithm, called SLD resolution. This algorithm is explained in the following subsection.

### 3.2.2 SLD Resolution

The grounding phase of URDF, in which all facts that are necessary to answer the query are produced, is computationally expensive. To keep query processing and rule evaluation as tractable as possible, several restrictions are enforced on queries and rules (see [The10]).

The first restriction states that queries have to contain at least one constant. This restriction ensures that the constructed dependency graph always forms a connected graph over all the possible facts. Moreover, this makes sure that the dependency graph is the same for all different grounded versions of the query. In addition, the SLD resolution does not have to start with an empty set of variable bindings. The second restriction forces variables and constants of head literals of soft rules to also occur in at least one of the body literals. This restriction prevents URDF from the need to instantiate a variable with all possible entities of the knowledge base. The third restriction forces the application of the closed-world assumption. By assuming a closed world, all facts which cannot be grounded in the database (including the application of the rules) cannot be true. Thus, these facts are not integrated into the dependency graph of the query.

An overview of the complete query execution process, divided into the grounding and the reasoning phase, is depicted in Algorithm 3.1. In the first four lines, the dependency graph is constructed, by using the aforementioned SLD resolution. In Line 3 and 4, all grounded soft and hard rules produced in one step of the SLD resolution are expanded into the respective set of grounded rules. In Lines 5 to 7, URDF builds a Boolean formula in CNF (Conjunctive Normal Form). This formula contains all the grounded base facts and grounded soft and hard rules. Then, the Weighted MAX-SAT problem over this Boolean formula has to be solved. By expanding the dependency graph in this way, consistent query answering is guaranteed. This means, the truth assignments for the facts in the dependency graph are the same as if the truth assignments for these facts would be obtained by applying the MAX-SAT solver over the entire knowledge base.
CHAPTER 3. URDF REASONING FRAMEWORK

Require: A knowledge base $KB$ with base facts $F$, soft rules $C$, and hard rules $S$

Require: A non-grounded conjunctive query $Q$

Initialize the dependency graph $D = \emptyset$

Ground all $q \in Q$ via SLD resolution (Alg 3.2) and add results to $D$

Expand $\mathcal{C}$ by grounded soft rules embedded in $D$

Expand $\mathcal{S}$ by grounded hard rules embedded in $D$

Construct CNF from all grounded soft rules $\mathcal{C}$, grounded hard rules $\mathcal{S}$, and base facts $f \in D \subseteq F$

Solve weighted MAX-SAT over the CNF

return $D$ with truth assignments $p(f)$ to all facts $f \in D \subseteq F$

Algorithm 3.1: URDF Reasoning Framework (see [The10]).

Require: A knowledge base $KB$ with base facts $F$, soft rules $C$, and hard rules $S$

Require: A non-grounded conjunctive query $Q$

Require: A dependency graph $D$

Initialize a temporary dependency graph $D' = \emptyset$

Sort all atoms $q \in Q$ in ascending order of estimated selectivity

For all $q \in Q$

· If there is a match $f \rightarrow q$ in $F$

· Add $f$ to $D'$

· For all $C \in \mathcal{C}$

· · Let $f_n$ be the head and $f_1, \ldots, f_{n-1}$ be the body of $C$

· · If the head $f_n$ of $C$ matches $q$

· · · Ground all $f_1, \ldots, f_{n-1}$ recursively via SLD resolution and add results to $D'$

· · · Add $f_n$ to $D'$

· If there is a conjunctive grounding of all $q \in Q$

· For all $S \in \mathcal{S}$

· · If a fact $f \in D'$ matches an ungrounded atom $q \in S$

· · · Ground $q$ recursively via SLD resolution and add results to $D'$

· Expand $D$ by all grounded facts in $D'$

Algorithm 3.2: SLD Resolution with Soft and Hard Rules (see [The10]).

Algorithm 3.2 presents the SLD resolution in detail. Here, the resolution algorithm iterates over all yet ungrounded query atoms, after rearranging them according to their selectivity. Query atoms and rule body atoms are processed the same way. Thus the algorithm recursively calls itself, first with query atoms, then with rule atoms. If a matching binding for a query atom is found in $KB$, then this fact is added to the temporary dependency graph $D'$. Additionally, all soft rules are examined for a head atom that matches the given query atom and then grounded, in case a match exists. First, the resulting body facts are added to $D'$, then the head fact (Lines 6 - 10). The
hard rule atoms are grounded separately in an additional step. If matched completely, all resolution result facts are added to \( D \).

Before we present the MAX-SAT algorithm in more detail, it is necessary to explain the lineage information of a fact. This is needed for a better understanding of the visualization presented in Chapter 6, especially in Section 6.4.2.

For every fact that is grounded by applying one or more hard or soft rules, URDF produces lineage information. The lineage information for a distinct fact contains all the grounded soft and hard rules that could be applied to produce that respective fact. And because the SLD resolution works recursively in a top-down manner, such lineage information has the shape of a lineage tree. This means, the lineage information corresponds to the derivation of the respective fact. Thereby, the different levels of such a lineage can contain different lineage information. The root node of the lineage of a fact is the fact itself. Then, in a recursive manner, nodes in this lineage tree can be differentiated into three cases:

- If the current grounded fact \( f \) of the lineage tree has no lineage information attached, then this fact is in the knowledge base. This means, there is no further subtree. Consequently, this fact is a leave of the lineage tree.

- In case \( f \) contains several grounded soft rules as lineage information for one distinct level of the lineage tree, then there is a subtree below \( f \). Multiple grounded soft rules are possible for the derivation of the same fact. In this case, \( f \) represents the parent node for the connected subtree. The child node (root node of the respective subtree) of \( f \) is a node denoted with a “\( \lor \)”. This node indicates that the \( f \) could be derived through grounded instances of several distinct soft rules. Consequently, the node denoted by a “\( \lor \)” represents a disjunction of the grounded soft rules that inferred \( f \). The child nodes of a disjunction node can only be nodes representing grounded soft rules.

- A grounded soft rule is represented by a node denoted with a “\( \land \)”. This node indicates that the fact \( f \) (which is represented by a node in the lineage tree) was inferred through the conjunction of several facts, the body facts of the aforementioned grounded soft rule. Moreover, all body facts of the respective grounded soft rule are child nodes of the node denoted with the “\( \land \)”. A conjunction in the lineage tree therefore represents the application of a grounded soft rule for a respective fact \( f \). This means that fact nodes can be followed by a disjunction node (several grounded soft rules for the fact), by a conjunction node (only one grounded soft rule possible for the fact), or by nothing (fact exists
in the knowledge base). A disjunction node, on the other hand, can only be followed by a conjunction node.

Consequently, we obtain a lineage tree consisting of nodes for the respective facts and nodes for the disjunction of grounded soft rules (multiple grounded soft rules for a distinct fact). In addition, a grounded soft rule is represented by a conjunction node and the connected fact child nodes. Next, the MAX-SAT solving algorithm of URDF is discussed in more detail.

3.2.3 Weighted MAX-SAT Algorithm

The MAX-SAT solving component of URDF provides an algorithm to solve a generalized version of the Weighted MAX-SAT problem\(^2\). After the SLD resolution in Algorithm 3.1 is finished, a CNF (see above) from all grounded soft rules, hard rules, and base facts is constructed. Now, this Boolean formula has to be evaluated. The task of the MAX-SAT solving algorithm is then to produce these truth assignments for all atomic facts contained in the formula, while maximizing the weights for the satisfied clauses. Thereby, the constraints given by the hard rules may not be violated. And because this task is computationally expensive (it is \textit{NP-hard}), it is not feasible for URDF to solve the MAX-SAT problem at query time in an exact manner. Thus, URDF uses a smart approximation algorithm to solve the Weighted MAX-SAT problem.

3.3 Summary

The URDF system, which has been developed at the Max Planck Institute for Computer Science, is an efficient reasoning framework for RDF data with uncertainty. It currently uses the ontological knowledge base YAGO as underlying information source. Moreover, URDF augments first-order reasoning by a combination of soft and hard rules. The framework computes facts to build the query results either by obtaining the facts directly from the knowledge base or by deriving them. To resolve occurring inconsistencies among the produced information, URDF uses a novel approximation algorithm for the Weighted MAX-SAT problem. By applying all these techniques, URDF achieves to deliver sophisticated query results.

After the introduction, the presentation of the related work, and the discussion of the URDF framework, we will show the technologies used to build our visualization system in the next chapter.

\(^2\)This is a generalized version of the Weighted MAX-SAT problem because hard rules are added to the CNF to solve.
Chapter 4

Information Visualization Technologies

In this chapter, we want to introduce the technologies used for the realization of the UViz system. Because of the specific features (e.g., graph-based visualization) of UViz and the used reasoning framework URDF, UViz belongs to the field of Information Visualization. Information visualization is a broad field, which spans countless subtopics, techniques, and algorithms. We want to focus on the topics that are related to UViz and this thesis.

As mentioned in Section 1.2, we use a common web browser and the Internet to provide the visualization frontend of our UViz system to the user. To build our web-enabled application, we use a client-server architecture as the “frame” of our system. Therefore, we will present in more detail the general architecture behind the client-server model in Section 4.1. Moreover, todays most common versions of this model will be discussed in this section as well. In addition, the current state-of-the-art web application frameworks will be introduced. Having discussed the general architecture and several current web application frameworks, we will then continue with a more detailed investigation of the Adobe Flex 3 “ecosystem” in Section 4.2. Thereby, technologies like Flash Player and BlazeDS will be discussed. In Section 4.3, we will present the visualization library that provides us with a suitable graph-visualization for UViz, the Flare visualization toolkit [Fla].

After this overview of the chapter, we will now discuss the general architectures of todays web applications and the current state-of-the-art frameworks to build interactive and visually appealing web applications.

4.1 Web Application Design

Today, the Internet is the leading source of information and offers a wide range of different types of web sites and web applications. It is possible to
watch movies online, listen to music, do shopping, sell belongings, use a
search engine to search for interesting web sites, and plenty of other things.
Many of the web sites on the Internet today are no longer simple electronic
pages to read, but rather rich user interfaces with powerful background ap-
plications, processing the more and more complex user tasks. Nowadays,
web applications often offer many of the features that were reserved only for
desktop applications a few years ago. There are differences in the way the
various web applications work, communicate and look, but the architectures
and design features follow general models, though.

To publish a web application to the Internet, it is necessary to *host* a web
application. The host of the web application takes care of the management of
user requests to the web application, security constraints, user authentication
for the application, access to a database, and many more tasks. Thereby, the
user interface and the logic behind this interface can be separated within a
web application. This naturally leads to the use of a client-server model for
the architecture of web applications. Thereby, the client provides the user
interface to access the web application, whereas the server usually executes
and manages the application logic that lies behind the user interface.

To get more insight into the most common client-server models used
by todays web applications and the current state-of-the-art web application
frameworks, we will examine the models and the frameworks in the following
subsection.

### 4.1.1 Client-Server Architectures

As already mentioned, a web application is usually divided into two logical
components, the user interface and the application logic that drives these in-
terfaces. This logical decomposition of the application opens the door for an
architectural decomposition. To use the Internet-provided data bandwidth
and the processing resources of the involved hardware efficiently, web applica-
tions are built following a client-server model. The client and the server
themselves are divided again into two parts, a hardware client(server) and a
software client(server). The software is running within the respective hard-
ware. Thereby, the software client requests data from the software server,
and the server answers in return.

A general client-server model for web applications is depicted in Figure
4.1. In case of web applications, the server is called *Web Server* or *Web
Application Server*. In Figure 4.1, we have a 2-Tier architecture, in which
we the server side (tier 2) processes the requests from the client (tier 1)
and manages the database (tier 2) accesses. The web server *hosts* the web
application and publishes it to the Internet. Thereby, the web server accesses
the database to receive requested data. The physical client in this case is a
common desktop computer, the software client is a common web browser,
like the Internet Explorer. Communication between the web server and the
A general web application execution involves some sort of user interaction in the user interface on the client side. This forces the client to request data from the server, by using HTTP. The server processes the request and sends the result back to the client. In the most basic scenario, which complies to the older and so-called Web 1.0, the returned answer is an HTML page. This is a data format the browser understands. The HTML page is either taken from a predefined pool of available HTML pages or dynamically created by an application running on the server. In general, the client then has to wait for the answer to arrive. This is a rather static model, where the data request takes place for every change in the web application. After the answer has arrived, the user interface on the web application is updated (refreshed). This model represents a synchronous data exchange between the server and the client. The client does nothing until the answer arrives. This web application schema is usually used for static web pages, rather than interactive web applications (in the true sense of the word application).

There are variations to that schema and different ways of distributing the processing of the data and the application logic between the client and the server side. The three main variations are the Thin Client model, the Fat Client model, and the Rich Client model. The already shown model with the synchronous data exchange follows the Thin Client idea. Before we discuss this model, we will first introduce the Fat Client model.
CHAPTER 4. INFORMATION VISUALIZATION TECHNOLOGIES

Fat Clients

In a **fat client** model, the client uses more processing resources and provides more functionality than in the **thin client** model. The fat client runs the user interface and does all of the application computation by itself and only accesses the server to request data. The consequence of this model is that there is less network traffic because most of the processing is done on the client. The drawback is the restriction of the client’s hard and software, because the fat client model requires more processing power on the client side. In addition to that, a fat client web application usually implies a rollout process to bring the “fat” application to the user. A good example for a “web application” that follows the fat client model is Google’s **Google Earth** application\(^1\) (see [Mül09], page 20). Google Earth provides a rich set of features, like 3D buildings, exploration of the Mars in 3D, exploration of the oceans in 3D, and plenty of other more or less processing intensive capabilities. It uses the Internet on occasion, but needs to be downloaded and installed. In addition, Google Earth requires a lot of processing power on the client to run properly. This is due to the fact that all the processing is done on the client.

Google provides also an alternative to the computationally demanding Google Earth, the far simpler **Google Maps**\(^2\). This application follows the opposite direction and leads us to the more common client-server model for web applications, the **Thin Client** model.

Thin Clients

The **thin client** model pursues the common idea behind the client-server model. The physical thin client can be a more processing weak device like a cellphone, a PDA, or a common desktop computer (an older machine is acceptable). The software client is an web browser and often only displays the user interface, with rather moderate processing to be done. Google Maps uses such a model for different reasons.

First, the client machine does not necessarily have to provide a lot of processing power. This is due to the fact that most or all of the computation-intensive tasks can be accomplished on the server. Second, by using a common web browser to run the user interface on the client, the web application is available on many devices and at many locations. Third, the user is always provided with the most current version of the program, without the need to reinstall or update the application manually every time. Fourth, the user can use the browser as a familiar environment to access the web application. Several other strong points advise the thin client model for web application design, only the most interesting ones are mentioned above.

---

\(^1\)http://earth.google.com/, 30.06.10
\(^2\)http://maps.google.com/, 30.06.10
Google Maps provides most of the aforementioned aspects, as it is available from every possible web browser, does not need to be installed, and shows its content in a simple graphical interface. In addition, such a type of web application can be accessed fast by only typing in the particular web site address and waiting on the response of the server. A couple of years ago, most of the web applications were rather static and acted more like electronic pages, as already mentioned at the beginning of this section. Google Maps is only one example for more sophisticated web applications, as it applies one of the currently mostly used approaches for web application development, the Ajax model. Ajax stands for Asynchronous JavaScript and XML. Ajax is no technology on its own, but a programming paradigm. A web application which follows the Ajax paradigm uses a compound of several techniques. The frame of the application that is shown in the browser is XHTML (see [SB09], Chapter 3). XHTML is supported by CSS (Cascading Style Sheets). CSS changes the look of the HTML pages. While XHTML is used to show the application content in the browser, JavaScript (a scripting language not related to Java) is used for application logic processing on the client side. JavaScript functions are executed within the JavaScript engine of the respective web browser. JavaScript loads data into the web browser by XMLHttpRequest requests to the server. The server then returns the produced data back to the client in one of the supported data formats, e.g., XML. All the communication is done asynchronously. This communication model lets the user interact with the user interface, while data requests take place in the background. In addition to that, less server calls are necessary. This is because small changes in the user interface can be accomplished by processing JavaScript functions, rather than calling the server for every small change and reloading the complete page. Consequently, Ajax-enabled web applications provide richer user interfaces and more interactivity. Among the myriad of web applications following the Ajax paradigm are web applications like Amazon, eBay, or the aforementioned Google Maps.

Some people claim that the Ajax paradigm belongs to the Web 2.0, because rich user interfaces are possible and asynchronous data exchange is provided. Other people believe that Ajax applications still belong to the old Web 1.0 (see [BEH+09], page 4). Nevertheless, Ajax was a major leap for the acceptance of web applications as serious applications accessible through the web browser. Apart from the current wave of Ajax-enabled web applications, we have a third type of client-server model for web applications, the so-called Rich Client.

---

3. JavaScript functions can be executed on the server as well.
4. Today’s web browsers like Firefox or Opera constantly improve their JavaScript engines to beat the opponents and publish performance benchmarks of their engines.
CHAPTER 4. INFORMATION VISUALIZATION TECHNOLOGIES

Rich Clients

A rich client is essentially a hybrid form of the thin and the fat client. Web applications following the rich client approach provide the best of the two worlds. This means, a rich client offers a rich and interactive user interface and runs autonomically on the client machine. Many functionalities can already be computed on the client. Consequently, more than just the user interface-related operations can be accomplished on the client, while the client application runs in its own client runtime. These client capabilities can reduce the network load substantially. Nevertheless, processing-intensive computations can be done on the server, as the rich clients usually provide several services to connect the client to the server in an asynchronous and efficient way. Thus, rich clients can accomplish many tasks on their own. If processing intensive operations are necessary, these can be realized on the server, though. Consequently, the rich client model provides desktop-like applications, accessible via the web browser. A rich client application is also called RIA (Rich Internet Application) (see [Mühl09], page 21). The technology behind RIAs is constantly evolving, and the development tools to produce rich client applications become more and more powerful (see [SB09], chapter 3). As already mentioned, a distinct runtime environment is needed to execute such a RIA within a web browser (see [Mühl09], page 21-23).

Although Ajax-enabled web applications are often called RIAs, Balderson et al. (see [BEH+09], page 4) claim that the main purpose of Ajax applications still is the textual representation. On the other hand, native RIAs have the purpose of providing rich media capabilities to the user. True RIAs reside entirely on the client, running in their own runtime environment, like a browser plugin. Usually, RIA development is accomplished by the use of mature development tools and powerful programming languages, provided by the company behind a distinct RIA framework. Because of the more advanced programming languages behind the current RIA frameworks, the development of RIAs follows more the traditional way of developing desktop software.

Some of the drawbacks of RIAs are that the runtime has to be downloaded and the application runtimes potentially consume more memory and processing resources. There are a few other drawbacks, but these are minor ones. The advantages of using the rich client model for a rich and dynamic web application prevail, though. In addition, disadvantages like the download of the runtime or the application itself are negligible. This is due to the fact that this downloading only has to be done if a new version of the RIA runtime is published. Moreover, the runtimes and programs usually consist of only a few megabytes. Additionally, the use of RIAs and their own runtimes prevents developers of adjusting their Ajax applications to be executed properly by the major web browsers. This has to be done because there are no standards for the JavaScript engines of the browsers. Consequently, an
Ajax application can run unexpectedly in a certain web browser.

Before we discuss several state-of-the-art web application frameworks, we want to give some examples for RIAs build with the RIA frameworks presented in the next subsection. A RIA that uses Silverlight (to be presented in the following subsection) is Surfity\(^7\). Surfity is a visual search engine that uses Bing and eBay to present results to a given query. Example applications for Flex (also presented later) can be found on several web sites, including a showcase web site of Adobe\(^8\). One particular example of a Flex application is the Nasdaq Market Replay application\(^9\). Regarding the third RIA framework, JavaFX, there are merely applications available. Example applications can be found on the showcase site of JavaFX\(^10\), though.

Having discussed the general client-server model and its variations, we will now close this section with an overview of the current state-of-the-art frameworks to build RIAs. Although UViz and its visualization component follow the rich client approach, we will present some of the frameworks to build Ajax-enabled applications as well.

### 4.1.2 State-of-the-Art Web Application Frameworks

There are plenty of commercial, free, and open-source frameworks available to develop Ajax-enabled web applications. Most of these frameworks provide all the functionality and tools necessary to produce such an Ajax-enabled application. Some of them are rather JavaScript libraries, others contain even development environments and user interface design tools. Among the most prominent frameworks are Dojo [Doj], JQuery [JQu], Prototype [Proa], and Google Web Toolkit [GWT]. Let us now discuss the three current state-of-the-art RIA frameworks, specifically developed to create web applications with rich user interfaces and powerful multimedia capabilities.

The first RIA framework is Microsoft Silverlight\(^11\) [Sil], the second framework is JavaFX\(^12\) [Jav], and the third framework is Flex\(^13\) [Fle]. We do not want to give the full details for all of the capabilities of the respective frameworks and their differences, as this would take several pages. Instead, we present the most important features of these frameworks and give reasons for our choice of the Flex framework and its related technologies. The first RIA framework to discuss is Silverlight.

\(^7\)http://surfity.com/#/Images
\(^8\)http://flex.org/showcase
\(^9\)https://data.nasdaq.com/MR.aspx
\(^10\)http://javafx.com/samples/
\(^11\)Silverlight is currently in version 4, as of 01.07.2010.
\(^12\)JavaFX is currently in version 1.3, as of 01.07.2010.
\(^13\)Flex is currently in version 4, as of 01.07.2010.
CHAPTER 4. INFORMATION VISUALIZATION TECHNOLOGIES

Among the most important features of Silverlight are the execution of Silverlight applications in the Silverlight runtime, loaded into the web browser as a plugin. In addition, Microsoft provides several mature development tools to build Silverlight applications. Thereby, the declarative language to define user interfaces for Silverlight applications is XAML\textsuperscript{14}. The languages to implement the application logic are C\# and Visual Basic. Silverlight applications themselves offer navigational techniques like zooming, a huge set of controls, media codec extensibility, a rich set of streaming functionalities, perspective 3D graphics, and many other features. Moreover, it is possible to place the Silverlight application on the user desktop to be accessed by a single click. In addition, Silverlight applications can also be used on mobile phones.

Another current RIA framework is JavaFX. JavaFX applications run within the Java Virtual Machine, either in the browser, on the desktop, or on mobile phones\textsuperscript{15}. Sun advertises JavaFX with similar capabilities as Microsoft does with its Silverlight framework and Adobe with its Flex framework. There are several multimedia capabilities and user interface-defining features. The declarative language JavaFX Script is used to develop applications in JavaFX. In addition, any Java library can be integrated into a JavaFX project. The main development environment is NetBeans\textsuperscript{16}, which even provides a GUI\textsuperscript{17} designer. Let us next present the reasons for our choice of the Flex framework.

Flex offers similar capabilities as the two contenders. But there were some really important facts to take into consideration, when we analyzed the potential framework to use for our visualization system. First of all, Ajax systems were discarded. This decision was made due to the potentially painful adjustment of the JavaScript code to run properly in all major browsers. Moreover, we did not find any JavaScript visualization libraries useful for our scenario. Second, we discarded JavaFX, because it is a relatively new technology and was not mature enough at the point this thesis started. In addition, there did not exist a suitable visualization library for JavaFX at that time. This applies to Silverlight as well. In fact, it is hard to find suitable graph-based visualization libraries for the available RIA and Ajax frameworks. Apart from this circumstance, the Silverlight runtime is only available for Microsoft operating systems. There is a free Linux port, called Moonlight, but this port is usually several versions behind the most current Silverlight framework.

As a consequence, we decided to use the Flex framework and its related technologies. This decision was also influenced by the availability of the so-

\textsuperscript{14}XAML is a Microsoft own variation of the XML standard.

\textsuperscript{15}As Sun claims, this will be possible in the near future.

\textsuperscript{16}\url{http://netbeans.org/features/javafx/}

\textsuperscript{17}GUI stands for Graphical User Interface.
phisticated Flare visualization toolkit and the Adobe developed data service BlazeDS. As we will see later on, BlazeDS provides services to easily connect Java-enabled backend applications to Flex applications on the client. Moreover, Flex applications run in Flash Player. And Flash Player still has the highest distribution rate of client systems among all the RIA frameworks. Additionally, the Flash Player (and consequently UViz) is available on most operation systems and all major Web browsers. This and more will be discussed in more detail in the next section about Flex, followed by a distinct section about the Flare visualization toolkit.

4.2 Adobe Flex

The Adobe Flex framework is a compound of several service and libraries, used to build RIA’s (see [BEH+09], page 30). The Flex framework is contained in the Adobe Flex SDK (Software Development Kit) (see [BEH+09], pages 29-30). The SDK is available as free closed-source and as an open-source version. Although Flex is currently available in version 4, only version 3 was available when this thesis started. Consequently, we will only talk about Flex 3 in this section. The Flex applications are developed in the programming languages MXML (a declarative programming language invented by Adobe and based on XML) and ActionScript (see [BEH+09], pages 3 and 13). Moreover, the final applications run in Flash Player (as a browser enabled web application) or as a desktop application in the Adobe Integrated Runtime (AIR)\(^\text{18}\).

\(^{18}\text{http://www.adobe.com/products/air/}\)
Several Flex-related services and technologies are available and build the Flex 3 “ecosystem” (see [BEH+09], Chapter 2). Thereby, the technologies interesting for this thesis are the following:

- **Flex Builder** - This is an Eclipse-based development environment (also available as a plugin for Eclipse). Additional features like a design mode for MXML are integrated. We used the Flex Builder to develop UViz. Nevertheless, we will not discuss the Flex builder any further.

- **Flex Software Development Kit (SDK)** - The open-source SDK, which we use, contains the debugger, the compiler, and the Flex framework. The Flex SDK will not be examined in this thesis, though.

- **BlazeDS and LiveCycle Data Services** - These technologies are frameworks to connect server-residing Java applications to Flex client interfaces. Moreover, these frameworks offer several services, like managing database accesses and messaging capabilities. We use the BlazeDS data service within UViz.

- **ActionScript and MXML** - These are the programming languages to develop Flex applications. Both languages are discussed later on.

- **Flash Player runtime** - This runtime executes the Flex applications on the client. Because the visualization frontend of UViz runs in Flash Player, we will discuss Flash Player in more detail in this section.

The schematic architecture of a Flex application is shown in Figure 4.2. In this case, we can see a comparison of a common web application with a Java backend on the server with a Flex application. Our UViz system is designed according to the schema of the Flex application in Figure 4.2. Thereby, the visualization frontend of UViz runs as Flex application in Flash Player. Instead of using HTML to define the visual layout, a Flex application uses MXML. The user can view the graphical application interface via the browser in both application types. The browser interprets the HTML code directly, in the case of a common web application (see [Mül09], pages 22-23). The MXML code is interpreted by Flash Player, which runs as a plugin inside the web browser. In a common web application, the program logic, which processes the user input and user interface related computations, is written in JavaScript. In a Flex application, the processing on the client side is accomplished by program logic written in ActionScript.

Going into more detail, regarding the development and compilation process of Flex applications, take a deeper look at Figure 4.3.

---

19Some of these services are not available in BlazeDS, as this is the functionality reduced open-source version of the LiveCycle services.
Both file types, MXML and ActionScript (which will be discussed in the following subsections), are compiled by the Flex compiler into an \textit{swf} file (see [BEH+09], pages 104-106). Thereby, libraries from the Flex framework, the AS 3 language and the Flash Player runtime are taken into account. The \textit{swf} file consists of ActionScript 3 bytecode (ActionScript ByteCode, ABC) (see [BEH+09], page 105). With the compilation, we obtain the main \textit{swf} application file and necessary additional files, e.g., \textit{supporting browser integration files}. These files are needed to enable the Flash Player plugin to load the application and execute it in the browser.

After this overview of the Flex 3 ecosystem and how Flex applications are built and compiled, we will next discuss several Flex related technologies in more detail. In the remaining part of this section, we will also examine the programming languages of Flex, the Flash Player runtime, and the BlazeDS data services. Again, all stated facts and discussed technologies of this section are related to Flex 3 (features and components may have changed for Flex 4).
4.2.1 Flex Programming Languages

Flex applications are developed with the help of two Adobe-invented programming languages, namely MXML and ActionScript. Both of them are now presented in more detail, starting with MXML.

MXML

MXML is an XML-based declarative programming language, developed by Adobe. It is solely used to layout the visual aspects and to visualize the Flex-related display components (see [BEH⁺09], pages 28-29). MXML is used to reduce the production time by leveraging the efficient markup style of MXML. It is possible to use ActionScript (AS) inside an MXML file (in a similar way to using JavaScript inside an HTML site). In addition, a developer can use non-visual classes, like data binding services, when declaring an MXML file. Moreover, it is possible to make use of CSS markup files (Cascade Style Sheets) to define the layout of Flex components (see [BEH⁺09], pages 28-29). The MXML markup language is part of the Flex framework. This is due to the fact that the Flex compiler contains the MXML interpreter.

An interesting feature of MXML is that the development environment Flex Builder provides a design mode for MXML files. This way, it is possible to see the visual aspects of a Flex user interface already during the development time. Consequently, there is no need to compile and run the application to see the applied visual changes. Hence, a direct visual feedback is delivered by the MXML markup language. An example of MXML code is depicted in Figure 4.4 (see Adobe20).

20http://www.adobe.com/de/devnet/flex/quickstart/coding_with_mxml_and_actionscript/
While the MXML language is used to define the visual aspects of a Flex application, ActionScript offers an object-oriented, Java-style approach to leverage the Flex capabilities.

**ActionScript**

ActionScript, version 3 (also called AS3), is an imperative and strongly typed OOP (Object Oriented Programming) language, developed by Adobe. AS3 conforms to the ECMA-262 Edition 4 specifications, which will also be the standard for the upcoming JavaScript 2 language (see [BEH+09], page 27). There exist many similarities between ActionScript and Java. Among the vast number of shared features are control structures, like if/else statements, the switch statement, and the conditional operator \( x ? y : z \). Moreover, loops in the form of for and while/do are possible. In addition, AS3 provides several data types, like Boolean, Number (double in Java), int, uint, and String. The special types void and null are supported by AS3, as well as complex types, e.g., Array and Object. Additionally, AS3 provides functionality to cast objects. In addition, several access modifiers, e.g., private and public are supported by AS3. More similarities exist. These can be examined on the various Flex-related web pages of Adobe. Many of the visual MXML components have ActionScript counterparts. Consequently, ActionScript can be used to build a whole application, including data services, the application logic, the user interface, and other features or parts of an application.

A simple comparison of AS3 and Java code is depicted in Figure 4.5. ActionScript 3 offers many more functionalities, so an extensive examination of AS3 would easily fill several pages. Hence, we recommend to read [BEH+09](Chapter 3) to gain more insight about the AS3 language. The next technology we want to discuss is the Flash Player runtime, which is used to execute the visualization frontend of UViz.

### 4.2.2 Flash Player Runtime

The Flash platform consists of three runtimes, namely the Flash Player runtime, the AIR and Flash Lite. We will only talk about the Flash Player runtime, as this is the relevant one for this thesis. Moreover, the version of the Flash Player discussed here is the Version 9. This is due to the fact that UViz is targeted at Flash Player 9.

Adobe Flash Player 9 executes the compiled swf files (see [BEH+09], pages 19-27). This is accomplished by two different virtual machines, the ActionScript Virtual Machine 1 (AVM1), and the ActionScript Virtual Machine 2 (AVM2). The AVM1 was the main runtime up to Flash Player 8. It runs legacy ActionScript code up to Version 2. Starting from Flash Player 9 (the current version is 10), Flash Player contains both virtual machines. This way, Flash Player is downward compatible. The AVM2 runs Action-
Figure 4.5: A simple comparison of ActionScript 3 and Java code. The example files are classes of the UViz application code.

Script code starting from Version 3. The AVM2 was changed dramatically, compared to the AVM1. The features of the AVM2 are (see [BEH+09], pages 20–21):

- **ABC parser** - The ABC parser parses the produced bytecode that is contained in the swf files.
- **Bytecode verifier**
- **Interpreter**
- **Runtime system** - This is used for the core API’s and the AS 3 language related ones.
- **Garbage collector and memory manager**
- **Just-in-Time (JIT) compiler**

The JIT compiler is one of the new features of the AVM2. It converts ActionScript bytecode into machine specific code, no matter whether the CPU
is a PowerPC or a X86-compatible one (see [BEH+09], pages 20-21). This translation into CPU specific code allows for a noticeable increase of execution performance. Even though current versions of browsers like Firefox and Opera have increased the performance of their JavaScript engines tremendously, the AVM2 is claimed to be still faster. Since Version 10, Flash Player supports 3D graphics and GPU hardware acceleration. This is accomplished by using DirectDraw or Direct3D for the PC, and by using OpenGL for the Mac (see [BEH+09], pages 22-23).

Apart from the advantage of having a really fast virtual machine, the Flash Player plugin has the benefit of running even on PDAs and many of the todays cellphones. Several sources state that Flash Player 9 has a distribution degree of over 98% (see [BEH+09], page 12). In addition, using bytecode and the JIT compiler, Flash Player is OS (Operation System) independent. Moreover, the single-threaded execution model of Flash Player simplifies the processing of events and data. This model waists the additional performance offered by today’s multi-core CPUs, though.

Before we discuss the Flare visualization toolkit, we will first discuss the Flex-specific service BlazeDS.

### 4.2.3 BlazeDS - The Flex Data Service

The Flash platform, which underlies Flex 3, offers a vast number of data formats, thereby supporting several network protocols. The Flash runtime, running on the client, accepts many different data formats to be sent and received by the executed Flex application, e.g., XML or JSON. Thereby, the runtime establishes the connection between the client and the server with the help of several supported network protocols (see [BEH+09], pages 37-
38). It is possible to connect the client to the server via HTTP, HTTPS, the Flex-specific messaging technologies RTMP (Real-Time Messaging Protocol), via sockets, and via several other protocols (see [BEH+09], Chapter 2). In addition to these communication options, Flex offers the open-source Adobe data service BlazeDS.

BlazeDS is a server-residing data service that provides functionality to connect a Flex client application to several kinds of server-side resources. An overview of the BlazeDS features is depicted in Figure 4.6. BlazeDS provides three different types of communication services, which are distributed over five different components accessed by the clients. We are only interested in the third of these services, the Remoting Service. All components interesting for us and related to the Remoting Service are colored yellow in Figure 4.6. Let us now examine this service in more detail.

Remoting Service

The Remoting Service of BlazeDS enables a Flex client to remotely call a Java-written and server-residing application. This is accomplished by using the RemoteObject component of Flex. By applying this technology, Flex client applications can call methods of server-residing Java objects transparently (see [BEH+09], pages 1054-1055). Thereby, BlazeDS handles all the client-server communication automatically. This means that the serializing ("encoding") of strongly typed Java objects and the sending of this data to the client is managed by BlazeDS. BlazeDS also handles the deserializing ("decoding") of this data into strongly typed ActionScript objects as well. These automatic translations work in both directions, from the server to the client, and vice versa. The distinct Java adapter of BlazeDS is used to offer the seamless ActionScript to Java connection. In addition to this transparent communication management, BlazeDS also uses its compact binary messaging format AMF (Action Message Format) (see next subsection). In contrast to the widely used XML data format, this reduces the amount of data to be sent noticeably. Consequently, this can improve the transmission performance considerably.

AMF

As previously mentioned, AMF Action Message Format is a compact binary format (see [BEH+09], page 33). It is used to efficiently transfer data from Java applications on the server to ActionScript using Flex applications on the client. AMF is currently available in Version 3 and supports complex data types, as well as native data types of the current Flash Player runtime AVM2 (see Subsection 4.2.2). This ActionScript-native origin of AMF efficiently reduces the necessity to parse and interpret received or send-ready data. Since opening the format to the public in late 2007, AMF has been
ported, among others, to PHP, Python, and .NET. AMF is an interesting technology to use, because of its efficiency. This is documented in the Census performance benchmark of James Ward\(^{21}\). All these features advise AMF as fast alternative to the XML data format, which is often used in client-server architectures (e.g., by WebServices). Therefore, we use AMF in our UViz system.

Apart from these technologies used within our UViz system, we also had to decide which technology to use for the visualization itself. We decided to use the Flare visualization toolkit, which is an ActionScript library for graph-based visualizations. This visualization library will be discussed in the following section.

### 4.3 Flare Visualization Toolkit

Flare [Fla] is an ActionScript visualization library targeted at Flash Player 9. Flare was developed by Jeffrey Michael Heer [Heeb]. He is currently an assistant professor at the Computer Science Department of the well known Stanford University. More specifically, Jeffrey Heer is a member of the Human-Computer Interaction & Visualization Groups. Flare is one of Jeffrey Heers more recent projects. In the following section, we will present the background of Flare and discuss related work of Jeffrey Heer.

#### 4.3.1 Background

Flare itself is not a completely new library. Flare is based on its Java-written ancestor Prefuse [Pre]. The first version of Prefuse was made public in 2006. A few months earlier, Heer published the paper Prefuse: A Toolkit for Interactive Information Visualization [HCL05]. Prefuse is a pure Java library to build user interfaces and highly dynamic visualizations. Thereby, Prefuse uses the Java 2D library\(^{22}\) and can easily be integrated into Java Swing\(^{23}\) applications or Java Web Applets\(^{24}\). The Prefuse toolkit is distributed over several distinct packages, while providing a rich set of functionalities. Among these are different interactive controls to operate the visualization, several layouts to build graphs (including tree structures), and classes for animations. Moreover, physical computations can be run with the help of distinct algorithms contained within the toolkit. In addition, multiple encodings of different kinds are provided to define the visualized data. Moreover, a huge utility library is contained in Prefuse as well. There are even more structures

\(^{21}\)http://www.jamesward.com/census/
\(^{22}\)http://java.sun.com/products/java-media/2D/index.jsp
\(^{23}\)http://java.sun.com/javase/6/docs/technotes/guides/swing/
\(^{24}\)http://java.sun.com/applets/
provided by Prefuse, and different kinds of demos are integrated to demonstrate the potential of Prefuse for interactive information visualizations. One thing to mention here is that Prefuse is open-source and published under the terms of the BSD license. After refining Prefuse in several later versions, Heer decided to use ActionScript for its new project, called Flare.

Flare can be easily integrated into Flex projects, even though it does not request for any Flex capability (Flare uses only Flash libraries). Flare shares many of the functionalities of Prefuse. Therefore, Flare has a similar feature set and similar structures. Using ActionScript and Flash Player as runtime environment, Flare is directly targeted at visualizations for the Internet (see [Heec]). Flare is an open-source software and released under the terms of the BSD license (like Prefuse). The latest version of Flare came out in early 2009 and was an alpha release (with several maintenance releases inbetween). Since then, there have not been any concrete statements when a new version of Flare could possibly be expected to be released. There is only a weak indication on the Flare web site [Fla], which says that further versions will migrate to Flash Player 10. But, Flash Player 10 has already been available for a while. Consequently, it is unclear if Heer still plans to develop a new Flare version. What is even more hinting at the end of the Flare project is the publishing of the declarative visualization toolkit Protovis [Prob] [BH09]. The focus of Heer seems to be on this project at the moment. With Protovis, Heer follows the route of visualization interface libraries that use declarative syntax, like the already mentioned Java 2D and Processing\textsuperscript{25}, instead of the more common approach of using an imperative syntax. In its most current publication, Jeffrey Heer talks about state-of-the-art visualization techniques and the way they can support the user [HBO10].

Before we end this section with some example visualizations made with Flare, we will give an overview of the Flare API.

4.3.2 The Flare API

As already said, the Flare toolkit is premised on the Prefuse toolkit and shares many structures and ideas. The Flare API, just as it is the case for Prefuse, is distributed over many packages. We will now examine the general API modules to give an insight into the Flare API. The following packages are provided by Flare:

- One of the most important packages is the vis package. Here, we have several subpackages to define and control the visualization. There is a special package with structures to control the visualization, e.g., drag or a hover controls. Moreover, this visualization specific package contains subpackages to define events and graph related data, like nodes

---

\textsuperscript{25}http://processing.org/
and edges. Additionally, the package operator is provided. Here, among others, we find classes for distortions (e.g., the fisheye-distortion), to compute layouts for graphs (including trees), and to apply visual filters to the visualization. Finally, the main visualization class for Flare-enabled applications, the class Visualization, is contained in the vis package. The Visualization class holds all the necessary properties to configure and run a Flare visualization.

- The analytics package provides algorithms to cluster the visualized nodes and edges of graphs. The computation of graph properties like the Shortest Path between two nodes or the Spanning Tree of a graph are possible as well.

- The animate package provides several classes to control the way updates to the visualization are handled. Moreover, interpolation algorithms are contained in this package. These algorithms can be used to smooth the defined animations.
- Classes to specify and configure the visualization source data are encapsulated in the \textit{data} package. This package also provides several classes to load and convert common data files, like \textit{GraphML} or \textit{JSON} files.

- The \textit{display} package holds Flare distinctive classes that are extended versions of the Flash class \textit{Sprite}. This class is the basic building block for display objects in ActionScript 3 (see [BEH$^+$09], page 68).

- Physical computations are possible by using the classes of the \textit{physics} package. Possible physical computations are the application of drag forces. Thereby, a drag force based on the velocity of particles is applied to the particles of the simulation. Other possible forces provided are gravity forces. Here, a global gravitational pulling force on particles is simulated. Additional force simulations are supported as well. This package is used by Flare to build a force-directed graph layout.

- Moreover, the API contains the packages \textit{query} and \textit{scale}. These packages provide structures and methods to directly query a database or define different scales to be applied to values (like ordinal or logarithmic scales). In addition, there is a package \textit{flex}, which holds a structure to embed a Flare visualization in a Flex application.

This is just a rough overview of all the provided functionality of Flare. For a more in-depth study of Flare, we recommend to access the aforementioned web sites of Flare and Jeffrey Heer. Let us now conclude this section with some example visualizations built with Flare.
4.3.3 Example Applications

The first example is the visualization of the Flare API package structure\textsuperscript{26}, as it is shown in Figure 4.7. The visualization is done with the circle layout contained in Flare. Here, several techniques are applied. Edges are bundled to route along the tree structure. On hovering over a class with the mouse, the dependencies to other classes and packages are shown. Hereby, text labels represent the respective classes. Labels closer to the center display the circumstance that these packages and their contained classes reside higher in the package hierarchy. Different colors encode the dependency, inheritance and access between classes and packages. In addition, the visualization is highly interactive, thereby guaranteeing a smooth and seamless user experience.

Apart from this example, another interesting demo of the Flare capabilities is the Job Voyager application of Jeffrey Heer\textsuperscript{27} (see Figure 4.8). To quote Heers documentation for this application:

\begin{quote}
The visualization shows stacked time series of reported occupations in the United States Labor Force from 1850-2000. The data has been normalized: for each census year, the percentage of the polled labor force in each occupation is shown. The data is originally from the United States Census Bureau and was provided by the University of Minnesota Population Center (ipums.org).
\end{quote}

The Job Voyager application uses different color encodings to indicate that a job is done by men or women, or by both genders. There are percentage values to show the proportion of people working in a specific job. Information are popping up when hovered over the respective section. If the user enters a job name, the application changes the visualization accordingly. Again, the application is highly dynamic.

Our last example presented is the main Flare demo application, which is the first place to start to investigate the capabilities of Flare. In Figure 4.9, the aforementioned demo application is depicted. There are options for most of the important Flare features, and selected options are immediately visualized underneath the application menu. In the screenshot, a simple graph with a radial layout is shown. The demo shows general features of Flare and indicates what Flare is capable of.

After the discussion of so many topics with this level of detail, let us now summarize the just presented information. In addition, we will also present the architectural overview of UViz, with the integration of the technologies discussed in this chapter.

\textsuperscript{26}http://flare.prefuse.org/launch/apps/dependency_graph
\textsuperscript{27}http://flare.prefuse.org/launch/apps/job_voyager
CHAPTER 4. INFORMATION VISUALIZATION TECHNOLOGIES

Figure 4.9: The Flare main demo application, which shows the main Flare capabilities.

4.4 Summary

In this chapter, we presented several technologies to build applications for information visualization, especially Web-enabled applications. We started with an overview of different client-server architectures and state-of-the-art web application frameworks in Section 4.1. We continued with the introduction of Adobe Flex and its related technologies in Section 4.2. The chapter ended with the discussion of the Flare visualization toolkit in Section 4.3. Most of the technologies and models presented in this chapter are used in UViz. To get a better picture of the UViz architecture, we now want to give an overview of the UViz architecture, including the important technologies used.

The UViz system is build according to the client-server model. To be more precise, UViz is a Flex-enabled system. Thereby, the visualization frontend, called UViFace, runs in the Flash Player inside a common web browser. In addition, we use the BlazeDS data service and AMF to connect URDF on the server-side with the Flex visualization frontend. To build an abstraction layer between the client side and the server side, we also developed a new API for URDF. This API is also used for the efficient data transport between server and client. Moreover, UViz integrates the Flare visualization toolkit to provide a graph-based visualization to the user via Flash Player.

Having summarized this chapter and introduced the architecture of our UViz system, let us now have a more detailed discussion about the UViz system architecture in the following chapter.
Chapter 5

UViz - The URDF Visualization System

In this chapter, we present UViz (URDF Visualization), a complete visualization system for URDF. UViz uses the aforementioned URDF framework (see Chapter 3) to answer given queries and deliver the information for the visualization. The system is built as a web-enabled visualization system, according to the rich client model (see Subsection 4.1.1). Thereby, the visualization interface of UViz will run in Flash Player and is accessible via a common web browser\(^1\). This way, the visualization related computations can be accomplished on the client without the need to call the server. In case the user submits a query, the query request is sent to the server. Then, the reasoning engine of URDF, which resides on the server, executes the query and sends the result data back to the visualization. Distributing the processing-intensive reasoning to the server, the system is able to efficiently compute the query results, while staying interactive at the client side. In addition to this provided consistent program interactivity, the system will be mostly independent from the browser, the operation system, and the client platform. This is accomplished by using Flash Player, which is installed on most computers today, as runtime to execute the visualization frontend of UViz. To use the Flash Player, we decided to write the client-side UViz components in ActionScript and MXML (see Section 4.2.1), using the Adobe Flex RIA framework (see Section 4.2). Thereby, we use the ActionScript visualization library Flare (see Section 4.3) to provide the visualization components and leverage Flash Player. In addition, UViz integrates the BlazeDS data service (see Subsection 4.2.3) to connect the Java-enabled URDF framework to the client-side visualization. The client-server communication is thereby supported by the compact binary data format AMF.

\(^1\)By applying only a few changes in the source code and using the desktop runtime AIR of Adobe, the visualization frontend of UViz will be able to run even as a desktop application, if needed.
The UViz system architecture consists of several components, technologies and programming languages. To get an overview of the system, let us first show the general architecture of UViz. Omitting all the details of the architecture, the overall system looks like the one depicted in Figure 5.1.

Here, we can see the already explained client-server architecture of the system. In addition, the URDF API (Application Programming Interface), specifically developed for UViz, is shown in the picture. This interface will be discussed in more detail in the next section. Keeping this system overview in mind, the remaining part of this chapter is divided into several sections, namely:

- Section 5.1 concentrates on the server side of the UViz system. This section will discuss technologies used and give reasons for their application in UViz decisions.

- Section 5.2 deals with the client side in more detail.

- The system specific communication is then the focus of Section 5.3.

- After presenting the system architecture and communication scheme, we will discuss the general application work-flow of UViz in Section 5.4.

- In Section 5.5, we demonstrate the efficiency of the client-server communication within UViz, by providing several performance measurements.

- This chapter closes with a small summary in Section 5.6.
5.1 Server-Side Architecture

The server side itself is divided into several components. Among these are the application server, the database server with the database (manages the knowledge base YAGO), the URDF reasoning framework, and the URDF API. The first three components are discussed in the following subsection, whereas the URDF API will be explained in Subsection 5.1.2.

5.1.1 Data Processing and Managing

The server software that deploys the visualization frontend of UViz to the Internet is the Apache Tomcat Server\(^2\). Tomcat is an open-source software implementation of the specifications for the Java Servlet\(^3\) and JavaServer Pages\(^4\) technologies. The Apache Tomcat is a servlet container, which acts as a pure Java web server in the usage scenario of the UViz system. Tomcat provides all the functionality necessary to connect the client-side visualization with the server-side URDF reasoner (see also Figure 5.1). In addition, Tomcat manages the access to the database on the database server.

The database server manages the knowledge base YAGO, which is accessed by URDF during the reasoning process. Thereby, the database server is part of the server side as well. In the case of UViz, the database is provided by Oracle\(^5\). To improve the throughput of URDF and the access times to the database, the Oracle database is located on the same computer as the Apache Tomcat server. Running the database on another computer within the MPI network is feasible as well.

Not only the UViz hosting Tomcat and the UViz backing knowledge of the database is located at the server side, but also the URDF reasoning framework. As explained in Section 3.2, URDF accesses the database to receive information for the reasoning process. With the help of the rich client model, all the processing intensive tasks of the URDF reasoner can be accomplished on the server side. It is worth mentioning here that the current URDF version is implemented purely in Java (with an even faster C++ version in development).

Further investigations of the process work-flow of UViz will follow in the remaining sections. Before we can present the client-side architecture and the work-flow, we have to discuss the URDF API, though. This abstraction layer represents the intermediate tier between the client-side visualization and the server-side URDF reasoning engine.
Figure 5.2: An overview of the Java URDF API of UViz.
5.1.2 URDF API

As mentioned before, the URDF API is implemented as an intermediate layer between the frontend on the client and the backend of the UViz system on the server. The API is also located at the server side. The URDF API, just like the current URDF version, is implemented in Java. The URDF API is used for the data exchange between the visualization and URDF. An overview of the API structures and their inheritance hierarchy is depicted in Figure 5.2.

Let us now discuss the API structures shown in Figure 5.2 in more detail:

- Above the hierarchical class structure of the URDF API are the interfaces Serializable and Externalizable. These interfaces are implemented by the API classes to be able to manually specify which information should be send back and forth between the client and the server. Thereby, the Externalizable interface gives us more control over the serialization and deserialization processes, in particular with the different types of the URDF API. This way, it is possible to explicitly define the data type of an object to serialize or deserialize.

- The UObject class is the topmost class in the hierarchy of the URDF API. This class is used to represent a generalization for all the following classes of the API. The interesting thing here is the id property. This is a distinct numerical identifier. It is applied to circumvent redundant hash computations with the name of UObject instances, when using the instances within hashing structures.

- The class UReasoner is the main entry point on the server side. Its methods are remotely called by the client application (which will be explained in Section 5.4). The UReasoner class manages the internally used reasoning rules, by providing two lists of default rules, one for the soft rules (see Subsection 3.1.1) and one for the hard rules (see Subsection 3.1.2). These lists are encapsulated into an instance of the URuleStore class. This object is finally used to reload the default rules into the client application. Moreover, the UReasoner class provides three distinct methods to be remotely called by the client. The method processQuery uses the URDF reasoner to process a query and return the results to the client. The method loadData uses the URDF reasoner to load data for a specified entity. This method is called when a user wants to receive more information about a specific entity representing

---

2 http://tomcat.apache.org/
3 http://java.sun.com/products/servlet/overview.html
4 http://java.sun.com/products/jsp/
6 There is a similar ActionScript version of the URDF API on the client side as well, as we will see later on.
a node in the visualization (which will be explained in Chapter 6). The third available remote method `requestRules` reloads the sets of default soft and hard rules into the client visualization.

- The `UBindingSet` class manages all mappings between variables or constants and their bound values.

- The `UStatistics` class stores information about the server-side reasoning. An instance of this class is send to the client, every time a query result is returned by URDF. These reasoning statistics can then be used within the visualization.

- The hard and soft rules are represented by the classes `UHardRule` and `USoftRule`. Queries submitted via the visualization interface of UViz are represented by the class `UQuery`. All three classes are inherited from the class `URule`, because of the similar and shared structures. The USoftRule objects store an additional head literal to be grounded by the URDF reasoning engine. In addition, USoftRule objects contain a weight to reflect the importance of a soft rule (see Subsection 3.1.1).

- The triples style literals in form of the `ULiteral` class, as well as their grounded counterparts in form of the `UFact` class, are inherited from the `UTriplet` class. The constituents of these classes are of type `UArgument`. These arguments inherit relations and entities of types `URelation` and `UEntity`, respectively. Literals can contain variables and constants, whereas facts only contain constants in the form of predefined relations or entities received from the database. Facts have truth values attached to them. This truth values were produced by the application of the MAX-SAT solver of URDF (see Subsection 3.2.3). Additionally, each UFact object stores its lineage information of type `ULineage`. Each ULineage object represents the derivation information for the respective fact (see Subsection 3.2.2). Moreover, the `UDomain` class is used to define the domain of arguments.

- Java objects of type `UFactSet` represent query results and grounded hard and soft rules. Grounded soft rules are soft rules which contain only facts inferred beforehand. Semantically, such a `UGroundedSoftRule` instance means that the head fact was inferred by conjoining the body atoms, or by receiving the fact directly from the knowledge base. Objects of type `UGroundedHardRule` are the grounded versions of hard

---

7 Literals in UViz represent RDF triples of the form `(subject, predicate, object)` (see Subsection 1.1.1). These literals can contain variables and constants.

8 We have no atoms in the URDF API anymore. Atoms are replaced by their signed counterparts literal in the form of the type ULiteral.
rules. Basically, these rules represent sets of facts that are not allowed to occur simultaneously in one distinct answer of a query.

In Figure 5.2, there are only the properties of the respective classes visible. Nearly all of these properties are hidden (only made visible for convenience) and only accessible via defined methods. Methods are omitted here, to simplify the visual appearance of the diagram. The structure of the API is modular and can be extended later on, if necessary. Since we have shown the server side architecture, we can now go on with the client side.

5.2 Client-Side Architecture

In this section, we will concentrate on the client side of the UViz system. As already mentioned in the beginning of this chapter, UViz is targeted at the Flash Player, by using Adobe Flex (see Section 4.2) to develop the client side UViz components. Moreover, we integrate the Flare visualization library into our UViz system. The question now is, what does the architecture on the client side look like?

First of all, we have a version of the URDF API on the client side as well. This API is basically the same as the one residing on the server side. Nevertheless, there exist a few differences. The client-side API is written in a language Flash Player understands, namely ActionScript (see Subsection 4.2.1). Consequently, some of the used data structures had to be adjusted. But overall, the API provides similar capabilities and data structures (including naming conventions) as the Java version of the URDF API. The ActionScript URDF API is used by BlazedDS to serialize and deserialize data to and from the server. Additionally, with the help of the ActionScript URDF API, we can use the binary data transmission format AMF, which is provided by BlazeDS (see Subsection 4.2.3). This will be explained in more detail in Section 5.3.

Apart from the ActionScript API, there are several other components in the frontend of UViz. The implemented ActionScript and MXML classes are divided into user interface-related classes and visualization-related ones. The user interface-related classes deal with the user interface panels. These are used to help the user interact with the visualization. Among these classes are the panels for the queries, the options, the statistics, and for the rules. The visualization-related classes implement algorithms for the graph visualization of the received URDF results. In addition to this main distribution of functionalities within the frontend, there are packages for events and helper structures, as well as for external components. The just explained issues are reflected in the Figure 5.3.

The Flare library, the client-side URDF API, and the visualization-related UViz classes are pure ActionScript. The user interface-related related classes are mixtures of ActionScript and MXML. By applying Flex to build
our client component of UViz, we can use Flex-specific technologies, as well as Flash-specific ones. In combination with the Flare library and running in the Flash Player runtime, we achieve a visualization frontend with many strong features, among which are the following:

- The visualization frontend can make use of the Flash Player garbage collector to manage memory consumption.

- The mature state of the Flash platform provides many features in a stable version. Moreover, the Flash Player runtime delivers a decent performance.

- The Flare library delivers sophisticated algorithms and smart implementations to leverage Flash Player capabilities even more effectively.

- Flex supports many user interface features, which makes building a appealing user interface easier.

Having explained the architecture of the UViz system on the client side, we continue with the examination of the system communication, which will show some interesting features.

### 5.3 System Communication

Up to this point, we have only explained the general architecture of the UViz system. We have not yet talked about how the system communicates. In this
In this section, we will discuss the system communication, which involves the data exchange between the server and the client side of the UViz system.

Before we go into more detail, we present an overview of the communication between the server side and the client side in Figure 5.4. Here, we can see the most important component used for the communication within UViz, the BlazeDS data service of Adobe (see Section 4.2.3). BlazeDS provides the technology to remotely call server-side Java objects from a Flex client. Moreover, BlazeDS allows to use the efficient data transmission format AMF.

The communication between server and client applications is always a crucial task because data transmission usually involves the usage of special, often inefficient, data formats. These data formats usually influence the transmission time. Consequently, the response time of the server is influenced as well. This leads to a slowed down web application and a negative user experience with the application. Apart from this circumstance, web applications have to transform their data into these transmission data formats before sending the data from the server or the client. When the server or the client application receive the transmitted data, they have to be transformed back into objects of the respective programming language to be processed further. This is a time consuming operation on both sides of a client-server system.
This is the reason, why we use the Remoting Service of the BlazeDS data service technology (see Section 4.2.3). This open-source software component, provided by Adobe, gives us the possibility to directly call a function of a Java Remote Object, residing on the server. By applying BlazeDS technology, we are also able to use the messaging data format AMF (see Subsection 4.2.3) to transmit data to and from the server. The BlazeDS and AMF usage scenario is depicted in Figure 5.4. BlazeDS resides on the server, but it manages the communication between the Flex frontend application and the Java backend application. The data transformation and communication handling is done by BlazeDS in an automatic fashion, but can be controlled by the developer as well, if needed⁹. Figure 5.4 also shows the application of AMF. Using the Remoting Service and AMF, strongly typed ActionScript objects are mapped to their strongly typed Java counterparts, and vice versa (see [BEH⁺09], chapter 59). Therefore, classes of the ActionScript URDF API are marked with a RemoteClass tag. This is the indication for BlazeDS how to map data received on the client and the server.

⁹In the case of UViz, this manual configuration was necessary.
In Figure 5.5, the manual configuration of the remoting service is depicted. In a), a snippet from the configuration file for the remoting is shown. In this file, Java classes that should be remotely accessed by the client have to be manually specified. Moreover, instances of these classes have to be attached with a new identifier. This identifier is then used within the client application, to remotely call the named Java object. This represents a mapping scheme to introduce the Java object to the client application. In b), the named Java object is transparently used in the Flex client application, as if the object would reside on the client. Moreover, the remote methods to call are manually specified. To handle the method results from the remote call accordingly, distinct result handling functions can be defined on the client.

Using the BlazeDS technology, UViz is able to reduce the server response time considerably. Moreover, we can work transparently with our API within our UViz system.

Before we show performance measurements for the communication within UViz, we want to demonstrate the general application work-flow of UViz, by means of a simple execution example.

### 5.4 Application Work-Flow

This section will present a work-flow example to clarify the general application work-flow within UViz. The example work-flow is depicted in Figure 5.6. In this example, we assume that the user submits a new query on the client side. The query request is then sent to the URDF reasoner, which resides on the server side. The reasoner then computes the query result, and the result is sent back to the client. In the figure, there are labels on the edges. These labels represent the steps involved in the processing. The processing starts at the query interface. The processing ends in the updating of the user interface elements, as well as in the updating of the graph visualization. Thereby, red arrows stand for the direction from the client application to the server application. And the black arrows represent the back direction. The API has an exceptional position in this schema, as the API is internally applied on both sides of the UViz system. Hereby, the API data structures are used as parameters and as callback results (when a remote method was called).

Let us now analyze the example work-flow of UViz (depicted in Figure 5.6) in a step-by-step manner:

1. The user defines and submits the query in this step. This task is accomplished by using the query interface of UViFace (the UViz visualization interface on the client side (see Chapter 6)).

2. The UViz application entry class on the client receives the query event and the ActionScript object that represents the query to be sent to the
server. The main class then calls the \texttt{processQuery} method. BlazeDS maps this method call to the UReasoner Java remote object, by checking the \texttt{destination} and \texttt{source} properties defined in the respective configuration file (see Figure 5.5). If BlazeDS recognizes this mapping as valid, the query object, as well as the rules for URDF, are serialized. This is done by translating them from strongly typed ActionScript objects into binary AMF data.
c) In this step, the AMF data arrived and is deserialized again into strongly typed Java objects. Again, this is accomplished automatically by BlazeDS, using the mappings specified in the respective classes of the ActionScript version of the URDF API (see Figure 5.4). Now, the UReasoner Java object hands the query and rules received from the client application to the instance of the URDF reasoner.

d) Now, the reasoner starts with the grounding phase by using the already mentioned SLD resolution (see Subsection 3.2.2). Thereby, the reasoner applies its rules. Moreover, the URDF reasoner instance uses the passed query and its rules to specify access parameters for the database. This means, the reasoner examines which arguments are variables, and which arguments are constants. Then, the access to the database is managed accordingly.

e) After finishing the grounding phase, URDF then constructs a CNF and applies its MAX-SAT solver (see Subsection 3.2.3). In this separate reasoning step, possible inconsistencies between the knowledge base and the inference rules are resolved and the final query result produced.

f) The current step involves the passing of result information to the UReasoner object. This object is used throughout the whole lifetime of a user-controlled instance of UViz on the client side. The UReasoner object represents a communication bridge between the client and the server side.

g) Step g) is similar to step c). Now we collect the result data in an instance of the class UQueryResult and BlazeDS serializes this object into AMF data for us. Then, the data is sent back to the client application.

h) Finally, the data arrives at the client and is deserialized back again into strongly typed ActionScript objects of our ActionScript version of the URDF API. As a last step, the running client application uses the received result data to update the respective user interface elements and the graph visualization accordingly.

This was just an example for the work-flow within the UViz system, more complex scenarios are possible. The last section before the summary deals with selected performance measurements that show the efficiency of the chosen communication implementation.

---

10Either as a fault event or a result event. This forces the client application to handle the respective case.
5.5 Performance Measurements

We have measured 8 queries, differing in the amount of data to send and the complexity of the reasoning. We submitted every query 10 times and averaged the measured values. The measured values can be seen in Table 5.1. Thereby, the distinct queries used are presented in the appendix (Chapter 8).

<table>
<thead>
<tr>
<th>Dependency Graph Size</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q7</th>
<th>Q8</th>
</tr>
</thead>
<tbody>
<tr>
<td># Grounded Soft Rules</td>
<td>6</td>
<td>9</td>
<td>6</td>
<td>23</td>
<td>8</td>
<td>37</td>
<td>30</td>
<td>38</td>
</tr>
<tr>
<td># Grounded Hard Rules</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>14</td>
<td>4</td>
<td>46</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>Execution Time (ms)</td>
<td>198</td>
<td>187</td>
<td>283</td>
<td>343</td>
<td>17729</td>
<td>2151</td>
<td>364</td>
<td>242</td>
</tr>
<tr>
<td>Send Time (ms)</td>
<td>39</td>
<td>44</td>
<td>25</td>
<td>51</td>
<td>43</td>
<td>106</td>
<td>55</td>
<td>54</td>
</tr>
<tr>
<td>Message Size (byte)</td>
<td>7338</td>
<td>2928</td>
<td>7162</td>
<td>12006</td>
<td>3670</td>
<td>28571</td>
<td>6965</td>
<td>6400</td>
</tr>
</tbody>
</table>

Table 5.1: Measurements for 8 different queries submitted in UViz and executed on a local Apache Tomcat server (client and server on the same computer).

What does that table show? The table consists of 8 query columns (plus one column for the row identifiers). The first row shows the measured sizes for the respective dependency graphs (all facts produced during reasoning, see Subsection 3.2.1). The second and third row present the number of grounded soft and hard rules for a certain query. Row four states the execution time (in milliseconds) the server (URDF) needed for a respective query. In the fifth row, the time needed (also in milliseconds) to send the result data from the server to the client is measured (the other direction is omitted here). The measurement for this value starts with the execution of the return statement in the processQuery method on the server. This means, the serialization time is included in the measured value. The measurement for the value ends with receiving the result event on the client. Moreover, the result was casted into ActionScript objects before the measurement ended. Consequently, deserialization is included in this value as well. The last row shows us the size of the return message (query result) from the server in bytes. Now, what can we conclude?

The size of the messages is relatively small. Additionally, the times to send the result data back, including serialization on the server and deserialization on the client, are low as well. Nevertheless, we have to keep in mind that the reasoning is the time consuming operation here. Consequently, the time to get the data from the server, after the reasoning has finished, has to be considered high for fast executed queries. This means, the faster the reasoner executes the queries, the more impact on the overall response time of the server has the serialization/deserialization and transmitting process. Moreover, the size of a query answer is no direct indication for slower server responses. There are several factors that can influence the time a query re-
results needs to return to the client. An important factor seems to be the serialization and deserialization, though. In addition, the combination of a complex serialization and deserialization object tree and a huge amount of result data potentially leads to slower response times. This is an obvious observation.

Summarizing the observations, we conclude the following:

- The communication implementation we use seems to deliver fast response times for the user. For easy to process queries, the communication overhead can not be ignored, though.

- The sizes of the transmitted query results are relatively small and lead to an efficient utilization of the respective Internet connections of the users.

We have to admit that we did the measurements with a locally running server (server and client on the same computer) and a database within the MPI network. Moreover, several factors can negatively influence the measured times. Consequently, the measured values are only indications for the efficiency of the communication implemented. These values can change considerably for the access of UViz over the Internet. We now want to conclude the whole chapter with a small summary.

## 5.6 Summary

In this chapter, we discussed the general system architecture of UViz. Moreover, we examined the URDF API, which was developed as additional access layer between the client and the server applications. In addition, we presented the communication of the UViz system. Afterwards, we demonstrated the application work-flow of UViz by examining an example usage scenario step by step. Finally, we presented several measurements to show the efficiency of the implemented UViz communication.

In the following chapter, we will discuss the visualization frontend of UViz, named UViFace.
Chapter 6

UViFace - The UViz Visualization Interface

In this chapter, we finally present *UViFace* (UViz Visualization Interface), the visualization interface of UViz.

UViFace originates in the idea of providing the visualization frontend to the already existing and still evolving URDF reasoner. With the special

![Figure 6.1: An example graph of UViFace.](image-url)
features being inherent in URDF, there were special needs to reflect these features in a decent way. Repeating the objectives for our UViz system, we expected several features from our visualization component of UViz. Because it restricted our choices of technologies, we have to mention the browser as “frame” for our visualization again. This influences the visualization engine of our UViz system noticeable and narrowed our amount of choices down to a few RIA frameworks (see Subsection 4.1.2). Moreover, the frontend should provide the user with an enticing visualization. Thereby, the visualization should encourage the user to work with the system. Finally, we decided to use Flare, Flex, and Flash Player to accomplish this goal. Another objective was to provide common visual tricks to enhance the user experience. Among these visualization-related techniques are zooming, collapsing/expanding of graph nodes, and animations. With Flare, we found the perfect tool to solve this issue. In addition, our visualization should be able to present the reasoning steps involved in generating the reasoning results. Further, dynamic reloading of data and reflecting of changes to the used rule base should be part of the visualization as well. Again, Flare and Flex provided us with the necessary tools to accomplish these tasks. Summarizing the aforementioned objectives and adding not mentioned ones, we end obtain this list of goals for our UViFace component:

- The visualization should provide an easy to understand graph layout and general exploration of the result graph. The query and the used rules, as well as the resulting dependency graph (see Subsection 3.2.1), should be reflected appropriately in the visualization.

- Dynamic loading of data from the knowledge base, in case the user wants to explore the visualization graph, should be supported as well. In addition, it would be useful to collapse/expand parts of the graph, in case of a changed user focus.

- Zooming and summarizing of parts of the graph, according to the density of the parts, can also help the user to understand the visualization.

- The lineage information (see Subsection 3.2.2) contained in the query answers have to be visualized in a way that enables the user to examine the produced lineage in a step-by-step manner.

- It would be interesting to provide rule and data updates to the knowledge base via the user interface. Changes to the rules or the knowledge base, as well as changes between different queries, should be reflected in the visualization. This could be accomplished by using different colors or other visualization techniques.

An example screenshot of the the UViz visualization is depicted in Figure 6.1. Here, we can see UViFace in action. Before we discuss the achieved goals...
and go into more detail of the visualization, we want to give a short introduction to the subject of Information Visualization. This is the general field the UViz system belongs to. More specifically, UViz belongs to the subject Graph Visualization. These topics will be discussed in the next section.

6.1 Information Visualization

What is the meaning of visualization in general? Zhing formulates this term as follows (see [Zha08], pages 3-4):

> Visualization is the process of transforming data, information, and knowledge into graphic presentation to support tasks such as data analysis, information exploration, information explanation, trend prediction, pattern detection, rhythm discovery, and so on.

Many of the mentioned aspects hold for our UViz system. And the general idea behind visualization, the way Zhing formulates it, is captured very well by UViz. To come back to information visualization, Zhing divides the term visualization into two categories (see [Zha08], pages 3-4). One of the categories is scientific visualization, and one category is information visualization. Scientific visualization often deals with the augmentation of the human sensory system, usually to slow down or enlarge things which are not appropriately perceptible for human beings. Examples are shapes of molecules or fluid dynamics. Information visualization, on the other hand, is usually used to visualize abstract information (see [Zha08], pages 3-4). Examples are visual reasoning, visual data modeling, or information retrieval visualization. Scientific and information visualization share many aspects. But information visualization poses some real challenges. Most importantly, there is no spatial structure or framework for the semantic relationships among data in information visualization. Consequently, these structures must be defined to display the data in a decent way. While this gives the people more freedom in how they process and display the given data, it requires smart solutions to be found. Data in an information space may be multi-faceted, relationships among data can be interwoven, and the data can have a diverse nature. As a consequence, it is not an easy task to present the data in a meaningful way. Thereby, extracting interesting information from data and synthesizing the data relationships and semantic aspects into the visual space is hard to accomplish (see [Zha08], page 4). In the end, information visualization is the subject of meaningful visualizing information extracted from data.

A subcategory of information visualization is Graph Visualization. This subject is the determinant topic to which our UViz system can be assigned to. Before we can examine the visualization in UViz, we will discuss the topic Graph Visualization.
Referring to Huang (see [Hua09], page 6), graphs can be defined as an information space $G = (V, E)$, where $V$ is the set of entities and $E$ the set of relationships between those entities.

This is the general definition of graphs. There exists a variety of graph types, e.g., graphs can have multiple edges between two distinct nodes (called multi-graph). The above definition sets the basic idea behind graphs. The variations to visualize graphs are numerous. Consequently, the field of graph visualization is huge. Nevertheless, there exist some common graph related subtopics which define the properties of a graph visualization very accurately. As graph visualization is a subtopic of information visualization, it shares the same problematic issues. A suitable visualization has to provide meaningful data representations. Moreover, a visualization should support the user in its search for interesting information within the visualization. Nowadays, the task of providing efficient and easy to understand graph visualizations is harder than ever [Cui07]. This is due to the overwhelming amount and complexity of data that comes from sources like large social networks. The algorithms and techniques developed to cope with these challenges are plentiful. We now want to present some of these techniques.

In [Cui07], Cui shows many of the current hot topics related to graph visualization. In [HMM00], Herman et al. discuss many of today’s most interesting issues in graph visualization. We will present some of the topics presented in these two reports, but only giving an overview\footnote{For a more detailed examination, we advise to have a look at the respective papers.}.

The first big topic for graph visualization is the choice of the graph layout. A lot of distinct layout algorithms exist, but these are more or less useful for a
graph visualization, depending on the size of the graph and the complexity of the data (additional reasons exist). Another point to consider for the layout decision is the choice of the dimensionality. 2D and 3D graphs have their advantages and disadvantages, depending on the usability scenario. Among the vast diversity of graph layouts are the radial layout, the tree map, and the force layout. The radial layout positions nodes in a radial shape around a center node. The treemap displays nodes as boxes and places smaller boxes into bigger ones. The force layout computes forces between nodes, by using the connecting edges as springs between the nodes. In contrast to the first two layouts, which are static, the force layout is a dynamic layout. The radial layout and the force layout are both supported by UViz and depicted in Figure 6.2. Graph layouts are further divided into classical graph and tree layouts, as trees are special forms of graphs. Apart from the diversity in graph layouts, the navigation and interaction within graphs is another interesting issue for graph visualizations.

Graph navigation and interaction extend the graph usability, in case it cannot be provided by the aforementioned graph layouts. One important feature that can be applied to navigate and interact with a graph is panning and zooming of graphs (see [HMM00], pages 10-11). Panning allows the positioning of the graph, according to user needs. Zooming allows to set the distance of the graph to the virtual camera, which represents the eye of the user\(^2\). Other important techniques to support the user in understanding a graph visualization belonging to the navigation techniques are focus+context techniques (see [HMM00], pages 11-12). One of these techniques is the fisheye distortion. Here, a set of “focus” nodes are magnified visually, while nodes out of the “focus” are shrunk down visually (see [HMM00], pages 11-12). This technique can increase the readability of large graphs. Apart from these techniques, incremental exploration and navigation techniques can be applied. Here, the graph is only shown partially. This improves the readability of the graph for the user (see [HMM00], pages 13-14).

One last subtopic of graph visualization we want to mention is the issue of clustering of graph data. Clustering of data can reduce the amount of information to show noticeably. Thereby, nodes with certain attributes\(^3\) are collapsed or merged into several “super nodes”. This reduces the amount of visual information to analyze by the user and can potentially improve the performance of the drawing routine. Further details can be found in [HMM00](pages 14-16).

This is only an overview of the currently available visualization techniques for graph visualizations. Having introduced these techniques and our goals for the UViz system, we will now examine the UViFace frontend component

---
\(^2\)This way, magnifying and shrinking the graph in a visual way is accomplished.
\(^3\)As a possible condition, nodes within a certain distances can be clustered.
CHAPTER 6. UVIFACE - THE UVIZ VISUALIZATION INTERFACE

of UViz in more detail. The remaining sections of this chapter will therefore be arranged in the following order.

- In Section 6.2, we will present the graph visualization component and its applied techniques (see also this section).
- The user interface control panels provided by UViFace will then be discussed in Section 6.3.
- In Section 6.4, we will show several user interface tasks and discuss how UViFace handles these.
- We close this chapter with a small summary in Section 6.5.

We start with the graph visualization component of UViFace.

6.2 UViFace Graph Visualization

Our graph visualization component contains several common features, as well as some URDF-specific ones. Among these are the visualization of the reasoning produced lineage information and the visualization of changes/updates to the rules and queries. We therefore divide this section further into three subsections. Subsection 6.2.1 deals with the default mode of UViFace. In this mode, UViFace visualizes the URDF produced dependency graph (see Subsection 3.2.1) as an entity-relationship graph. Subsection 6.2.2 discusses the explanation mode provided by UViFace. In this mode, the user is able to investigate the URDF produced lineage data. Finally, in Subsection 6.2.3, we will present the comparison mode of UViFace. Here, the user can examine the differences between several queries and changes to the used URDF rules in a visual way. Let us start with the default mode of UViFace.

6.2.1 Default Mode

In this subsection, we talk about the default mode of UViFace. More precisely, we will examine the visual encodings, the applied graph layouts, the possibilities to interact with and navigate the visualization directly (ignoring the user interface) and to dynamically load data from the knowledge base. In general, we visualize the dependency graph produced by URDF (see Subsection 3.2.1). More precisely, we visualize a transformed dependency graph. Here, all facts produced during the reasoning are visualized in a connected entity-relationship graph. Thereby, nodes represent entities of the knowledge base, whereas directed edges represent relations between them. As an example, the fact ⟨Woody All, isMarriedTo, Soon-Yi Previn⟩ leads to a directed edge from the node for the entity Woody Allen (the subject of the fact in this case) to the node for the entity Soon-Yi Previn (the object of
true fact, represented by a green edge
false fact, represented by a red edge
the size of the node and its label font are determined according to the node degree
true fact, represented by a green edge

Figure 6.3: An example UViz graph for the query: *To whom is Woody Allen married?*

the fact in this case). There are multiple edges possible between nodes. Consequently, our entity-relationship graph is a *multigraph*. An example of a visualized graph can be seen in Figure 6.3. Thereby, the typed in query was: *To whom is Woody Allen married?*

Let us now discuss several features to explore in Figure 6.3, by starting with the *visual encodings*.

**Visual Encodings**

First of all, the nodes have different sizes\( ^4 \), according to their number of connected edges (known as *degree*\( ^5 \)). The more connected edges a node has, the bigger the node. This way, its importance for the graph is reflected. The nodes themselves have a dark-red fill color and a yellow border color, in their general form. This composition looks nice. Moreover, the entity name printed on the node is easily readable. Because the URDF reasoner uses hard rules (see Subsection 3.1.2) and its MAX-SAT solver (see Subsection 3.2.3) to assign truth values to facts, we can have false and true facts. We encode these truth values by applying green and red as colors. True facts are represented by green-colored edges between entities. Consequently, false facts are represented by red-colored edges. Moreover, there exist confidence values (in the range \([0,1]\)) for the facts. These values are computed by URDF and represent the degree of believe in the correctness of a certain fact. These confidence values are reflected in UViFace, by assigning alpha values between 0 and 1 to the edges. Therefore, a dark-colored edge between two entity nodes

---

\(^4\)There are 8 different sizes for nodes possible in the current version of UViFace.

\(^5\)In the case of a directed graph, the *degree* of a node is computed as the sum of *indegree* (number of edges incoming) and *outdegree* (number of edges outgoing).
represents a high confidence for the fact that is represented by this particular edge and the two connected nodes. As a consequence, a light-colored edge means a low confidence for the respective fact. In addition, we use shadows for the nodes and edges, to simulate the effect of depth in the graph. This leads to a more 3D looking visualization.

Having discussed the general visual encodings, we will now present the used graph layouts.

**Graph Layouts**

For our entity-relationship graph, we use two layouts already mentioned in Subsection 6.1. The first one is the radial layout. This layout is discussed now.

**Radial Layout** The used radial layout essentially is a tree layout. It places tree nodes in a radial shape around a center node, thereby laying out different tree depths along circles of growing radii. We took the layout from Flare (see Section 4.3) and use it the way it is provided. The layout algorithm itself is implemented by Jeffrey Heer [Heeb]. The algorithm is an adjustment of a technique presented in [YFDH01]. Our version of the layout is shown in
Figure 6.4. This is an expanded version of the graph shown in Figure 6.3. Expanded here means that we dynamically loaded all related data about the entities Sharon Stone and Soon-Yi Previn, by using the URDF reasoning engine. This figure shows the features of the used radial layout in a clear way. The layout uses an “artificial” root node to layout the nodes and edges. This “root node” is necessary due to the radial layout being a tree layout, as mentioned before. Thereby, the nodes are positioned at increasing radii around this defined root node. We can choose another node as root node and get a changed layout, while the changes are animated smoothly. Moreover, it is possible to increase or decrease the radii between the different “depth levels” of the graph.

The second layout type provided by UViFace is the force-directed layout.

**Force-Directed Layout** The force layout of UViFace originates from Flare and has not been changed by us so far. It positions nodes, based on a physical simulation of forces interacting with each other. Thereby, nodes simulate charged particles with a mass, and edges act as springs between these charged particles. In the general form, nodes repel each other. Thereby, drag forces, comparable to air resistance, are applied to this force simulation. Moreover, parameters like the spring length for the edges and the mass for the nodes can be adjusted as needed. One of the important underlying physic forces computed is the N-Body force. This force simulates the already mentioned charged particles with a pairwise interaction, e.g., gravity or an electrical charge. The algorithm used by Jeffrey Heer is the one explained by J. Barnes and P. Hut in [BH86]. The application of this layout can help to reorder the graph in a cleaner way, as the nodes potentially build clusters. This can lead to a graph with less edge crossings. We use the force layout in a dynamic way, running it incrementally on a permanent basis. The drawback of this layout is the increased demand for processing resources. But, as we will see in Subsection 6.3.1, we are able to disable the node and edge shadows. This increases the performance substantially. An example graph using the force layout is depicted in Figure 6.2 b).

After having presented the layouts used for the entity-relationship graph, we will now focus on the interaction and navigation techniques used for the visualization.

**Interaction and Navigation**

We implemented multiple interaction schemes. Thereby, we use the controls provided by Flare in adjusted versions. These controls are explained in the next paragraph, before we show our provided way of navigating the entity-relationship graph.

---

6In the case of the UViFace radial graph, depth levels represent the path lengths starting in the artificial root node.
Graph Interaction  We support the most common controls used for graph visualizations. Some of them are applied in our already used example graph, depicted in Figure 6.5. The first interaction control to mention is the hover control. With hovering, we want to show to the user that he has moved the mouse pointer over a specific node or edge. This way, an interactivity with the visualization is already indicated to the user by selecting a visual item of the graph. Thereby, our chosen hover color temporarily fills the nodes and edges with an intensive blue. The next control to talk about is the drag & drop control. With the help of dragging and dropping, the user can reposition the nodes as needed\(^7\). This control over the nodes is helpful to reposition nodes not positioned in a pleasing way by the layout, as this may be the case from time to time. Especially in the force layout, drag&drop increases the feeling of the user to have control over the graph layout. In addition, this feature increases the dynamic appearance of the computed force layout. An additional control used is the tooltip control. This control is applied with the purpose of displaying additional information, by pointing the mouse over a node or edge.

Now, it is time to present our way of navigating the graph, hereby using an incremental exploration technique.

Graph Navigation  In fact, we combine several techniques to achieve an incremental exploration of the graph. What does incremental exploration mean in our case? We use the GraphDistanceFilter class of Flare for our purpose. This filter is used to hide nodes and edges that are not to within

\(^7\)It makes only sense to drag and drop nodes, as edges are automatically adjusted to changed source and target node positions.
Figure 6.6: Comparing the result of a dynamic data loading: In a), we see again the result graph of the query: *To whom is Woody Allen married?* In b), we dynamically loaded all related facts of the entity *Hugh Grant*, by using URDF.

a specified distance to the current root node. By incrementing the size of the node neighborhood, regarding the current root node, we can expand the currently visualized graph. By changing the root node, the radial layout positions the new root node in the middle of the screen and shows only the nodes and edges within the specified distance to this root node. Consequently, with the help of these techniques, we can restrict the number of simultaneously visualized nodes and edges. Moreover, we can incrementally navigate the graph. This works also in the case of the force layout. For the force layout, this can reduce the processing demand substantially, in case the amount of shown nodes and edges is reduced noticeably by the distance filter. As a further technique, UViFace provides *panning & zooming*. With the help of the panning control, the user can reposition the whole graph as one visual element on the screen. In addition, zooming can magnify or shrink the graph “virtually”. This way, the user is able to focus on interesting parts of the visualized graph.

Apart from the navigation within and the interaction with the graph, we provide a mechanism to dynamically load data from the knowledge base (if requested). This feature will be discussed in the proceeding subsection.

**Dynamically Data Loading**

The dynamical loading of data is accomplished by triple-clicking on a node. Again, we use the query from Figure 6.3 to show an example in Figure 6.6.
In this case, two queries with only one literal\(^8\) per query are built. The first query contains a literal with the label of the clicked node as first argument. The relation, as well as the second argument of the literal, are variables. The literal of the second query contains similar values. But this time, the second argument is the node label, and the first argument and the relation are variables. Again, each node is a graphical representation of a knowledge base entity. The node labels are string representations of the entities and reflect their entity name in the knowledge base. As a consequence, the first query returns every fact that contains the knowledge base counterpart of the clicked node as a first argument. This means, we get all entities representing nodes and their relations representing edges outgoing from our clicked node. In the case of the second query, we receive facts whose graphical representations contain the clicked node as target node. Currently, we use the rules of the reasoner during the reasoning process to dynamically load information related to the clicked entity node. This approach always produces additional reasoning information in the form of lineage data. A possible and easy addition to that would be to ignore the URDF rules during reasoning, to dynamically load information. This would reduce the amount of data and the execution time of the server substantially. But this would also result in less information (less related facts) to visualize to the user. Alternatively, it is planned to provide the choice between applying and ignoring the URDF rules for this task.

After this discussion of the default mode, we want to present the two special modes provided by UViFace. The first one is the explanation mode. The second special mode is the comparison mode. We want to start with the explanation mode.

### 6.2.2 Explanation Mode

The explanation mode of UViFace visualizes the lineage information of a fact. As already mentioned, in the default mode, we visualize an entity-relationship graph. Thereby, nodes represent entities of the knowledge base. Edges between two distinct nodes represent relations between these knowledge base entities. Consequently, a fact produced by URDF is visually represented by two entity nodes and a connected edge. Thus, we do not show facts directly. To show lineage information about a fact, we therefore regard the edges as “virtual” representatives of the facts in the entity-relationship graph. By double-clicking on an edge, the user can enter the explanation mode. In this mode, we change the layout to the node-link tree layout of Flare. We want to discuss our explanation layout in more detail now.

---

\(^8\)Again, a literal represents an RDF triple of the form \((subject, predicate, object)\).
Figure 6.7: The figure shows the lineage tree of the fact \((Woody\_Allen, isMarriedTo, Sharon\_Stone)\), with a certainty of 0.33 and the truth value false.

The explanation layout is a tree layout, as already mentioned. The lineage (explanation) tree (see Subsection 3.2.2) of a clicked fact is built over several steps. An example lineage tree is depicted in Figure 6.7. How is this lineage tree constructed? First, the two nodes connected by the clicked edge are hidden. These nodes represent the subject and the object of the fact whose lineage information should be visualized. All edges between those nodes are hidden as well. Then, a new lineage node, which represents the fact, is constructed. Because we build a lineage tree and want to show some data context, we visualize all nodes at most one edge away from the original subject and object node of the respective fact. In addition, the edges incoming into and outgoing from the original subject and object node are visualized as well. The original subject and object node, as well as the edges between them, are merged in this way. Then, we construct the lineage tree belonging to the specified fact. If this fact exists only in the knowledge base (could not be inferred through reasoning), then there is no lineage subtree beneath the fact, but only the context nodes and edges. Otherwise, there is a lineage tree. And the tree itself represents a flattened version of the complex lineage data. This means, a derived fact can occur multiple times within the lineage data, and therefore, within the lineage tree. This stems from the application of the rules during the execution of the URDF reasoner. Distinct facts can be produced at several levels, within different rules. Visualizing these potential duplicate nodes can increase the size of the lineage tree noticeable, but shows the derivation steps as they were applied. At this
point, we should mention that we use a mixed granularity for the lineage layout. We show entity nodes, in combination with fact nodes and intermediate nodes. What are these intermediate nodes? As shown in Figure 6.7, the lineage tree contains one smaller node. This node represents a conjunction (for an explanation see Subsection 3.2.2, the part about lineage). In this case, the fact \((Woody\_Allen, is\text{MarriedTo}, Sharon\_Stone)\) was derived through the grounded soft rule:

\[
\text{isMarriedTo}(Woody\_Allen, Sharon\_Stone)[0, 33]|false\]
\[
\leftarrow\text{actedIn}(Woody\_Allen, Picking\_Up\_The\_Pieces)[0, 97]|true\]
\[
\land\text{actedIn}(Sharon\_Stone, Picking\_Up\_The\_Pieces)[0, 97]|true\]
\[
\land\text{notEquals}(Woody\_Allen, Sharon\_Stone)[1.0]|true\]

There exists another derived fact, in which \(Woody\ Allen\) could also be married to \(Hugh\ Grant\). Moreover, there is an additional fact, which says that \(Woody\ Allen\) could be married to \(Soon\_Yi\ Previn\) (see Figure 6.3). And this fact about \(Soon\_Yi\ Previn\), which is directly available in the knowledge base (no grounding of a soft rule necessary), has a higher probability attached. Consequently, this fact is set to true, the two derived ones are set to false. The truth values are indicated by painting a true fact node in green and a false one in red, respectively. It is possible for the user to click on any node of the lineage subtree, and the subtree of the respective node will be collapsed in a smooth animation. Clicking a second time on the same node expands the subtree again. This way, the user can reduce the visualized information and focus on interesting parts of the lineage tree. Moreover, by double-clicking any node of the lineage tree, UViFace will change from explanation mode back into the default mode again.

After the presentation of the explanation mode, we will now continue with the discussion of the comparison mode.

### 6.2.3 Comparison Mode

The comparison mode was developed to compare a new query result with an already visualized graph. In this mode, we keep all the received data in the graph until the user leaves the comparison mode again and submits a new query. This mode has two application scenarios. In the first one, the user changes rules used by the URDF reasoner. When the UViFace component is in comparison mode and the user sends the same query as before, the user can see the changes induced by the rule changes. In the second scenario, the user sends a second query, after submitting the first one and changing to
Figure 6.8: Comparing the results of the queries *To whom is Woody Allen married?* and *To whom is Kevin Bacon married?*. In a), we see the result of the Woody Allen query, in b) the query for Kevin Bacon. In c), the query results are compared. Both results have the node *Sharon Stone* in common, which is marked in red in all three screenshots from UViFace.

An example of a query comparison is depicted in Figure 6.8. The first query submitted was: *To whom is Woody Allen married?* The second query submitted was: *To whom is Kevin Bacon married?* The nodes have different border colors now. What do they mean? If a node has already been in the graph before the query to compare was submitted, two cases have to be differentiated. In the first case, the node was only in the old graph and is not in the result of the current query. This means that the graph and the query result do not share this node. Then, this node obtains a black border color to indicate that this node is “old”. If the previous graph and the result of the current query share a node, then this node stays in its default border color *yellow*, just like in the default mode. If a node only occurs in the result graph of the current compare query and not in the old graph, then the border of the node is colored in white. This indicates a “new” (fresh) node.
Changes in edges are not so easy to visualize. This is due to the fact that we already use the colors red and green to indicate the truth value of an edge (fact). Therefore, we developed the *glow mode* for edges.

In the glow mode, which is only available in comparison mode\(^9\), we use several versions of the *Glow Filter* provided by Flex. This mode is also shown later on. What we essentially do is using neon-like glow colors to enhance the visual effect of different colors. The black color for old nodes is changed to a *dark grey*. This is due to the change of the UViFace background color to black. This black background color enhances the glow effect even more. In glow mode, we can now visualize changes for edges as well. The glow filters consume a lot of memory and processing power for their displaying. This leads to a severe performance decrease if the glow mode is used in combination with the force layout. Therefore, the user can switch between default and glow mode as he likes to.

After discussing the visualization of UViFace, we want to present the user interface control panels of UViFace next.

### 6.3 Control Panels

In addition to the visualization, the user interface control panels are additional components of UViFace. An overview of all control panels is shown in Figure 6.9. The control panels were not the main task of this thesis. Nevertheless, we achieved to add some interesting features to support the user in

\(^9\)In fact, it is a special feature of the comparison mode.
its ambition to understand the visualized information. We will not examine the panels in all details, as this would fill several pages. In this section, we will only give an overview of the control panels provided by UViz. At first, we want to introduce the statistics and options panels.

6.3.1 Statistics Panel and Options Panel

The Statistics panel, as well as the Options panel, contain several sub-panels, available via menus. We omit a distinct picture of the Statistics panel. Nevertheless, we want to introduce the Statistics panel now.

The Statistics panel consists of sub-panels for server-related statistics, e.g., the time needed to execute the query on the server. There are three different sub-panels for this purpose. One of these panels provides statistics regarding the latest returned query result. The second panel of this type displays the same values for all submitted queries. This means that all query statistics so far are accumulated here. The third sub-panel uses the total values to provide the averaged values for the query statistics. Moreover, an additional menu panel shows the number of used nodes and edges for the current graph. This fourth panel also shows the current memory consumption and the current frame rate of Flash Player. While the Statistics panel only shows values, the Options panel lets the user change values.
The Options panel provides two sub-panels. The first one, which is not depicted here, gives the user the possibility to directly access nodes (entities) and facts. This accessing is possible, even if the clicked nodes or edges are not visible on the screen at that moment. The second panel contains many visualization-related parameters to tweak and play with. This is shown in Figure 6.10. There are options to change the distance parameter of the graph distance filter and the radius increment for the radial layout. In addition, several options exist to change parameters for the nodes and edges. Additionally, it is possible to activate or deactivate the shadows. The length and intensity of the shadows can be changed as well. Moreover, the force layout can be adjusted in different ways. Additionally, there are several options to configure the lineage layout, e.g., the distance between child nodes in a subtree or the distance between different levels of the lineage tree are adjustable. Other options are the switches between the radial and the force layout, and between the enforcing of the visualization bounds for the layouts and the ignoring of the bounds. At the bottom of this sub-panel is a button-field provided. This button-field contains buttons to re-center the visualization, reset the parameters, and to update the visualization with the given parameters. Changing parameters can be done by the user at any time during the visualization. Not all changes are applied in all modes, though.

After the discussion of the Statistics and Options panel, we will now focus our attention on the Rule panel.

6.3.2 Rule Panel
The Rule panel is divided into a hard rules sub-panel and a soft rules sub-panel. We omit the discussion and presentation of the hard rules sub-panel here. Only the soft rules sub-panel is discussed (both panels provide similar functionality, although for different purposes). The soft rules sub-panel is depicted in Figure 6.11. Going from the left to the right, we have the following options:

- There is a list of available soft rules. New rules can be added and existing ones can be deleted. Moreover, it is possible to select one particular rule or all rules. Selected rules are copied to the list of selected rules.

- Only the selected rules are used within the URDF reasoner. Selected rules can be deselected again, either rule by rule, or by deselecting all rules. This task can be accomplished by the Selected Soft Rules list component, depicted in the middle of Figure 6.11.

- By selecting a rule in the list of available rules, the literals, the name, and the weight for the current rule become visible in the third big visual component in Figure 6.11 (on the right). All these aforementioned constituents of a soft rule can be changed. Thereby, the head literal
and the body literals can be changed separately. It is also possible to add new body literals, or to delete existing ones. The weight to indicate “importance” of the soft rule can be adjusted as well.

- Finally, the rules from the selection list can be updated. Only these soft rules will be used for the next query. If needed, the list of available rules can be reset. By doing so, they are reloaded from the server.

As already mentioned, a similar architecture is used for the hard rules sub-panel. As a consequence, the complete list of soft and hard rules used for the URDF reasoner can be changed, according to the user needs. We will now close this section with a discussion of the Query panel.

### 6.3.3 Query Panel

The Query panel, which is depicted in Figure 6.12, is potentially one of the most frequently used control panels. We tried to create an easy to understand panel that shows only the most necessary information and provides only the most needed features. URDF uses a SPARQL-like query model and deals with RDF data. Therefore, we decided to provide a visual query representation that reflects the aforementioned models appropriately. Consequently, the Query panel displays literals in the form of \((subject, predicate, object)\). In the current version of UViFace, the query panel offers the possibility to
use p eight literals to define a query. Thereby, we designed the panel to only show one literal at a time. The user can switch seamlessly between the eight literals. To accomplish this task, there are eight pairs of click and select buttons provided. With the help of these, the user can select a literal to change it, or to mark it for deletion. The deletion of literals clears their two arguments and the connecting relation (predicate). Thereby, the predicate field provides autocomplete functionality to specify a relation variable, or to choose one of the URDF predefined relations. This component is taken from Hillel Coren\textsuperscript{10}. Furthermore, the Query panel provides buttons to enter or leave the compare and glow mode.

Before we go to the user interface tasks, we want to mention that all of the four main panels can be shown or hidden as needed. In case the user points the mouse to the top of the screen, the Query panel shows up. Pointing to the right makes the Statistics panel visible. Pointing to the left shows the Options panel. And finally, pointing to the bottom of the screen makes the Rule panel show up. This interface scheme of controlling which panel is visible gives the user the possibility to focus his attention on the visualization itself. Let us now proceed with the user interface tasks.

\subsection*{6.4 User Interface Tasks}

In this section, we want to present four different user tasks common for the usage of the UViz system. Each of these tasks has the purpose of exemplifying a certain usage scenario for the UViFace component. Thereby, a combination of actions is performed by the user in each scenario. The first user task we want to talk about is the basic exploration of the visualized graph in the

\footnote{http://hillelcoren.com/flex-autocomplete/}
default mode of UViFace (see Subsection 6.4.1). The second user interface task will show the usage scenario of the explanation mode (see Subsection 6.4.2). Then, the third and the fourth user interface task will deal with the comparison mode of UViFace (see 6.2.3). More specifically, the third task presents the comparison of different queries (see Subsection 6.4.3). In the fourth task, we will exemplify changes to the rules and the so induced visual changes to the results of the same query (see Subsection 6.4.4). We start with the user task for the general graph exploration.

6.4.1 Graph Exploration

The first task can be seen in Figure 6.13. The timeline goes from the top left picture to the top right picture, and ends in the bottom left picture. Now let us analyze the content of the figure:

1) In the first picture, the user receives the result to the query: To whom is Woody Allen married? The user can now examine the result graph and decide how to proceed.

2) The user is interested in Soon-Yi Previn. Consequently, the user changes the root node to Soon-Yi Previn. This is depicted in the second picture. Now the user wants to gather additional information about Soon-Yi Previn. Therefore, the user dynamically loads data to this entity.

3) In the third picture, the facts related to Soon-Yi Previn are loaded and added to the graph. The appearance of the graph changes accordingly. Now, the user wants to change his focus to Sharon Stone.

4) Sharon Stone was set as new root node in the fourth picture.

5) The user wants to get more information to Sharon Stone. Consequently, the user dynamically loads information related to Sharon Stone. Moreover, the root node is changed to Woody Allen. This rearranges the graph to a cleaner appearance. Finally, the user wants to experience the force layout.

6) The layout of the default mode is changed in the sixth picture. Now, the layout is the force layout. The nodes are arranged in clusters around the nodes with the highest degrees.

The scenario shown is only one of the many possible scenarios. But this example indicates how the user can examine the visualized information in the default mode. Now, we want to present the second user interface task. In this scenario, the user wants to investigate the lineage information of facts in the explanation mode.
CHAPTER 6. UVIFACE - THE UVIZ VISUALIZATION INTERFACE

Figure 6.13: Exploring the result graph in the default mode of UViFace.
6.4.2 Reasoning and Explanation

The second task is depicted in Figure 6.14. The timeline for the pictures is the same as for the first task (left to right, top to bottom, right to left). What can we see in the figure?

1) In the first picture, the user receives the result to the query: Where does Al Gore live? It is now possible for the user to work with the result graph.

2) In the second picture, the user hovers over the edge livesIn between Al Gore and Washington D.C. The user now clicks this edge to see the lineage information for the fact that is represented by the clicked edge.

3) The third picture shows the lineage tree for the fact \((\text{Al}_\text{ Gore}, \text{livesIn, \ Washington, }_\text{D.C.})\). It is a rather big lineage tree, with only Tipper Gore as context information. The user now has to decide how to proceed. The user decides to explore the lineage tree in more detail.

4) The user uses panning and zooming to arrange the graph in a way that allows the user to explore the graph according to his current interest. In addition, the user collapses several subtrees to reduce the amount of information shown on the screen. The expansion control in UViFace provides the user with the necessary tool to collapse and expand the nodes. This gives the user the opportunity to focus his interest.

5) After examining the lineage tree, the user decides to return to the general visualization. Now, the user wants to learn more about Tipper Gore, the wife of Al Gore. The user therefore hovers over the edge livesIn between Tipper Gore and Washington D.C.

6) The user clicks the aforementioned edge livesIn between Tipper Gore and Washington D.C. This edge represents the fact \((\text{Tipper}_\text{ Gore, livesIn, \ Washington, }_\text{D.C.})\) in the entity-relationship graph of the default mode. After clicking the edge, UViFace changes to the explanation mode. The lineage information to the previously clicked fact is shown now. As we would expect from the shown lineage graph to the fact \((\text{Al}_\text{ Gore, livesIn, \ Washington, }_\text{D.C.})\), Tipper Gore is known to live in Washington D.C. Moreover, this fact could not be inferred through additional applications of the URDF rules. Thus, only the context node Al Gore is shown in the explanation mode for the fact \((\text{Tipper}_\text{ Gore, livesIn, \ Washington, }_\text{D.C.})\).

The depicted user interface task shows how the user can use the explanation mode of UViFace to examine the lineage information of a derived fact. The next task will deal with the comparison of different queries, which are related, though. Thereby, UViFace is in its comparison mode.
Figure 6.14: Examining the lineage information of distinct facts in the explanation mode of UViFace.
6.4.3 Comparing Query Results

The third task is depicted in Figure 6.15. The timeline is the same as before. What do the 6 pictures tell us?

1) The first picture shows the state of UViFace after receiving the result to the query: *To whom is Woody Allen married?* There are several entity nodes present in the result graph. The user can now decide what to do next. The user decides to use the comparison mode of UViFace to compare the current query to the new (related) query: *To whom is Hugh Grant married?*

2) The user enters the comparison mode and submits the query about Hugh Grant. The resulting combined graph is depicted in the second picture of Figure 6.15. Both queries share the entities *Woody Allen, Hugh Grant, and Small Time Crooks.* This is indicated by the yellow node borders. Why is that so? In URDF, there currently exists a soft rule that says if two entities played in the same movie and are not the same entities, then they are married. And because *Woody Allen* and *Hugh Grant* played both in the movie *Small Time Crooks,* the aforementioned soft rule applies here. *Soon-Yi Previn, Sharon Stone,* and *Picking up the Pieces* were present only in the first query about Woody Allen. This circumstance is indicated by the black borders around the respective entity nodes. The aforementioned entities (except for *Woody Allen*) are not related to *Hugh Grant* and could therefore not be inferred. This is the reason they do not exist in the result of the current query about *Hugh Grant.* The entities *Sharon Small* and *About a Boy* are contained only in the second query about Hugh Grant. These fresh entities are represented by their respective white bordered entity nodes. The reason is that *Sharon Small* and *About a Boy* could only be inferred for *Hugh Grant.*

3) The user submits again the query: *To whom is Hugh Grant married?* Now, we can see that only the entity nodes that were indicated as “old” in the previous graph stayed in their comparison color. The nodes for the entities *Sharon Small* and *About a Boy* are now colored with yellow borders. This makes sense, as the formerly “fresh” entity nodes now are shared by the result graph to the *Hugh Grant* query and the graph that was extended by exactly the same query result one step before.

4) After submitting the same query twice, the user decides to send a new query. Therefore, the user switches back to the default mode and submits the (related) query: *To whom is Sharon Stone married?* The result graph is shown in the fourth picture of Figure 6.15.
RESULT GRAPH OF QUERY: “TO WHOM IS WOODY ALLEN MARRIED?”

ENTERED COMPARISON MODE AND SUBMITTED QUERY: “TO WHOM IS HUGH GRANT MARRIED?”

AGAIN, SUBMITTED QUERY: “TO WHOM IS HUGH GRANT MARRIED?”

ENTERED DEFAULT MODE AND SUBMITTED QUERY: “TO WHOM IS SHARON STONE MARRIED?”

ENTERED COMPARISON MODE AND SUBMITTED QUERY: “WHO ACTED IN PICKING UP THE PIECES?”

SUBMITTED QUERY: “WHERE DOES KEVIN BACON LIVE?”

FIGURE 6.15: COMPARING DIFFERENT QUERIES FOR SIMILARITIES IN THE COMPARISON MODE OF UVIFACE.
5) In the fifth picture, we see the result graph of the query: *Who acted in Picking up the Pieces?* The user wants to know if other persons than *Woody Allen* and *Sharon Stone* acted in the specified movie. But there are no other persons in the knowledge base, nor could URDF infer any new ones. The entity nodes for *Kevin Bacon* and *He said She said* have black borders, which represents the circumstance that these entities are not related to the result of the previously submitted query. Finally, the user submits a last query.

6) We saw the entity node for *Kevin Bacon* in the last two pictures. *Kevin Bacon* played in one movie together with *Sharon Stone*. Now, the user shows some interest in *Kevin Bacon* after finding out this relationship. Consequently, the user submits a query about *Kevin Bacon*. The query is: *Where does Kevin Bacon live?* Thereby, the query is submitted while still being in comparison mode. The result graph is depicted in the sixth picture of Figure 6.15. The previous graph is completely contained in the new combined result graph. This means that the dependency graph for the previous query is contained in the dependency graph for the new query. The conclusion is that *Kevin Bacon* is strongly related to *Sharon Stone* and *Woody Allen* (because of the movies they played in together and the URDF rule shown in the next subsection).

The presented user interface task exemplifies how the user can work with the comparison mode of UViFace. But this was only the first usage scenario of the comparison mode. It is also possible to compare the difference between results of the same query, after changing hard rules or soft rules of URDF. This scenario is discussed in the next subsection.

### 6.4.4 Comparing Rule Changed Results

The fourth and last task we want to present is displayed in Figure 6.16. Again, the timeline for the pictures is the same as before. In this task, the user sends the same query several times, thereby changing the rules for URDF. By using the comparison mode of UViFace, the user is then able to see changes induced by the changed rules. Let us now examine the pictures shown in Figure 6.16:

1) The first picture shows the result to our beloved query: *To whom is Woody Allen married?*

2) In the second picture, the user updates the list of hard rules. The list is empty now. Consequently, URDF will not use any hard rule for the reasoning.

3) The user enters the comparison mode and submits the same query as before. The result graph changes. Now, we have no red-colored edges
Result graph of query: "To whom is Woody Allen married?"
Hard rules updated with empty list
Entered comparison mode and submitted query: "To whom is Woody Allen married?"
Deselect soft rule 16 and update soft rule list to use for URDF
Again, submitted query: "To whom is Woody Allen married?"
Entered glow mode and changed background color to see edge changes

Figure 6.16: Changing the URDF rules and comparing the changed query results of the same query in the comparison mode of UViFace.
The user deselected the soft rule 16 from the list of used soft rules. Soft rule 16 has the following form:

$$\text{isMarriedTo}(\?X, \?Z) \iff \text{actedIn}(\?X, \?Y)$$
$$\land \text{actedIn}(\?Z, \?Y)$$
$$\land \text{notEquals}(\?X, \?Z)[0.002]$$

Now, the user updates the list of soft rules and submits the same query about Woody Allen again.

5) The new result graph is shown in the fifth picture. The user can now see that the deletion of the soft rule shown above leads to several changes. Only the nodes for the entities Woody Allen and Soon-Yi Previn are colored with a yellow border. This is an indication that only those two entities occur in the old and the new graph. This makes sense, as only the fact ($\text{Woody}_{-}\text{Allen, isMarried}_{-}\text{Soon-Yi}_{-}\text{Previn}$) is in the knowledge base. Without the respective soft rule, no further possible spouses of Woody Allen can be inferred by URDF.

6) To even visualize the change for the whole fact, the user switches to the glow mode of UViFace and changes the background color. This is shown in the sixth picture. Now, the user is able to also see the changes in the edges. This is usually useful if only the truth values for edges (facts) changes. Then, the glow mode can be used to visualize these changes by applying the glow filters to the nodes and edges.

This was the last user task we wanted to present. Consequently, this last user interface task closes our presentation of the UViFace component of UViz. We will now close the whole chapter with a small summary of the information just presented.

### 6.5 Summary

This chapter discussed UViFace, the user and visualization interface of UViz. We introduced this component of UViz with some overall information. Then, we gave reasons and objectives for the design and implementation of UViz and its interface UViFace. Afterwards, an introduction to information visualization and its sub-category graph visualization was presented in Section 6.1.
In Section 6.2, UViFace was discussed in more detail. We presented the three different modes provided by UViFace in this section. Moreover, we examined how to navigate and control the visualization in UViFace. In Section 6.3, the user interface control panels of UViFace were introduced. These control panels enable the user to control the visualization in an intuitive and flexible way. In the last section before this summary, we finally presented four different user interface tasks to exemplify common usage scenarios of UViFace. In the next chapter, we will conclude this thesis and discuss limitations and potential future plans for the UViz system.
Chapter 7

Discussion

7.1 Conclusion

In this thesis, we presented the graph-based visualization system UViz. This system has been developed to visualize the specific features introduced by the URDF reasoning framework (see Chapter 3).

The features of URDF include the use of Datalog-style (recursive) soft inferencing rules, as well as hard rules, which represent mutually exclusive sets of facts. Moreover, URDF provides a novel approximation algorithm to solve the Weighted MAX-SAT problem. With the help of these techniques, URDF applies logical reasoning to answer given queries. By accessing a knowledge base (currently the YAGO knowledge base), URDF can infer facts not originally existing in the knowledge base. Moreover, URDF is able to deal with uncertainty inherent in the underlying knowledge base and resolve inconsistencies between the knowledge base and the used inference rules. As a consequence, URDF achieves to present reasoning-extended query results to the user and even uncertainty information about the inferred data. But this enhanced expressiveness also bears a tough challenge for a possible visualization of the increased amount and complexity of the returned information.

Therefore, we had several objectives for a possible visualization of the URDF returned results in mind. The final visualization should be available not just for a small group of experts, but for everyone interested in working with the visualization and URDF. Additionally, the visualization should be easily accessible by the user. Moreover, the URDF system and the visualization should operate on their own and communicate by defined interfaces. Additionally, the visualization had to find ways to present the query results in an easy to understand and visually enticing way. Apart from the goals just mentioned, many other objectives were defined for the design of the intended visualization system. To challenge these objectives, we developed UViz. The goals achieved with the UViz system in this thesis are discussed now.
Our first objective was to find an application type for our UViz system. We decided to build UViz as a web-enabled application whose visualization uses a common web browser as “presentation frame”. Consequently, the architecture of UViz follows the client-server model (see Subsection 4.1.1). More specifically, we follow the Rich Client approach (by using Flex), where the web application runs inside a distinct runtime inside the browser, and the client can work autonomically to accomplish several tasks on its own. This way, the visualization can be executed on the client without the need to constantly call the server for every small change in the visualization. Thereby, URDF is executed on the server. Consequently, we decoupled the processing-intensive reasoning from the visualization-related computations. This decoupling is supported by the development of the URDF API, which represents an intermediate layer between the server-side URDF reasoner and the client-side visualization (see Subsection 5.1.2). To run inside the browser and deliver a suitable visualization for the query results produced by URDF, we used several state-of-the-art technologies.

To run the visualization frontend of UViz in the browser, we found Adobe Flash Player (see Subsection 4.2.2), along with Adobe Flex (see Section 4.2) and the BlazeDS data service (see Subsection 4.2.3), as the software of our choice. Flash Player provides a decent performance and a lot of graphical capabilities. Moreover, Flex offers a rich set of features and good development tools. Moreover, BlazeDS lets us easily connect our Java version of URDF on the server with the visualization Flex frontend on the client. In addition, BlazeDS enables UViz to use the space efficient transmission data format AMF (see Subsection 4.2.3) to transmit data between the client and the server. Consequently, we could improve the response time of UViz, when queries are submitted. Besides these architectural and performance-related achievements of UViz, we succeeded in developing a visualization that visualizes the URDF produced information in a decent way. The visualization itself is a crucial component of UViz. We implemented a dynamic and feature rich user and visualization interface for UViz, called UViFace. This could be achieved using the Flare visualization toolkit (see Section 4.3), which is an ActionScript library for graph-based visualizations in the Flash Player.

With the help of all these tools and our general architecture for UViz, we were able to design and implement UViz such that several visualization-related achievements became possible.

The UViz visualization interface UViFace is able to present a visually appealing entity-relationship graph in its default mode. This is the general way to visualize the returned query results (see Subsection 6.2.1). Thereby, UViFace provides two different layouts, a radial and a force-based layout. Consequently, the user can switch between a clean, but static layout, and a dynamic self-organizing layout. In addition, UViFace offers the possibility to dynamically load data related to a certain knowledge base entity, repre-
sent by a node in the graph. This way, the user can explore the knowledge base incrementally. Further, we achieved to integrate different controlling schemes, e.g., the dragging of nodes. Additionally, UViFace provides a flexible graph navigation by the usage of panning and zooming. Thereby, parts of the graph can be hidden in UViFace. Moreover, by using the explanation mode of UViFace, we achieved to visualize the important lineage information produced by URDF in a convincing way (see Subsection 6.2.2). In this mode, we provide a lineage tree, whose different subtrees can be collapsed and expanded as needed. This gives the user the possibility to easily examine the produced reasoning information. In addition, UViFace provides a special comparison mode (see Subsection 6.2.3). In this mode, UViFace is able to compare the results of different queries. Moreover, it is even possible to compare the results of submitting the same query twice, by applying a changed set of rules the second time.

Apart from the graph visualization, UViFace provides a several user interface control panels. There are distinct interface panels for queries, visualization statistics, visualization options, and to manage the used soft and hard rules. The user has the possibility to adjust the visualization by defining several visualization related parameters in the options panel. The statistics panel provides multiple server and graph statistics. The rule panel, on the other hand, enables the user to add, delete and change rules. Last but not least, the query panel provides an intuitive query interface, which offers several special features.

Having discussed the context and the achieved objectives of the UViz system, we want to talk about the system limitations now.

7.2 Limitations

In our current version of UViz, we are restricted to a 2D visualization. This is due to Flash Player 9 as our target platform and the Flare visualization toolkit. There is no 3D capability provided by Flash Player 9, at least not in a native and performance efficient way\(^1\). Moreover, we would not be able to use Flare anymore. The current UViz system is not restricted to Flash Player 9, but cannot use the improved graphical capabilities of the Flash Player 10. Thus, our performance gains with the current Flash Player version\(^2\) are limited. Another limitation is the use of the layouts. We use two different layouts for the entity-relationship graph and an additional one to display the lineage information. No further layout options are implemented in the current version of UViz. Additionally, no clustering algorithms are used so far.

\(^1\)There exists several 3D engines for flash. But these engines are external libraries not natively integrated into the Flash Player.

\(^2\)The current version is 10.1 http://get.adobe.com/de/flashplayer/, 08.07.10.
But we have many ideas for future extensions and enhancements to UViz (and specifically UViFace). We want to present these ideas in the following section.

7.3 Outlook

The use of additional layouts or even specifically designed ones are feasible additions for a later version of UViz. As the current version uses the existing Flare graph layouts in unchanged or slightly changed versions, the design of a layout specifically designed for UViz could improve the clarity and expressiveness of the result graph. Moreover, a useful further addition could be the integration of algorithms that reduce the number of edge crossings. The Flare given layouts produce pleasing results in UViFace most of the time. But because our entity-relationship graph is a multi-graph, which means there are several edges between two particular nodes possible, the number of edge crossings can be substantial. As a consequence, the graphs can potentially contain a severe number of edge crossings. So, this enhancement to UViFace could be a really useful one. In addition, the application of clustering algorithms, with the purpose of reducing the amount of data shown simultaneously on screen, is a viable option as an extension to the visualization. Moreover, as already mentioned in Section 7.2, supporting the changes to the used knowledge base via the UViFace component is a further possible extension to UViz. Besides these visualization-specific features, architectural and performance influencing extensions and enhancements to the UViz system are possible as well.

One of the big extensions possible on the software side is the change from Flash Player 9 to version 10 as target platform. With a migration from version 9 to 10, we could improve the performance of UViz. Moreover, we would be able to raise the more flat look of the visualization of UViFace into the third dimension. To use the 3D capabilities and the graphics card via the Flash Player, we would have to make some tough decisions, though. With the current version of Flare, the third dimension is not supported. And although Jeffrey Heer promised on its web site to migrate with Flare to the Flash Player 10, there is no indication so far when this migration will happen. We could use one of the free 3D engines for Flash. But again, we could not use Flare anymore and would have to change a substantial part of our code, if not most of it. Another totally different idea is to possibly change to another rich client platform like Silverlight or JavaFX. This would lead to major changes or a complete rewriting of our program code as well, though. To even extend this idea of radically changing our program code, another idea would be to develop our own visualization engine behind UViz. Depending on the used RIA framework, or even recent technologies like HTML 5 or 3D graphics accelerating APIs like OpenGL or Direct3D for the web browser, it
would be an option to transform UViz into a visualization framework. This means, we could lift up UViz to a general visualization toolkit, like the ones mentioned in Chapter 2. Either way, these possible changes would be major ones and are less feasible as future plans for UViz.

Summarizing the just discussed issues, several further extensions and improvements to UViz are possible. We now want to close this chapter, and with this, the content part of this thesis. We thank you for reading this thesis and refer to the appendix for further explanations and the demonstration of the URDF rules and the queries used in Section 5.5.
Chapter 8

Appendix

On the following two pages, we present the soft rules currently used by URDF. Moreover, we show the 8 queries submitted for our performance measurements discussed in Section 5.5.

URDF Soft Rules

So far, the soft rules for URDF are hand-crafted. To show the consequences of their application to the reasoning process, also noisy rules, e.g., rule \textit{C16}, are currently integrated into the reasoning. Nevertheless, it is easy to exclude soft rules from the reasoning process, or to add new ones. The hard rules used are omitted. The soft rules are the ones defined in [The10].

Applied Queries

The queries shown were used for the performance measurements of the implemented UViz communication, discussed in Section 5.5. These 8 queries differ in the complexity of the reasoning and in the complexity of the resulting lineage information. These queries are also taken from [The10].
People usually do not live longer than 85 years.
\[ C_1: \text{bornOnDate}(a, b) \land \text{diedOnDate}(a, c) \rightarrow (c - b \leq 85) \]

People born before 1900 typically did not live longer than 65 years.
\[ C_2: \text{bornOnDate}(a, b) \land \text{yearBefore}(b, 1900) \land \text{diedOnDate}(a, c) \rightarrow (c - b \leq 65) \]

Everyone is born before his/her parent died.
\[ C_3: \text{hasChild}(a, b) \land \text{bornOnDate}(b, c) \land \text{diedOnDate}(a, d) \rightarrow (c \prec d) \]

If somebody was born in a place and lived in that place, he/she also died in that place.
\[ C_4: \text{bornIn}(a, b) \land \text{livesIn}(a, c) \rightarrow \text{diedIn}(a, c) \]

If somebody is a citizen of a country, he/she was also born in that country.
\[ C_5: \text{isCitizenOf}(a, b) \land \text{locatedIn}(b, c) \rightarrow \text{bornIn}(a, c) \]

If somebody graduated from a university, he/she was born after the university was established.
\[ C_6: \text{graduatedFrom}(a, b) \land \text{establishedOnDate}(b, c) \land \text{bornOnDate}(a, d) \rightarrow (c < d) \]

If somebody has an academic advisor who works at a university, then he/she graduates from that university.
\[ C_7: \text{hasAcademicAdvisor}(a, b) \land \text{worksAt}(b, c) \rightarrow \text{graduatedFrom}(a, c) \]

If two people have the same academic advisor, they also graduate from the same university.
\[ C_8: \text{hasAcademicAdvisor}(a, c) \land \text{hasAcademicAdvisor}(b, c) \land \text{graduatedFrom}(a, d) \land (a \neq b) \rightarrow \text{graduatedFrom}(b, d) \]

If two people are married, they also live in the same place.
\[ C_9: \text{marriedTo}(a, b) \land \text{livesIn}(a, c) \rightarrow \text{livesIn}(b, c) \]

If two people are married and one was born in some place, then the other person was also born in that place.
\[ C_{10}: \text{marriedTo}(a, b) \land \text{bornIn}(a, c) \rightarrow \text{bornIn}(b, c) \]

If two people are married and were both born in the same place, then one of them also lives in that place.
\[ C_{11}: \text{marriedTo}(a, b) \land \text{bornIn}(a, c) \land \text{bornIn}(b, c) \rightarrow \text{livesIn}(a, c) \]

If two different people have a child in common, then they are married.
\[ C_{12}: \text{hasChild}(a, c) \land \text{hasChild}(b, c) \land (a \neq b) \rightarrow \text{marriedTo}(a, b) \]

People are not married to their children.
\[ C_{13}: \text{hasChild}(a, b) \land \text{marriedTo}(a, c) \rightarrow (b \neq c) \]

If two people are married, live in the same place and have a child in common, then the child was born in the same place.
\[ C_{14}: \text{marriedTo}(a, b) \land \text{livesIn}(a, c) \land \text{livesIn}(b, c) \land \text{hasChild}(a, d) \land \text{hasChild}(b, d) \rightarrow \text{bornIn}(d, c) \]

If somebody is director of a movie, he/she is unlikely to be an actor in the same movie.
\[ C_{15}: \text{directorOfMovie}(a, b) \land \text{actedInMovie}(c, b) \rightarrow (a \neq c) \]

If two different people acted together in the same movie, then they might be married.
\[ C_{16}: \text{actedInMovie}(a, c) \land \text{actedInMovie}(b, c) \land (a \neq b) \rightarrow \text{marriedTo}(a, b) \]

Table 8.1: These are the 16 soft rules currently used by URDF (see [The10]).
Single-fact queries:

Where does Al Gore live?
\(Q_1: \text{livesIn}(\text{Al} \_ \text{Gore}, x)\)

Who is Woody Allen married to?
\(Q_2: \text{marriedTo}(\text{Woody} \_ \text{Allen}, x)\)

Chain queries:

Who acted in the movie Total Recall and was born in Thal, Austria?
\(Q_3: \text{actedInMovie}(x, \text{Total} \_ \text{Recall}) \land \text{bornIn}(x, \text{Thal} \_ \text{Austria})\)

Who acted together with Arnold Schwarzenegger in the same movie, and where were they born?
\(Q_4: \text{actedInMovie}(\text{Arnold} \_ \text{Schwarzenegger}, y) \land \text{actedInMovie}(x, y) \land \text{bornIn}(x, z) \land \text{notEquals}(\text{Arnold} \_ \text{Schwarzenegger}, x)\)

Who was born in Oxford, where did he or she graduate from, and who was his/her academic advisor who himself had graduated from the University of Cambridge?
\(Q_5: \text{bornIn}(x, \text{Oxford}) \land \text{graduatedFrom}(x, y) \land \text{hasAcademicAdvisor}(x, z) \land \text{graduatedFrom}(z, \text{University} \_ \text{of} \_ \text{Cambridge})\)

Star query:

Who acted in the movie Total Recall, where was he/she born, where does he/she live, when was he/she born, and who is he/she married to?
\(Q_6: \text{actedInMovie}(x, \text{Total} \_ \text{Recall}) \land \text{bornIn}(x, y) \land \text{livesIn}(x, z) \land \text{bornOnDate}(x, a) \land \text{marriedTo}(x, b)\)

Clique queries:

Who is Emma Thompson married to, and in which movies did she act in which also her spouse acted?
\(Q_7: \text{marriedTo}(\text{Emma} \_ \text{Thompson}, x) \land \text{actedInMovie}(\text{Emma} \_ \text{Thompson}, y) \land \text{actedInMovie}(x, y)\)

Which movies were directed by Martin Scorsese, and which two different people acted in this movie?
\(Q_8: \text{directorOfMovie}(\text{Martin} \_ \text{Scorsese}, z) \land \text{actedInMovie}(y, x) \land \text{actedInMovie}(z, x) \land (y \neq z) \land (y \neq \text{Martin} \_ \text{Scorsese}) \land (z \neq \text{Martin} \_ \text{Scorsese})\)

Table 8.2: These are the 8 queries (see also [The10]) used for the performance measurements of the implemented UViz communication, discussed in Section 5.5.
List of Figures

1.1 YAGO provides only concrete answers. In a), the query *Where does Al Gore live?* is specified and delivers no answer. In b), the query *To whom is Al Gore married* delivers the one and only answer *Tipper Gore.*

1.2 In a), the query result of the query: *Where does Al Gore live?* is shown. In b), an excerpt from the produced lineage data to the aforementioned query is presented. The grounded soft and hard rules are displayed in c). The pictures are screenshots, captured from an execution of URDF in the Eclipse IDE.

3.1 A small example knowledge base about computer scientists (see [The10]).

3.2 An example query graph (see [The10]).

4.1 A diagram of a 2-Tier client-server architecture for web applications (see also [OHE99], page 23).

4.2 A comparison of a common web application and a Flex application, both with a Java backend (see [Müll09]).

4.3 A schematic overview of the generation process of a Flex application (see [BEH+09], page 104).

4.4 A simple Flex application with a button declared in MXML. This example is taken from Adobe (see reference in text).

4.5 A simple comparison of ActionScript 3 and Java code. The example files are classes of the UViz application code.

4.6 An overview of the BlazeDS features (see [BEH+09], page 1054).

4.7 The Flare API package structure, visualized by Flare.

4.8 The Flare *Job Voyager* application.

4.9 The Flare main demo application, which shows the main Flare capabilities.

5.1 An overview of the visualization system *UViz.*

5.2 An overview of the Java URDF API of UViz.

5.3 An overview of the client side architecture of the visualization system *UViz*, the *UViFace* component (see Chapter 6).
5.4 An overview of the client-server communication within the UViz system. .......................... 61
5.5 The usage of the BlazeDS Remoting Service. In a), the manually configured remoting file for BlazeDS is depicted. In b), the manually defined remote procedure calls on the visualization frontend of UViz are shown. .................. 62
5.6 The example work-flow of submitting a new query over the UViz visualization interface UViFace. .......... 64
6.1 An example graph of UViFace. .................. 69
6.2 The radial layout of UViFace is depicted in a), and the force layout of UViFace is shown in b). .................. 72
6.3 An example UViz graph for the query: To whom is Woody Allen married? .......................... 75
6.4 The radially shaped UViFace graph after expanding the nodes Sharon Stone and Soon-Yi Previn of the original graph, depicted in Figure 6.3. .......................... 76
6.5 The example graph from Figure 6.3, with the hovering and tooltip control applied. .................. 78
6.6 Comparing the result of a dynamic data loading: In a), we see again the result graph of the query: To whom is Woody Allen married? In b), we dynamically loaded all related facts of the entity Hugh Grant, by using URDF. .................. 79
6.7 The figure shows the lineage tree of the fact (Woody_Allen, isMarriedTo, Sharon_Stone), with a certainty of 0.33 and the truth value false. .................. 81
6.8 Comparing the results of the queries To whom is Woody Allen married? and To whom is Kevin Bacon married?. In a), we see the result of the Woody Allen query, in b) the query for Kevin Bacon. In c), the query results are compared. Both results have the node Sharon Stone in common, which is marked in red in all three screenshots from UViFace. .................. 83
6.9 An overview of the user interface control panels of UViFace. .......................... 84
6.10 The options panel of UViFace. .................. 85
6.11 The soft rule sub-panel of the UViFace rule panel. .................. 87
6.12 The query panel of UViFace. .................. 88
6.13 Exploring the result graph in the default mode of UViFace. .......................... 90
6.14 Examining the lineage information of distinct facts in the explanation mode of UViFace. .................. 92
6.15 Comparing different queries for similarities in the comparison mode of UViFace. .................. 94
6.16 Changing the URDF rules and comparing the changed query results of the same query in the comparison mode of UViFace. .......................... 96
Bibliography


[ECD+04] Oren Etzioni, Michael J. Cafarella, Doug Downey, Stanley Kok, Ana-Maria Popescu, Tal Shaked, Stephen Soderland, Daniel S.


<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[SPA]</td>
<td>SPARQL Query Language for RDF. <a href="http://www.w3.org/TR/rdf-sparql-query/">http://www.w3.org/TR/rdf-sparql-query/</a></td>
</tr>
<tr>
<td>[SWR]</td>
<td>SWRL - A Semantic Web Rule Language. <a href="http://www.w3.org/Submission/SWRL/">http://www.w3.org/Submission/SWRL/</a></td>
</tr>
<tr>
<td>[Uni]</td>
<td>uniprot RDF: A project to provide all UniProt protein sequence and annotation data in RDF format. <a href="http://dev.isb-sib.ch/projects/uniprot-rdf/">http://dev.isb-sib.ch/projects/uniprot-rdf/</a></td>
</tr>
</tbody>
</table>