Hypermedia Presentation Generation for Semantic Web Information Systems

Flavius Frăsincar

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Frăsincar, Flavius

Hypermedia Presentation Generation for Semantic Web Information Systems/ by Flavius Frăsincar.
Eindhoven: Technische Universiteit Eindhoven, 2005. Proefschrift.

ISBN 90-386-0594-3
NUR 983

Keywords: hypermedia / information systems; Internet / ontology / knowledge management
C.R. Subject Classification (1998): H.5.4, H.5.2, H.5.1, D.2.2, H.2.4, H.3.4, I.2.4

SIKS Dissertation Series No. 2005-07
The research reported in this dissertation has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

Cover design: Paul Verspaget
Printed by University Press Facilities, Eindhoven, the Netherlands.

Hypermedia Presentation Generation for Semantic Web Information Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven,
op gezag van de Rector Magnificus, prof.dr.ir. C.J. van Duijn,
voor een commissie aangewezen door het College voor Promoties
in het openbaar te verdedigen
op maandag 20 juni 2005 om 16.00 uur

door

Flavius Frăsincar

geboren te Boekarest, Roemenië

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. P.M.E. De Bra
en
prof.dr.ir. G.J.P.M. Houben

Copromotor:
prof.dr. J. Paredaens

Preface

At the end of the software technology program of Stan Ackermans Institute at Eindhoven University of Technology I met professor Paul De Bra who offered me a PhD position on the NWO-sponsored Dynamo project. The Dynamo project focuses on the semi-automatic hypermedia presentation generation. After a short discussion with professor Paul De Bra I was immediately attracted by the research topics proposed in the Dynamo project and the knowledge and kindness of professor Paul De Bra.

The Dynamo project was a joint project with the Multimedia and Human-Computer Interaction Group of CWI, Computer Science Department of Eindhoven University of Technology, and New Media Systems and Applications Group of Philips Research. In this way I was able to participate in interesting research discussions with people from different institutes and with different backgrounds, among which I would like to mention professor Lynda Hardman, Jacco van Ossenbruggen, Lloyd Rutledge, and Warner ten Kate. The part of the Dynamo project carried out at the Eindhoven University of Technology was done in the frame of the Hera project, a more general project for modeling Web information systems.

I would like to thank professor Geert-Jan Houben, my daily supervisor and chair of the Hera project, for the research freedom and support offered during my PhD. Despite his volume of research and teaching, professor Geert-Jan had always his door open for me, ready to help me whatever was the issue. Through professor Geert-Jan Houben I had the great pleasure to meet and work on various database subjects with professor Jan Paredaens and his colleagues at University of Antwerpen. I would like to thank professor Paul De Bra for his continuous support during my years of PhD studies. I would like to thank both professor Paul De Bra and professor Geert-Jan Houben for the teaching opportunities that they gave me in the areas of databases and Web information systems.

I also want to thank all the people I worked with during my PhD studies. I would like to thank my colleagues Richard Vdovjak and Peter Barna for the numerous research discussions and papers that we wrote together. I would like also to thank my students Bas Rutten, Bert Okkerse, and Martijn Schuijers for their help in building support software tools for our research. Special thanks deserves my friend Alexandru Telea for the fruitful research that we done together by applying data visualization techniques for Semantic Web representations. A debt of gratitude is owed to my friend Cristian Pau for helping me at the beginning of my PhD. I had useful discussions and collaborations with Zoltan Fiala from the AMACONT project at Department of Computer Science of Dresden University

of Technology. I would like to thank Ad Aerts, Reinier Post, and Jan Hidders for their help and support for my research.

Finally I would like to thank my parents for their continuous encouragement during my PhD.

Flavius Frasincar

Eindhoven, June 2005

Contents

Chapter 1

Introduction

Since its birth in the early nineties the World Wide Web has grown into one of the most popular channels for reaching a very diverse audience on different platforms worldwide and 24 hours per day. Its success is overwhelming and its impact on the humankind is tremendous. Some compare its importance with Gutenberg's invention of the printing press. As a result of Web popularity, many information system have been made available through the Web, resulting in so-called *Web Information Systems* (WIS)[1] [Isakowitz et al., 1998].

The early WIS presented information in terms of carefully authored hypermedia documents. This approach fails to meet the growing demand to make available on the Web large amounts of data. The *data-intensive* aspect of WIS is present in many applications which can be found today on the Web: institutional portals, community Web sites, online shops, digital libraries, etc. Going from data to a user appealing Web presentation is a complex process that asks for a highly structured and controlled approach to WIS design. Conventional *software engineering* practices are useful for the design of the back-end of these applications. The hypermedia paradigm specific to the Web application front-end asks for a Web-specific engineering approach. A new emerging discipline called *Web engineering* [Murugesan et al., 2001] is concerned with the establishment of systematic approaches to the development of WIS.

A typical scenario in a WIS is the following: in response to a user query the system (semi-)automatically generates a hypermedia presentation. The content of the hypermedia presentation is gathered from different, possibly heterogeneous, sources that are distributed over the Web. A characteristic aspect of a WIS is the *personalization* of the generated hypermedia presentation. The one-size-fits-all approach that is so typical for traditional hypermedia is not suitable for delivering information at run-time to different users with different platforms (e.g., PC, PDA, WAP phone, WebTV) and different network connections (e.g., dial-up modem, network copper cable, network fiber optic cable). The WIS support for user interaction (e.g., by means of forms) enables the user to influence the generated hypermedia presentation based on his explicit needs.

[1] From now on we will refer by WIS to both Web Information Systems and a Web Information System, as a matter of convenience.

Several methodologies have been proposed for the design of WIS. In the plethora of proposed methodologies we distinguish the *model-driven methodologies* (e.g., WebML, OOHDM, RMM, UWE, OO-H, OOWS, OntoWebber) that use models to specify the different aspects involved in the WIS design. The advantages of such model-based approaches are countless: better understanding of the system by the different stakeholders, support for reuse of previously defined models, checking validity/consistency between different design artifacts, (semi-)automatic model-driven generation of the presentation, better maintainability, etc.

The next generation Web, the *Semantic Web* [Berners-Lee et al., 2001], is an extension of the current Web in which data will have associated semantics to it. Several Semantic Web languages (e. g. RDF(S), OWL) have been proposed to represent data semantics (as metadata) in a uniform way at different abstractions levels. WIS that use Semantic Web technologies we call Semantic Web Information Systems (SWIS). The Semantic Web will enable the interoperability between different SWIS. The benefits that the Semantic Web brings to SWIS are remarkable: a large amount of annotated data accessible by any SWIS, exchange and reuse of data models between different SWIS, flexible representation of the Web semistructured (meta-)data, etc. Model-driven SWIS design methodologies that exploit the advantages of the Semantic Web, will help the construction of successful SWIS on the future Web.

1.1 Motivation

Despite the existence of many model-driven methodologies for designing WIS there are not many model-driven methodologies targeting the design of SWIS. The existing SWIS design methodologies (e. g. XWMF, SEAL, OntoWebber, SHDM) fail to support the complete design process of a SWIS. Personalization, a critical aspect in a SWIS, is very often neglected in present SWIS design methodologies. Also, the ability to model the user interaction with the system (besides the 'link following' mechanism) is missing from these methodologies.

A methodology that comes with CASE tools will be better accepted by WIS developers. Most of the existing CASE tools (e.g., WebRatio, OOHDM-Web, RMCase) do not target the development of SWIS. Based on our knowledge the CASE tools that support the design of SWIS are now either work-in-progress (e.g., SHDM) or do say very little about how the implementation of such a system was realized (e.g., OntoWebber) in order to make these results available to the SWIS research community.

An aspect very often neglected is the ability to reuse design artifacts in building WIS. Despite the fact that there are methodologies that support the reuse of components at implementation level (e.g., AMACONT), reuse at design level is poorly sustained. The WIS design patterns in existing methodologies (e.g., WebML) usually do not consider the great potential for reuse which the Semantic Web has to offer. Also, Web services have been primarily used as a way to share data between WIS. Web services that provide a certain functionality to a WIS (e.g., data presentation or presentation adaptation) are not yet available. The existence of such services, as building blocks, would make the WIS more

robust, of a better quality, and it will shorten the development time.

RDF is the foundation language for the Semantic Web. SWIS typically use RDF (or a higher-level language like OWL, also RDF-based) to represent both models and input data. This data needs to be transformed and queried in a number of subsequent steps[2]. The available RDF query languages (e.g., SeRQL, RQL, RDQL) are very often at an early development stage in which query optimization issues are not yet (or poorly) considered. It is important to note that once these languages are used for large amounts of data (as it is the case in a SWIS) the query optimization aspects are crucial for the WIS success. Also the possibility to analyze these large amounts of data can be facilitated by appropriate visualization techniques.

Hera is a SWIS design methodology. It proposes two phases: data collection, which retrieves data from different sources, and presentation generation, which produces Web presentations for the retrieved data. Hera encapsulates the best aspects from existing methodologies: the ontology-based approach from OntoWebber, the reusable components from AMACONT, the modeling style of WebML, the simplicity of RMM, etc. In addition, it focuses on the adaptation aspects involved prior or during user browsing that are considered early in the design process. A CASE tool, the Hera Presentation Generator (HPG), was developed in order to support the presentation generation phase of Hera. Several applications have been built using the HPG: a review system for the Hera papers, a shopping site for vehicles, a portal for a virtual paintings museum (without user interaction), and a shopping site for posters depicting paintings, etc. As Hera uses RDF for its specification language, significant work was also done on query optimization and data visualization of RDF data.

For all these reasons Hera is a unique answer to the complex task of SWIS design. It is based on a blend of traditional software engineering design steps with (Semantic) Web specific design steps. Each step is supported by appropriate concepts, notations, and techniques.

1.2 Research Questions and Approaches

The contribution of this dissertation is the proposal of the presentation generation phase of the Hera methodology. For this purpose several questions have to be answered:

Question 1: How to design the presentation generation for SWIS?
First we look at existing (S)WIS design methodologies identifying their main features. Then we we can ask ourselves how can one devise a SWIS design methodology that will have all these characteristics and some additional ones that we consider useful. Because we do not want to propose yet another SWIS design methodology, we would like to build upon the strong points of existing methodologies taking in consideration the facilities offered by the Semantic Web. Also one needs to explore how easily the design specifications can

[2]Note that most of the WIS methodologies use the pipeline architecture in which the input of one step is the output of the previous step.

be translated in an automatic way to software components that produce a ready-to-use SWIS. This dissertation concentrates on the front-end of SWIS design, i.e., the presentation generation.

Question 2: How can we support adaptation during the design of the presentation generation for SWIS?
One of the most important features that a SWIS needs to have is its ability to support adaptation of the hypermedia presentation prior and during user browsing. We need to consider which are the adaptation "hot-spots" inside such a methodology and how one can make the adaptation specifications work in practice. The modeling of the user interaction with the system (by means of forms) is an important adaptation aspect as it allows for a better personalization of the system according to user needs.

Question 3: What CASE tools can support the design of the presentation generation for SWIS?
The benefits of CASE tools associated to a SWIS design methodology are tremendous. It does not only simplify the designer's tasks but it will also show how one can build a SWIS using the proposed methodology. Based on the methodology steps and their associated output specifications we need to consider which are the necessary support tools and how one can build them. Having these tools applied in realizing different SWIS (possibly from different domains) will validate our proposed methodology.

Next to these major research questions, there are two other research questions that we encountered while building SWIS.

Question 4: How can one realize query optimization inside a SWIS?
SWIS usually deal with large amounts of data. As such a query optimization mechanism that will shorten the system response time (to user actions) is very important[3]. Taking in consideration the query languages used in a SWIS one needs to investigate what are the possible query optimization techniques and how they can be made available for these query languages. As we consider the Semantic Web foundation language, RDF, we will ask this question in the context of RDF query languages.

Question 5: What are suitable visualization techniques for the data used by a SWIS?
Having to deal with large amounts of data (input data or even large specification models) it is important to support the SWIS designer with techniques to get a better insight into the data properties. Visualization techniques proved to be successful in the past in realizing different software engineering objectives (e.g., reverse engineering). A legitimate question would be how one can apply existing visualization techniques in analyzing a given set of data in a SWIS. As most of the considered data representations are RDF representations, will ask this question in the context of RDF data.

[3]The user's level of patience with SWIS is usually very limited, the user moves immediately to a different SWIS if he experiences long response times.

1.3 Outline of the Dissertation

The dissertation has seven chapters. Each chapter starts with an abstract underlying the main results that are presented. The first section of each chapter provides a motivational introduction. Chapter 2, Chapter 3, and Chapter 4 do not have a related work section because Chapter 2 describes the related work for both Chapter 3 and Chapter 4. Chapter 5 and Chapter 6 have separate sections that analyze the related work and its shortcomings. After the introduction (Chapter 3 and Chapter 4) or the related work (Chapter 5 and Chapter 6) the next sections focus on the proposed solution. The last section concludes a chapter suggesting possible future work.

Chapter 2 and Chapter 3 answer research Question 1. Chapter 2 looks at existing (S)WIS design methodologies and identifies their main characteristics. Based on these characteristics and some additional ones that we consider useful we propose the Hera SWIS design methodology. Chapter 3 describes the Hera SWIS design methodology. At the core the Hera methodology there are different models that specify, based on the separation of concerns principle, different aspects involved in the design of a SWIS. The concepts involved have graphical representations and (usually) model specifications are diagrams. Each diagram has an RDF(S) representation. The focus of this dissertation is on the design of the presentation generation phase for SWIS.

Chapter 3 answers research Question 2. One of the main features of the presentation generation phase of the Hera methodology is the adaptation support for the built hypermedia presentations. The first type of adaptation that is supported is the static adaptation of the presentation based on user preferences and device capabilities. This adaptation will be performed before the user starts browsing the generated presentation. In order to better personalize a SWIS we also support a second type of adaptation, i.e., dynamic adaptation by means of forms. In this way the user is able to change the generated hypermedia presentation during the browsing session. Most of the functionality of the dynamic adaptation is given by RDF queries. Parts of Chapter 3 have been previously published in [Frasincar et al., 2001; Frasincar and Houben, 2002; Frasincar et al., 2002b; Houben et al., 2003; Vdovjak et al., 2003; Houben et al., 2004; Fiala et al., 2004].

Chapter 2 and Chapter 4 answer research Question 3. Chapter 2 recalls the existing design tools that support (S)WIS design methodologies. Chapter 4 describes the Hera Presentation Generator (HPG), a CASE tool aiming at supporting the WIS designer that uses the presentation generation phase of the Hera methodology. The HPG also produces in an automatic way a SWIS based on the designer's specifications. The chapter concentrates on how the tool supports the design steps proposed by the presentation generation phase of the Hera methodology.

Chapter 5 answers research Question 4. In order to support RDF query optimization one can define an algebra composed of a data model and a set of operators that fulfill certain equivalence laws. An example of such an RDF algebra is RAL. This algebra was developed from a database perspective in the sense that it provides similar extraction operators (with similar equivalence laws) as the ones found in relational algebra. Differently than the relational algebra RAL provides construction operators for building new data elements.

Based on the identified algebra equivalence laws, a heuristic query optimization algorithm is proposed. Chapter 5 was previously published as [Frasincar et al., 2004b]. It is based on previous work published in [Frasincar et al., 2002c]. This chapter also appears in [Vdovjak, 2005].

Chapter 6 answers research Question 5. The input data and design specifications of a SWIS built with the Hera methodology are RDF data. As RDF data has a graph representation we were able to successfully apply a general purpose graph visualization tool to analyze large sets of RDF data inside a SWIS. Based on the proposed visualization techniques one can answer complex questions about this data and have an effective insight into its structure. Chapter 6 will to be published as a book chapter [Frasincar et al., 2005]. It is based on previous work published in [Telea et al., 2003].

The last chapter, Chapter 7, gives a summary of the main results and indicates some possible future research directions.

Chapter 2

Methodologies for Web Information Systems Design

Modern Web Information Systems (WIS) are characterized as data-intensive systems in which data integration and personalization aspects play important roles. Designing such WIS is far from trivial: the good old software engineering principles need to be adapted to the peculiarities of the Web. With respect to this researchers have proposed several WIS design methodologies among which we mention the model-driven methodologies due to the advantages they offer. Several WIS design methodologies and their accompanying tools are presented in this chapter. The emerging Semantic Web offers numerous benefits/opportunities for the WIS designers. WIS that use Semantic Web technologies are called Semantic Web Information Systems (SWIS). In this chapter, we also present some of the pioneering work in developing SWIS design methodologies. A brief comparison of the presented (S)WIS methodologies is given emphasizing for each methodology its strong and weak points.

2.1 Introduction

The Web of today has more than ten trillion pages and around one billion users. Its success lies in its very characteristics: it is unbound in space and time (it is available everyday, around the clock, and around the world), it uses the hypermedia paradigm (it provides flexible access to information according to the associative nature of the human mind [Bush, 1945]), it is distributed (it uses the popular client/server architecture with multiple clients and servers), and it is for free (there is no organization that owns the Web). If the nineteenth century was dominated by the "industrial revolution", the beginning of this century is marked by the "information revolution" having the Web as its main engine.

A Web Information System (WIS) [Isakowitz et al., 1998] is an information system that uses the Web to present data to its users. The first generation of WIS presented the data in terms of carefully authored hypermedia documents. Typically, this involved the hand-

crafting of a static collection of pages and links between these pages in order to convey information to the users.

Due to its popularity there was an increasing need to make available on the Web more data sources to present up-to-date information. As such the second generation of WIS was characterized as data-intensive applications, that usually produced on-the-fly Web presentations from data stored in databases. The databases connected to the Web form the so-called "deep Web" as opposed to the "surface Web" composed of static pages. It is the "deep Web" that is the most interesting one for WIS developers as it is 500 times larger and offers much better quality than the "surface Web".

The next generation Web, the Semantic Web, is an "extension of the current Web in which information is given well-defined meaning, better enabling computers and people to work in cooperation" [Berners-Lee et al., 2001]. One can view the Semantic Web as a large decentralized global knowledge representation system. The third generation WIS are the Semantic Web Information Systems (SWIS) , i.e., WIS that make use of the Semantic Web technologies.

Some typical examples of Web Information Systems are: commerce sites, online newspapers, educational sites, Web order tracking systems, etc. All these systems share a number of characteristics. First of all, as indicated above, they are data-intensive applications. This data can come from a single source or from different sources that need to be integrated. Based on the (integrated) data a WIS automatically generates a hypermedia presentation. The one-size-fits-all approach for traditional hypermedia is not suitable for delivering information dynamically to different users with different platforms (e.g., PC, PDA, WAP phone, WebTV) and different network connections (e.g., dial-up modem, network copper cable, network fiber optic cable). The personalization component in a WIS will take care exactly of these user/platform features so that the user has a pleasant browsing experience.

The lack of rigor in developing WIS leads to serious problems (with respect to maintenance, evolution) when the complexity of these applications grows. Web Engineering, a new discipline, is responsible for proposing a systematic approach to the successful development of Web applications [Murugesan et al., 2001]. As in software engineering, Web engineering emphasizes the need to carefully design your application before implementing it. Also the existence of reusable components simplifies a lot the development of new Web applications. Differently than the classical software engineering approach, Web engineering needs to consider the peculiarities of Web applications, e.g., the navigational aspects of these application.

The design of WIS is a highly complex task that needs to consider all WIS features (e.g., data-intensive, data integration, automatic generation of the presentation, personalization). It is the consideration of these WIS characteristics at an early stage in the WIS development life-cycle, i.e., at design time, that, in our opinion, ensures the Web application success. We also believe that a methodology that clearly identifies different steps for coping with different WIS aspects will greatly reduce the WIS development effort.

Several design methodologies have been proposed to help the designer to specify WIS. A distinguished group of methodologies are the model-driven methodologies, i.e., methodologies that use models to specify the different aspects of a WIS. A model-based approach

for WIS design has numerous benefits: better communication and understanding of the system among stakeholders, model reuse, improved system maintainability and evolution, possibility for checking validity and consistency between models, etc.

Figure 2.1 shows on a timeline some of the most popular (S)WIS design methodologies.

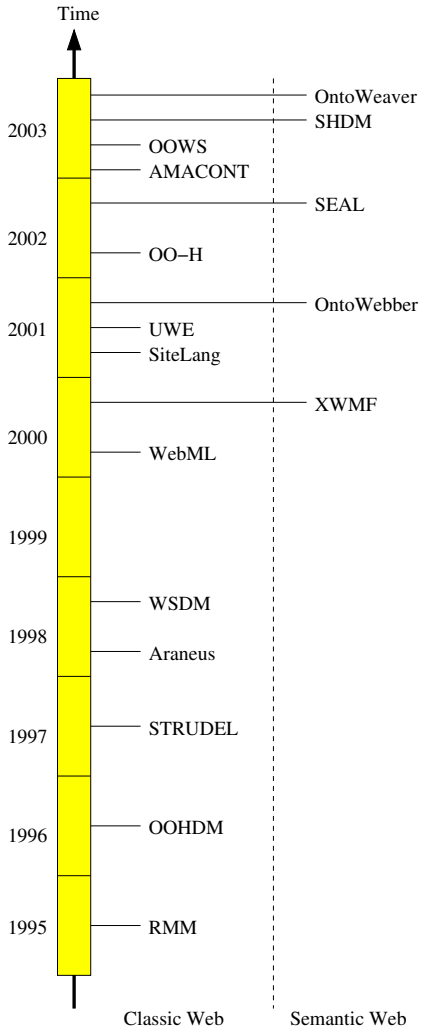


Figure 2.1: WIS methodologies.

Some of the most well-known WIS design methodologies are: RMM [Diaz et al., 1997], OOHDM [Schwabe and Rossi, 1998], STRUDEL [Fernandez et al., 2000], Araneus [Mecca et al., 1998], WSDM [De Troyer and Leune, 1998], WebML [Ceri et al., 2003], SiteLang [Thalheim and Dusterhoft, 2001], UWE [Koch et al., 2001], OO-H [Gomez and Cachero, 2003], AMACONT [Fiala et al., 2004], and OOWS [Pastor et al., 2003].

There are few design methodologies that exploit the potential of the Semantic Web. Some of the pioneering methodologies for designing SWIS are: XWMF [Klapsing and Neumann, 2000], OntoWebber [Jin et al., 2001], SEAL [Maedche et al., 2002], SHDM [Lima and Schwabe, 2003a], and OntoWeaver [Lei et al., 2003]. Common to all these systems is

the use of ontologies (as specifications of conceptualizations) [Gruber, 1993] for describing models. These ontologies are supported by inference layers that use ontology rules (axioms) to deduce new facts based on existing facts.

The main goal of using such an approach is application interoperability. By application interoperability is meant not only the WIS interoperation at data level (the output of one system is the input of another system) but also the ability to reuse existing WIS models in building new WIS. Also the model verification of such systems becomes simpler (compared with the verification of traditional WIS design models) as part of the verification is a direct application of the ontology semantics. The semi-structured aspect of Web data asks for the use of semi-structured representations (as opposed to the structured data present in classical databases) which are supported by some of the Semantic Web languages.

The rest of the chapter is structured as follows. Section 2.2 presents some of the most well-known methodologies for WIS design. Section 2.3 describes some pioneering methodologies for SWIS design. Section 2.4 gives a brief comparison of (S)WIS design methodologies emphasizing for each methodology its strong and weak points. Finally, section 2.5 concludes the chapter suggesting possible evolutions of SWIS design methodologies.

2.2 Methodologies for WIS Design

2.2.1 RMM

Methodology

The Relationship Management Methodology (RMM) [Isakowitz et al., 1995; Diaz et al., 1997] uses a "relationship management" approach for modeling WIS. By "relationship management" it is meant the management of relationships among information objects. RMM is developed from a database perspective by using the popular Entity-Relationship (E-R) diagram. At the core of the methodology there are four different activities: *E-R design*, *application design*, *user interface design*, and *construction/testing*.

The *E-R design* produces an E-R diagram in order to depict the entities and relationships relevant to a particular application domain. Each entity has attributes that describe its data characteristics. The relationships are associative relationships as they depict associations between different entities. These associative relationships have cardinality one-to-one or one-to-many. Similar to database modeling, many-to-many relationships are decomposed into two one-to-many relationships.

The *application design* produces an application model. The application model has two types of navigational elements: slices and access structures. The most significant access structures are indexes, guided tours, and links. A slice groups information into a meaningful presentation unit. The information is given by attributes of E-R entities. Slices can be aggregated in order to form higher level slices. The slices that correspond to pages are called top-level slices. The simple slices contain only one attribute. A slice can be the anchor of a link which has as destination a top-level slice. A slice is owned by an entity from the E-R model and as such can be viewed as an entity extension. Embedded slices need to

specify the E-R relationship that associates the embedded slice owner with the embedder slice owner, in case that the two owners are different. For one-to-many relationships the application model designer can choose between indexes and guided tours, as corresponding access structures.

The *user interface design* describes the presentation of each application model element. This involves the layout of slices, anchors, indexes, and guided tours. RMM suggests the use of the low-fidelity (paper and pencil) and high-fidelity prototyping methods. The paper and pencil prototyping gives a high-level overview of how one views the application. The working prototype was built in order to better understand application's behavior. In an interactive manner, based on the observations made available from the running prototype, the application model might be readjusted.

The *construction/testing* phase, as in traditional software engineering projects, builds and tests the WIS based on the previous specifications. Special care needs to be considered for the testing of all navigational paths. More on the details of this phase (the WIS construction) can be found in the next subsection.

Tools

RMCase [Diaz et al., 1995] is an environment to support the development of WIS using RMM. The environment takes in consideration cognitive issues of hypermedia software development. The three cognitive requirements used in RMCase are: feedback loops across methodological phases, manipulation of design objects, and lightweight prototyping. The tool supports: bottom-up, top-down, and middle-out software development styles. RMCase has six contexts (views): *E-R design context*, *application (navigation) context*, *node-link conversion context*, *user interface context*, *hyperbase population*, and *prototyping (simulation) context*. One can recognize four of these contexts as corresponding to the four RMM phases. The designer can easily navigate from one context to another by means of the tool navigation bar.

In the *E-R design context* the user specifies the entities, attributes, and relationships of the application domain. In the *application context* one builds an application model composed of slices, links, indexes, and guided tours. In the *node-link conversion context* one can automatically convert the application model to a node-link web. In the *user interface context* the designer builds HTML templates that are associated to the previously generated nodes. Each attribute or link anchor has an associated "slot" in the HTML template. In the *hyperbase population context* one adds instances into the application database. These instances can be "pumped" into the nodes (data hard-coded in the application) or queries (SQL) can obtain data on-demand from the underlying database. The *prototyping context* enables the application designer to test the capabilities of the future Web application by means of an automatically generated prototype.

2.2.2 OOHDM

Methodology

The Object-Oriented Hypermedia Design Methodology (OOHDM) [Schwabe et al., 1996; Schwabe and Rossi, 1998] uses an object-oriented approach for modeling WIS. The object-oriented approach chosen for OOHDM is motivated by the fact that object orientation is supported by successful standards (e.g., UML) and it offers powerful object view mechanisms. At the core of the methodology there are five different activities: *requirements specification*, *conceptual design*, *navigation design*, *abstract interface design*, and *implementation*.

The *requirements specification* identifies the users of the system and the activities the users would like to perform with the system. A user can play one role or multiple roles in the system. Scenarios, provide narrative descriptions of user activities for a certain role. Related scenarios are aggregated in a use case that informally describes the user interaction with the system without considering the internal aspects of the application. A more formal description of this interaction is provided in a user interaction diagram [Guell et al., 2000]. The user interaction diagram can be used for the derivation of the models specific to the conceptual design and the navigation design of the application.

The *conceptual design* produces a conceptual schema using a notation very similar to a UML class diagram. The conceptual schema captures the domain semantics as independently as possible from the different types of users and tasks. Conceptual classes may use different relationships like inheritance, aggregation, and association. In addition to the well-known UML constructs, OOHDM proposes attribute perspectives, i.e., different media types that can characterize a certain attribute (e.g., a building description can have a text and an image associated to it).

The *navigation design* produces a navigation class schema complemented with a context diagram. The original conceptual schema needs to be reorganized such that the application fits the user needs. Different navigation models need to be built for different applications using the same domain data. A navigational class schema has three types of navigational elements: nodes, links, and access structures. Recognizing the need to group class attributes from a presentation perspective, navigation classes (also called nodes) are defined as views on the conceptual schema using an object-oriented query language. The attributes perspectives are mapped to different navigational class attributes. Similar to the class relationships between conceptual classes, links reflect navigational relationships between navigational classes to be explored by users. For each link, some of the navigation class attributes are marked as anchors. Access structures like indexes and guided tours are other possible ways to access the navigation nodes.

In order to structure the navigation space one can define navigational contexts. A *navigational context* is a set of navigational objects. There are five types of navigational contexts: simple class derived (objects of a class that satisfy a property, e.g., buildings with address Rio de Janeiro), class derived group (a set of simple class derived contexts, where the defining property of each context is parameterized, e.g., building by location),

simple link derived (all objects related to a given object, e.g., buildings designed by Oscar Niemeyer), link derived group (a set of link derived contexts, where the the source of the link is parameterized, e.g., buildings designed by architect), and enumerated (set of elements explicitly enumerated). Associated to contexts there are access structures like indices. A navigational context contains the specifications of its elements, an entry point, and an associated access structure. If the elements of the context depend on the user browsing behavior, the context is said to be dynamic (e.g., history and shopping basket). "InContext" classes extend certain nodes with attributes when these classes are navigated in a particular context.

The *abstract interface design* produces abstract data views (ADV) and abstract data view charts (ADV-charts). ADVs are abstract in the sense that they represent the interface and the state, and not the implementation. ADVs define the interface appearance of navigational classes and access structures, and other useful interface objects (e.g., menus, buttons, etc.). ADVs capture the statical part of the interface: the perception properties and the interface's events. The dynamical part of the interface is given by ADV-charts, a derivative of the UML state charts that specify the system's reaction to external events.

Among the object-oriented design patterns used in OOHDM we distinguish the observer pattern for the navigational classes and ADVs, and the decorator pattern for the "InContext" classes. From the UML notation we recognize the class diagrams for conceptual schemas and the state diagrams for the ADV-charts. Aggregation and inheritance are used in both conceptual schemas and ADVs.

The *implementation* phase produces a WIS based on the previous OOHDM specifications. The designer has to decide on how to store the conceptual and navigational objects. Also he needs to decide on how to realize the static and dynamic aspects of the interface using HTML and some extensions. More on the details of this phase can be found in the next subsection.

Tools

OOHDM-Web [Schwabe et al., 1999] is an environment to support the development of WIS using OOHDM. It is based on the scripting language Lua [Ierusalimschy et al., 1996] and the CGILua environment [Hester et al., 1997]. Navigational classes and contexts are stored in relational databases. ADVs and ADV-charts are implemented by a combination of HTML templates (ADV structure) and OOHDM-Web library function calls (ADV behavior). OOHDM-Web has three interfaces: the *authoring environment*, the *browsing environment*, and the *maintenance environment*.

In the *authoring environment* the designer specifies the navigation schema generating database definitions. OOHDM assumes that the navigation schema is given (no need to translate it from a conceptual schema) and stores it in a relational database. Each navigational class and link are implemented as tables. For navigational classes each column stores an attribute and each arrow corresponds to an object of that class. All class tables have an attribute "key" defined. The link tables define relations between the corresponding object keys. Attribute perspectives are stored in a separate table. There is also a table

to store all context names and their types. Each context type has a corresponding table that stores all context of this type. These tables refer to HTML templates. OOHDM also provides functions to manipulate the state of different contexts.

In the *browsing environment* the designer specifies HTML templates according the corresponding ADV specifications. An HTML template is combined with OOHDM-Web function calls to return dynamically computed data. Examples of OOHDM-Web functions are "Index", "Attrib", "PrevLink", "NextLink", etc. The different kind of events can be handled by inserting scripting code in the HTML page.

In the *maintenance environment* the designer specifies interfaces to allow the insertion/change of the instance data (nodes and contexts). Also by using summary screens the environment allows one to check the design. For example one can see which are the previously stored navigational classes and contexts.

OOHDM-Web is also supported by a CASE environment [Lyardet and Rossi, 1996] that helps the designer to describe the conceptual, navigational, and interface models of its application using the OOHDM notation. Based on the chosen run-time environment the tool is able to generate appropriate implementations.

OOHDM-Java2 [Jacyntho et al., 2002] introduces a business model as a generalization of the conceptual model and the application's transactional behavior. In response to a user request the business model is possibly updated based on the application's business rules (stored in the same business model). The navigational nodes and contexts are created based on data stored in the business model. The requested page template is then populated with the data coming from navigational nodes and contexts. In this implementation OOHDM models are stored in XML and the page templates are defined in JSP. The implementation environment was Java2EE (Java 2 Enterprise Edition), a popular platform for implementing robust distributed multi-tier architectures.

2.2.3 WSDM

Methodology

The Web Site Design Method (WSDM) is an audience-driven design methodology for building WIS [De Troyer and Leune, 1998]. An audience driven philosophy takes as a starting point the explicit modeling of different target users, i.e., audiences and derives from it the system's navigation structure. WSDM consists out of five phases: *mission statement*, *audience modeling*, *conceptual design*, *implementation design*, and *implementation*. Recently, WSDM has been extended to support localization [De Troyer and Casteleyn, 2004] and adaptive behavior [Casteleyn et al., 2003, 2004].

The *mission statement* expresses the purpose, the subject, and the intended audiences for the WIS. The intention of this phase is to make clear from the very beginning what is the goal of the WIS, what is the subject that is involved in realizing the goal, and that no intended visitors are forgotten.

During *audience modeling*, the mission statement is taken as a starting point to classify the different audiences into a hierarchy, based on their information and functional require-

ments. Each group of visitors with similar functional and informational requirements form an audience class. Visitors with extra requirements are subclasses, and appear under their parent audience class in the audience class hierarchy. The top of the hierarchy is called the visitor audience class: it groups the most general requirements that all visitors to the website share. Furthermore, for each audience class, their specific characteristics, usability and navigation requirements are specified. These are later taken into account when deciding how to present information to these particular visitors.

The *conceptual design* phase consists out of two important sub-phases: *task and data modeling* and *navigation design*.

During *task and data modeling*, for each requirement, a task model is specified. This task model decomposes each high-level requirement specified during audience modeling into elementary requirements. WSDM uses CTT (Concurrent Task Trees) [Paterno, 2000] to specify the task models, which allows next to standard task decomposition, also the specification of temporal relations between the different subtasks. Then, for each elementary task derived in the task models, a tiny data model, called an object chunk, is specified. Such an object chunk formally describes the information and/or functionality needed to fulfill the elementary requirement it covers. Currently, WSDM uses ORM (Object Role Modeling) [Halpin, 1995] (with some extensions for functionality) to express object chunks, but a shift to Web Ontology Language [Bechhofer et al., 2004] is being done at the time of writing.

In the next sub-phase, the *navigation design*, the conceptual navigation structure for the website is defined. This is done by constructing a graph of nodes with links between them, and connecting the object chunks to the nodes. A node is thus a conceptual navigation entity, containing information/functionality (i.e., object chunks) that logically belongs together. The basic navigation structure is derived from the audience class hierarchy, subdividing the global site navigation space into different "sub-sites", one for each audience class. Each so-called audience track contains all but only the information/functionality required for that particular audience class. Using the task models, and more particularly the temporal relations specified between elementary tasks, each audience track is further refined.

The *implementation design* phase consists out of three sub-phases: *page design*, *presentation design*, and *data design*.

During *page design*, the designer decides which conceptual navigation nodes from the navigation design will be grouped into one page. To do so, the characteristics of the audience classes are taken into account. Possibly, different site structures can be defined, for example by targeting different browsing devices.

The *presentation design* defines the look-an-feel of the application. This is done again by taking in consideration the different requirements of the audience classes. With this respect for each page a template depicting the page layout (i.e., positioning of nodes and chunks) is defined. In addition it is specified the style (e.g., font type, size, color, etc.) for all pages.

In the *data design*, based on the conceptual schema defined by the object chunks, the schema of the underlying database is specified.

Finally, taking as an input the object chunks, the navigation design, and the implementation design, the actual *implementation* can be generated in the chosen implementation language. Both static and dynamic sites are supported. The transformation is performed fully automatically.

Tools

The Audience Modeler is a CASE tool that supports the audience modeling phase of WSDM. It allows the designer to graphically describe the different audience classes and their requirements. Furthermore, the tool also support the audience class matrix algorithm [Casteleyn and De Troyer, 2001], which automatically constructs the audience class hierarchy based on a simple series of yes/no questions. Naturally, a combined approach, generating the audience class hierarchy and afterward manually adjusting it, is also supported. This tool has been implemented using Java and Wx Windows (now renamed to wxWidgets [Anthemion Software, 2004]) for the graphical parts.

The Chunk Modeler is a CASE tool that supports the data (i.e., information and functional) modeling phase (part of the conceptual design) of WSDM. It allows the designer to graphically model object chunks using ORM. The extensions to ORM needed to model functionality are also supported. This tool has been implemented using C++ and Wx Windows (now renamed to wxWidgets) for the graphical parts.

WSDM has also a prototype of the implementation generation phase. This prototype takes as inputs (in XML format) the object chunks, the navigation design, and the implementation design of a WSDM design, and outputs an actual implementation. The implementation is done using XSLT [Kay, 2005b] transformations.

2.2.4 WebML

Methodology

WebML (Web Modeling Language) [Ceri et al., 2000, 2003] is a high-level modeling language for the specification of WIS. It uses conceptual modeling techniques for describing hypertext. WebML concepts have visual representations and are serialized in XML. The WebML design methodology comprises three main phases: *data design*, *hypertext design*, and *implementation*.

The *data design* produces the data schema composed from several sub-schemas: core, access, inter-connection, and personalization schema. In the same way as for RMM the data schema is an E-R diagram. The core sub-schema identifies the main entities in the application domain. The access sub-schema enriches the core sub-schema with access entities. In this phase one can define the categories and localizations of the previously identified entities. In the inter-connection schema the core entities can be connected using relationships. The personalization schema adds to the data schema, entities depicting the user and its preferences. In this phase, for example, the user will be associated to its preferred language.

The *hypertext design* defines the navigational structure of the application. The result of this phase is the hypertext model. The hypertext model is based on the concepts of units and links. WebML uses a hierarchical organization of data in units, pages, areas, and site views. The most primitive element is the unit, an atomic piece of information. Units are grouped into pages. A page provides information to be presented at once to the user in order to achieve a certain communication purpose. Pages that deal with a homogeneous subject are grouped in areas. At the top of the hierarchy there is the site view which can contain areas and pages. A site view defines all the information accessible in a WIS by a certain user.

There are five types of basic units: data units, multidata units, index units, scroller units, and data entry units. Data units display information about one object. Multidata and index units show information about a set of objects. Differently than the multidata unit, the index units allow the selection of one object out of a set. Scroller units support the scrolling (moving backward/forward) through a set of objects. Data entry units enable the user to insert new data into the system. The first four types of units have associated sources and selectors. Sources are used to identify the entity (type) providing the unit data. Selectors define define predicates to select the object(s) (of source type) to be presented by the unit.

Between units/pages one can define links which are directed connections. WebML distinguishes several types of links: navigational, automatic, and transport links. These links can carry information from the source to the destination. The information is stored in the link parameters. The link parameters are used in the unit selectors (selectors with link parameters are called parametric selectors). The navigational links require user intervention. Both automatic and transport links are traversed without user intervention. For automatic links once the source is presented also the associated destination is showed. The transport links do not define navigation and are solely used to transport information. Besides the link parameters, which use a point to point communication, one can define also global parameters. A global parameter is small piece of information extracted during user navigation which will be made available to all units in the system. The global parameters are accessed/modified using the get/set units.

In addition to the previous units, WebML defines operation units used to invoke different operations. These operations can be for example associated to data entry units in order to process information entered by a user. There are some predefined units to model the content management operations like create, delete, and modify units (for entities), and connect and delete units (for relationships). Also one can use operation units in order to call in a synchronous or asynchronous manner Web services. The hypertext model can be organized, if necessary, in a workflow-driven hypertext [Brambilla et al., 2002]. With this respect WebML defines a workflow data model (composed of processes, activities, etc.), workflow-specific operations (like start activity, end activity), and workflow-specific units (like switch unit).

The *implementation* comprises two phases: the data implementation and hypertext implementation. In the data implementation phase the data schema is mapped to data sources using standard database techniques. In the hypertext implementation phase WebML pages

are mapped to JSP page templates. Such a JSP page contains a query to retrieve the relevant data and a layout template used to present this data. For contextual links besides the fixed part of the URL one needs to provide also the parameters associated to the link. The operations used in the hypertext model are implemented also as JSP templates. These JSP templates will not present information but will have only a side-effect to implement a particular operation.

Tools

WebRatio [Ceri et al., 2003] is an environment to support the development of WIS using WebML. It is one of the most comprehensive CASE tools for WIS development that we have encountered so far. WebRatio has three interfaces that help to graphically define the application models: *data and hypertext design*, *data mapping*, and *presentation design*.

The *data and hypertext design interface* allows the construction of both the E-R model and the hypertext model of the application. This interface has two work areas: one for the E-R model and one for the hypertext model. The *data mapping interface* assists the designer in connecting entities and relationships to tables. The *presentation design interface* is used to define XSL stylesheets. In order to support style reuse (among units/pages) these stylesheets refer to unit placeholders instead of the actual units.

After defining all the above models one can trigger the *code generator* to produce implementations for different target platforms. In case that HTML is the language of the target platform, the output of the code generator will be a set of JSP page templates. In addition to the JSP template pages, the code generator will produce the operation actions (Java), and a set of XML descriptors. These XML descriptors are used to specify the properties of units, pages, and links. For example a unit descriptor will specify the query, input and output parameters associated with a unit.

2.2.5 SiteLang

Methodology

SiteLang (Site Development Language) [Thalheim and Dusterhoft, 2001; Schewe and Thalheim, 2004] is an abstract language with a strong theoretical foundation for the specification of WIS. The language is based on the integration of three approaches: *extended E-R (Entity-Relationship) modeling* [Thalheim, 2000], *theory of media objects* [Schewe and Thalheim, 2001], and *website storyboarding* [Thalheim and Dusterhoft, 2001]. SiteLang proposes a codesign methodology for the integrated specification of structuring, functionality, distribution, and interactivity for WIS.

The *extended E-R model*, called High-Order Entity-Relationship Model (HERM) [Thalheim, 2000] adds to the classical E-R model, behavior specification, i.e., generic operations to support the update/retrieval of data. Also the database types are extended hierarchically such that one can build higher-order types (a type of level k is a set of types of level $k-1$).

Storyboarding is the activity of defining the application story. A WIS can be described as an edge-labeled directed multi-graph in which nodes are scenes and edges are transitions between scenes. Each transition has a label which denotes a certain user action. If the label is absent the transition is based on simple link following. Actions have associated a triggering event, a precondition, and a postcondition.

A scene is a conceptual location at which dialogues (between the user and the WIS) occur. To each scene there are associated: a dialogue step expression, a media object, a set of actors involved in it, a representation, and a context. The dialogue step expression is defined by means of a dialogue step algebra, recently replaced by a story algebra. A media object is an instance of a *media type*, i.e., a view of some database and a defining query. The classical database view is extended to support the notion of a link. With this respect the query is able to create object identifiers that links can reference. Additionally a media objects has associated dynamic operations and adaptivity specifications for a controlled form of information loss. The adaptivity with respect to the device is stored in the representation associated to the media object (e.g., delete images for devices that are not capable to display images). The context specifies the access channel (e.g., high-speed, low-speed), device (e.g., PC, WAP phone), and browsing history of the user.

The storyboarding language is a story algebra [Schewe and Thalheim, 2004]. To each action is associated a scene. Based on actions and scenes one can define inductively processes which are the arguments of the story algebra. Some of the algebra operators are: sequence, skip, parallel, choice, iteration, etc. Processes can have preguards and postguards associated to them. It has been showed that the story algebra is in fact a Many-sorted Kleene Algebra with Tests (MKAT) where the sorts are the scenes (bundles of actions at a certain WIS location) and the tests are the guards associated to processes. In this way a site can be expressed as a MKAT expression. MKATs can be used to formalize the personalization and information needs of the user by means of equations. These equations will result in a simplification of the original story space.

Tools

The Storyboard Editor [Thalheim et al., 2004] is an environment to support the development of WIS using SiteLang. It has several interfaces, each covering a certain WIS design aspect as specified by the SiteLang methodology. This editor is backed by a database that stores the WIS structure and functionality. In addition the tool also supports the automatic generation of the WIS each time the content, structure, and functionality of the system are changed. The WIS specifications expressed in SiteLang are stored as XML documents. The graphical representations built using the Storyboard editor can be automatically serialized in XML.

2.3 Methodologies for SWIS Design

2.3.1 XWMF

Methodology

The eXtensible Web Modeling Framework (XWMF) [Klapsing and Neumann, 2000; Klapsing et al., 2001] is a modeling framework for designing SWIS using RDF. The core of the framework is the *Web Object Composition Model* (WOCM), a formal object based language used to define the structure and content of a Web application.

WOCM is a directed acyclic graph with complexons as nodes and simplexons as leaves. Complexons define the application's structure while simplexons define the application's content. Components are a special kind of complexons representing a physical entity (e.g., a Web page). The representation of an WOCM is done in RDF(S).

By means of RDFS inheritance mechanism one can define different views on the simplexon for different browsing devices (e.g., HTML or WML). Such an implementation (of the view) uses variables to refer to the concrete instances having the same type as the original simplexon. Using RDFS multiple resource classification mechanism, a simplexon instance can be an instance of more than one simplexon class. In this way the same object can have different implementations for different platforms. These implementations share the same object, a property which fosters object reuse. Complexons are defined for a specific view (e.g., HTML or WML) in case that the included simplexons are instances of more than one class.

The RDF extensibility feature allows the integration of different schemas in the same WOCM. For example, in a content management system the data will be annotated with the property "expires" in order to determine if a certain piece of information became obsolete.

Tools

XWMF is supported by a tool suite in order to create, process, and analyze its models. All tools are written in Extended Object Tcl (XOTcl) [Neumann and Nusser, 1993] and in Prolog. As WOCM has an RDF representation a number of tools have been developed to facilitate RDF editing and processing.

The RDF parser was implemented using TclXML parser. RDF Handle provides an interface to query RDF models. Gramtor is a graphical RDF editor able to work with both RDF/XML and RDF triple notation. The WebObjectComposer is able to store WCOMs (in RDF) as XOTcl classes and objects, and to generate a corresponding Web implementation.

The graphical user interfaces were done using the Motif version of Wafe [Neumann and Zdun, 2000], a Tcl interface for XToolkit. For exploring RDFS representations an RDFS parser on top of SWI-Prolog RDF parser [Wielemaker, 2000] was built. In addition to the RDF rule set, one can define new rules to capture the semantics of user-specific predicates.

2.3.2 OntoWebber

Methodology

OntoWebber [Jin et al., 2001] is an ontology-driven design methodology for building SWIS. Here, by ontology, is meant a set of terms (real-world or abstract objects) and their relationships (with semantic significance). The system's architecture is composed of three layers: *integration layer*, *composition layer*, and *generation layer*.

In the *integration layer*, first the syntactic differences between the different data sources are resolved. As a semi-structured data format for data representation was chosen RDF. In the second phase, the semantic differences between the different data sources are resolved. With this respect a reference ontology (domain ontology) is built for a specific domain. The articulation ontology bridges the semantical gap by mapping the concepts and relationships between data sources and the reference ontology.

The *composition layer* uses four ontologies that captures different aspects involved in designing a WIS: the *navigation ontology*, the *content ontology*, the *presentation ontology*, the *personalization ontology*, and the *maintenance ontology*. For a WIS, each ontology will be instantiated with a corresponding site model. All site models are integrated in one graph called the site-view.

The *navigation model* defines the basic elements of the site-view graph: cards, pages, and links. A card is a minimal unit of information. Pages contain one or more cards and correspond to the Web pages. Links connect cards in order to define the WIS navigational structure. Cards are classified as dynamic (depend on the source data change) or static (do not depend on source data change). There are four types of dynamic cards: fact cards (one instance), list cards (index of instances), slide cards (guided-tour of instances), and query cards (input properties).

The *content model* defines the data that will populate the navigation model. For static cards the static elements (types text, image, or anchor) are defined. For dynamic cards an entity from the domain model is specified. Also one needs to specify the entity properties that will be presented for these cards. There are two types of links: foreign (link to an external Web page) and native (link to a internal page). Native links are further classified as: static (no information flow) or dynamic (with information flow). Dynamic links have three properties associated: a query property, which produces the content of the destination card, a binding variables property, which stores values of the source entity, and an initiating property which defines the anchor in the source card. The queries are expressed in the TRIPLE [Sintek and Decker, 2002] RDF query language.

The *presentation model* specifies the look-and-feel of the application. Both cards and pages have associated style elements (font, color, etc.). Card style overrides the style of its embedder (a page). There are three type of layouts: flow layout (one row), grid layout (a table), and frame layout (composed of one static frame and one dynamic frame).

The *personalization model* supports both fine-grained (user) and coarse-grained (group) adaptation. For a specific user/group a site view and user model are defined. The user model contains the capacity property (e.g., name, age, gender, etc), interest property (the

navigation, content, and presentation models), and request property (triggers, e.g., site view update, that will be fired if some conditions are fulfilled).

The *maintenance model* focuses mainly about content maintenance (maintenance of the functionality is not considered here). It defines an administrator user and the maintenance rules (triggers) associated with him. The administrator can update the source data and the site view specifications (note that among these models is the personalization model which he can rewrite for some users).

The *generation layer* has two phases: the constraint verification phase and the site view instantiation phase. In the first phase the constraints imposed to the WIS are checked. As ontologies are defined in RDFS some of the constraints are automatically verified by directly applying the RDFS semantics. One can formulate additional constraints like structural constraints (e.g., every dynamic card has an incoming link), semantic constraints (e.g., the query card should have the same entity associated with it as the destination card), presentation constraints (e.g., suppress images for small devices). All these constraints are expressed as TRIPLE rules. Based on the data sources and site view specifications a site view instantiation is generated in the second phase of this layer.

Tools

OntoWebber is supported by an integrated development environment (IDE) in order to develop WIS [Jin et al., 2002]. The main components of the environment are: Ontology Builder, Site Builder, Site Generator, and Personalization Manager.

The Ontology Builder assists the WIS designer in developing the domain ontology. The Site Builder is used to create the site view (graph) which is exported in three models: navigation, content, and presentation model. Additionally the Site Builder is used to define rules for checking integrity constraints on the site view. The verification of these rules is done in the same tool component. The Site Generator instantiates the site view with data. The Personalization Manager defines model-rewrite rules for the site views in order to present personalized information to its users.

2.3.3 SEAL

Methodology

The Semantic PortAL (SEAL) [Maedche et al., 2002, 2003] is a domain ontology-driven design methodology for building WIS that represent Web portals. By Web portal is meant a WIS which has a large collection of information related to specific topics and often organized in a hierarchical manner. SEAL proposes a number of steps for building a Web portal: *ontology design*, *data integration*, *site design*, and *implementation*.

In the *ontology design* one creates the domain ontology in RDFS and refines it using F-Logic [Kifer et al., 1995] axioms.

The next step, the *data integration*, lifts all the data sources to a common data model, RDF. Several wrappers have been developed, e.g., for HTML, XML, and rela-

tional databases. Of course the RDF data present on the Web is ready to be used (no need of wrapping). In addition SEAL can use data coming from an Edutella Peer-2-Peer (P2P) network. Edutella provides in RDF the metadata infrastructure of P2P networks. For the integration SEAL uses the warehouse approach to combine information coming from different sources.

In the *site design* the *navigation model*, *input model*, and *personalization model* are built. The *navigation model* defines the navigational structure over the warehouse. It is generated by combined queries for schema (ontology) and content. First the users are offered a view on the ontology by using different types of hierarchies (e.g., isA, partOf). Second for each shown ontology part the corresponding content is presented. The *input model* is used for knowledge acquisition by defining forms from the ontology. These forms have associated queries that will update the warehouse with user entered data. In the *personalization model* both navigation and input model are tailored for a specific user.

The last step is the *implementation* of the WIS using the above models. For the presentation of different WIS pages specific templates are defined. More on the details of this phase can be found in the next subsection.

Tools

SEAL is supported by a number of tools included in the KArlsruhe ONtology and Semantic Web (KAON) [AIFB, University of Karlsruhe, 2004] tool suite, an ontology management infrastructure.

OntoEdit [Storey et al., 2002] is a tool used for building the domain ontology. The metadata (RDF) available on the Web can be collected by the KAON Syndicator. KAON Reverse is a visual tool to map the logical schema of relational databases to the domain ontology.

The KAON Portal Maker produces a WIS implementation based on the different SEAL models. It uses the model-view-controller design pattern. In this pattern the models are the SEAL specification models, the view is defined by the presentation template, and the default controller provides standard application logic (update data, generate links to the next objects to be presented). The default controller can be replaced with a custom-made controller.

2.3.4 SHDM

Methodology

The Semantic Hypermedia Design Method (SHDM) [Lima and Schwabe, 2003a,b] is an ontology-based SWIS design methodology. It extends the power of expression of OOHDM (see subsection 2.2.2 for a brief OOHDM description) by defining ontologies for each of the OOHDM models. These ontologies are specified in OWL [Bechhofer et al., 2004], a more expressive language than RDFS. In the same way as OOHDM, SHDM identifies four different phases: *conceptual design*, *navigation design*, *abstract interface design*, and

implementation.

The *conceptual design* builds the conceptual class schema for the application domain. This schema is described in UML extended with a few new characteristics like the ability to specialize relations. The UML diagram is mapped to an OWL model according to some heuristics rules. In addition to the previously defined OWL classes one can define new classes by using boolean expressions specifying necessary and sufficient conditions for class membership. These last type of classes are called inferred classes and are represented graphically by UML stereotypes.

The *navigation design* defines the navigational class schema and the navigational context schema. The main navigational primitives are navigational classes (nodes), navigational contexts, and access structures. In the same way as for the conceptual class schema, one can specialize navigational relations. The mappings between the conceptual schema and navigational class schema are defined using RQL [Karvounarakis et al., 2002]. The navigation context allows the description of sets of navigational objects. The new definition for concepts is more expressive than the one from OOHDM. For example one can create groups of contexts by using the subclassing mechanism. It is the user who will decide which particular specializations he wants to see. Context have associated with them access structures (e.g., indexes). SHDM introduces the powerful concept of facet (i.e., category) access structure that simplifies a lot the description of navigational context schema. This will represent any combination of the classes and their subclasses for navigation with the restriction given by explicitly specified invalid facet combinations.

The *abstract interface design* defines the abstract widget ontology and concrete widget ontology [Moura and Schwabe, 2004; Schwabe et al., 2004]. The abstract widget ontology defines the following widgets: EventActivator (reacts to external events, e.g., link or button), ElementExhibitor (presents some content type, e.g., image), VariableCapturer (receives the value of some variables, e.g., input fields), and any composition of the above. Abstract interface widgets must be mapped on concrete interface widgets in order to appear on the interface (e.g., an EventActivator can be mapped to a Link). Abstract interface widgets must also be mapped to specific navigation elements (e.g., an ElementExhibitor is associated to a certain navigational class attribute).

The *implementation* phase produces a SWIS based on the previous SHDM specifications. In this phase a converter from the UML representation to the OWL representation of the models is needed. More on the details of this phase can be found in the next subsection.

Tools

SHDM is supported by several tools in building a WIS: an SHDM2OWL mapping tool, an ontology editor, the Sesame storage, inference and query environment [Aduna, BV, 2005] and a presentation builder.

The SHDM2OWL mapping tool [Lima and Schwabe, 2003b] is used to convert the SHDM class schema, SHDM navigational class schema, and navigational context schema into OWL representations. The OWL representations are subsequently stored in a Sesame repository. Similar to OOHDM, SHDM defined templates for pages and the styles associ-

ated to pages. The OOHDM queries have been replaced by RQL queries.

The ontology editor [Lima and Schwabe, 2003b] allows the designer to directly build one of the SHDM ontologies in OWL or to visualize the ontologies produced by the SHDM2OWL mapping tool. One can also use as an ontology editor the Protege [Noy et al., 2001] environment.

The presentation builder [Schwabe et al., 2004] has an architecture composed of the following components: Request Handler, Template Engine, View Manager, Navigation Manager, Data Manager, Template Engine, and Output Postprocessor. The Request Handler gets the user request specifying the view name with possibly some navigational parameters. The Request Manager communicates with the View Manager, Navigation Manager, and Data Manager (in this order) to get to the right data associated to a user requested view. The page template associated to this view is given by the Template Engine that is responsible for producing the final page. Optionally the Template Engine can call the Output Postprocessor to convert the produced page (e.g., in XML) to a Web browsable format (e.g., HTML).

2.4 Discussion

Model-driven methodologies have proven to fulfill the practical needs that the WIS designer experiences. For example WebML was successfully used for projects inside Microsoft, Cisco Systems, TXT e-solutions, and Acer Europe, and SiteLang was used for several city information, learning, e-government, and community sites in Germany. Being at their infancy SWIS design methodologies were merely used in research. For example SEAL was used to develop the institute portal of AIFB, University of Karlsruhe and OntoWebber was exploited for realizing the Semantic Web community portal.

In the rest of this section we use several comparison criteria in order to stress the strong and weak points of some of the most representatives WIS and SWIS design methodologies. These comparison criteria are:

- *methodology*: does the methodology provide design steps and guidelines for each step in order to produce models?
- *tools*: is the methodology supported by CASE tools?
- *automation*: is there support to automatically build Web presentations based on previously specified models?
- *data integration*: does the methodology consider the integration of data coming from different heterogeneous sources?
- *personalization*: does the methodology support adaptation mechanisms in order to realize the application personalization?

- *user interaction*: does the methodology support complex forms of user interaction (e.g., by means of forms) with the system?
- *task model*: does the methodology explicitly model the tasks of the user in a separate task model?
- *presentation model*: does the methodology explicitly model the presentation aspects (the look-and-feel aspects, separate from navigation) of the application?
- *verification*: can the methodology models be easily verified for their validity and consistency among each other?
- *reuse*: does the methodology support the design of reusable components?

Most of the (S)WIS design methodologies identify two models: the domain model, which describes the application domain, and the navigation model, which depicts the navigation (linking) aspects through the data.

Table 2.1 shows an overview of the characteristics of some WIS design methodologies with respect to the previously selected criteria.

	RMM	OOHDM	WSDM	WebML	SiteLang
Methodology	Yes	Yes	Yes	Yes	Yes
Tools	Yes	Yes	Partial	Yes	Yes
Automation	Partial	Yes	Partial	Yes	Yes
Data integration	No	No	No	No	No
Task model	No	Partial	Yes	Partial	Yes
Personalization	Partial	Yes	Yes	Yes	Yes
User interaction	No	Partial	No	Yes	Yes
Presentation model	No	Yes	No	No	No
Verification	No	No	No	No	Yes
Reuse	Yes	Yes	Yes	Yes	Partial

Table 2.1: WIS methodologies comparison.

The analyzed WIS design methodologies have a well-structured methodology and good tool support. One of the tools is the code generator that builds in an automatic way a Web presentation based on previously specified models. RMM and WSDM are examples of methodologies that have tools that only partially support this automatic process.

The WIS requirements describing what the user actually wants to do with such a WIS are very often neglected by WIS design methodologies. A notable exception is WSDM that models the audience that the WIS targets. Its characteristic feature is that it proposes a task model associated with a certain audience. The task model is subsequently used to derive the navigation model in which a navigation track corresponds to a certain audience. Similar to the task models are the storyboards used in SiteLang. An alternative approach

is proposed by OOHDM which models the interaction between the user and the system in a user interaction diagram. The WSDM task model and the SiteLang storyboard are more expressive than the user interaction diagram as they have complex task operators (like concurrency, choice, iteration, etc.) which are not available in user interaction diagrams.

For all the examined WIS design methodologies the data integration issue was ignored. This is mainly due to the lack of representation languages on the classic Web to express data semantics. It is the Semantic Web with its support for expressing data semantics that fosters application interoperability. Knowing the data semantics one can easily integrate data coming from different sources.

In order to personalize a WIS, the designer identifies user profiles, which stores characteristics of the user and his browsing platform. These user profiles can be aggregated in group profiles which cluster the users with the same requirements. To a user profile or group profile one can attach specific views. The conceptual model is augmented with user-specific entities and relationships that can refer back to the application domain model. In WebML these extensions form the so-called personalization sub-schema. In addition to the above personalization techniques, OOHDM proposes the personalization of the entities (attribute content of an entity) in the conceptual model and of the layouts in the interface model (based on user preferences or selected devices).

Another feature neglected by the examined methodologies is the design support offered for the presentation aspects (the look-and-feel aspects) of WIS. Most of the methodologies refer to templates (for example XSL templates) that describe the styling information of the systems. An exception is OOHDM which has an explicit presentation model. This model does not only depict the static presentation characteristics of the system (e.g., layout) but also the dynamic aspects of the system, i.e., what will be the system reaction to external events.

Modern WIS allow the user to interact with the system in order to let him influence the next page to be generated in the hyperspace. These systems need to make available complex user interaction (e.g., by means of forms). WebML supports the modeling of user input and its processing using data entry units and processing units, respectively. SiteLang defines activities for gathering user input and variables to store the data input by the system. These variables can be used by processing activities that will perform computations based on user input. The results can be made available for display in the scenes associated to the processing activities.

Verification is yet another aspect that was ignored by most of the WIS design methodologies that we analyzed. An exception is SiteLang, a WIS design methodology with strong theoretical foundations. Having strong theoretical grounds SiteLang produces very concise WIS representations by means of formulas. This formulas can be easily verified for well-formedness. Moreover a SiteLang formula can be minimized (optimized) by considering the equations that model certain user characteristics (like preferences or information needs). In this way one can considerably reduce the complexity of a WIS specification.

All examined methodologies provide primitives that can be reused in the model specification. Models can be refined by means of inheritance, enabling thus the reuse of existing specifications. An object-oriented approach like the one used in OOHDM or the extended

E-R Modeling from SiteLang fosters also the reuse of the behavior specifications. In addition, object-oriented approaches benefit from the reuse of design patterns (e.g., observer pattern, decorator pattern). The OOHDM team is also actively involved in defining navigational patterns to support coarse-grained reuse in navigational models. With respect to expressivity and fine-grained reuse in navigation specifications we found that WebML has one of the most extended set of navigational primitives that satisfy most of the WIS designer needs.

Table 2.4 shows an overview of the characteristics of some SWIS design methodologies with respect to the previously selected criteria.

	XWMF	OntoWebber	SEAL	SHDM
Methodology	Partial	Yes	Partial	Yes
Tools	Yes	Yes	Yes	Yes
Automation	Yes	Yes	Yes	Yes
Task model	No	No	No	Partial
Data integration	No	Yes	Yes	No
Personalization	No	Partial	No	Yes
User interaction	No	No	Partial	Partial
Presentation model	No	Partial	No	Yes
Verification	Partial	Yes	Partial	Partial
Reuse	Yes	Yes	Yes	Yes

Table 2.2: SWIS methodologies comparison.

Less mature than the WIS design methodologies, many SWIS design methodologies focus less on the steps needed to build SWIS. These methodologies emphasize how ontologies can be used for their model representations and their support tools. The representation language ranges from RDF (in XWMF) to OWL (in SHDM). Having such a standardized mean to express application semantics greatly improves system's interoperability. SWIS design methodologies (e.g., SHDM) that extend an existing WIS methodology (e.g., OOHDM) benefit from the reuse of the methodological steps and are better structured than the rest of the examined SWIS design methodologies.

Most of the examined SWIS design methodologies have good tool support. Tools for automatic generation of WIS based on previously defined models do also exist. Being at an early development stage most of these methodologies do not provide an integrated development environment like the ones we found for WIS design methodologies (e.g., RMCase, OOHDM-Web, WebRatio).

The SWIS specifications describing what the user tasks in a SWIS are neglected by most of the examined SWIS design methodologies. SHDM is the only SWIS design methodology that addresses this issue by means of the user interaction diagram that it inherits from OOHDM. As SWIS design methodologies will become more mature and their applicability will go beyond research laboratories we hope that more attention will be given to the specification of user tasks.

Data integration is a topic of special interest for SWIS design methodologies. Having the necessary technologies to describe the data semantics facilitates the integration of data coming from different sources. Common to all these methodologies is the wrapping of the data sources in a semantic representation. These semantic representations are mapped to a common (application) data representation. The only methodology that doesn't consider data integration is XWMF, as the authors focus on the presentation aspects of a SWIS.

Most of SWIS design methodologies have defined models and ontologies to describe the model semantics. Seen as an advanced feature few of these methodologies provide model "hot-spots" to support system's personalization. A notable exception is SHDM which reuses the adaptation mechanisms from OOHDM. An interesting approach is provided by OntoWebber which briefly sketches personalization by means of model re-write rules.

There is very little support in the analyzed SWIS design methodologies to model more advanced forms of user interaction with the system than simple link following. SEAL offers a limited form of user interaction by defining forms only for changing the system input data not affecting thus the application's hyperspace. The only form of user interaction that OOHDM supports is the user selection of items from existing data.

As for WIS design methodologies, a feature also neglected in the examined SWIS design methodologies is the design support offered for the presentation aspects (the look-and-feel aspects) of SWIS. OntoWebber briefly sketches similar presentation specification mechanisms.

Differently than WIS design methodologies, SWIS design methodologies support model verification. This is largely due to the direct application of the model semantics as specified in the associated ontologies. Besides the validation feature offered by the use of ontologies, some methodologies (e.g., OntoWebber) offer the possibility to express structural, semantical, and presentational constraint verification.

SWIS design methodologies support reuse by inheritance mechanism not only at concept level but also at property level in their models. This is due to the property-centric view of the Semantic Web languages (e.g., RDF, OWL) that support property specialization. XWMF also shows how the multiple instantiation mechanism can enable the reuse of the same data object for different implementations. SHDM introduces the faceted navigation structure that has a concise representation which reduces the effort of specifying navigation models.

2.5 Conclusions

In this chapter we presented the current situation with respect to (S)WIS design methodologies. While WIS design methodologies did reach maturity, more research has to be done for WIS methodologies that make use of Semantic Web technologies (the so-called SWIS design methodologies). Realizing the benefits of the Semantic Web platform (e.g., interoperability, inference capabilities, increased reuse of the design artifacts, etc.) traditional WIS design methodologies like OOHDM or WSDM are now focusing on designing SWIS. New methodologies like OntoWebber were specifically designed by considering the

Semantic Web peculiarities.

Realizing the importance of a personalized (S)WIS, during the last years a lot of attention was given to the design of the (S)WIS adaptation aspects. Good results were obtained for the static system adaptation (e.g., OOHDM, WebML), i.e., adaptation performed before the user starts browsing the (S)WIS. More work has to be done for the dynamic adaptation of these systems, i.e., adaptation performed during user browsing of the (S)WIS. Research done in the adaptive hypermedia [De Bra et al., 1999] field can prove to be useful for designing adaptive (S)WIS.

As the Semantic Web matures, we hope that the same will happen with SWIS design methodologies. One of the main obstacles in building SWIS is also the absence of standardized query languages and the lack of data transformation languages for the Semantic Web. By defining standards for integration, conceptual, navigational, presentation, and personalization modeling one will greatly contribute for SWIS application interoperability. Also by having the user profile defined in a standard way will enable the reuse of user profiles among SWIS.

An important factor to assure the success of a WIS design methodology is the existence of tool support. A powerful methodology that is not accompanied by adequate tools will make the designer's tasks very difficult to fulfill. Most of the WIS design methodologies have powerful CASE tools. There are very few SWIS design methodologies with a good tool support.

Chapter 3

The Presentation Generation Phase of Hera

Hera is a model-driven methodology for designing Semantic Web Information Systems (SWIS). The presentation generation phase of the Hera methodology builds a Web presentation for some given input data. Based on the principle of separation of concerns, Hera defines models to describe the different aspects of a SWIS. These models drive the specification of the data transformations used in the implementation of the Hera presentation generation phase. The Hera presentation generation phase has two variants: a static one that computes at once a full Web presentation, and a dynamic one that computes one-page-at-a-time by letting the user influence the next Web page to be presented. The dynamic variant proposes, in addition to the models from the static variant, new models to capture the data resulted from the user's interaction with the system. The implementation of the static variant is based on XSLT data transformations and the implementation of the dynamic variant is based on Java data transformations.

3.1 Introduction

Hera is a SWIS design methodology. It proposes design steps that, based on the separation of concern principle, specify different aspects of a SWIS. These specification aspects are given by models that have graphical representations. The implementation of a SWIS using the Hera methodology is based on data transformations driven by Hera models. Hera has its origins in the RMM design methodology [Diaz et al., 1997]. Differently than RMM, Hera specifies also other features of a SWIS like the look-and-feel aspects, the user interaction with the system, or the presentation adaptation.

Figure 3.1 shows the main phases in Hera: *data collection* and *presentation generation*. The Hera methodology comes also with a straightforward implementation in which the Hera's main phases and the design steps corresponding to these phases are naturally

mapped to components in a pipeline software architecture. We point out that the software based on this architecture is just one of the possible implementations of SWIS given the specifications required by the Hera methodology.

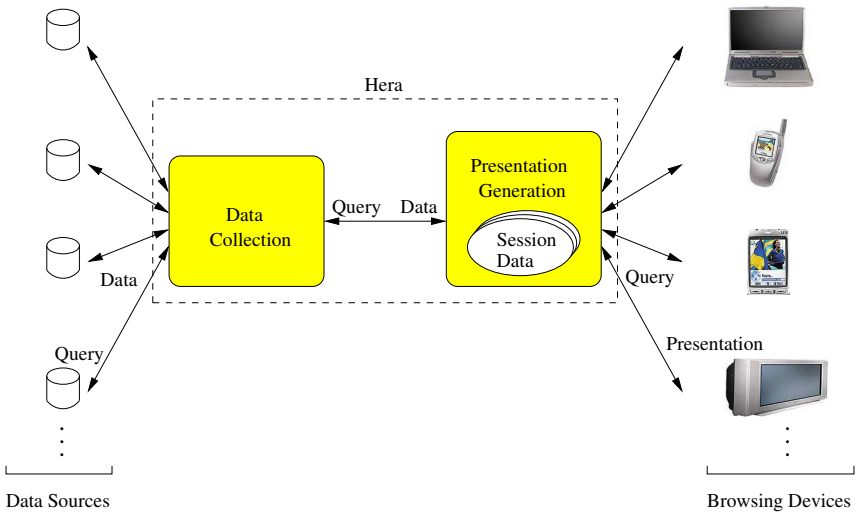


Figure 3.1: Hera's main phases.

The data collection phase helps to make the data available from different sources, such that in response to a user query a data result set is obtained. In this phase of the process the integration model is defined that maps data from the different sources to a common data representation. This mapping is needed whenever for a given query the instances that compose the query result need to be retrieved. The data collection phase is outside the scope of this thesis. More information on this phase can be found in [Vdovjak et al., 2003].

The presentation generation phase builds a hypermedia presentation for the retrieved data. It is based on a sequence of data transformations driven by several models. These models depict different application aspects that are relevant in this process: what is the domain of the application, what is the navigation structure for data from this domain, how to arrange and style the data on the user's display, and how can we tailor the generated presentation based on user preferences and user browsing platform. As can be seen from Figure 3.1 the generated hypermedia presentations can target different platforms like PC, WAP phone, PDA, etc.

The presentation generation phase has two variants: a static one in which the user is unable to change the content of the generated hypermedia presentation and a dynamic one which considers the user interaction with the system in the process of building the next hypermedia page. In the static variant all pages are generated before the user browses the presentation and in the dynamic variant one page is generated-at-a-time during the browsing.

In order to better support the description of Hera's presentation generation phase we use a running example based on real data coming from the painting collection in a museum, the Rijksmuseum in Amsterdam.

The remainder of this chapter is structured as follows. Section 3.2 explains why we chose RDF as a model representation language. Section 3.3 presents the static presentation generation phase of Hera. Section 3.4 presents the dynamic presentation generation phase of Hera. Section 3.5 concludes the chapter and presents future work.

3.2 RDF(S)

For the Hera specifications RDF(S) [Lassila and Swick, 1999; Brickley and Guha, 2004] is used. RDF(S) is the foundation language of the Semantic Web. There are several reasons for choosing RDF(S): it is flexible (it supports schema refinement and description enrichment), it is extensible (it allows the definition of new resources/properties), and it fosters Web application interoperability (it provides a framework to describe in a uniform way the data semantics). As RDF(S) doesn't impose a strict data typing mechanism it proved to be very useful in dealing with semi-structured (Web) data. On top of RDF(S) high-level ontology languages (e.g., DAML+OIL [Connolly et al., 2001], OWL [Bechhofer et al., 2004]) are defined, which allows for expressing axioms and rules about the described classes giving the designer a tool with larger expressive power. We believe that choosing RDF(S) as the foundation for describing models enables a smooth transition in this direction.

Hera models are described in RDFS. An RDFS vocabulary is developed for each model in order to define the model's concepts (which are the classes and properties to be used in a model). Model instances have an RDF representation which are validated against their corresponding schema (model). Having such standards to express models enables the model reuse between different applications. The use of RDFS allows us also to reuse existing RDFS vocabularies for expressing for example domain models or user profiles.

In some applications built with Hera we successfully reused the domain model developed for museum descriptions in the TOPIA (Topic-based Interaction with Archives) project [Rutledge et al., 2003] and the User Agent Profile (UAProf) [Wireless Application Protocol Forum, Ltd., 2001], a Composite Capability/Preference Profiles (CC/PP) [Klyne et al., 2004] vocabulary for modeling device capabilities and user preferences.

3.3 Presentation Generation (Static)

The typical structure of the static variant of the presentation generation phase is given in Figure 3.2 in terms of three layers: the conceptual layer defines the content that is managed in the SWIS, the application layer provides the navigation structure on the data, and the presentation layer gives the presentation details that are needed for the generation of the hypermedia presentations on a concrete platform. As can be noted from Figure 3.2, in the static variant for the presentation generation phase the whole Web presentation is produced at once in response to a user query.

The presentation generation phase distinguishes the following steps: the conceptual design, the application design, the presentation design, and the implementation. Each

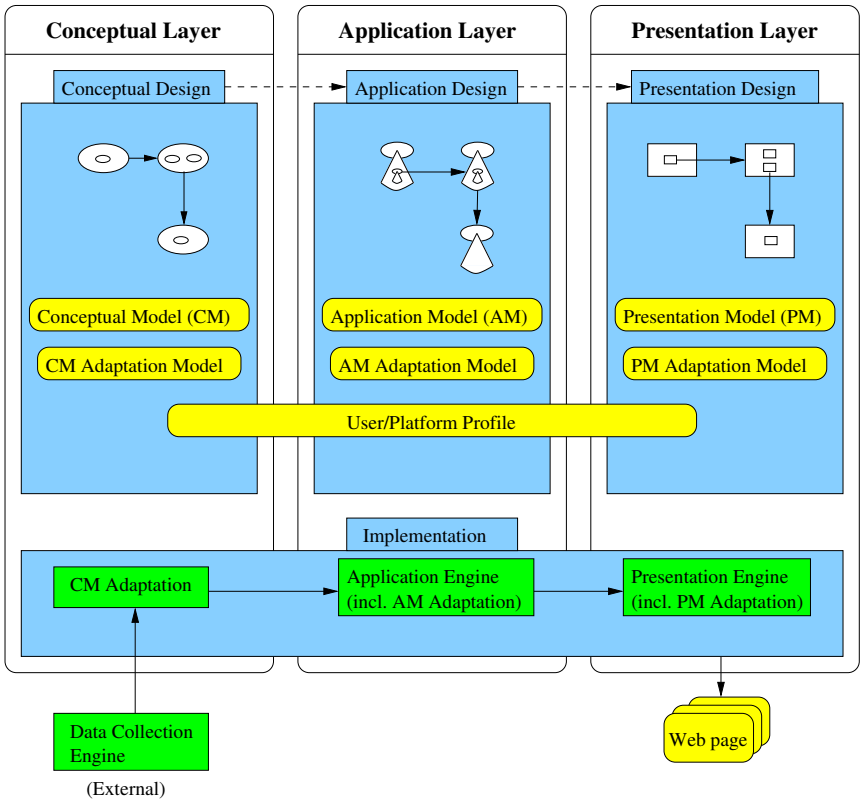


Figure 3.2: Presentation generation phase (static).

design step produces appropriate models that capture the design aspects specific to this step. A model uses concepts from a model-specific vocabulary. In order to ease the specification of the models the model concepts have associated graphical representations. In this way a model can be showed as a diagram to facilitate the designer development and understanding of a certain model.

Adaptation [Frasincar et al., 2004a] is not seen as a separate design phase because this process is distributed through all the previously identified design steps. In the adaptation design the user/platform profile (UP) is defined, i.e., it is determined which are the user preferences and platform characteristics that can influence the Web presentation before the user starts the browsing session. The adaptation model specifies adaptation conditions (Boolean expressions) used to tailor the Hera models based on the UP attributes. An excerpt of the UP vocabulary is given in Figure 3.3.

We present the adaptation model when we show the different design steps. If the designer is not interested in adapting the system he can ignore the adaptation aspects in the proposed methodological steps. The adaptation presented here is a fine-grained adaptation. A coarse-grained adaptation is achieved by using group profiles, instead of UPs. In this approach users with similar characteristics are assigned a group profile. One of the advantages of coarse-level adaptation is the decrease in the system's workload, as the performed adaptation is relevant for several users.

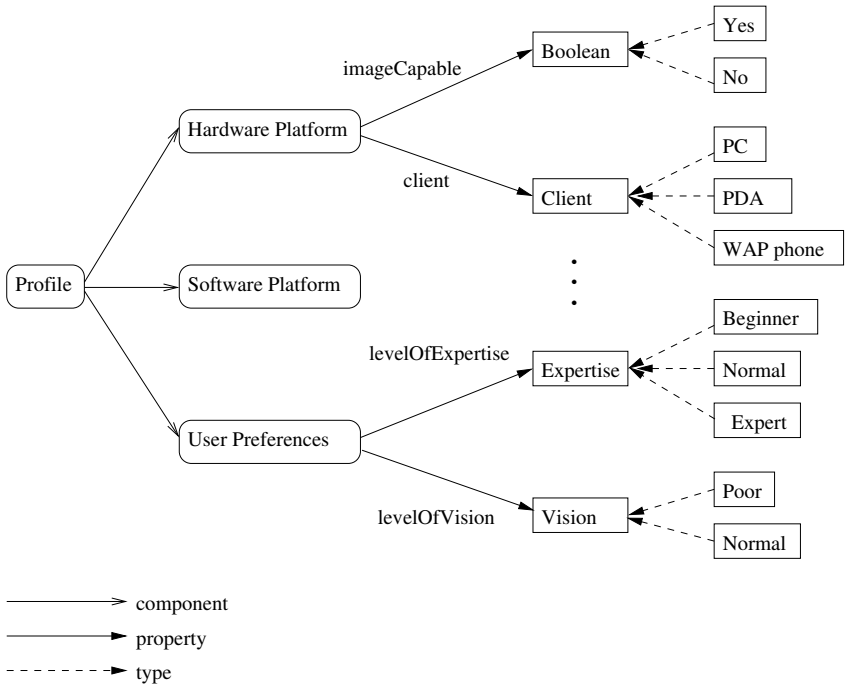


Figure 3.3: User/platform vocabulary.

3.3.1 Conceptual Design

The conceptual design specifies the input data in a uniform manner, independent from the input sources. The result of this activity is the *conceptual model* (CM). From a database point of view, the CM defines the schema for the data that needs to be presented. The CM serves also as the interface between the data collection phase and the presentation generation phase of the Hera methodology.

Figure 3.4 shows the CM vocabulary. It defines the following notions: *concept*, *concept attribute*, and *concept relationship*. A concept represents a certain entity in a particular application domain. Concept attributes and concept relationships refer to media types and other concepts, respectively, in order to describe the properties that characterize a concept. As CM vocabulary we did use the standard RDFS concepts with three extensions: one for modeling the *cardinality* of the concept relationships, one for representing the *inverse* of the concept relationships, and one for depicting the media types, the so-called *media vocabulary*. Similar to database modeling, many-to-many concept relationships are decomposed into two one-to-many concept relationships. In this way we have only two types of cardinalities: one-to-one and one-to-many.

Figure 3.5 shows the type hierarchy in the media vocabulary. In the same way as AMACONT [Fiala et al., 2003], we base our media vocabulary on a subset of the MPEG-7 standard [Martinez, 2003]. The basic media types are: *Text*, *Image*, *Audio*, and *Video*. The figure also shows the attributes of the media types, for example the *nrChars* of a text or the *width* and *height* of an image. For the refinement of the *Text* media types the XML

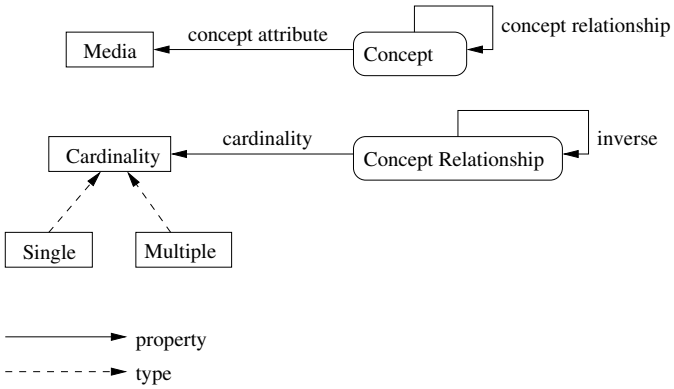


Figure 3.4: Conceptual model vocabulary.

Schema Datatypes (e.g., *String* or *Integer*) are used (not shown in the figure). One of the focus points of the Hera methodology is to reuse as much as possible the existing Web standards providing thus a higher degree of application interoperability.

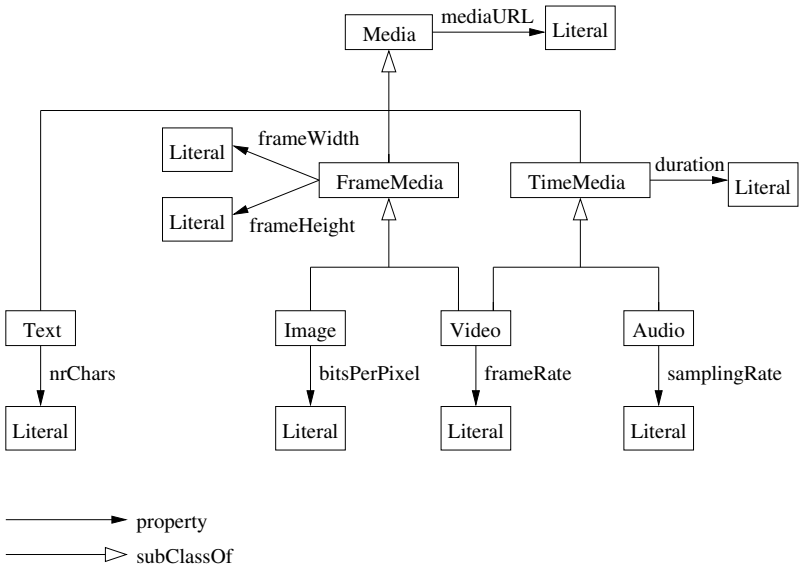


Figure 3.5: Media vocabulary.

Media adaptation selects the most appropriate media items for the technical system parameters provided by different network environments and client devices. Figure 3.6 shows a few media adaptation examples. For devices that are not able to display images (like certain WAP phones), the images are removed from the presentation. Based on display size, large strings and images are selected for PC, and small versions of the same strings and images are selected for PDA.

Figure 3.7 shows am excerpt of the CM for the running example. Concepts are represented as ovals and media types as rectangles. There are three concepts: *Technique*,

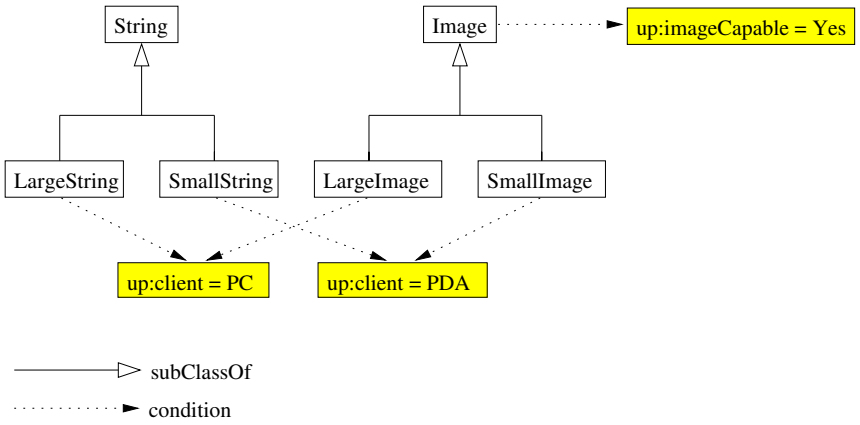


Figure 3.6: Media adaptation.

Artifact, and *Creator*. A *Creator* has two concept attributes attached to it, *cname*, for the creator's name, and *biography* for the creator's biography, both depicted by *String* items. A *Creator* is associated using the concept relationship *creates* to an *Artifact*. The cardinality of this concept relationship is one-to-many, i.e., one creator creates many artifacts. The inverse of the *creates* concept relationship is the *created_by* concept relationship. Note that both concept relationships and concept attributes are denoted as concept properties.

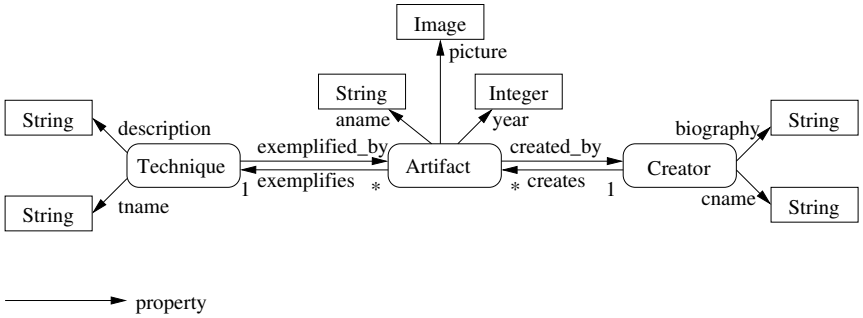


Figure 3.7: Conceptual model.

The conceptual model presented in Figure 3.7 depicting any creator, artifact, or technique can be refined to a specific artistic domain. Figure 3.8 shows the specialization (in a type hierarchy) of the previous conceptual model to the painting domain. Concepts are specialized by the *subClassOf* property and concept relationships are specialized by the *subPropertyOf* property. For example, the *Creator* is specialized as a *Painter* and the *creates* relationship is specialized as *paints*.

CM adaptation selects concepts or concepts attributes from the CM to be used in the presentation. Figure 3.9 shows an adaptation example in the conceptual model. In this example the *description* of the painting technique is removed from the whole presentation if the user is not an *Expert*. This is the so-called *context-independent adaptation*, i.e., adaptation that affects the entire presentation. An example of *context-dependent adaptation*,

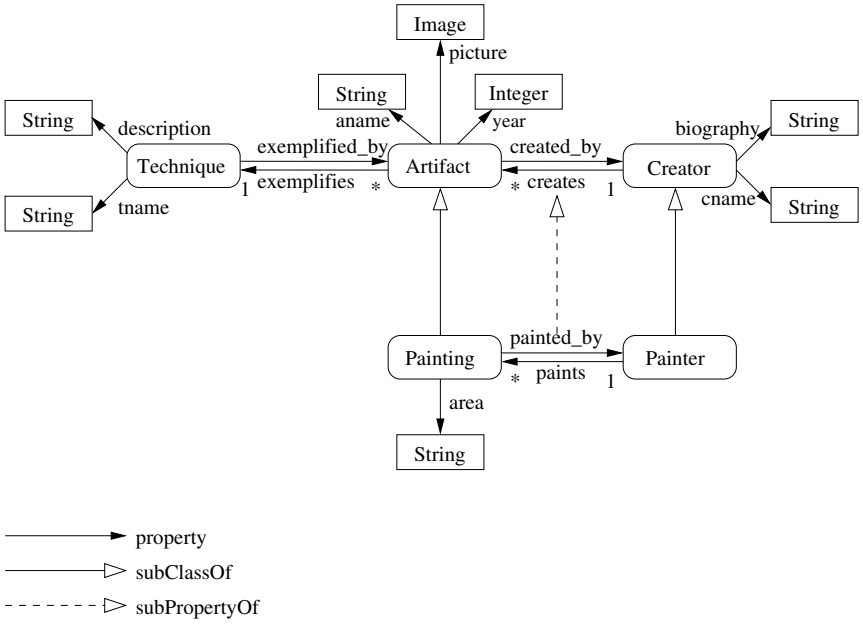


Figure 3.8: Specialization in the conceptual model.

i.e., adaptation that affects only a certain situation in a presentation, is provided in the next section.

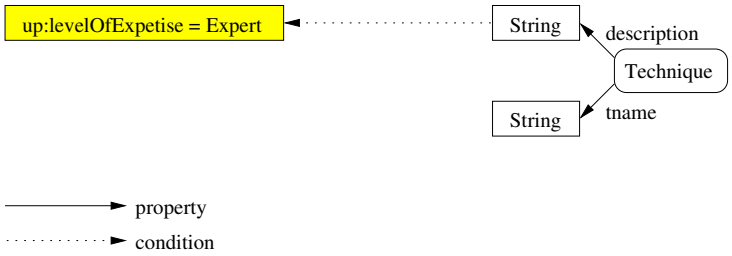


Figure 3.9: Adaptation in the conceptual model.

3.3.2 Application Design

The application design defines the navigational aspects of the presentation that is generated. A CM does not suffice to model a Web application [Rossi et al., 1999]: one needs to define the navigational view over the CM. The result of this activity is the *application model* (AM). From a database point of view, the AM is a view over the CM extended with navigation primitives.

Figure 3.10 shows the AM vocabulary. It defines the following notions: *slice*, *slice attribute*, and *slice relationship*. A slice [Isakowitz et al., 1998] is a meaningful presentation unit that fulfills a certain communication purpose. Slice attributes are used to refer to

media types. There are two types of slice relationships, *slice aggregation* and *slice navigation*. The first type of slice relationship facilitates the inclusion of a slice into another slice and the second type of slice relationship is used to define navigation between slices. An *empty slice*[1] is a slice that has its content defined at design-time. Such a slice has only one attribute that refers to a media type added at design-time. A *non-empty slice* has its content defined at run-time. In order to know from where the content is to be extracted at run-time slices have associated to them an *owner* concept from CM. The *owner* attribute for an empty slice can be any concept, as the slice content is defined at design-time.

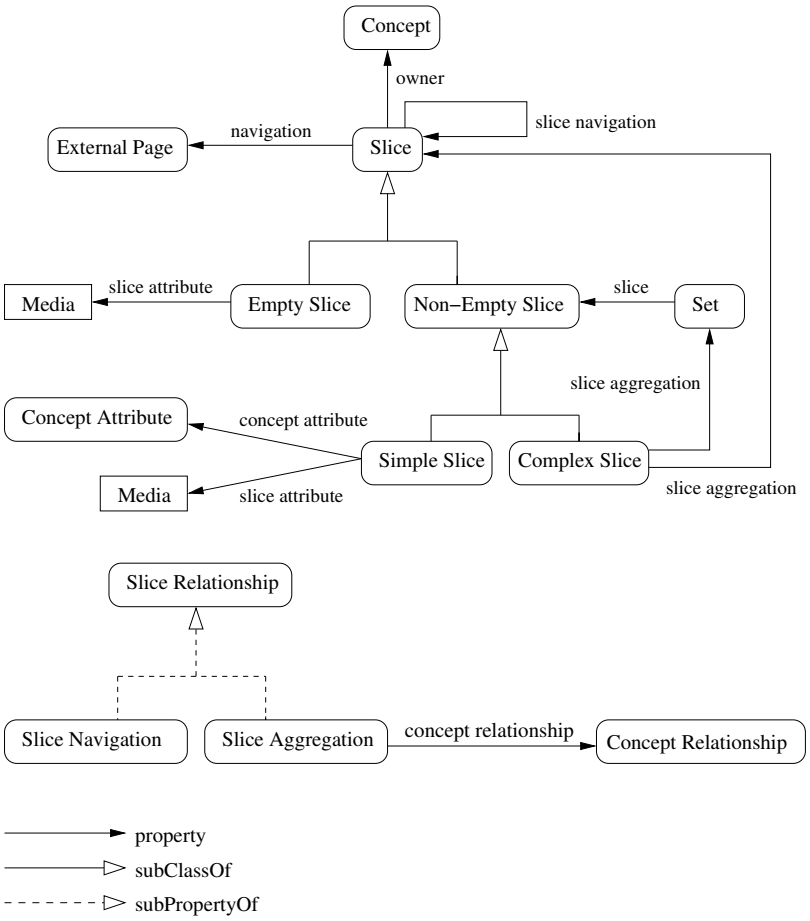


Figure 3.10: Application model vocabulary.

The definition of a non-empty slice is recursive: a slice can be a *simple slice* or can contain other slices[2]. A simple slice has only one slice attribute that refers to the same media

[1]Dealing with data-intensive applications, by 'empty' is meant that there is no content that will populate this type of slice at run-time.

[2]Due to their nested nature, slices are also called M-slices where 'M' stands for Matryoshka, the Russian doll [Diaz et al., 1997].

as the *concept attribute* of the owner concept from CM. A slice that aggregates other slices is called a *complex slice*. The recursion is defined by utilizing the *slice aggregation* relationship. The aggregation relationship between two slices that have two different owners needs to specify the *concept relationship* (or a relationship derived from the CM by relationship chaining) between the two owner concepts from the CM that made such an embedding possible. In case that the cardinality of this concept relationship is one-to-many the *Set* construct needs to be used. A *top-level slice* corresponds to a Web page. Using a slice navigation relationship, a slice (the anchor) can be linked to a top-level slice. Additionally a slice can be linked to an external Web page.

Figure 3.11 shows an excerpt of the AM for the running example. Slices are depicted (as their name suggests) by pizza-slice shapes. There are two slices, the main slice owned by *Technique* and the main slice owned by *Artifact*. We use the convention to denote the slice (long) name by *Slice.<concept name>.<slice short name>*, in order to distinguish them from concept names or slices with the same short name but owned by different concepts. The name of the slice owned by *Technique* is thus *Slice.Technique.main*. The slice *Slice.Technique.main* aggregates (by means of slice aggregation relationships) two simple slices and one complex slice. The simple slices *Slice.Technique.tname* and *Slice.Technique.description* are owned by *Technique*. The complex slice that aggregates *Slice.Artifact.picture* is owned by a different concept, i.e., *Artifact*. The aggregation relationship used for this embedding refers to the *exemplified_by* concept relationship between *Technique* and *Artifact*. As the cardinality of *exemplified_by* is one-to-many the *Set* construct is also inserted. In a similar manner the slice *Slice.Artifact.main* is defined. As *created_by* has cardinality many-to-one (inverse of *creates*), the *Set* construct is not used in this case. The slice navigation relationship connects the picture of an artifact *Slice.Artifact.picture* with the slice giving detailed information about that artifact *Slice.Artifact.main*.

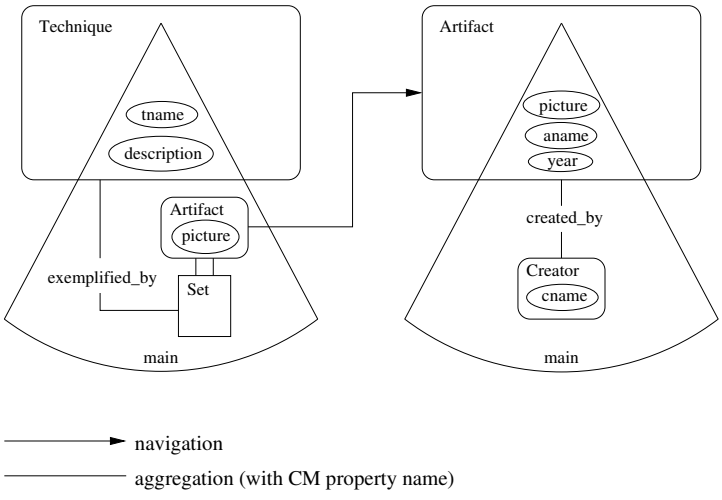


Figure 3.11: Application model.

The AM presented in Figure 3.11 depicting the main slices for techniques and ar-

tifacts can be refined to a specific artistic domain. Figure 3.12 shows the specialization (in a type hierarchy) of the previous AM to the painting domain. Slices are specialized by the *subClassOf* property. For example, the slice *Slice.Creator.main* is specialized by the slice *Slice.Painting.main*. *Slice.Painting.main* inherits all the slice relationships of *Slice.Technique.main* and adds three new slice relationships to it: two slice aggregations and one slice navigation. The aggregation relationships refer to the slice *Slice.Technique.area* and *Slice.Technique.tname*. The navigation relationship links backwards the *Slice.Technique.tname* with the *Slice.Technique.main*.

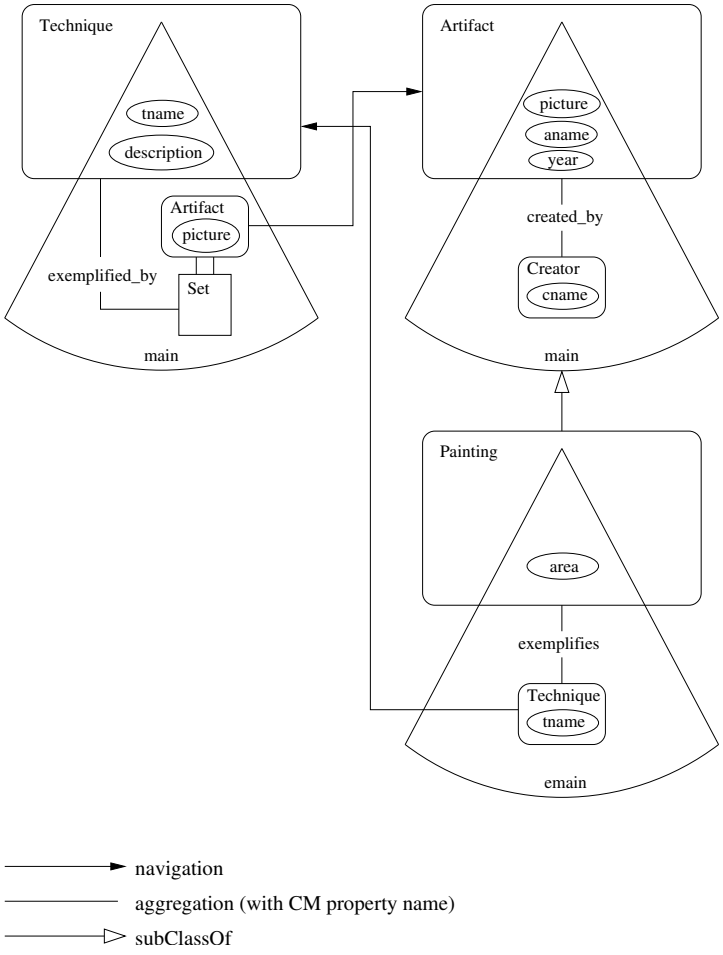


Figure 3.12: Specialization in the application model.

The AM adaptation [Frasincar and Houben, 2002] is based on two typical adaptation mechanisms: *conditional inclusion of fragments* (fragments are slices in our context) and *link hiding* [Brusilovsky, 2001] (links are slice navigation relationships in our context). A link is hidden when its destination slice has an invalid condition.

Figure 3.13 shows an adaptation example in the AM. In this example the *description* of the painting technique is removed from the main slice of this technique if the user is not an *Expert*. Later on in the presentation, the description of the painting technique can

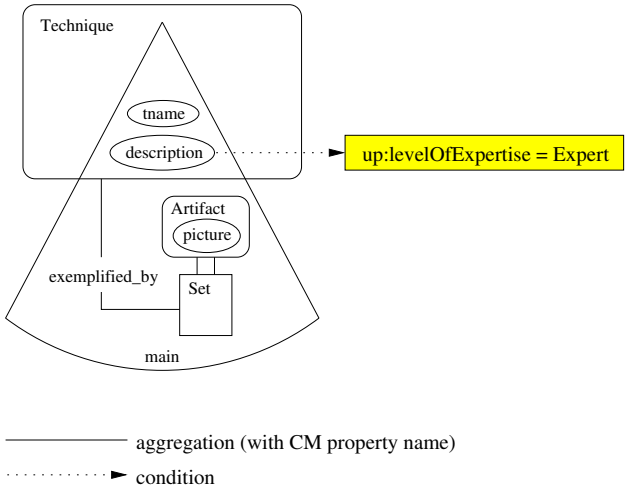


Figure 3.13: Adaptation in the application model.

appear also for users that are not *Expert*s (at that point in the presentation, the system can consider that the user is now ready to digest more advanced information). This is the so-called *context-dependent adaptation*, i.e., adaptation that affects only the current slice (by current slice is meant the top-level slice that contains the slice with the condition). Slices that have attached conditions outside the scope of a container slice have a *context-independent adaptation*, i.e., these slices will be removed from the whole presentation, no matter where they appear. This is similar to the *context-independent adaptation* for conceptual model adaptation showed in Figure 3.9. Note that the removal of a concept or concept attribute from a presentation has as its consequence the removal of all associated slices (i.e., slices for which the concept is an owner) and of the slice that refers to that concept attribute, respectively.

3.3.3 Presentation Design

The presentation design specifies the look-and-feel aspects of the presentation that is generated, independent from the implementation. The result of this activity is the *presentation model* (PM). It describes the layout and style information of the presentation. Both aspects are not to be neglected because they might have an immediate impact on the user choice for a certain application among applications offering similar functionality.

Figure 3.14 shows the PM vocabulary. It defines the following notions: *region*, *region attribute*, and *region relationship*. A region is an abstraction for a rectangular part of the display area where the content of a slice will be displayed. Each region is associated to a slice, the so-called region *owner*, from which the region content will be derived. The definition of region is very similar to that of a slice with a few simplifications and some additions. Region attributes are used to refer to media types. There are two types of region relationships, *region aggregation* and *region navigation*. The first type of region relationship facilitates the inclusion of a region into another region and the second type

of region relationship is used to define navigation between regions. The classification empty/non-empty does not apply for regions as regions get their content from the slice owner always at run-time.

The definition of regions is recursive: a region can be a simple region or can contain other regions. A *simple region* has only one region attribute that refers to the same media as the slice attribute of the corresponding simple slice from AM. Differently than for slices, one doesn't need to specify a corresponding concept attribute. A region that aggregates other regions is called a *complex region*. The recursion is defined by utilizing the region aggregation relationship. Another difference from slices is that for aggregation relationships there is no need to specify concept relationships. The *Set* construct, aggregation, and navigation relationships are copied for a region from the corresponding (by the owner relationship) slice. A *top-level region* corresponds to a Web page and is owned by a top-level slice.

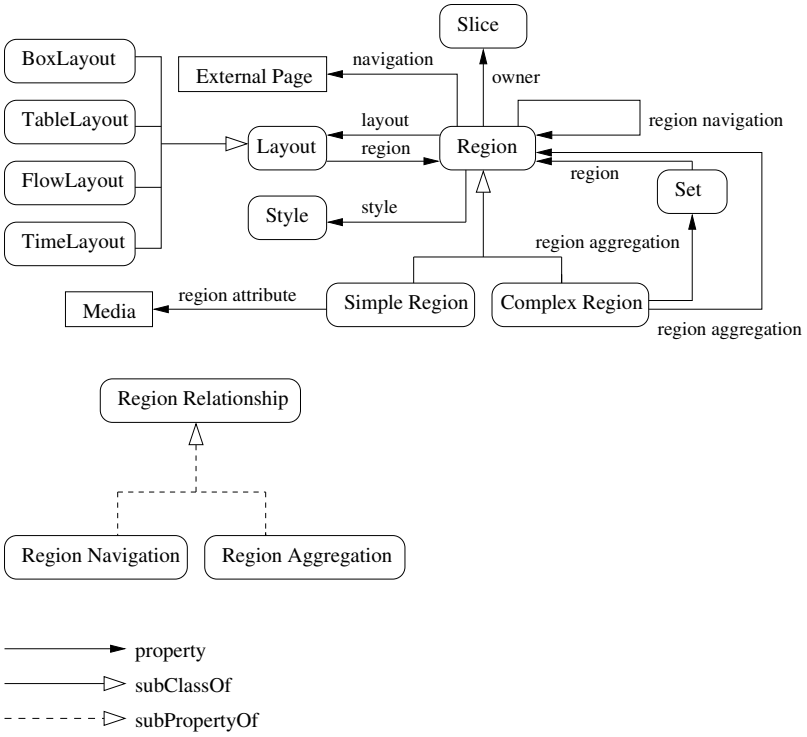


Figure 3.14: Presentation model vocabulary.

A region has a particular *layout manager* and *style* associated with it. There are four abstract layout managers: *BoxLayout*, *TableLayout*, *FlowLayout*, and *TimeLayout*. The layout managers describe the spatial/temporal arrangements of regions embedded into another region. The list of layout managers can be easily extended with other layouts like *BorderLayout*, *OverlayLayout*, *GuidedTourLayout*, etc.

[Frasincar et al., 2001] presents an alternative way of defining layouts by using qualitative and quantitative constraints for regions. These constraints are associated to region

relationships which are further classified as temporal, navigational, and spatial. Temporal relationships express the notion of time, navigational relationships represent (hyper)links, and spatial relationships define the spatial arrangements in presentations.

The layout managers were inspired from the abstract user interface (XML) representations from AMACONT [Fiala et al., 2004], UIML, and XIML [Souchon and Vanderdonckt, 2003]. These layout managers describe client-independent layouts that allow to abstract from the exact features of the browser's display. Note that because regions can be aggregated, layouts can also be aggregated (by means of regions), and thus one is able to build complex layouts.

The style information describes the colors, fonts, backgrounds to be used in a region, etc. Regions that do not have explicitly associated style information associated with them inherit the style of their container. In this way the designer is not forced to specify style information if that is not necessary.

The BoxLayout arranges the inner regions on one row or one column. Table 3.1 summarizes the possible attributes of the BoxLayout. The *height*, *width*, *border*, and *space* attributes have integer values that represent number of pixels.

Attribute	Meaning	Usage	Values
axis	orientation of the layout	required	"x"|"y"
rows	number of rows	optional	integer
columns	number of columns	optional	integer
height	height of the layout	optional	integer|percentage
width	width of the layout	optional	integer|percentage
border	size of the layout border	optional	integer
space	space between content and border	optional	integer

Table 3.1: BoxLayout attributes.

TableLayout arranges the inner regions in a table. Though it can be realized by nested *BoxLayout*s, we implemented it separately because SWISs often present dynamically retrieved sets of data in a tabular way. Table 3.2 summarizes the possible attributes of the TableLayout. Due to the dynamic nature of SWIS applications, the number of items in a complex region that uses the *Set* construct is not known at design-time. In such cases one should use only one of the dimensions: *rows* or *columns*. The missing dimension is automatically computed at run-time.

FlowLayout arranges the inner regions in the same way as words on a page: the first line is filled from left to right, then does the same for the lines below. Table 3.3 summarizes the possible attributes of the FlowLayout.

TimeLayout shows the inner regions in a time sequence that produces a slide show. Table 3.4 summarizes the possible attributes of the TimeLayout. The *duration* attribute has a float value that represents number of seconds. TimeLayout is used for platforms that support time sequences for presenting media items, e.g., Timed Interactive Multimedia

Attribute	Meaning	Usage	Values
rows	number of rows	optional	integer
columns	number of columns	optional	integer
height	height of the layout	optional	integer|percentage
width	width of the layout	optional	integer|percentage
border	size of the layout border	optional	integer
space	space between content and border	optional	integer

Table 3.2: TableLayout attributes.

Attribute	Meaning	Usage	Values
border	size of the layout border	optional	integer
space	space between content and border	optional	integer

Table 3.3: FlowLayout attributes.

Extensions for HTML (HTML+TIME) [Schmitz et al., 1998] and Synchronized Multimedia Integration Language (SMIL) [Ayars et al., 2005].

Attribute	Meaning	Usage	Values
duration	play time for a sequence element	optional	integer
repeat	number of times to repeat one sequence	optional	"indefinite"|integer

Table 3.4: TimeLayout attributes.

Table 3.5 summarizes the possible layout-related attributes for a region used inside a BoxLayout, TableLayout, or FlowLayout. These attributes describe how each referenced region has to be arranged in its surrounding layout. For example, the regions embedded in a layout form a sequence for which the order needs to be specified. For this purpose the *order* attribute is used. Note that for the TableLayout, the cell elements are counted from left to right and from top to bottom. The *sort* attribute specifies the sorting criteria for region instances. For example *alpha(Slice.Technique.tname,ascending)* specifies an alphabetical sorting in ascending order based on the name of artistic techniques. Besides the existing sorting functions like *alpha* and *num*, for alphabetical and numerical sorting, one can use its own sorting function (e.g., a multi-sort for data with different facets). If the sort criteria is not provided, the regions will be arranged in the order in which region content (data) is given by the data collection phase.

Even though most attributes are platform-independent, there are platform-dependent attributes in order to consider the specific card-based structure of WML presentations. The optional attribute *wml_visible* determines whether in a WML presentation a region should be shown on the same card. If not, it is put onto a separate card that is accessible

by an automatically generated hyperlink, the text of which is defined in *wml_description*. The *wml_description* attribute can refer to a constant string or one of the simple slices that give some of the content for a region. Note that this kind of content separation provides scalability by fragmenting the presentation according to the small displays of WAP phones.

Attribute	Meaning	Usage	Values
valign	vertical alignment	optional	"left"|"center"|"right"
halign	horizontal alignment	optional	"top"|"center"|"bottom"
ratio	space to be filled	optional	percentage
order	order in the sequence	optional	integer
sort	sorting criteria	optional	string
wml_visible	show on same card	optional	boolean
wml_description	anchor description	optional	string

Table 3.5: Layout-related region attributes inside BoxLayout/TableLayout.

Table 3.6 summarizes the possible layout-related attributes for a region used inside a FlowLayout. It is a subset of the previous set of attributes.

Attribute	Meaning	Usage	Values
order	order in the sequence	optional	integer
sort	sorting criteria	optional	string
wml_visible	show on same card	optional	boolean
wml_description	anchor description	optional	string

Table 3.6: Layout-related region attributes inside FlowLayout.

Table 3.7 summarizes the possible attributes for a region used inside a TimeLayout. The *begin*, *duration*, and *end* attributes have float values that represent the number of seconds.

Attribute	Meaning	Usage	Values
begin	(absolute) start time	optional	float
duration	play time	optional	float
end	(absolute) end time	optional	float

Table 3.7: Layout-related region attributes inside TimeLayout.

Table 3.8 presents some of the possible style attributes. These attributes refer to the font characteristics (e.g., size, color), background, link colors, etc. The definition of these attributes is inspired from Cascading Style Sheets (CSS) [Bos et al., 2004].

Attribute	Meaning	Usage	Values
font-family	the family of a font	optional	“times”|“helvetica”| ...
font-style	the style of a font	optional	“normal”|“italic”
font-size	the size of a font	optional	“small”|“medium”|“large”
font-color	the color of a font	optional	“red”|“green”|...
font-weight	the weight of a font	optional	“normal”|“bold”|...
background-color	the color of the background	optional	“red”|“green”|...
link-color	the color of a not-visited link	optional	“red”|“green”|...
visited-color	the color of a visited link	optional	“red”|“green”|...
...			

Table 3.8: Style attributes.

The layout managers need to be instantiated in order to be used in the PM. The layout manager instances are used for complex regions. Also when referencing a region (or set of regions) one needs to define values for the layout-related region attributes corresponding to the layout associated to the container region.

Figure 3.15 shows an excerpt of the PM for the running example. Regions are depicted as rectangles. There are two top-level regions: *RegionFullT* and *RegionFullA*. *RegionFullT* and *RegionFullA* are owned by *Slice.Technique.main* and *Slice.Artifact.main*, respectively. We use the convention to denote the region (long) name by *Region.<Slice full name>.<Region short name>*. The short name of a region can be omitted from its full name, if the full name unambiguously identifies the region. The full name of *RegionFullT* is *Region.Slice.Technique.main.RegionFullT*. As the full names are quite long in the rest of the explanation it is used the short name of regions when these short names are available.

The region *RegionFullT* aggregates (by means of slice aggregation relationships) three regions: one contains the technique name, one contains the technique description and, one contains the set of pictures that exemplify a painting technique. As simple regions, the first two regions do not need a layout. The third region, a complex region, has a *TableLayout* specified for arranging the set of pictures. All three regions are arranged using a *BoxLayout* specified in the *RegionFullT*. The style information is given by the *DefaultStyle*. As can be seen from the figure the inner regions do not have the style information explicitly defined which means that they inherit the style information from the container region. In a similar manner is defined the region *RegionFullA*. The region navigation relationship connects *RegionBottomA* with *RegionFullA*.

Figure 3.16 shows some of the layout attributes and layout-related region attributes for our running example. *RegionFullT* has a BoxLayout with two attributes defined: *axis* with value *y* which indicates that this layout has a vertical arrangement and *width* with value *100%* which means that this layout will completely fill the width of its container. As *RegionFullT* is a top-level region, the container is the user's display. In *BoxLayout1* there are three regions embedded in the order specified by the *order* attribute. All three regions have the *halign* layout-related attribute defined in order to specify that their hor-

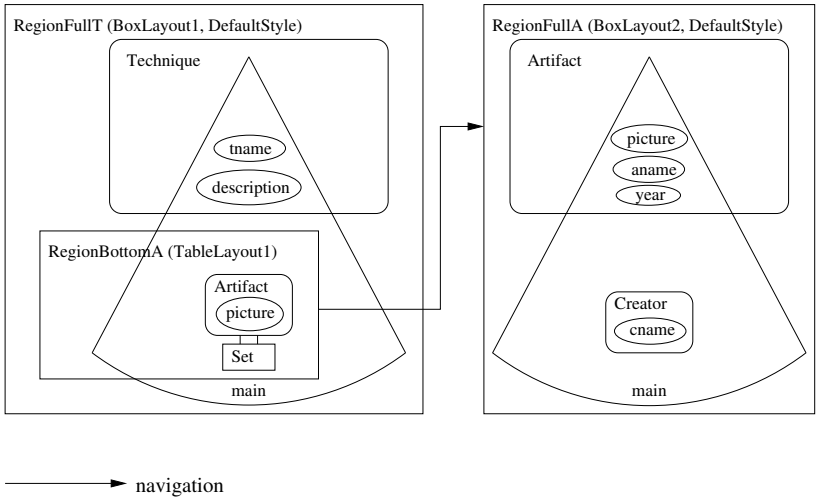


Figure 3.15: Presentation model.

izontal alignment will be centered. The third layout, *RegionBottomA* has two attributes defined: *cols* with value *3* which indicates that this layout has three columns and *width* with value *100%* which means that this layout will completely fill the width of its container. The container is in this case *RegionFullT*. *RegionBottomA* contains pictures for which the horizontal alignment is centered.

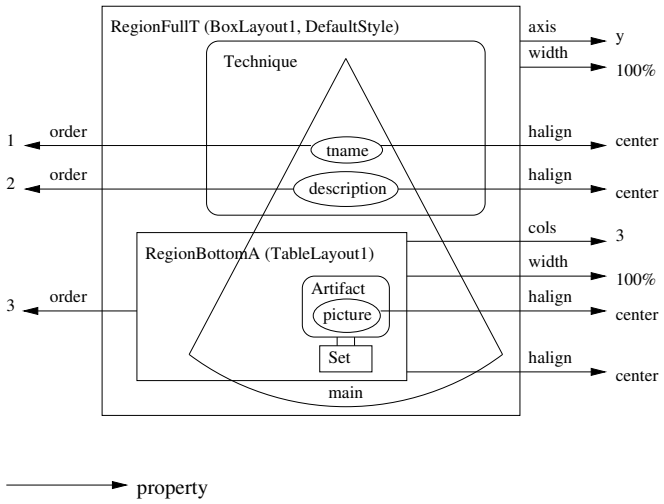


Figure 3.16: Layout and layout-related region attributes.

The PM presented in Figure 3.15 depicting the main regions for techniques and artifacts can be refined to a specific artistic domain. Figure 3.17 shows the specialization (in a type hierarchy) of the previous PM to the painting domain.

Regions are specialized by the *subClassOf* property. For example, the region *RegionFullA* is specialized by the region *RegionFullP*. *RegionFullP* inherits all the region relation-

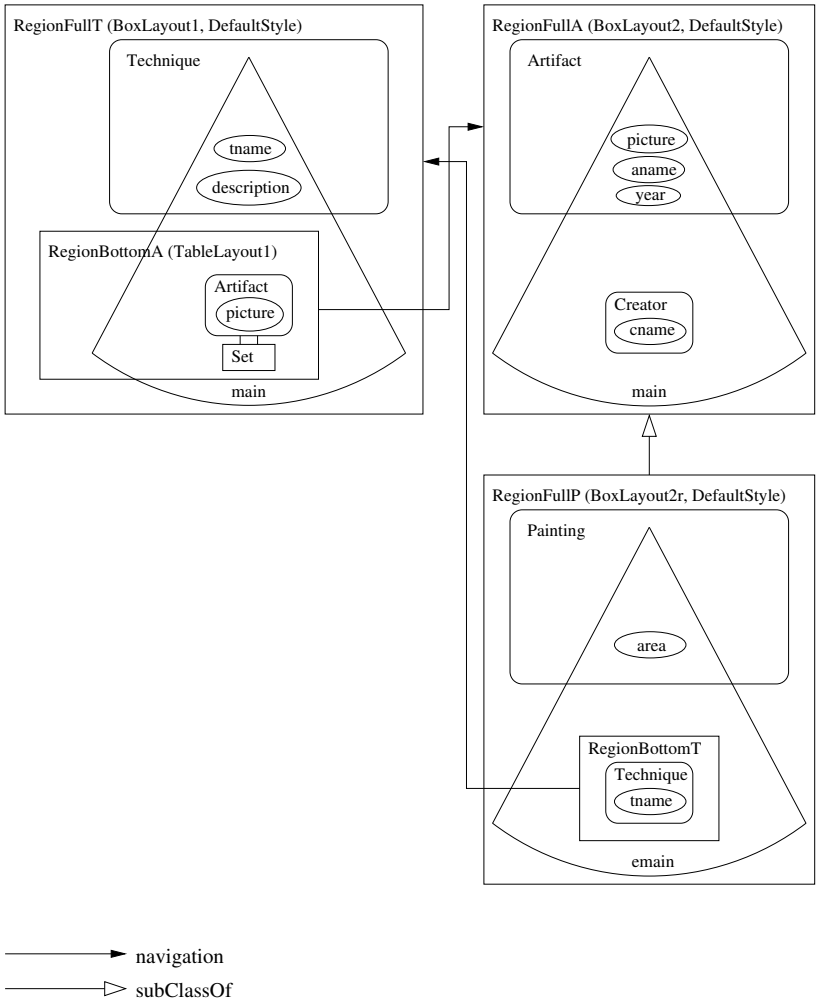


Figure 3.17: Specialization in the presentation model.

ships of *RegionFullA* and adds three new region relationships to it: two region aggregations and one region navigation. The aggregation relationships refer to the regions *Region.Slice.Painting.area* and *RegionBottomT*. As *RegionFullP* contains more regions than *RegionFullA*, the *BoxLayout2* is replaced with *BoxLayout2r* which among other things specifies in which order the added regions are placed. The navigation relationship links backwards the *RegionBottomT* with *RegionFullT*.

PM adaptation selects layouts or styles from PM to be used in the presentation. Figure 3.18 shows two adaptation examples in PM. In one example, depending on the size of the screen, the *RegionBottomA* uses a *BoxLayout* for PDA and a *TableLayout* for PC. The small screen size of the PDA requires a vertical arrangement of the data. In the other example the *DefaultStyle* uses medium fonts for a user with a average level of vision and large fonts for a user with a low level of vision. Other possible adaptation examples are: increasing the font of links for users with limited manual dexterity, eliminate colors for

color-blind users, etc.

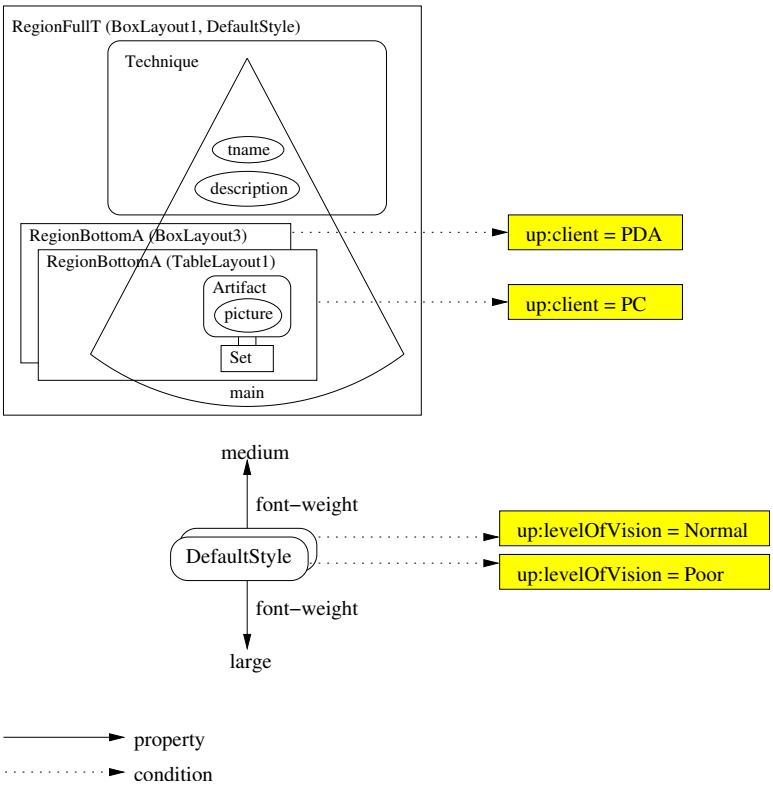


Figure 3.18: Adaptation in the presentation model.

3.3.4 Implementation

The implementation of the static variant of the Hera presentation generation phase is based on several data transformations specified by XSLT [Kay, 2005b] stylesheets. These transformations operate on the RDF/XML [Beckett, 2004] serialization of the RDF models. The XSLT processor used for interpreting XSLT stylesheets is Saxon [Kay, 2005a]. Figure 3.19 shows the transformation steps for the static variant of the Hera presentation generation phase. Each transformation step has a label associated with it. Some of these transformations have substeps which are labeled using a second digit notation.

In Figure 3.19 there are two types of dashed arrows: "is used by" to express that an RDFS model is used by another RDFS model and "has instance" to denote that an RDFS model has as instance an RDF model. A model vocabulary, a model, a model instance, and the generated presentations are depicted by rectangles. The transformation specifications are represented by ovals.

There are three types of model/transformation specifications: application-independent, application-dependent, and query-dependent. The application-independent specifications

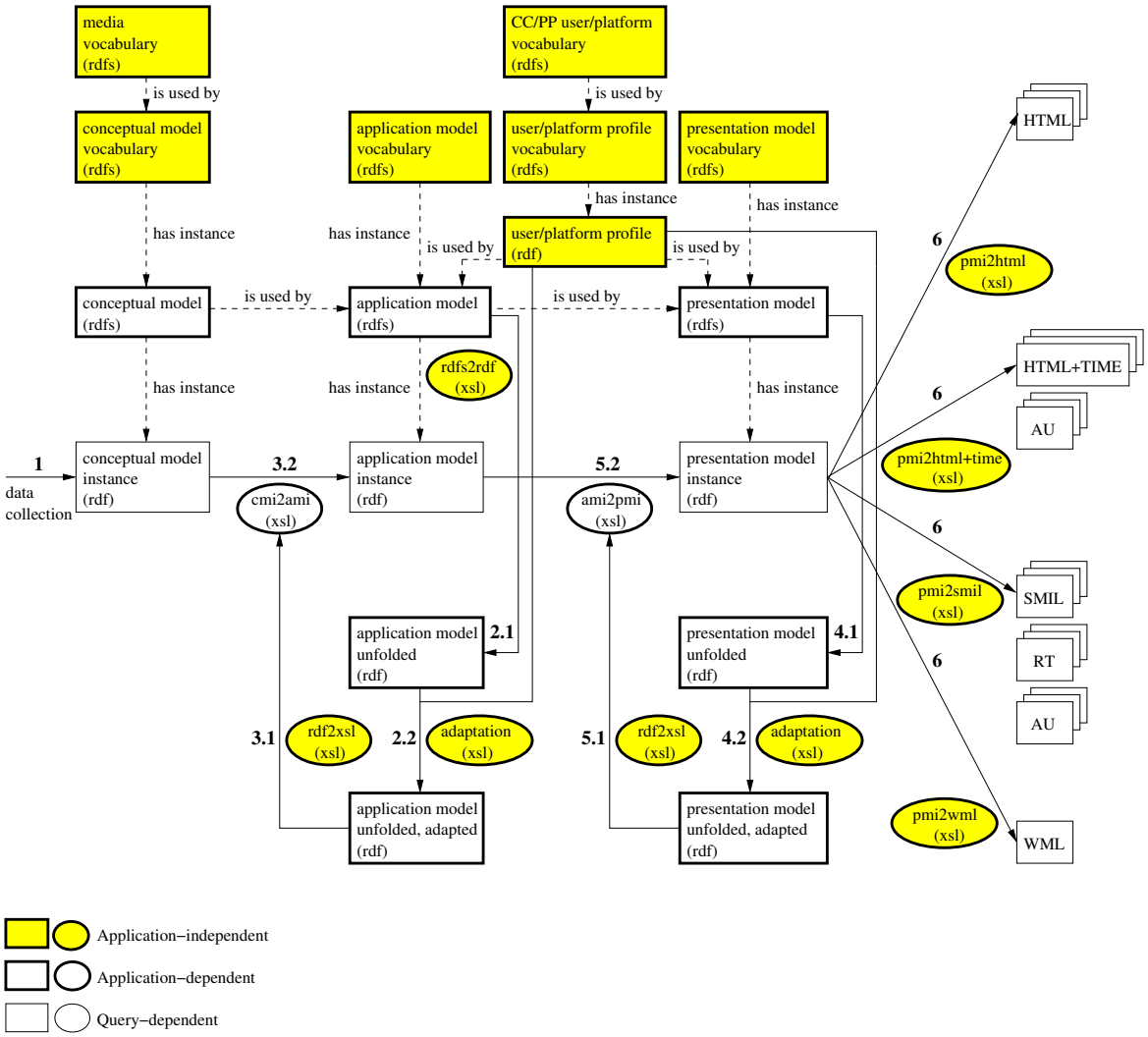


Figure 3.19: Presentation generation using XSLT.

do not refer to SWIS models (CM, AM, and PM), the application-dependent specifications refer to SWIS models, and query-dependent specifications refer to the SWIS models and the retrieved data (e.g., model instances). One can note that the query-dependent transformations are also application-dependent transformations. Transformations that are application-independent are also called generic transformations. Transformations that are application-dependent are also called specific transformations.

The input to the presentation generation phase is the conceptual model instance (CMI), i.e., the data retrieved in response to a user query. This data is produced in the *data collection* phase from a given set of input sources. This is step 1 in the figure and is not described here. More information on step 1 can be found in [Vdovjak et al., 2003]. At the current moment CM and media adaptation are carried on in the AM adaptation. Future implementations will separate the CM and media adaptation from the AM adaptation.

Step 2, the *AM generation*, builds an adapted AM template. This step contains two substeps: the *AM unfolding* and the *AM adaptation*.

Step 2.1, the *AM unfolding*, generates the AM template. The AM template represents the structure of an AM instance (RDF) based on the AM schema (RDFS). Such a template will ease the specification of an XSLT stylesheet used to convert a CM instance (CMI) to an AM instance (AMI). By unfolding the AM we mean repeating the process of adding properties inside the subject classes until slice references or media items are reached. In this way one obtains an AM template which will be filled later on with appropriate instances.

Step 2.2, the *AM adaptation*, executes the adaptation specifications on the AM template. The transformation stylesheet of this step has two inputs: the AM template and the UP. The UP attributes are replaced in the conditions by their corresponding values. The slices that have the conditions not valid are discarded and the hyperlinks pointing to these slices are disabled.

Step 3, the *AMI generation*, instantiates the AM with the retrieved data. This step is composed of two substeps: the *AMI transformation generation* and the *AMI creation*.

Step 3.1, the *AMI transformation generation*, builds the transformation stylesheet that will convert a CMI to an AMI. This step uses an XSLT stylesheet that will generate another XSLT stylesheet. One should note that an XSLT stylesheet is a valid XML file that can be produced by another XSLT stylesheet. This technique was also successfully used in the previous version of the implementation which was XML-based [Frasincar and Houben, 2001]. This transformation is based on the *owner* of a slice and the *concept attribute* of a simple slice. The following name convention is used: a slice instance name (e.g., *Slice.Painting.main_ID1*) is obtained from the slice name (e.g., *Slice.Painting.main*) concatenated with the suffix (e.g., *ID1*) of the associated concept instance identifier (e.g., *Painting_ID1*). The implemented algorithm is straightforward: instantiate all slices for all the corresponding retrieved concept instances and each time a slice is referenced add its identifier based on the above name convention.

The transformation used in this phase is a generic one, but the output that it produces is used for a specific transformation (the next step).

Step 3.2, the *AMI creation*, converts the CMI to an AMI. The XSLT stylesheet obtained in the previous substep is applied to the CMI to yield an AMI. As opposed to the previous transformations, this stylesheet will operate for inputs and outputs that are both query-dependent. For each query, Hera will dynamically instantiate the AM with the query result, i.e., a CMI.

The PM-related transformation steps (steps 4 and 5) are realized in a similar manner as the AM-related transformation steps (steps 2 and 3).

Step 4, the *PM generation*, builds a PM template. This step contains two substeps: the *PM unfolding* and the *PM adaptation*.

Step 4.1, the *PM unfolding*, generates the PM template. The PM template represents the structure of a PM instance (RDF) based on the PM schema (RDFS). Such a template will ease the specification of an XSLT stylesheet used to convert an AM instance (AMI) to a PM instance (PMI). By unfolding the PM we mean repeating the process of adding properties inside the subject classes until slice references or media items are reached. In this

way, one obtains a PM template which will be filled later on with appropriate instances.

Step 4.2, the *PM adaptation*, executes the adaptation specifications on the PM template. The transformation stylesheet of this step has two inputs: the PM template and the UP. The UP attributes are replaced in the conditions by their corresponding values. The layouts and styles that have the conditions not valid are discarded.

Step 5, the *PMI generation*, instantiates the PM with data from the AMI. This step is composed of two substeps: the *PMI transformation generation* and the *PMI generation*.

Step 5.1, the *PMI transformation generation*, builds the transformation stylesheet that will convert an AMI to a PMI. As in step 3.1, an XSLT stylesheet that will generate another XSLT stylesheet is used. This transformation is based on the *owner* of a region and the fact that simple regions are associated to simple slices. The following name convention is used: a region instance name (e.g., *Region.Slice.Painting.main.RegionFullA_ID1*) is obtained from the region name (e.g., *Region.Slice.Painting.main.RegionFullA*) concatenated with the suffix (e.g., *ID1*) of the associated slice instance identifier (e.g., *Slice.Painting.main_ID1*). The implemented algorithm is straightforward: instantiate all regions for all the corresponding slice instances and each time a region is referenced add its identifier based on the above name convention.

Step 5.2, the *PMI creation*, converts the AMI to a PMI. The XSLT stylesheet obtained in the previous substep is applied to the AMI to yield a PMI. As opposed to the previous transformations, this stylesheet will operate for inputs and outputs that are both query-dependent.

Step 6, the *presentation data generation*, transforms the PMI into code specific for the user's browser. Note that a set of Web pages is generated at-a-time. Some of supported formats are: HTML, HTML+TIME, WML, and SMIL. For each type of serialization a specific stylesheet is used. The stylesheets used for the HTML, HTML+TIME, and SMIL use the ability of XSLT 2.0 [Kay, 2005b] to generate multiple outputs (this feature is not supported in XSLT 1.0 [Clark, 1999]). In order to generate multiple outputs the XSLT 2.0 *result-document()* function was used.

For HTML(+TIME), BorderLayout and TableLayout are implemented using tables. An HTML presentation is composed from the *index.html* document (starting point of the presentation) and a set of HTML pages each corresponding to a top-level slice.

The *FlowLayout* is supported by any HTML browser (the content of a table cell is automatically wrapped if it doesn't fit one line). *TimeLayout* is supported only by HTML+TIME and SMIL browsers.

For WML, there is only one layout supported, i.e., the *BorderLayout* with a vertical alignment. Because lists are not available in WML, they are implemented as simple sequences of items without any visual cues. To each top-level region corresponds a WML *card*. A WML presentation is composed from a single WML document, *a deck* that contains a set of cards. The first card is the starting point of the presentation.

For SMIL, there is an explicit part to describe the layout of a document. As tables/flow are not supported in SMIL, one needs always to fully define the layout information for *BoxLayout*, *TableLayout*, and *FlowLayout*. The *TimeLayout* was defined using the *seq* container for regions. Hera regions are implemented as SMIL regions. A SMIL presentation

is composed from a main SMIL document (starting point of the presentation), a set of SMIL documents each corresponding to a top-level region, a set of RealText (RT) clips, one per each text media, and a set of audio clips (AU), one per each audio media.

3.4 Presentation Generation (Dynamic)

Recently the Hera methodology has been extended in order to accommodate more complex forms of user interaction in addition to simple link-following, e.g., interaction by means of forms in which the user can enter data [Houben et al., 2004]. In this way the user can better personalize the SWIS according to his needs, specially regarding the dynamics within a browsing session. Figure 3.20 shows the "loop" with which we extended the presentation generation to support this additional dynamics and to allow the user to influence the generation of the Web presentation. Note that in response to a user query only one page is generated at-a-time instead of the full Web presentation as is the case for the static variant of the presentation generation phase. Generating one-page-at-a-time allows the system to consider the user input before generating the next Web page. The request contains the (owner) concept instance identifier and the slice type of the next slice to be generated (i.e., the one corresponding to the next Web page).

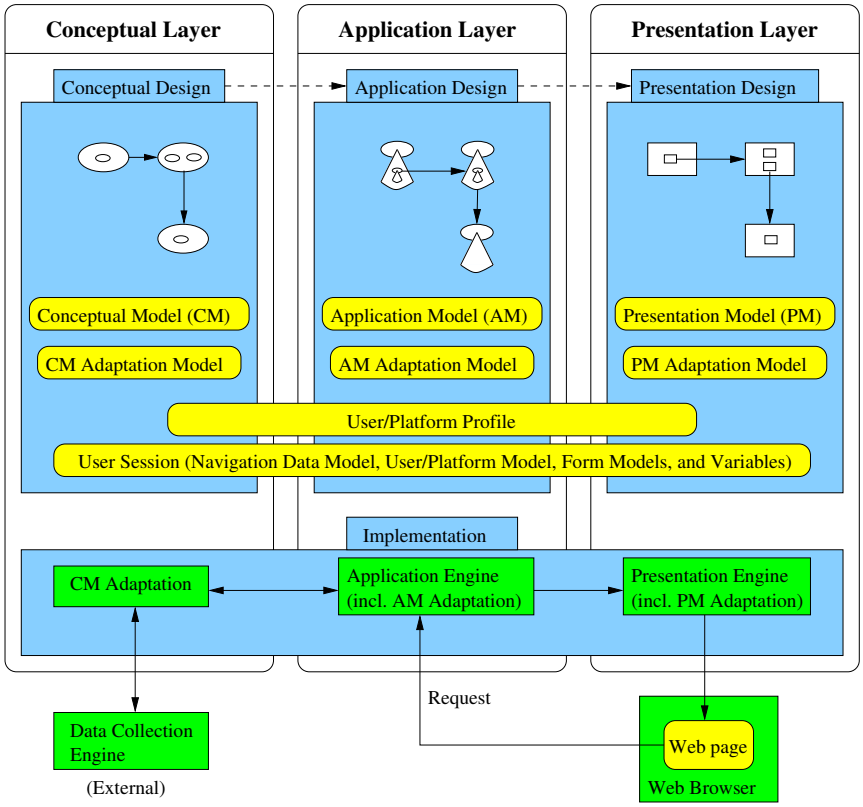


Figure 3.20: Presentation generation phase (dynamic).

In order to illustrate the dynamic version of the presentation generation the running example is extended such that it allows the visitor to buy posters of the paintings in the museum. For simplicity we didn't model explicitly the posters, assuming a one-to-one correspondence with the depicted painting. Also, after buying a certain painting, the user will not be presented with the same painting again.

In addition to the data from CM, AM, and PM, interaction requires a support for creating, storing, and accessing data that emerges while the user interacts with the system. This support is provided by means of the user session (US). US is composed of the *navigation data model*, *user/platform model*, *form models*, and *variables*.

The purpose of the navigation data model (NDM) is to complement the CM with a number of auxiliary concepts that do not necessarily exist in the CM (although this is the decision of the designer in concrete applications) and which can be used in the AM when defining the behavior of the application and its navigation structure.

The user/platform model (UM) stores user preferences and device capabilities that change during user browsing (e.g., network connection speed, user knowledge on some of the displayed topics, etc.). In Section 3.3 the UP was defined. The UP-based adaptation is done at the beginning of the user browsing session in order to adapt the CM, AM, and PM. In a similar way the UM is used to adapt the CM, AM, and PM. Differently than for UP, the UM-based adaptation is done before each Web page is generated.

The form models (FM) describe the data that is entered by the user by means of forms. Each form has a so-called form model associated with it. The data input by the user in a form populates the associated form model. Similar to XForms [Dubinko et al., 2003], a form separates presentation from content. FM describes the form content. The presentation-related issues of forms are given in the AM.

The session variables are the concept instance identifier, i.e., *instanceid*, and the slice type, i.e., *slicetype*, of the previous slice (the one from which a request originated), and a number of variables to store temporary data created during a user browsing session (e.g., for storing the URIs of newly created resources).

We remark that from the system perspective the concepts in the NDM can be divided into two groups. The first group essentially represents views over the concepts from the CM, the second group corresponds to a locally maintained repository. A concept from the first group can be instantiated only with a subset of instances of a concept existing in the CM, without the possibility to change the actual content of the data. A concept from the second group is populated with instances based on the user's interaction, i.e., the data is created, updated, and potentially deleted on-the-fly. The AM can refer to the concepts from NDM as if they were representing "real" data concepts.

The NDM of our example is depicted in Figure 3.21; it consists of the following concepts: *SelectedPainting*, *Order*, and *Trolley*. The *SelectedPainting* concept is a subclass of the *Painting* concept from the CM. It represents those paintings which the user selected in a selection form. The *Order* concept models a single ordered item consisting of a selected painting (the property *includes*) and the *quantity* represented by an Integer. The *Trolley* concept represents a shopping cart containing a set of orders linked by the property *contains*.

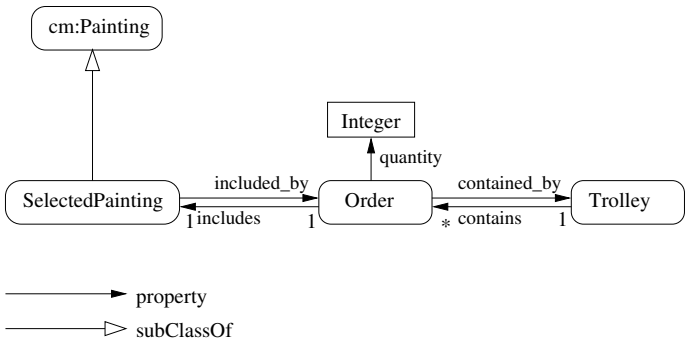


Figure 3.21: Navigation data model.

In the example the *SelectedPainting* concept belongs to the group of view concepts whereas both the $Order$ and the $Trolley$ are updatable concepts with the values determined at run-time. This is reflected also in the navigational data model instance (NDMI) depicted in Figure 3.22 that results from the user's desire to buy 1 poster of the selected painting. The instance $Painting1$ comes from the CM, i.e., it is not (re)created: what is created however, is the *type* property associating it with the *SelectedPainting* concept. Both instances $Order1$ and $Trolley1$ are created during the user's interaction; they, as well as their properties, are depicted in bold in Figure 3.22.

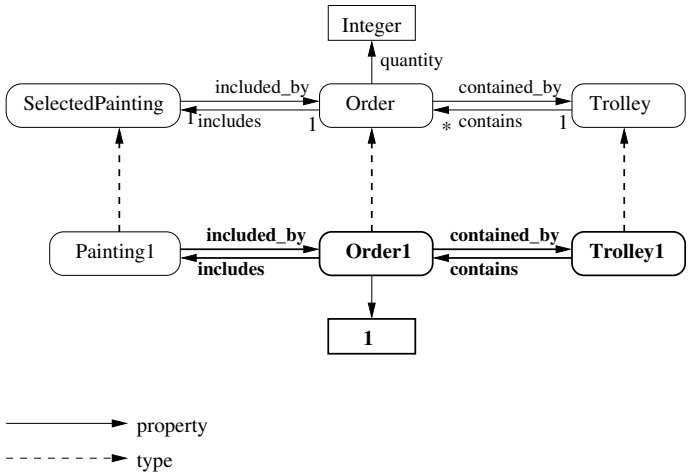


Figure 3.22: Navigation data model instance.

The application model vocabulary from Figure 3.10 was extended in order to support forms. Figure 3.23 shows these extensions, inspired by the XForms standard. Similar to XForms, a form separates presentation from content. The presentation-related issues of forms are associated to the AM. In AM, a *form* is a particular type of slice which has controls associated with it. Some of the supported form controls (as in XForms) are: *Select1* (*S1*), selects one instance from a set; *SelectN* (*SN*), selects several instances from a set; *Input (I)*, accepts one line of input text, etc.

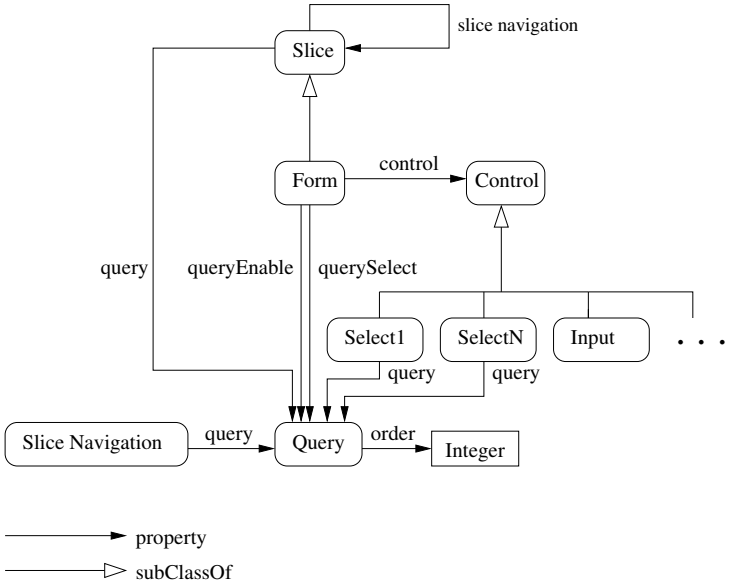


Figure 3.23: Extended application model vocabulary.

The dynamics of the application is given by a set of AM queries used for selection, deleting, or updating of data. These queries can be attached to:

- *slices*, to express user-independent updates (e.g., creation of a trolley),
- *form controls*, to get values for these controls (e.g., select all names of paintings that are not in the trolley),
- *forms*, (1) to enable/disable a form (e.g., if the user has already added all paintings to his trolley, there is no painting left to be offered to the user for the next selection, and therefore the selection form is disabled) or (2) to select the concept instance for the next slice (e.g., after selecting a painting, the main slice of the selected painting is presented),
- *slice navigation*, to express user-dependent updates (e.g., create order and add it to the trolley).

By a query that enables/disables a form it is actually meant a condition that uses some query results for enabling/disabling a form. The identification of the query with the condition is done because the condition usually is a very simple one (in most of the encountered cases it is a comparison of the query result with '0'). An element from AM can have attached a single query or a sequence of queries. The order in which the sequence queries will be executed is given by the *order* attribute.

The content of the form is based on a form model (FM), i.e., the schema of the data associated with a certain form. The data of the form that populates (at run-time, based on user actions) the FM is the so-called form model instance (FMI). The mappings (bindings)

of the data provided by the form controls to the form model instance is outside the scope of this description as this is done by an external XForms processor. Figure 3.24 shows an example of a form model and its instance.

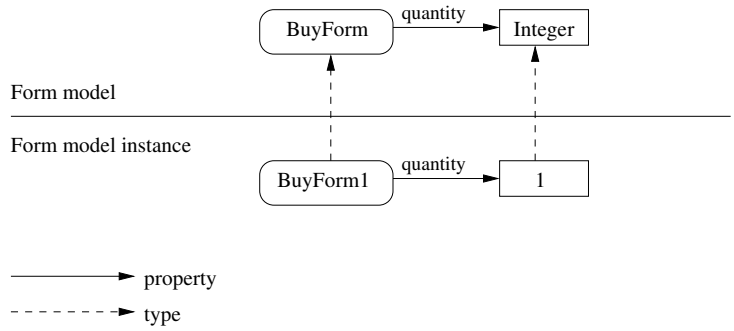


Figure 3.24: Form model and form model instance.

Figure 3.25 shows two form slices that can be embedded in an AM. The short names of the forms are *SelectForm* and *DeleteForm* and the long names are *Slice.Painting.SelectForm* and *Slice.Trolley.DeleteForm*, respectively. The owner of the *SelectForm* is *Painting* and the owner of the *DeleteForm* is *Trolley*. Two queries are used to enable/disable the forms: *QEnableSF* and *QEnableDF*. Both forms have one control field defined *S1* (selects one instance from a set). The values from which the user makes one selection are given by the queries *QSelectSFPn* and *QSelectDFPn*. The first form has *QSelectP* a query that selects a painting instance identifier based on the user's choice. The second form has a slice navigation relationship with an update query defined, i.e., *QDeleteO*.

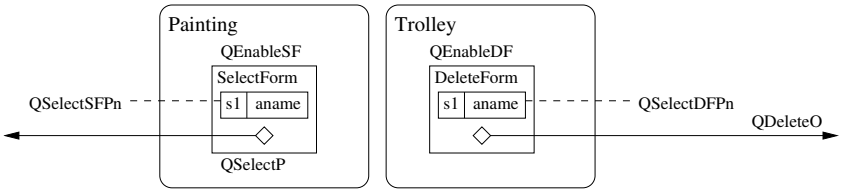


Figure 3.25: Form in application model.

Figure 3.26 shows the application model extended with forms. The main slice of a painting depicts information related to the painting. It also contains the *BuyForm*, a form that allows the user to make an order by specifying the quantity of desired posters for the presented painting. In order not to produce too much visual clutter, we do not show in the figure the concept owner of the form (this is the same as the owner of the destination slice when one navigates from that form). The main slice of the trolley displays the orders contained in the trolley. Note that when the user makes an order, this order is immediately added to the trolley. In addition the main slice of the trolley has two other forms *SelectForm* and *DeleteForm*. *SelectForm* is used to select paintings by their name, paintings which do not have posters in the trolley. *DeleteForm* is used to delete orders from the trolley.

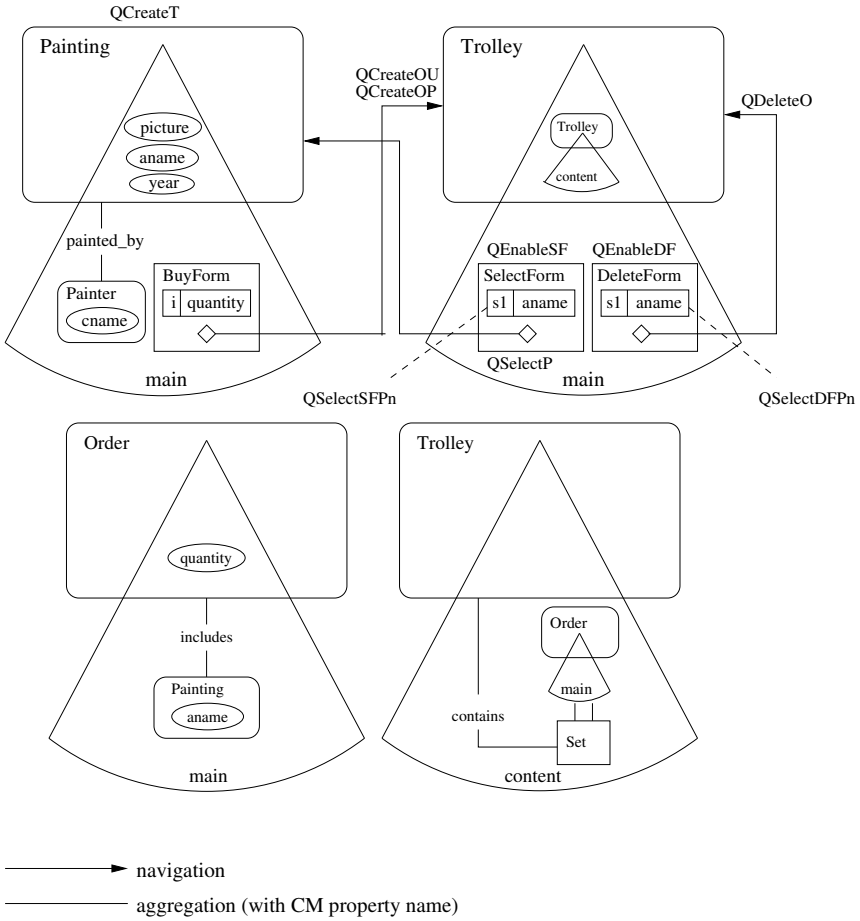


Figure 3.26: Extended application model.

Because models are represented in RDF(S), the AM queries are described using an RDF query language. As an RDF query language it was chosen SeRQL [Aduna, BV, 2005], one of the most expressive RDF query languages that supports not only the selection of RDF data but also the creation of new RDF data. In the rest of this section several queries are presented in their SeRQL syntax. Due to the fact that SeRQL doesn't support nested queries some queries are expressed in RQL [Karvounarakis et al., 2002]. In the rest of this section the queries from Figure 3.26 are presented.

Figure 3.27 shows *QCreateT* a query attached to the main slice of a painting. It is used to create a trolley for the user. The SeRQL was extended with the `new()` function that is able to create a URI (identifier) unique in the application for a new resource. The newly created URI is stored in the user session variable `trolleyid`.

```
CONSTRUCT {new()}<rdf:type><ndm:Trolley>
```

Figure 3.27: QCreateT (create trolley).

QCreateOU and *QCreateOP* are a sequence of queries attached to the slice navigation from *BuyForm* to the main slice of the trolley. Figure 3.28 depicts *QCreateOU*, a query that creates a new order. The newly created URI is stored in the user session variable *orderid*.

```
CONSTRUCT {new()}<rdf:type><ndm:Order>
```

Figure 3.28: QCreateOU (create order).

Figure 3.29 shows *QCreateOP*, a query that fills the order properties and adds the order to the trolley. Note that the order is captured in NDM, the owner concept instance identifier of the current slice and the newly generated order identifier are user session variables, and the user input (the poster's quantity) is captured in *BuyForm1*, the form model instance of the form *BuyForm*.

```
CONSTRUCT
     {x}<ndm:contains>{y},
     {y}<ndm:contained_by>{x},
     {y}<ndm:includes>{z},
     {z}<ndm:included_by>{y},
     {y}<ndm:quantity>{v}
FROM
     {session}<var:trolleyid>{x},
     {session}<var:instanceid}{z},
     {session}<var:orderid>{y},
     {BuyForm1}<bf:quantity>{v}
```

Figure 3.29: QCreateOP (add order to trolley).

Figure 3.30 shows *QEnableSF*, a query attached to the *SelectForm* form in order to enable/disable this form. If all paintings have orders associated with them, the *SelectForm* is disabled, as there are no paintings left for user selection. SeRQL was extended with aggregation functions like the *count()* function.

```
(SELECT count(x)
 FROM {x}<rdf:type><cm:Painting>
 WHERE NOT x IN SELECT y
                FROM {session}<var:trolleyid>{v},
                     {v}<ndm:contains>{w},
                     {w}<ndm:includes>{y}) > 0
```

Figure 3.30: QEnableSF (condition that enables/disables SelectForm).

Figure 3.31 shows *QSelectSFPn*, a query attached to the control of the form *SelectForm* in the main slice of trolley. Note that *QSelectSFPn* is a nested query: first the paintings included in the order are computed and the result is subtracted from the set of all the paintings. The query returns the name of the paintings that are not in the trolley.

```
SELECT xname
FROM {x}<rdf:type><cm:Painting>,
     {x}<cm:aname>{xname}
WHERE NOT x IN SELECT y
               FROM {session}<var:trolleyid>{v},
                    {v}<ndm:contains>{w},
                    {w}<ndm:includes>{y}
```

Figure 3.31: QSelectSFPn (select paintings (names) that are not in the trolley).

Figure 3.32 shows *QSelectP*, a query attached to the *SelectForm* in order to select the concept instance that owns the next slice to be presented (i.e., the main slice of a painting). In the future we would like to exploit this selection feature (based on queries) at a more general level, i.e., in the navigation between any two slices and not just between forms (form slices) and slices. In this way the restriction that slice navigation relationships connect slices that have the same owner will be eliminated. Nevertheless one should ensure that only one instance of the destination slice is created.

```
SELECT x
FROM {SelectForm1}<sf:aname>{yname},
     {x}<cm:aname>{yname}
```

Figure 3.32: QSelectP (select painting).

Figure 3.33 shows *QEnableDF*, a query attached to *DeleteForm* in order to enable/disable this form. If the trolley is empty, *DeleteForm* is disabled, as there are no orders to delete.

```
(SELECT count(x)
 FROM {session}<var:trolleyid>{y},
      {y}<ndm:contains>{x}) > 0
```

Figure 3.33: QEnableDF (condition that enables/disables DeleteForm).

Figure 3.34 shows *QSelectDFPn*, a query attached to the control of the form *DeleteFrom* in the main slice of trolley. The query returns the name of the paintings that are in the trolley.

```
SELECT xname
FROM {session}<var:trolleyid>{y},
     {y}<ndm:contains>{x},
     {x}<cm:aname>{xname}
```

Figure 3.34: QSelectDFPn (select paintings (names) that are in the trolley).

Figure 3.35 shows the query *QDeleteO* associated to *DeleteForm* used to delete a selected painting order from trolley. The SeRQL query language was extended with the `DELETE` construct. Basically it is a deletion of statements from an RDF model. The deletion of resources from an RDF model can be easily done by deleting statements of the form

`{x}<rdf:type>{rdf:Resource}`, where `x` is the URI of a resource. A garbage collector will make sure that the properties of the deleted resources will be also removed from the model.

```
DELETE
     {x}<ndm:contains>{y},
     {y}<ndm:contained_by>{x},
     {y}<ndm:includes>{z},
     {z}<ndm:included_by>{y},
     {y}<ndm:quantity>{a}
FROM
     {session}<var:trolleyid>{x},
     {DeleteForm1}<df:aname>{yname},
     {y}<cm:aname>{yname},
     {y}<ndm:includes>{z},
     {y}<ndm:quantity>{a}
```

Figure 3.35: QDeleteO (delete selected order from trolley).

In the above queries we did need to extend Se(RQL) with new constructs like URI generators, aggregation functions, and `DELETE` statements. We do hope that future RDF query languages will be equipped with all these constructs.

3.4.1 Implementation

The implementation of the dynamic variant of the Hera presentation generation phase is based on several data transformations realized in Java. The Se(RQL) queries are executed by Sesame [Aduna, BV, 2005] and the data transformations are implemented in Jena [Hewlett-Packard Development Company, LP, 2005]. In this way the data transformations exploit more of the RDF(S) semantics given by the Hera models than the ones based on XSLT. A transformation language for XML documents like XSLT cannot use the full RDF semantics stored in the RDF/XML serialization of an RDF model.

Figure 3.36 shows the transformation steps for the dynamic variant of the Hera presentation generation. Each transformation step has a label associated with it. Some of these transformations have substeps which are labeled using a second digit notation. In Figure 3.36 there are two types of dashed arrows: “is used by” to express that an RDFS model is used by another RDFS model and “has instance” to denote that an RDFS model has as instance a certain RDF model. A model vocabulary, a model, a model instance, and the generated presentations are depicted by rectangles. The transformation specifications are represented by ovals. In the same way as for the static variant of the implementation models and transformation specifications are classified as application-independent, application-dependent, and query-dependent.

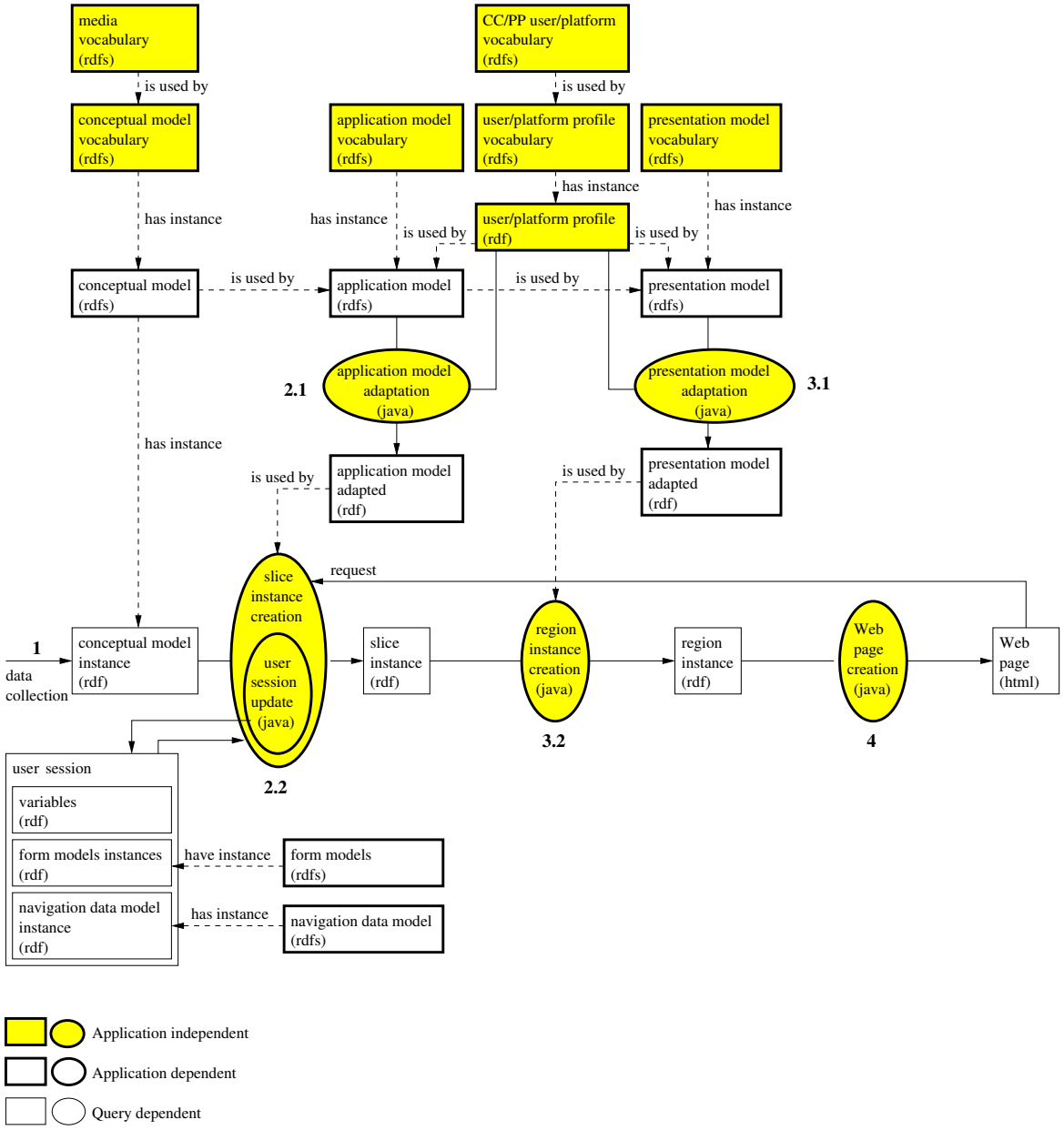


Figure 3.36: Presentation generation using Java.

Step 1, the *data collection* phase, is the same as in the static variant of the implementation. The result of this step is the CMI, i.e., the data retrieved in response to a user query. More information on step 1 can be found in [Vdovjak et al., 2003].

Step 2, the *slice instance generation*, computes a top-level slice instance in response to a user request. This step contains two substeps: the *AM adaptation* and the *slice instance creation*.

Step 2.1, the *AM adaptation*, executes the adaptation specifications on the AM. This transformation has two inputs: the AM and the UP. The UP attributes are replaced in

the conditions by their corresponding values. The slices that have the conditions not valid are discarded and the hyperlinks pointing to these slices are disabled. This step is executed only once at the beginning of a user session. In the current version of the implementation, AM adaptation based on the user model is not performed. Future versions of the implementation, that will make use of the user model will execute this step at each user request.

Step 2.2, the *slice instance creation*, creates the next slice instance. The user request provides: the slice type and the concept instance identifier of the slice instance corresponding to the next Web page to be computed, and possibly form model information, in case that request originates from a form. The first user request in a session specifies also the Hera models that will be used in the current session. The queries associated with the slice navigation that initiated the request and the queries associated to the slice to be computed are executed in the *user session update*. Besides updating the NDMI, the *user session update* also stores in the user session the form models and the value of the variables associated to queries.

Step 3, the *region instance generation*, computes the top-level region instance corresponding to the previously computed slice instance. This step contains two substeps: the *PM adaptation* and the *region instance creation*.

Step 3.1, the *PM adaptation*, executes the adaptation specifications on the PM. This transformation has two inputs: the PM and the UP. The UP attributes are replaced in the conditions by their corresponding values. The layouts and styles that have the conditions not valid are discarded. Similar to step 2.1, this step is executed only once at the beginning of a user session. In the current version of the implementation, PM adaptation based on the UM is not performed. Future versions of the implementation, that will make use of the UM, will execute this step at each user request.

Step 3.2, the *region instance creation*, creates the region instance for the previously computed top-level slice instance.

Step 4, the *Web page creation*, transforms the region instance generated in the previous step into code specific to the user's browser. Note that only one Web page is generated at-a-time. At the current moment only HTML is supported by the implementation.

3.5 Conclusions

Hera is a model-driven methodology for designing Semantic Web Information Systems. The presentation generation phase of the Hera methodology builds a Web presentation for some given input data. The Hera presentation generation phase has two variants: a static one that computes at once a full Web presentation, and a dynamic one that computes one-page-at-a-time by letting the user influence the next Web page to be presented. The design of both variants uses models that are specified in RDF. The implementation of the static variant is based on XSLT data transformations and the implementation of the dynamic variant is based on Java data transformations.

As future work we would like to improve the design and implementation of the Hera

presentation generation phase. For the static variant we would like to implement the CM and media adaptation as given in the design specifications as a separate (from AM adaptation) data transformation. The design of the dynamic variant can be extended by adding specifications for UM-based adaptation. With respect to this we anticipate to reuse some of the work done in the adaptive hypermedia field [De Bra et al., 1999]. The implementation of the dynamic variant needs to be extended with other code generators like HTML+TIME, WML, and SMIL.

Also we would like to investigate the use of a declarative RDF transformation language (similar to XSLT but exploiting better than XSLT the RDF semantics). In [van Ossenbruggen et al., 2005] it is proposed the use of XSLT stylesheets in combination with SeRQL queries (for selections) as a possible RDF transformation language. This hybrid solution is easy to implement and it exploits more of the RDF semantics than XSLT. Nevertheless it relies on the RDF/XML serialization of RDF models and it is less elegant than a solution based on the RDF data model. Lacking an RDF data transformation language based on the RDF data model, we plan investigate the definition and implementation of such a language.

At the current moment Hera doesn't support the requirements phase of the development life cycle of a SWIS. We would like to extend our methodology with a task (activity) model that will specify the activities that can be performed by a user with the system. Once devising a task model one can generate the navigation structure of the application from the task model eliminating the design effort for defining new application models. The task models can be assigned to a particular user or to a group of users (users that share the same task model) facilitating thus the definition of coarse-grained adaptation at navigation level.

Chapter 4

Hera Presentation Generator

The Hera Presentation Generator (HPG) is the integrated development environment that supports the Hera methodology. It is based on a number of software tools created for the Hera methodology that we integrated into one common environment. The HPG fills the existing gap for development environments supporting SWIS design. There are two versions of HPG: HPG-XSLT and HPG-Java. HPG-XSLT corresponds to the static variant of the Hera presentation generation phase and HPG-Java corresponds to the dynamic variant of the Hera presentation generation phase. A comparison of the two implementations based on their advantages and disadvantages is given. We also present a distributed architecture for the HPG based on Web Services.

4.1 Introduction

The success of a WIS design methodology is often depending on the existence of software tools that support the proposed methodologies. As shown in Chapter 2, many of the model-driven methodologies for SWIS design do not provide integrated development environments (IDE) similar to ones found for WIS design (e.g., RMCase, WebRatio) in order to help the design and construction of SWIS. An IDE has the advantage of supporting all design steps of a methodology from a single tool.

In Chapter 3 it was presented the presentation generation phase of Hera, a SWIS design methodology. The presentation generation phase has two variants: a static variant in which the user is unable to influence the generated hypermedia presentation, and a dynamic variant that considers user input before each hypermedia page is generated. The static variant uses XSLT data transformations and the dynamic variant uses Java data transformations.

The Hera Presentation Generator (HPG) is an IDE to support the development of SWIS using the Hera methodology. Based on the two Hera implementation variants, two versions of the HPG were realized: HPG-XSLT which corresponds to the static variant, and HPG-Java which corresponds to the dynamic variant.

The remainder of this Chapter is structured as follows. Section 4.2 describes HPG-XSLT. Section 4.3 presents HPG-Java. The two version of HPG are compared in Section 4.4 Section 4.5 shows a Web Service-Oriented Architecture for HPG. Section 4.6 concludes the chapter and presents future work.

4.2 HPG-XSLT

HPG-XSLT is an IDE that assists the designer of the static variant of the Hera presentation generation phase. It integrates several tools built during the last couple of years in the Hera project into one common environment. Besides its practical purpose, HPG-XSLT has also an explanatory purpose as it offers an explicit view over the data flow in the Hera presentation generation phase.

HPG-XSLT has the following graphical interfaces: *CM design interface*, *AM design interface*, *PM design interface*, *UP design interface*, and *implementation interface*, that correspond to the design steps in the presentation generation phase of Hera. For building (and visualizing) the CM, AM, and PM several Visio solutions were implemented. We chose to build the Hera models using Visio because: (1) Visio is widely used in industry, (2) it provides a graphical interface to build model specific shapes, (3) it is based on a simple programming language, i.e., Visual Basic, which makes it easy for one to define the behavior of the application. A solution is composed of a stencil (which has the model shapes) and a template (which has the load/export feature for the RDF/XML serialization of models). At the current moment the adaptation conditions are not supported by the Visio solutions, the designer has to insert them after a model is exported.

Each model needs to fulfill a set of model constraints. In case that the designer uses the HPG-XSLT interfaces to develop models these constraints are automatically fulfilled as they are enforced during model construction. Nevertheless, in case that the designer uses a different tool to build models, the resulted specifications need to be checked if they fulfill their associated constraints. For this purpose a separate Java program based on Jena [Hewlett-Packard Development Company, LP, 2005] was developed for checking the constraints of a model.

4.2.1 CM Design Interface

HPG-XSLT provides a graphical interface for building CM. Figure 4.1 shows a snapshot of the CM design interface. On the left-hand side there is the stencil that contains shapes for all CM elements. On the right-hand side there is the drawing frame in which a CM is built. For all drawn shapes one can set specific properties to them (e.g., for a concept relationship shape there are attributes that specify the inverse and cardinality of this relationship).

Some of the CM constraints that this interface enforces are: (1) only the media types defined in the media vocabulary can be used for attributes, (2) all concept relationships need to have a domain and a range , (3) every concept relationship needs to have its inverse

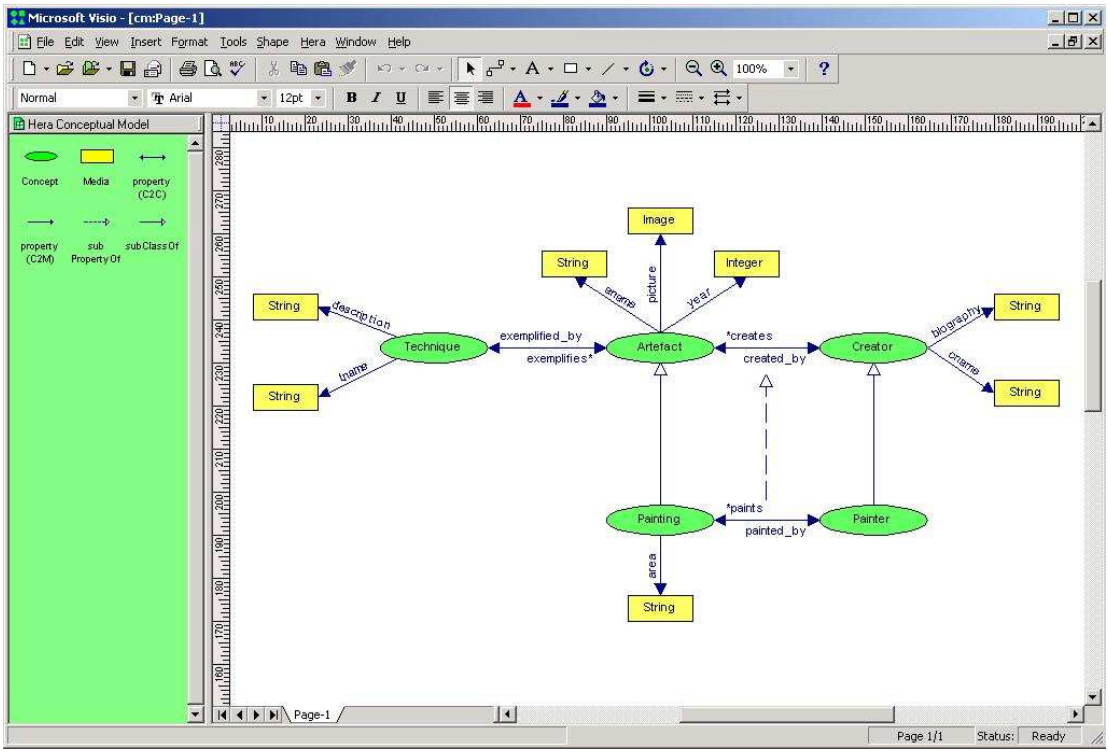


Figure 4.1: Conceptual model interface.

relationship defined, (4) every concept relationship needs to have its cardinality specified, etc.

4.2.2 AM Design Interface

HPG-XSLT provides a graphical interface for building AM. Figure 4.2 shows a snapshot of the AM design interface. On the left-hand side there is the stencil that contains shapes for all AM elements. On the right-hand side there is the drawing frame in which an AM is built. For all drawn shapes one can set specific properties to them (e.g., for a slice shape there is an attribute that specifies the name of the owner concept).

Some of the AM constraints that this interface enforces are: (1) only concepts defined in the associated CM can be used as owners of slices, (2) all region relationships need to have a source and a destination, (3) slices related by slice aggregation relationship need to specify a valid concept relationship between the slice owners, if the owners are different, (4) all slices can be reached from the start main slice, etc.

4.2.3 PM Design Interface

HPG-XSLT provides a graphical interface for building PM. Figure 4.3 shows a snapshot of the PM design interface. On the left-hand side there is the stencil that contains shapes for all PM elements. On the right-hand side there is the drawing frame in which a PM

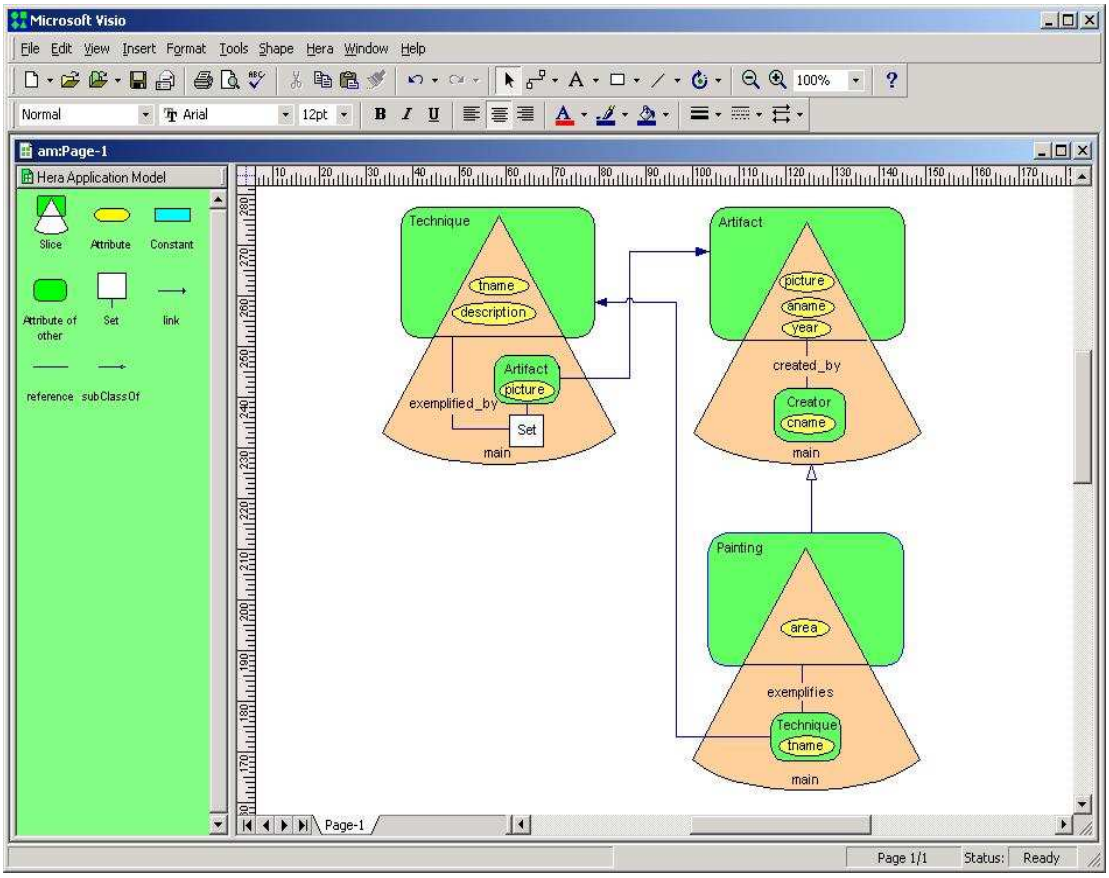


Figure 4.2: Application model interface.

is built. For all drawn shapes one can set specific properties to them (e.g., for a region shape there is an attribute that specifies the name of the owner slice). The layout and style information are associated to a region by means of shape attributes.

Some of the PM constraints that this interface enforces are: (1) only the slices defined in the associated AM can be used as owners of regions, (2) all region relationships need to have a source and a destination, (3) complex slices need to have the layout information specified, (4) all regions need can be reached from the start region, etc.

4.2.4 UP Design Interface

HPG-XSLT supports also the specification of a UP definition and of a UP instantiation. As shown in the previous chapter, the UP is used in the adaptation conditions in AM and PM. Taking this in consideration the UP was split in two parts, one relevant for AM and another part relevant for PM.

The profile definition is a CC/PP vocabulary [Klyne et al., 2004]. It defines three components: *HardwarePlatform*, *SoftwarePlatform*, and *User* (preferences). Each component has a number of attributes for which the types are also specified. Among the supported

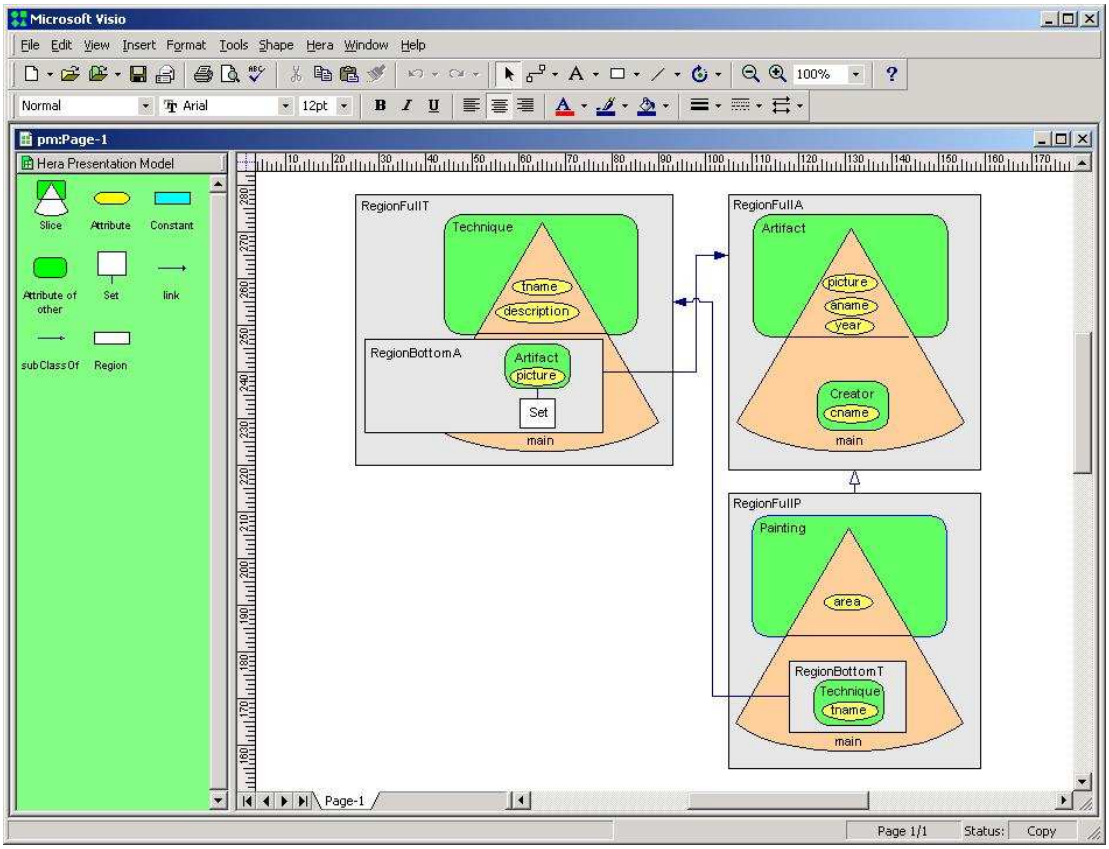


Figure 4.3: Presentation model interface.

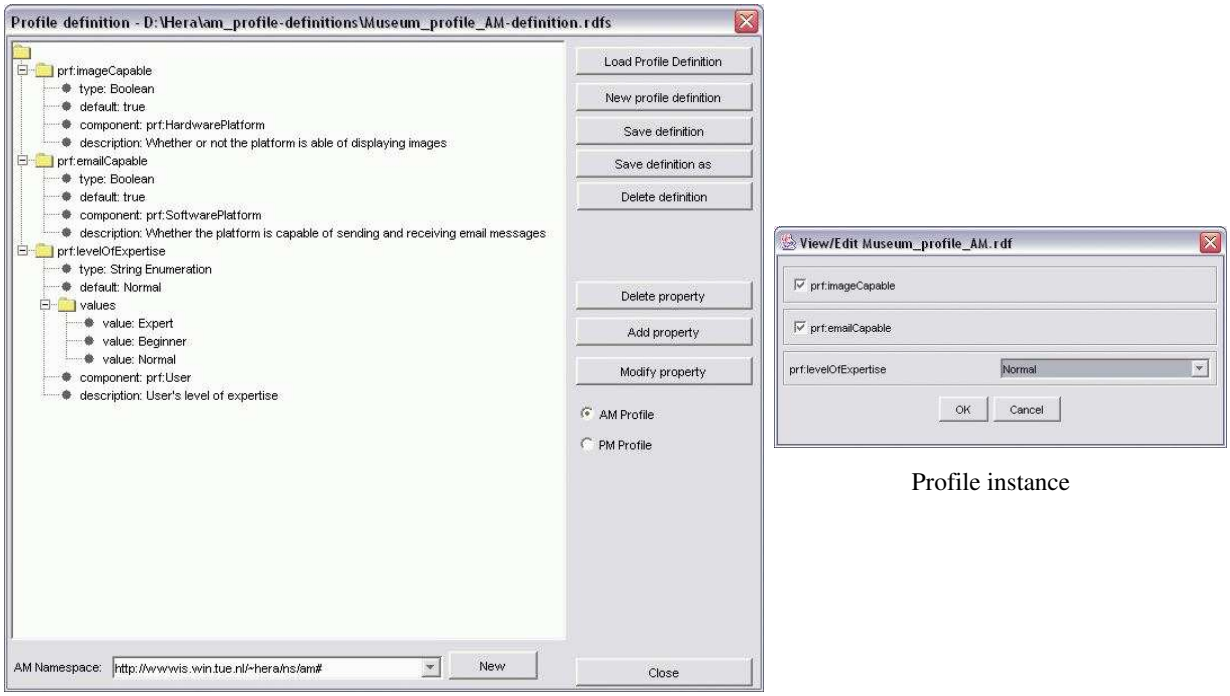
types are *Boolean*, *Integer*, *String*, and *Enumeration*. The instantiation of this profile is done by means of an interface automatically generated from a profile definition.

Figure 4.4 shows the UP definition and instantiation for AM. In this UP, *imageCapable* is defined as a *HardwarePlatform* attribute with an *Boolean* type. For the instantiation of this attribute one can check the associated radio button to indicate the value *True* (otherwise the value is *False*). In the current example this attribute was set to *True*.

Figure 4.5 shows the UP definition and instantiation for PM. In this UP, *client* is defined as a *HardwarePlatform* attribute with an enumerated type *PC*, *PDA*, or *WAP phone*. For the instantiation of this attribute one can select only one of the three values of the attribute type. In the current example this attribute was set to *PC*.

4.2.5 Implementation Interface

Figure 4.6 shows the advanced-designer view in HPG-XSLT. Inexperienced designers will be presented with another interface which follows the popular wizard paradigm (in which the more complex user interface is split into a sequence of smaller, easy-to-use interfaces). As one can notice from Figure 4.6, this advanced view shows two important parts: a left-hand side responsible for converting a CM instance into an AM instance based on the AM



Profile definition

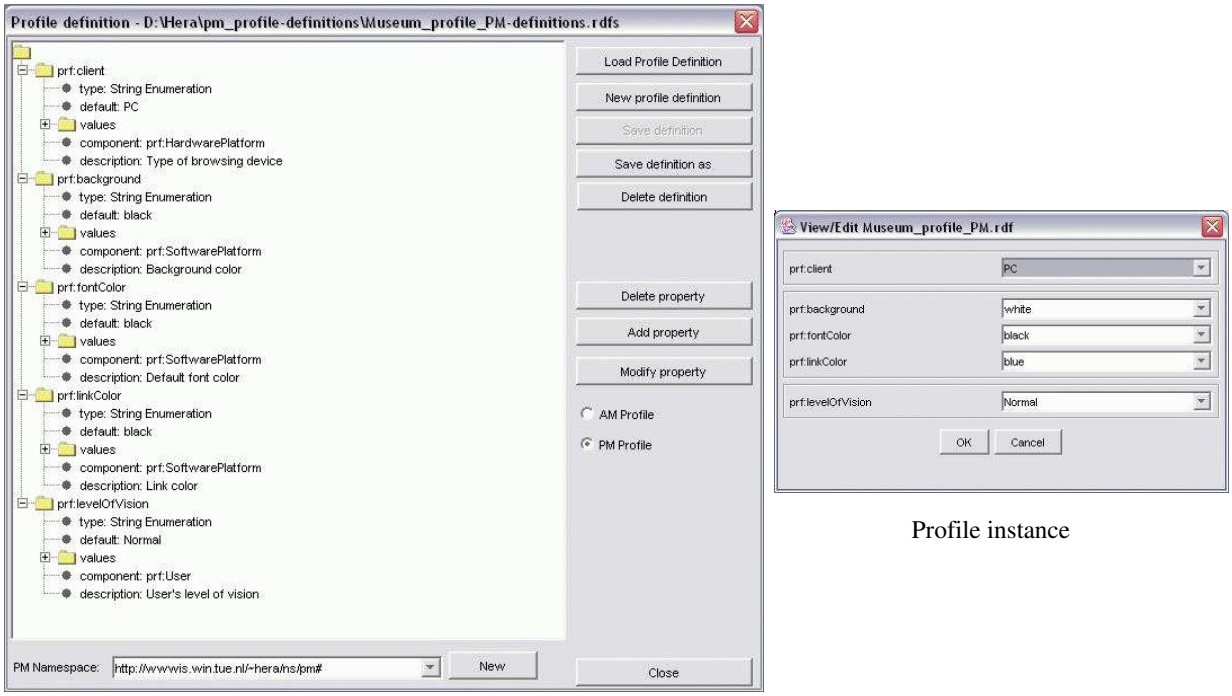
Profile instance

Figure 4.4: The user/platform profile for AM.

and a right-hand side accountable for converting this AM instance into a Web presentation based on a PM.

Each step in this advanced HPG view has associated with it a rectangle labeled with the step's name (e.g., *Conceptual Model*, *Unfolding AM*, *Application Adaptation*, etc.). In each step there are a number of buttons connected with within-step arrows and between-step arrows that express the data flow. Such a button represents a transformation or input/output data depending on the associated label (e.g., *Unfold AM* is a transformation, *Unfolding sheet AM* is an input, and *Unfolded AM* is an output). The arrows that enter into a transformation (left, right, or top) represent the input and the ones that exit from an transformation (bottom) represent the output. The transformation steps that can be triggered at a given moment (all inputs are present) have their buttons enabled while the inhibited transformation steps (not all inputs are present) have their buttons disabled. These visual cues in the advanced view are extremely useful for the understanding and good functioning of the whole transformation process.

All models are represented in RDFS and model instances are represented in RDF; both models and model instances have corresponding RDF/XML serializations. In HPG-XSLT we use XSLT transformations in order to convert one RDF/XML file into another RDF/XML file. The XSLT stylesheet that drives such a transformation process is one of the transformation's inputs. All models and transformation specifications are available for inspection: the *View* button is used to display models, or specific buttons labeled with the



Profile definition

Profile instance

Figure 4.5: The user/platform profile for PM.

name of the model (*AM Instance*, *PM Instance*, etc.) or the name of the transformation (e.g., *Unfolding sheet AM*, *Adaptation sheet AM*, *XML2XSL sheet AM*, etc.) are used to display models or transformations. The basic inputs for HPG are a CM, an AM, a PM, and UP (input specifications), and a CM instance (input data). The output is a Web presentation for the input data that fulfills all the input specifications.

The transformation process starts with the selection of a CM. In case that such a CM doesn't exist the designer can create one using the Visio solution as described in Section 4.2.1. After selecting a CM, the user can choose an AM from the available AMs that correspond to the chosen CM. Again, if such an AM doesn't exist the designer is offered the possibility to build one using the Visio solution presented in Section 4.2.2. The unfolding step is a preparation step in the sense that it restructures the AM in a format more fit (than the original AM) for the next transformation step.

Based on the UP for AM selection, the original AM is adapted. Slices with conditions invalid are discarded and the hyperlinks (slice relationships) referring to these slices are disabled. For example, if the user is not an *Expert* he will not see the painting technique *description*.

All transformations that we have seen so far are generic. Unless otherwise specified a transformation refers to a generic transformation. Based on the adapted AM one can use a generic transformation to produce a specific transformation (*CMI to AMI sheet*) that will convert a CMI to an AMI.

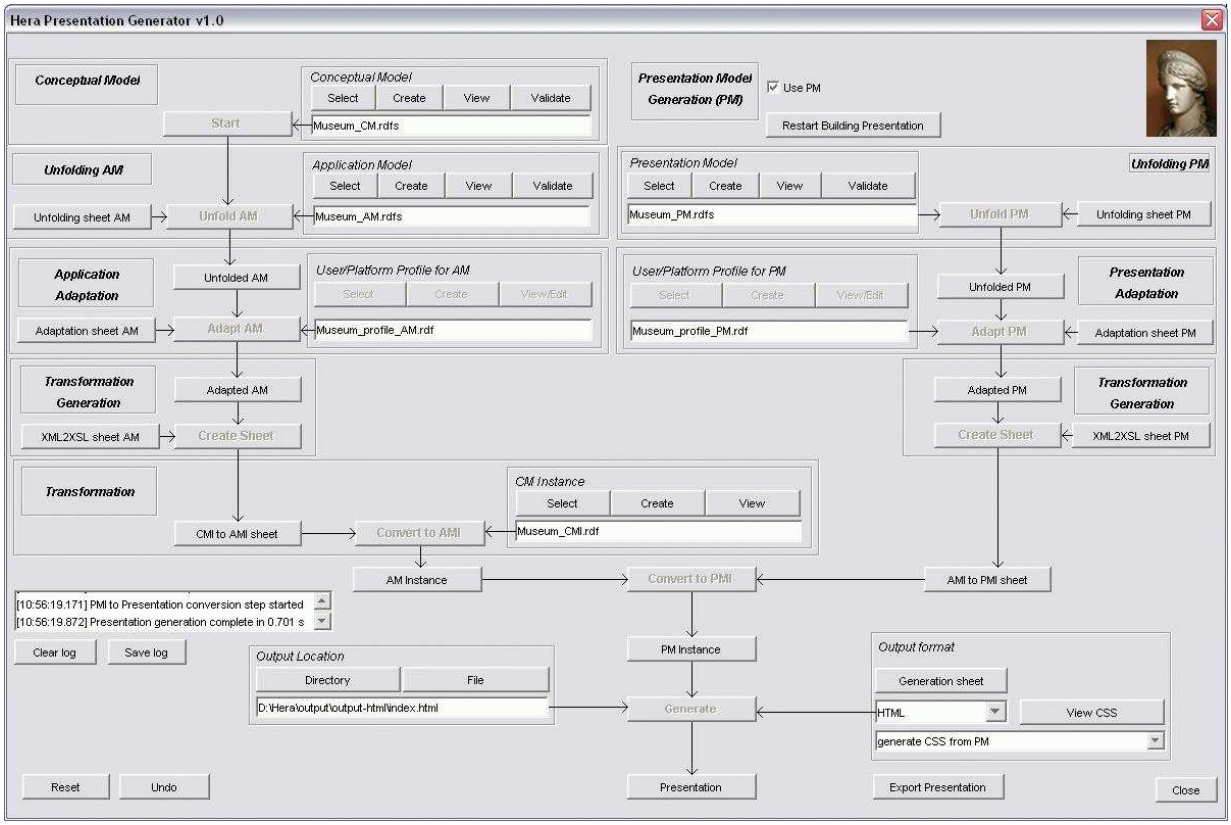


Figure 4.6: The HPG-XSLT interface.

Until now we have presented the transformations at AM level (left-hand side of Figure 4.6). Similar to the above transformations, there are PM-driven transformations (right-hand side of Figure 4.6). Again as a technical convenience, the unfolding mechanism is also used with the PM.

Based on the UP for PM selection, the original PM is adapted. For example, for presenting an index of paintings images one would use a *TableLayout* for a *PC* and a *BoxLayout* for a *PDA* or *WAP phone*. The PM has not only information on the layout but also on the style of the Web presentation. If the user has a *poor* vision, a style with *large* fonts will be used instead of the default style with *medium*-size fonts.

In a similar way as for the AM but this time based on the adapted PM one can use a generic transformation to produce a specific transformation (*AMI to PMI sheet*) that will convert a particular AMI to a PMI.

The last transformation generates code in the format suitable to the user's browser (HTML, HTML+TIME, SMIL, or WML). If the browser supports CSS also a CSS stylesheet is generated according to the style given in the adapted PM. The designer is offered the choice of specifying the directory where the Web presentation will be generated. Note that such a presentation can include thousands of files that might require a lot of disk memory.

Figure 4.7 presents the four different snapshots for four different browsing platforms: HTML for PC, SMIL for PC, HTML for PDA, and WML for WAP phone. These presentations are in accordance with the adaptation with respect to the user browsing device, i.e., the *client* in the UP, as given in the design specifications. Note that, as described in the PM adaptation, the paintings images have a *TableLayout* for HTML on PC, are arranged using *TimeLayout* for SMIL, and a *BoxLayout* (on the vertical axis) for the HTML for PDA and WML presentations. According to the media adaptation, the PDA and the WAP phone use a shorter text version for the painting technique description compared with the one on the PC. Also, the WAP phone presentation doesn't contain pictures. In a similar way, the presentation can be further adapted by considering other attributes from UP, e.g., the level of expertise of the user, the user visual capabilities, etc.

HTML for PC SMIL

HTML for PDA WML

Figure 4.7: Presentations in different browsers.

The first XSLT processor used to carry out the transformations specified by the differ-

ent XSLT stylesheets was Xalan 1.2D02 [Apache Software Foundation, 2004]. Since this processor supported only XSLT 1.0 [Clark, 1999] it was replaced with Saxon [Kay, 2005a], a more powerful XSLT processor that supports XSLT 2.0 [Kay, 2005b]. In order to speed-up the execution of these stylesheets (note that the used museum data has about 1000 art objects with their relations) several XSLT keys have been defined.

4.3 HPG-Java

HPG-Java supports the development of the dynamic variant of the Hera presentation generation phase. The design tools integrated in HPG-XSLT for the CM, AM, PM, and UP building can be used also for HPG-Java. HPG-Java doesn't have a graphical user interface similar to the implementation interface of HPG-XSLT.

One of the disadvantages of HPG-XSLT was the fact that it used XSLT stylesheets to transform RDF models. In this way it was difficult to make use of full semantics of a model as given by the model's RDFS-closure. HPG-Java eliminates this shortcoming by defining Java transformations based on Jena [Hewlett-Packard Development Company, LP, 2005]. For querying and updating models it is used the (Se)RQL implementation of Sesame [Aduna, BV, 2005].

4.3.1 Designing HPG-Java

Being a dynamic system able to react to user actions, HPG-Java was developed based on Java servlet technology. It runs as a Java servlet on an Apache Tomcat Web server. Figure 4.8 shows an excerpt of the class association diagram of HPG-Java.

The main class that receives user requests is the HeraServlet, which extends the Java *HttpServlet* class.

In order to build robust and flexible applications we used several design patterns. A servlet specific pattern is the delegation event model. All request handlers implement the *RequestHandler* interface. The request handlers are registered in the HeraServlet. Based on the value stored in a hidden field for the *GET* request, the HeraServlet is able to identify the particular request handler responsible for this event. Examples of events are login, logout, link following, each one having a corresponding request handler. In this way one avoids the building of complex *RequestHandler*s able to solve all request.

The façade pattern was used to hide from the HeraServlet the complexity of the different data transformations. Four classes were defined to perform data transformations: *AMController*, *SessionUpdater*, *PMController*, and *PresentationConverter*. The *AMController* and *PMController* are responsible for adapting and creating instances of the AM, respectively PM. The *AMController* has associated the *SessionUpdater* which manages the *Session*. The *Session* class is an extension of the Java *HttpSession* class. The *Session* class stores information that persist between user requests: the navigational data model, form models, variables (including the user name and password). *SessionUpdater* is used for updating the *Session* data as part of the process of generating the new slice instance.

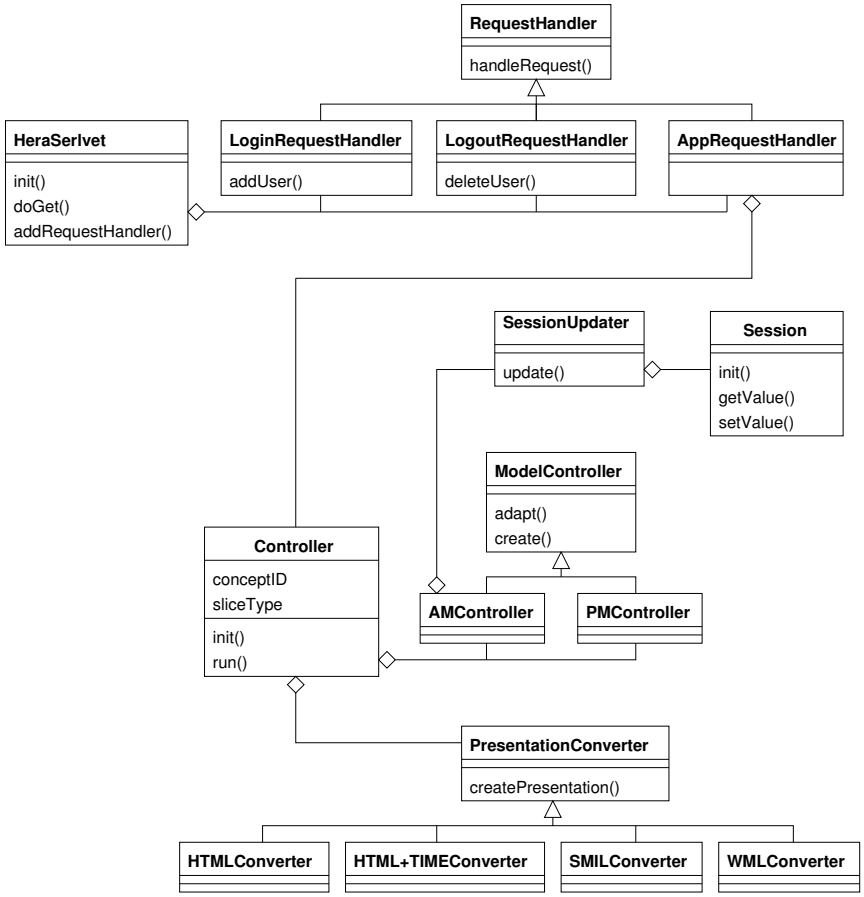


Figure 4.8: HPG-Java class association diagram.

As both slices and regions have a recursive structure we used the composite pattern to define them. In order to build a slice or region the *create()* function is used. Based on the UP the AM and PM will be adapted using the *adapt()* function. Based on the same UP a specific *PresentationGenerator* is chosen. The *createPresentation()* function is used to produce a presentations interpretable by the user browsing platform.

Figure 4.9 presents the exchange of messages between different class instances in response to a user query.

Suppose the *HeraServlet* receives a *doGet()* function call. In case that the request is originating from a link-following request, the *handleRequest()* function of *AppRequestHandler* is called. Next, the Controller *run()* function is called to manage the whole data transformation process.

Depending if the current session is a new session, the AM and PM are adapted by calling the *adapt()* function of their associated controllers. Note that at the present moment we support only static adaptation of the system, that is why the adaptation is performed at the beginning of the user session. With the AM and PM adapted the *create()* function calls will build a new slice instance and a region instance, respectively for the current page. Note that in the *create* function of the *AMController* the *update()* function of the *SessionUpdater*

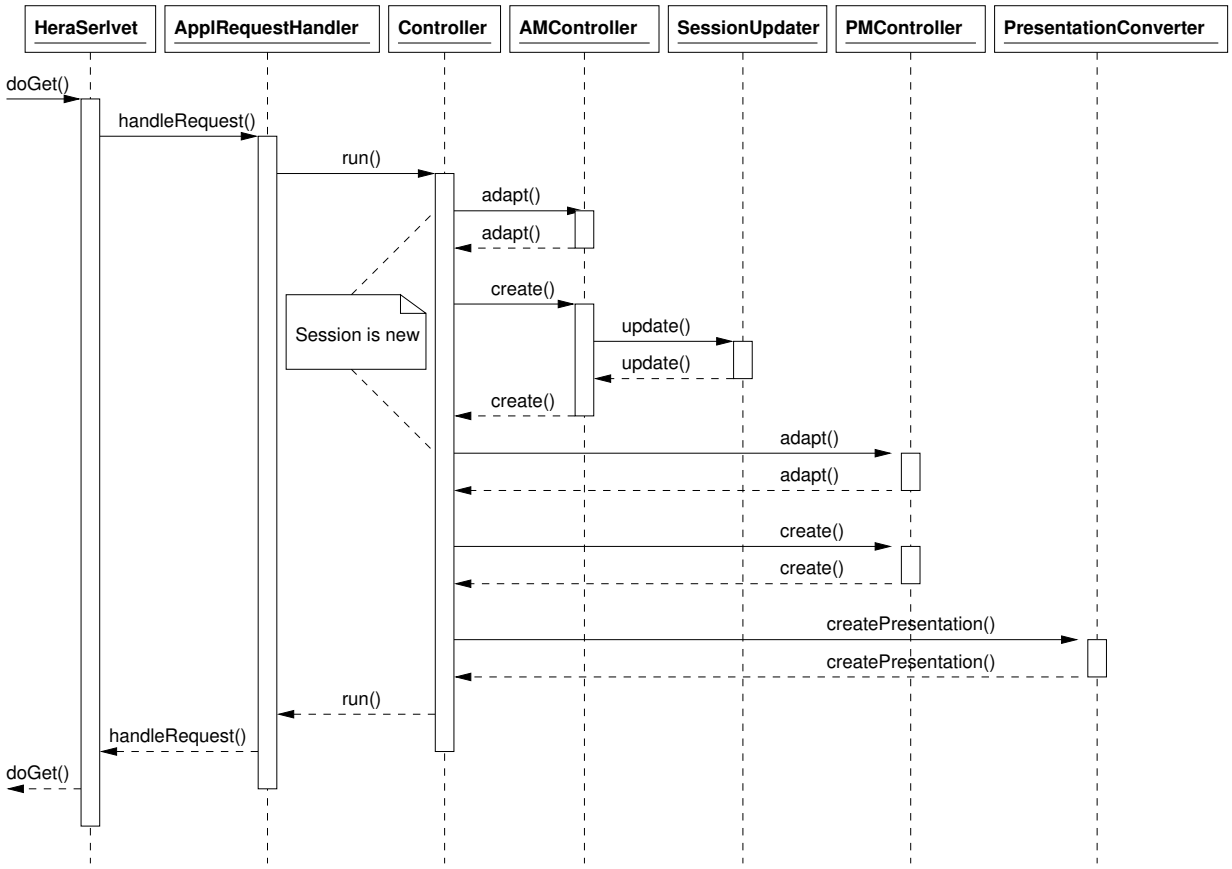


Figure 4.9: HPG-Java message sequence chart.

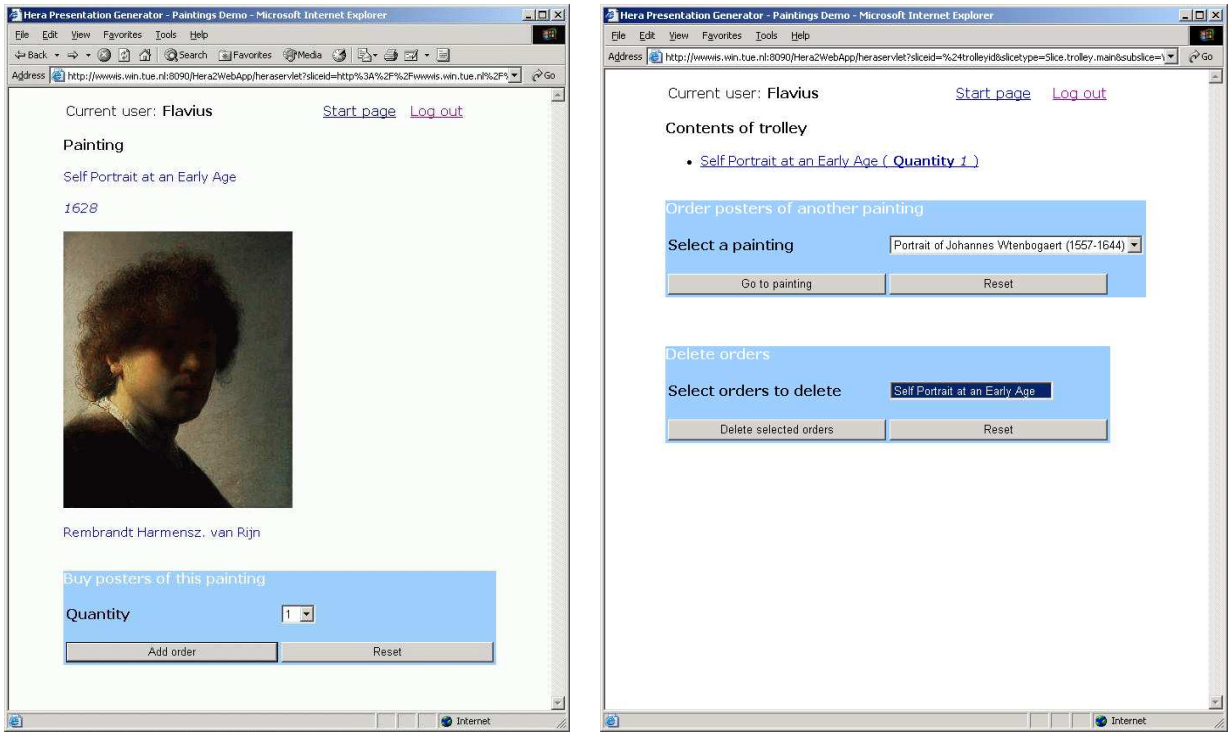
is called in order to update the *Session* content. The region instance is based on the previously generated slice instance. The last function is the *createPresentation()* which transforms the region instance into a Web page.

4.3.2 Using HPG-Java

Several applications were built using HPG-Java: a review system for the Hera papers, a shopping site for vehicles, a portal for a virtual paintings museum (without user interaction), and a shopping site for posters depicting paintings (with user interaction)[1]. We will use the example of the shopping site for posters in order to better illustrate the HPG-Java page generation process. Figure 4.10 (left) shows one page generated with HPG-Java.

Suppose the user wants to buy a poster of the shown painting. Based on such a user request, the next page to be displayed is computed. Let us have a closer look at what happens when the user presses the *Add order* button.

[1]Examples of applications built using HPG-Java are available from `http://wwwis.win.tue.nl:8090/Hera2WebApp`.



Previous slice Current slice

Figure 4.10: Pages generated by HPG-Java.

In order to be able to construct the next slice instance one needs to know the concept instance identifier that owns this slice instance and the slice type of this slice instance. These two elements, the concept instance identifier and the slice type are encoded in the user request.

In our example, there are two queries (attached to the current slice) that are triggered when the user presses the *Add order* button. The first query creates a new order. The second query fills the order properties and adds the order to the trolley. Based on the user request and AM specifications, a new slice instance is produced. This slice shows the list of ordered paintings and contains two forms: one for selecting the next painting and one for deleting an order from the trolley. The current slice instance is converted to a region instance by adding layout and style information to the slice as specified in the PM. In the last step, the region instance is converted to the next page (in this case a HTML page) to be presented to the user. Figure 4.10 (right) shows the next generated page.

4.4 HPG-XSLT vs. HPG-Java

HPG-XSLT and HPG-Java have both their advantages and disadvantages. Figure 4.1 compares the characteristic features of HPG-XSLT and HPG-Java.

	HPG-XSLT		HPG-Java
	generation of full Web presentation		generation of one page at-at-time
+	user interface	−	no user interface
+	deployable on any Web server	−	can be deployable only on Web servers supporting Java servlets
−	no form-support	+	form-support

Table 4.1: HPG-XSLT vs. HPG-Java.

The generation of the full presentation in HPG-XSLT requires usually a long time for computing the whole presentation. If one decides to deploy the resulted pages on a Web server this high computational time does not influence the system response time to a user. The user can browse the presentation at a reasonable speed if his network connection allows it because there is no computation performed on the server. The generation of one page at-a-time in HPG-Java has as consequence a longer response time than for a presentation generated with HPG-XSLT. Nevertheless if the HPG-XSLT presentation is built at run-time, the time needed for computing the whole presentation is higher than the computing of only one page at the beginning of the browsing process.

HPG-XSLT has a user interface that helps the designer for building models. It also allows the generation and execution of the data transformations based on previously defined models. At the current moment HPG-Java does not have such interfaces. It is planned in the future to build a graphical console for HPG-Java which will look similar to the implementation interface of HPG-XSLT.

The resulted Web pages from HPG-XSLT can be deployed on any Web server. Due to its dynamic nature, HPG-Java can be deployed only on Web servers that support Java servlets. Modern Web server (like Apache) do support Java servlets.

HPG-XSLT has no support for user interaction besides simple link-following. The user of a generated presentation cannot influence the content of the presentation. HPG-Java does allow for more advanced forms of user interaction (e.g., forms) as a way to let the user influence the content of the presentation. This is an extremely useful feature if the built application will be used for example, as a shopping site or as a review system.

Performing the data transformations in XSLT or Java have both advantages and disadvantages. Figure 4.2 compares the characteristic features of XSLT transformations and Java transformations used in HPG-XSLT and HPG-Java, respectively.

XSLT is a declarative programming language and Java is an imperative programming language. Depending on the programmer's preference one way of programming can be easier than the other one. As for many declarative languages, XSLT is supported by interpreters. Java is supported by many compilers that generate code interpretable directly by the machine. As such the execution time of the code generated by compilers is smaller than the time required by an interpreter to perform the same computations.

	XSLT stylesheets		Java code
	declarative		imperative
+	loosely-coupled, changing the system can be done by changing one stylesheet	−	strongly-coupled, internal dependencies makes the system harder to change
+	easy to understand	−	more difficult to understand
−	no IDEs	+	many IDEs
−	limited exploitation of models' RDFS semantics	+	full exploitation of models' RDFS semantics
−	XSLT processor offers little support for optimization leading to poorer performance	+	custom software can be optimized better leading to better performance
−	introduces extra steps	+	no need of extra steps

Table 4.2: XSLT stylesheets vs. Java code.

Changing the system can be done by changing only one XSLT stylesheet due to the nice separation of concerns provided by stylesheets. A similar change done for Java code might be harder to achieve due to the internal dependencies between software components. The use of design patterns can alleviate this problem in the Java code.

XSLT stylesheets are easy to learn, one can write fairly complex transformations after learning some of the basic XSLT concepts. Java code is harder to produce, the learning curve is usually higher than for XSLT programming.

At the current moment there is a lack of IDEs to assist the XSLT programmer. For Java, a more mature language, there is a huge number of developing IDEs that provide very advanced debugging facilities.

XSLT is a language for transforming XML documents. As there is no data transformation language for RDF it was decided to use XSLT for transforming the RDF/XML serialization of RDF models. Clearly these XSLT transformations have limitations as they are not able to exploit the full RDFS semantics of a model as given by the model RDF(S)-closure. The Java code is based on Jena and Sesame, two Java libraries that can fully exploit the RDFS semantics of models.

While developing HPG-XSLT it was noticed that there is very little support to optimize a data transformation. By optimization it is seeked the reduction of the time needed by a data transformation. The performance of HPG-XSLT is based on the performance of the data structures used by the XSLT processor. The Java code offers more room for optimizing the data transformations as one can define its own data structures and processing facilities.

HPG-XSLT introduces several extra steps, for unfolding models. These steps were developed in order to prepare an RDF/XML document in an XML format suitable for the next transformation step. These steps were not needed for HPG-Java as the Java code

directly operates on RDF models.

4.5 A Web Service-Oriented Architecture for HPG

HPG integrates several software components that together form one centralized application. In this section it is presented a distributed architecture for HPG in which components are mapped to Web Services (WS). The loosely coupled Hera WSs realize the plug-and-play software vision in the context of SWISs. For example, it is possible to generate a SWIS by composing a WS which provides up-to-date data, a WS that knows how to present this data, and a WS that is able to perform adaptation of the presentation based on user preferences and device capabilities.

It was decided for a WS solution for realizing the distributed Hera architecture because WSs have clear advantages compared to their predecessors CORBA, J2EE, and DCOM [O'Toole, 2003]. First of all WSs are based on the XML document paradigm, a human readable language that abstracts from the implementation details. WS interfaces are specified in a universally accepted Web Service Description Language (WSDL) [Christensen et al., 2001], an XML-based language. Last but not least, Web services use the popular HTTP protocol as the carrier of exchanged messages.

In the rest of this section we will mainly focus on HPG-XSLT, but one can implement similar services for HPG-Java that produces a single Web page at-a-time instead of a full Web presentation as is the case for HPG-XSLT. For this reason we will not distinguish between the two HPGs and we will refer to them with only one term, i.e., HPG.

Figure 4.11 presents a Web service-oriented architecture (WSOA) for HPG based on two WSs: the Data Service and the Presentation Service. The Data Service is responsible for delivering up-to-date data for which the Presentation Service will make a hypermedia presentation. A Client placed at a Web Server location will orchestrate the communication with the two services. The proposed WSOA has a star topology, with the Client in the middle. The communication between Client and services is done at SOAP [Box et al., 2000] level which resides on top of HTTP while the communication between the Web Browser and the Web Server is done in plain HTTP.

First the Client asks the Data Service to provide the data. Once the data is received it is passed to the Presentation Service. After receiving the data, the Presentation Service constructs a hypermedia presentation which is passed back to the Client. The Web Server that hosts the Client uses this presentation in providing pages to the Web Browser. The underlying assumption here is that the Data Service and the Presentation Service share the same CM.

Note that a service-based solution provides a lot of flexibility for such a system. Different Presentation Services (with different AMs and PMs) can be plugged into the system to produce different presentations for the same data. Moreover one Presentation Service may produce an HTML presentation, while another one can provide a WML presentation, ensuring thus the ubiquity of the built SWIS. Also different Data Services can be used in the same manner (assuming the fact that they agree with the Presentation Service on the

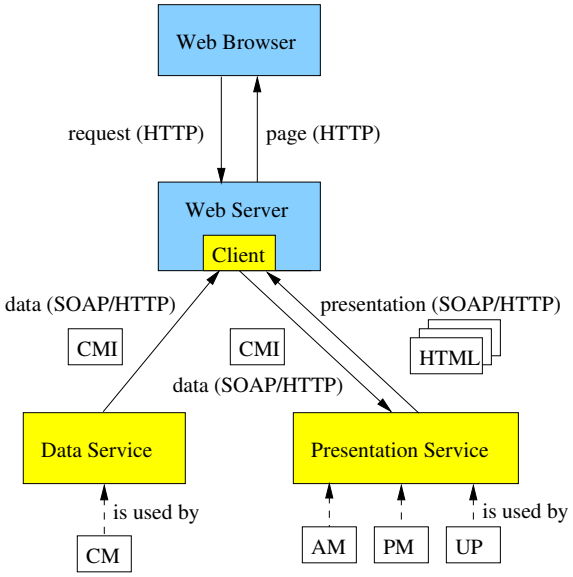


Figure 4.11: Web service-oriented architecture.

CM). In this way data that comes from two different sources but sharing the same domain can benefit from presentation capabilities from the same Presentation Service.

4.5.1 Web Service Descriptions

The interface of a WS is given in a WSDL specification. In addition to the service interface, such specifications give also the data types used by the service messages and the location of the service. In this subsection it is described only the interface as we only use existing XML Schema Datatypes [Thompson et al., 2001; Biron and Malhotra, 2001] (we did not need to define our own types) and the service location can be defined anywhere on the Web.

Figure 4.12 depicts an excerpt from the Data Service WSDL specification. First it specifies which are the messages, their embeddings, and the type of data that messages will carry. The `getDataRequest` message is an empty message. This message is used just to trigger the response from the service. The `getDataResponse` message has a `<wsdl:part>` containing the requested data string. The `<wsdl:portType>` associates the request and response messages with the `getData` operation of the Data Service. It also specifies the type of the message as input or as output for the operation. The data string returned is the CMI that the Data Service holds.

Figure 4.13 depicts an excerpt from the Presentation Service WSDL specification. The request message `getPresentationRequest` has a `<wsd:part>` named `in0` (this is the default naming convention used by our SOAP server for the operation arguments) containing one string. The response message `getPresentationResponse` will contain also a string. The `<wsdl:portType>` associates the request and response messages with the

```
    <wsdl:message name="getDataRequest">
    </wsdl:message>
    <wsdl:message name="getDataResponse">
         <wsdl:part name="getDataReturn"
                    type="xsd:string"/>
    </wsdl:message>
    <wsdl:portType name="DataService">
         <wsdl:operation name="getData">
              <wsdl:input message="impl:getDataRequest"
                          name="getDataRequest"/>
              <wsdl:output message="impl:getDataResponse"
                           name="getDataResponse"/>
         </wsdl:operation>
    </wsdl:portType>
```

Figure 4.12: Excerpt from Data Service WSDL.

`getPresentation` operation of the Presentation Service. As for the Data Service, it also specifies the type of the message as input or as output for the operation. The input data string is the CMI and the data string returned from the operation is the encoded (in one string) presentation.

```
<wsdl:message name="getPresentationRequest">
    <wsdl:part name="in0"
               type="xsd:string"/>
</wsdl:message>
<wsdl:message name="getPresentationResponse">
    <wsdl:part name="getPresentationReturn"
               type="xsd:string"/>
</wsdl:message>
<wsdl:portType name="PresentationService">
    <wsdl:operation name="getPresentation"
                    parameterOrder="in0">
        <wsdl:input message="impl:getPresentationRequest"
                    name="getPresentationRequest"/>
        <wsdl:output message="impl:getPresentationResponse"
                     name="getPresentationResponse"/>
    </wsdl:operation>
</wsdl:portType>
```

Figure 4.13: Excerpt from Presentation Service WSDL.

4.5.2 SOAP Messages

After we have defined the service interface we can have now a closer look at the actual representation of the service messages. Despite its name the Simple Object Access Protocol

(SOAP) is not a classic (communication) protocol. It is rather a one-way message exchange paradigm (or some others prefer to say a lightweight protocol to exchange information in a distributed system). A SOAP message is an XML message containing a SOAP envelope. A SOAP envelope has an optional SOAP header and a required SOAP body. It is the SOAP body that contains the data carried in a message. The current implementation uses SOAP RPC which means that all message communication is done synchronously.

Figure 4.14 presents a snapshot of Client-services communication, namely the SOAP messages exchanged between the Client and the Data Service. The `SOAP Request` window displays the `getDataRequest message`, an empty message as we already saw from the interface description. The `SOAP Response` window shows the `getDataResponse` message containing in its `getDataReturn` part an actual CMI. Note that `<`, `>` are escaped as we encoded the data as a string.

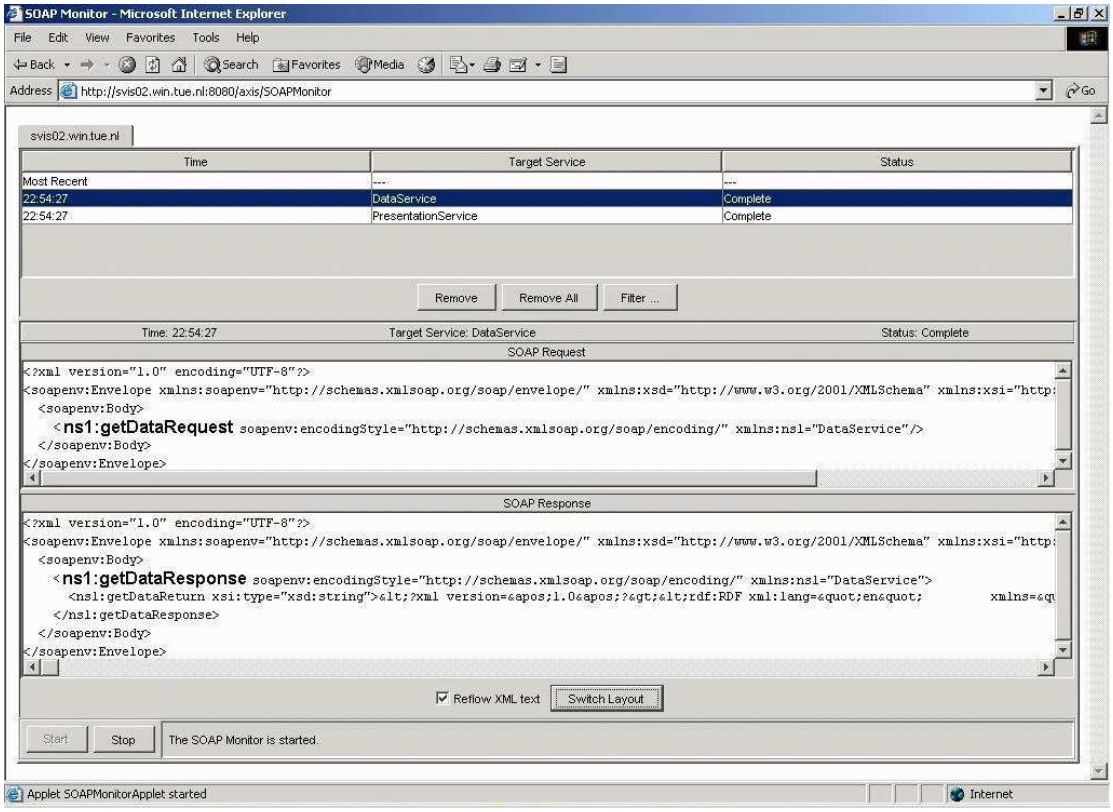


Figure 4.14: SOAP messages.

4.5.3 Tools

In order to experiment with the proposed architecture a Java-based Hera tool was developed. Tomcat 4.1 [Apache Software Foundation, 2005a] was used as the Web server that supports servlets. On this Web server we installed Axis 1.1 (Apache eXtensible Interaction

System) [Apache Software Foundation, 2005b], a SOAP 1.1 engine. By SOAP engine we mean a tool that supports both a SOAP server and SOAP clients. We did deploy on the SOAP server two services Data Service and Presentation Service. For their deployment we used appropriate Axis Web Service Deployment Descriptors. The SOAP Client that communicates with these services was installed on the Web server, outside the SOAP server. The WSDL specifications were generated by the Java2WSDL emitter. Both Java2WSDL and the SOAP Monitor are part of the Axis distribution kit. Tomcat and Axis are Java-based and freely available from the Apache Software Foundation. The services and the client were written in Java. All software is running on the Java 1.4 platform.

It is important to notice that when developing WSs with Axis, the programmer doesn't need to bother about making WSDL interfaces or the actual encoding of the SOAP messages. All these will be automatically done by the system. Making all WS details transparent to the programmer enables him to focus only on the application logic implementation in Java and makes thus the system less error-prone.

4.5.4 Adaptation in HPG Web Service-Oriented Architecture

Figure 4.15 presents a WSOA based on four services: Data Service, Presentation Service, Profile Service, and Adaptation Service. Note that this architecture doesn't have a star topology as the Client only communicates with three of the four services. In order to denote the order in which the messages will be passed we added a label to the continuous arrows. This label should be read in the increasing number order or alphabetical order. The two sequences (1, 2, 3) and (a, b) can be done in parallel. Step 4 is performed after completion of steps 3 and b.

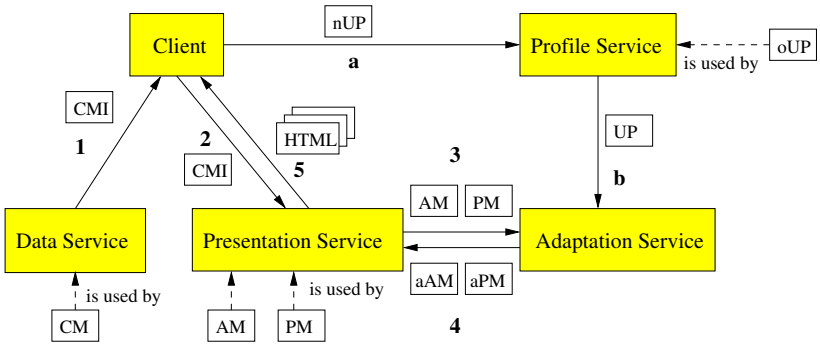


Figure 4.15: Extended Web service-oriented architecture.

The steps 1, 2, and 5 were already discussed in the beginning of this section. In step a the nUP ('n' stands for new) provided by the Client is sent to the Profile Service. The Profile Service can be viewed as a shared memory service for user profiles. By a shared memory service we simulate shared memory between services using one service. The profile attributes that are not defined in the nUP will be taken from the oUP ('o' stands for old), the old profile of the user, and result in a merged UP. In the request message sent to the

Profile Service the user may also specify if an update of the oUP with the UP should be done. The UP is sent to the Adaptation Service. Also, the Adaptation Service receives the AM and PM from the Presentation Service. After receiving these two messages (3 and b) the Adaptation Service computes the aAM and aPM ('a' stands for adapted) and sends them to the Presentation Service. In this WSOA the Presentation Service will use the aAM, instead of the AM, and the PM, instead of the aPM, to compute the presentation.

4.6 Conclusions

The Hera tool suite aims at supporting the design of SWISs using the Hera methodology. There are two versions of the software tools for the presentation generation in Hera: HPG-XSLT and HPG-Java. The XSLT stylesheets from HPG-XSLT are replaced in HPG-Java with Java code able to better cope with the RDF(S) semantics of the Hera models. Compared with HPG-XSLT, HPG-Java extends the functionality of a generated WIS with user interaction support. Nevertheless, HPG-Java lost the declarativity, simplicity, and reuse capabilities of the XSLT transformation templates. A declarative transformation language dedicated to RDF(S), which to our knowledge doesn't exist at the present moment, would probably be the best option for Hera transformations.

HPG has also a distributed architecture based on WS. We chose for a WS-oriented solution due to the popularity and easy-to-implement features of WSs. In this way WISs can be seamlessly built by composing appropriate WSs. The Axis distribution kit proved to be a very flexible set of tools to support WS development, deployment, and monitoring. We have also shown examples of WSDL specifications used to describe WSs and of SOAP messages exchanged with WSs.

As future work, a user interface (servlet console) will be developed for HPG-Java, very similar to the user interface in HPG-XSLT, in order to better trace and configure the Hera servlet activities. In both HPGs, the user interfaces for designing the CM, AM, and PM will be extended with adaptation specification support (for the appearance conditions). Depending on the future existence of a declarative RDF transformation language the Java code will be replaced with RDF transformation templates which will combine the best features of the two HPG versions: declarativity, simplicity, and reuse of templates as in HPG-XSLT, and the full RDF(S) semantics exploitation for the Hera models as in HPG-Java.

We would also like to extend the Web service-oriented architecture of the HPG with new services like a data query service, a data integration service, or a service able to generate adaptive hypermedia presentations.

Chapter 5

Query Optimization in Hera

While RDF and RDFS are widely acknowledged as a standard means for describing Web metadata, a standardized language for querying RDF metadata is still an open issue. Research groups coming both from industry and academia are presently involved in proposing several RDF query languages. Due to the lack of an RDF algebra such query languages use APIs to describe their semantics and optimization issues are mostly neglected. This chapter proposes RAL (an RDF algebra) as a reference mathematical study for RDF query languages and for performing RDF query optimization. We define the data model, we present the operators to manipulate the data, and we address the application of RAL for query optimization. RAL includes: extraction operators to retrieve the needed resources from the input RDF model, loop operators to support repetition, and construction operators to build the resulting RDF model.

5.1 Introduction

The Resource Description Framework (RDF) [Lassila and Swick, 1999; Brickley and Guha, 2004] is intended to serve as a metadata language for the Web and together with its extensions lays a foundation for the Semantic Web. It has a graph notation, which can be serialized in a triple notation (subject, predicate, object) or in an XML syntax [Beckett, 2004].

Compared to XML, which is document-oriented, RDF takes into consideration a knowledge oriented approach that is designed specifically for the Web and that is extremely useful for the Semantic Web. One of the advantages of RDF over XML is that an RDF graph depicts in a unique form the information to be conveyed while there are several XML documents to represent the same semantic graph. The central concept that RDF uses in modeling the metadata is that of resource: resources act as the objects or entities that are considered in the metadata. RDF's purpose to express metadata is met by its ability to define statements that assign values to properties of resources. In this way RDF expressions describe how resources are related to each other and to (concrete) values.

Object-oriented systems are object-centric in the sense that properties are defined in a class context. On the contrary, RDF is property-centric, which makes it easy for anyone to "say anything about anything" [Berners-Lee, 1998], one of the architecture principles of the Semantic Web. In RDF, concepts from E-R modeling are being reused for the modeling of Web ontologies. The concept of ontology is used to express a common understanding of resources that allows application interoperability [Decker et al., 2000]: identifying a common structure of resources supports the uniform understanding and treatment of metadata.

The language of RDF is composed from different parts. RDF Schema (RDFS) [Brickley and Guha, 2004] can be used to define application specific vocabularies. These vocabularies define taxonomies of resources and properties such that they subsequently can be used by specific RDF descriptions. RDFS is designed as a flexible language to support distributed description models. Unlike XML DTD or XML Schema [Thompson et al., 2001; Biron and Malhotra, 2001], RDFS does not impose a strict typing on descriptions: for example, one can use new properties that were not present in the schema, a resource can be an instance of more than one class, etc. The set of primitive data types in RDF is left on purpose poorly defined as RDFS reuses the work done for data typing in XML Schema [Klyne and Carroll, 2004]. We do hope that future versions of RDFS will bring clarification regarding RDF shortcomings of the present specification (e.g., missing set collection, difficult literals handling, etc.).

In order to use metadata for application interoperability it is not sufficient to just have a language to describe the metadata. A language for describing queries on that data is also needed. In the XML world there is already a winner in the quest for the most appropriate XML query language, i.e., XQuery [Boag et al., 2005]. As the Semantic Web initiative started recently, its supporting technologies are still in their infancy. Research groups coming from both industry and academia are presently involved in proposing several RDF query languages (see the next section). We observe that such query languages often use APIs to describe their semantics. Clearly, for a proper understanding and a sound theoretical foundation of these query languages there is a lack of an algebra in the spirit of the one we know from the relational model. As we also observe that optimization issues are mostly neglected, an algebra for RDF could help to build a platform for finding efficient rewritings of queries. This chapter identifies this need and proposes RAL, an RDF algebra suitable for defining (and comparing) the semantics of different RDF query languages and (at a later stage) for performing algebraic optimizations.

The remainder of this chapter begins with discussing the related work on RDF query languages. In Section 5.3 the definition of RAL starts by considering its data model. Section 5.4 presents the definition of the basic operators of the algebra, while some additional algebra features are presented in the next section. Section 5.5 also shows how the algebra can be used to express queries from other query languages like RQL. Section 5.6 discusses RAL equivalence laws and their application for query optimization. Section 5.7 concludes the chapter and indicates further research.

5.2 Related Work

In the previous section we addressed the role of an algebra for the definition and comparison of query languages and for query optimization. At present, there already exist a few RDF query languages but to our knowledge there is no full-fledged RDF algebra. The only algebraic description of RDF that we encountered so far is the RDF data model specification from Stanford [Melnik, 1999]. This specification is based on triples and it provides a formal definition of resources, literals, and statements. Despite being nicely defined, the specification does not include URIs, neglects the RDF graph structure, and does not provide operations for manipulating RDF models. Another formal approach, which aims not only at formalizing the RDF data model but also at associating a formal semantics to it, is the RDF Semantics (RS) [Hayes, 2004]. However, it does not qualify as an algebraic approach but rather, as a model-theoretic one. As RS is currently being considered a main reference when it comes to RDF semantics, we tried to make our algebra (especially the data model part) compatible with RS.

As implementation of RDF toolkits started before having an RDF query language, there are a lot of RDF APIs present today. Three main approaches for querying RDF (meta)data have been proposed.

The first approach (supported in the W3C working group by Stanford) is to view RDF data as a knowledge base of triples. Triple [Sintek and Decker, 2002], the successor of SiLRI (Simple Logic-based RDF Interpreter) [Decker et al., 1998], maps RDF metadata to a knowledge base in Horn Logic (replacing Frame Logic). A similar approach is taken in Metalog [Marchiori and Saarela, 1998], which matches triples to predicates in Datalog, a subset of Horn Logic. In this way one can query RDF descriptions at a high level of abstraction: the querying takes place at a logical layer that supports inference [Guha et al., 1998].

The second approach (proposed by IBM) builds upon the XML serialization of RDF. In the "RDF for XML" project (recently removed), an RDF API is proposed on top of the IBM AlphaWork's XML 4 Java parser. In the context of the same project a declarative query language for RDF (RDF Query) [Malhotra and Sundaresan, 1998] was created for which both input and output are resource containers. One of the nice features of this query language is that it proposes operators similar to the relational algebra, leaving the possibility to reuse some of the 25 years experience with relational databases. Unfortunately, the language fails to include the inference rules specific to RDF Schema, loosing description semantics.

Stefan Kokkelink goes even further with the second approach proposing RDF query and transformation languages that extend existing XML technologies. Similarly to XPath, he defines RDFPath [Kokkelink, 2001] for locating information in an RDF graph. The location step and the filter constructs were present also in XPath, but the primary selection construct is new in this language. With the RDF graph being a forest, one needs to specify from which trees the selection will be made. RDFT is an RDF declarative transformation language a la XSLT [Kay, 2005b], while RQuery, an RDF query language, is obtained by replacing XPath [Berglund et al., 2005] with RDFPath in XQuery [Boag et al., 2005].

However, this approach is not using the features specific for RDF, as the RDF Schema is being completely neglected.

The third approach (coming from ICS-FORTH in Greece) uses the RDF Graph Model for defining the RDF query language RQL [Karvounarakis et al., 2002]. It extends previous work on semistructured query languages (e.g., path expressions, filtering capabilities, etc.) [Catell et al., 2000] with RDF peculiarities. Its strength lies in the ability to uniformly query both RDF descriptions and schemas. Compared to the previous approach it exploits the inference given in the RDF Schema (e.g., multiple classification of resources, taxonomies of classes and properties, etc.) making it the most advanced RDF query language proposed so far.

Other query languages for RDF have been proposed during the last years: we name Algae [Prud'hommeaux, 2002] (W3C) and rdfDB Query Language [Guha, 2000] (Netscape) as graph matching query languages. RDF query languages similar to rdfDB Query Language are: RDFQL [Intellidimension Inc, 2002], David Allsop's RDF query language [Allsopp et al., 2002], SquishQL [Miller, 2002], and RDQL [Seaborne, 2001] (HP Labs) an implementation of SquishQL on top of the Jena RDF API [McBride, 2001] (HP Labs). Some other proposed RDF APIs are: Wilbur [Lassila, 2001] (Nokia), the RDF API from Stanford [Melnik, 2001], and Redland [Beckett, 2003]. DAML Query Language (DQL), a query language for ontology knowledge expressed in DAML+OIL [Connolly et al., 2001] (built on top of RDF), is currently under development.

We mention one characteristic aspect of all the languages. The proposed approaches disregard the (re)construction of the output: they leave the output as a "flat" RDF container of input resources. The focus is on the extraction of the proper resources for the given query, not on building a new RDF data structure. For the purpose of an RDF algebra we need to take into account also the construction part: deriving from the input data structure a new RDF data structure as the consequence of the query implies that the resulting RDF graph can contain new vertices and edges not present in the original RDF graph. To express RDF queries both the extraction and construction parts should be covered. The optimization of queries can be achieved not only in the extraction part, finding efficient ways of extracting the relevant resources, but also in the construction part when the actual output is produced.

5.3 Data Model

In this section we discuss the data model used with our algebra. We describe how the RDF data structures are represented in the input or output of the expressions formulated in RAL. We start by considering the concept of RDF model.

5.3.1 RDF Model

An RDF model is similar to a directed labeled graph (DLG) [Lassila and Swick, 1999]. However, it differs from a classical DLG since its definition allows for multiple edges between

two nodes. It also differs from a multigraph because the different edges between two nodes are not allowed to share the same label. The graph does not necessarily have to be connected and it is allowed to contain cycles.

The nodes in the graph are used to represent resources or literals. Literals (strings) are used to denote content that is not processed further by the RDF processor. The nodes that represent resources can be further classified as nodes representing URI references or blank nodes. URI references are used as universal identifiers in RDF. Each blank node, also called an anonymous resource, is considered to be unique in the graph despite the fact that it has no (explicit) label associated to it. The non-blank nodes are (explicitly) labeled with resource identifiers (URIs) or string values. The edges in the graph represent properties. These edges are labeled by property names. Edges between different pairs of nodes may share the same label and the same property can be applied repetitively on a certain resource. This RDF feature enables multiple classification of resources, multiple inheritance for classes, and multiple domains/ranges for properties. Both resources and properties are first class citizens in the proposed RDF data model.

We identify the following sets: R (set of resources), U (set of URI references), B (set of blank nodes), L (set of literals), and P (set of properties). At RDF level the following holds for these sets: $R = U \cup B$, $rdf{:}Property \in U$, $P \subset R$, $rdf{:}type \in P$, and U, B, and L are pair-wise disjoint.

The property $rdf{:}type$ defines the type of a particular resource instance. At RDF level any resource can be the target of an $rdf{:}type$ property. RDF supports multiple classification of resources, because $rdf{:}type$ (as any other property) can be repeated on a particular resource.

Definition 1 *An RDF model M is a finite set of triples (also called statements)*

$$M \subset R \times U \times (R \cup L)$$

Each triple or statement in an RDF model contains a resource, a URI reference (which stands for a property), and a resource or literal.

Definition 2 *The set of properties of an RDF model M is*

$$P = \{\ p \mid (s, p, o) \in M \vee (p, rdf{:}type, rdf{:}Property) \in M\}$$

The properties in an RDF model are the middle element of a triple in the model, or they are a resource with an $rdf{:}type$ property to the $rdf{:}Property$ resource.

Definition 3 *Formally the data model (graph model) corresponding to an RDF model M is*

$$\begin{aligned} G &= (N, E, l_N, l_E) \\ l_N &= N \rightarrow R \cup L \\ l_E &= E \rightarrow P \end{aligned}$$

using the following construction mechanism (N and E denote the nodes and edges, l_N and l_E their labels). For each $(s, p, o) \in M$, add nodes n_s, n_o to N (different only if $s \neq o$) and label them as $l_N(n_s) = s$, $l_N(n_o) = o$, and add e_p to E as a directed edge between n_s and n_o and label that as $l_E(e_p) = p$. In the case that s and/or o are in B, then $l_N(n_s)$ and/or $l_N(n_o)$ are not defined: blank nodes do not have labels.

The function $l_N(.)$ is an injective partial function, while $l_E(.)$ is a (possibly non-injective) total function: nodes that have a label have a unique one, edges always have a label but can share it with other edges.

We use quotes for strings that represent literal nodes to make a syntactical distinction between them and URI nodes. A URI can be expressed using qualified names (e.g., *s*:*Painting*) or in absolute form (e.g., *http://example.com/schema#Painting*). Blank nodes do not have a proper identifier which implies that they can be queried only through a property related to them. Compared to XML, which defines an order between subelements, in RDF the properties of a resource are unordered unless they represent items in a sequence container. We remark that not having the burden of preserving element order eases the definition of algebra operators and their associated laws.

5.3.2 Nodes and Edges

As we describe in Table 5.1 each node has three basic properties. The *id* of a node represents the (identification) label associated to it. The nodes from the subset of resources that represent the blank nodes do not have an *id* associated to them. There are two *type*s of nodes: *rdfs*:*Resource* and *rdfs*:*Literal*. The *nodeID* gives the unique internal identifier of each node in the graph. *nodeID* has the same value as *id* for the nodes that have a label, but in addition it gives a unique identifier to the blank nodes. The internal identifier *nodeID* is not available for external use, i.e., it is not disclosed for querying.

Basic property	Result for resource $u \in U$	Result for literal $l \in L$
id	$l_N(u)$	$l_N(l)$
type	*rdfs*:*Resource*	*rdfs*:*Literal*
nodeID	internal ID	internal ID

Table 5.1: Basic properties for nodes.

Each edge has three basic properties as described in Table 5.2. Compared with nodes, which have unique identifiers, edges have a *name* (label), which may be not unique. There can be several edges sharing the same *name* but connecting different pairs of vertices. The *name* of an edge is (lexically) identified with the *id* of the resource corresponding to the property associated with the edge. The *subject* of an edge gives the resource node from which the edge is starting. *object* returns the resource or literal node where the edge ends, i.e., the value of the property.

Basic property	Result for edge e from $r \in R$ to $o \in R \cup L$
name	$l_E(e)$
subject	r
object	o

Table 5.2: Basic properties for edges.

Definition 4 *Two non-blank nodes are considered to be equal if they have the same id. Two blank nodes are considered to be equal if they have the same (RDF) properties and the corresponding (RDF) property values are equal (in case of loops, pairs of blank nodes already visited are not further tested for equality).*

All non-blank nodes that are considered equal are internally mapped into one node in the graph.

Definition 5 *Two graphs are considered to be equal if they differ only by re-naming the nodeIDs of their blank nodes.*

Note that two graphs for which all their nodes are equal (in terms of node equality) may be not equal themselves (in terms of graph equality) if some corresponding non-blank nodes have different properties and/or different property values.

5.3.3 RDFS

RDF Schema (RDFS) [Brickley and Guha, 2004] provides a richer modeling language on top of RDF. RDFS adds new modeling primitives by introducing RDF resources that have additional semantics (in the previous section we already mentioned $rdfs$:*Resource* and $rdfs$:*Literal*). If one chooses to discard this special semantics, RDFS models can be viewed as (plain) RDF models.

The RDFS type system is built using the following primitives: $rdfs$:*Resource*, rdf:*type*, rdf:*Property*, $rdfs$:*Class*, $rdfs$:*Literal*, $rdfs$:*subClassOf*, $rdfs$:*subPropertyOf*, $rdfs$:*domain*, and $rdfs$:*range*. The distinction between *rdf* and *rdfs* namespaces to be used for different resources is more due to historical reasons (RDF was developed before RDFS) than due to semantical ones. Figure 5.1 depicts graphically these RDF/RDFS primitives.

The inheritance mechanism incorporated in RDFS supports taxonomies at class level (using the $rdfs$:*subClassOf* property) and at property level (using the $rdfs$:*subPropertyOf* property). It also defines constraints: names to be used for properties, domain and range for properties, etc. These constraints need to be fulfilled by RDF descriptions (later on called instances) in order to validate these instances according to the associated schema.

Every resource that has the rdf:*type* property equal to $rdfs$:*Class* represents a type (or class) in the RDF(S) type system. Types can be classified as primitive types ($rdfs$:*Resource*,

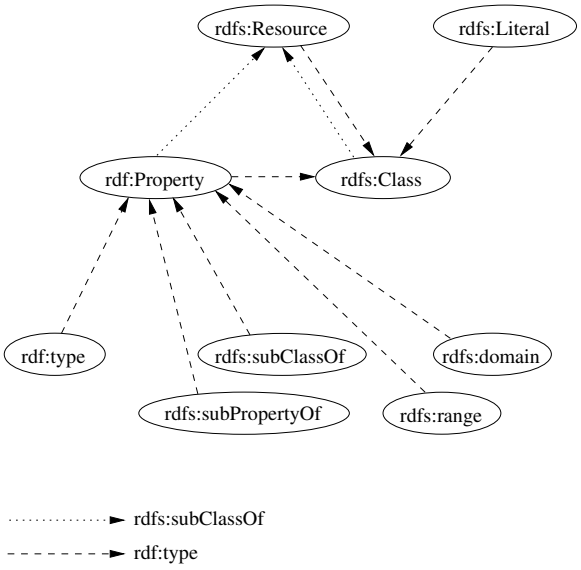


Figure 5.1: RDF/RDFS primitives.

rdf:$Property$, $rdfs$:$Class$, or $rdfs$:$Literal$) or as user-defined types (those are resources defined explicitly by a particular RDF model to have the rdf:$type$ property equal to $rdfs$:$Class$). The type of the resource $rdfs$:$Class$ is defined reflexively to be $rdfs$:$Class$. The resource $rdfs$:$Class$ contains all the types, which is not the same thing as saying that it includes all the values (instances) represented by these types.

We extend the data model with the set C (set of classes). At RDFS level the following holds: $C \subset R$, $rdfs$:$Resource \in C$, rdf:$Property \in C$, $rdfs$:$Class \in C$, and $rdfs$:$Literal \in C$.

Definition 6 *The set of classes of an RDF model M is*

$$C = \{\ c \mid (c, rdf{:}type, rdfs{:}Class) \in M\}$$

The most general types are $rdfs$:$Resource$ and $rdfs$:$Literal$ which represent all resources and literals, respectively. According to the data model these types are disjoint. Subclasses of the class $rdfs$:$Resource$ are $rdfs$:$Class$ and $rdfs$:$Property$, $rdfs$:$Class$ representing all types (already stated above), and $rdfs$:$Property$ containing all properties. The distinction between properties and resources is not a clear cut one as properties are resources with some additional (edge) semantics associated to them. A property (edge) can be used repetitively between nodes (similar in a way to repeating a particular type in the definition of its instances) which justifies the existence of an *extent* function (defined later on) for properties, as well as for classes. Moreover, property instances can have the $rdfs$:$subPropertyOf$ property defined in the same way as one can use the $rdfs$:$subClassOf$ property for classes.

The most important properties (each instance of rdf:$Property$) are: $rdfs$:$subClassOf$, $rdfs$:$subPropertyOf$, $rdfs$:$domain$, and $rdfs$:$range$. The properties $rdfs$:$subClassOf$ and

$rdfs$:$subPropertyOf$ are used to define inheritance relationships between classes and properties, respectively. Based on the RDF Test Cases [Grant and Beckett, 2004] the properties rdf:$subClassOf$ and rdf:$subPropertyOf$ can produce cycles, a useful mechanism if we think about class or property equivalence. A resource of type rdf:$Property$ may define the $rdfs$:$domain$ and the $rdfs$:$range$ associated to that property: the type of the subject and object nodes of the property edge. Inspired by ontology languages, like OWL [van Harmelen et al., 2003], $rdfs$:$domain$ and $rdfs$:$range$ can be multiply defined for one particular property and will have conjunctive semantics.

There is one particular class called $rdfs$:$Literal$ that represents all strings. Note that the RDF Semantics [Hayes, 2004] identifies two types of literals: plain literals and type literals. A plain literal is a 2-tuple (lexical form, language identifier) and a typed literal is a 3-tuple (lexical form, language identifier, datatype URI). The datatype URI is an XML Schema datatype [Biron and Malhotra, 2001] or rdf:$XMLLiteral$ for XML content. In the data model we simplify the literal definition considering just the character string (the lexical form) for literals. Note that literals are not resources, i.e., one cannot associate properties to them. On the other hand, there are resources that have type $rdfs$:$Literal$ and thus can have properties attached to them. Nevertheless one cannot say which literal this resource denotes. RDF defines also the container classes rdf:Seq, rdf:Bag, and rdf:Alt to model ordered sequences, sets with duplicates, and value alternatives. The properties rdf:rdf_1, rdf:rdf_2, rdf:rdf_3, etc., refer to container members.

5.3.4 Class and Property Nodes

As shown in Table 5.3 each node representing a class has three schema properties. Schema properties associated to nodes are short notations (like a macro) for expressions doing the same computation based only on basic properties. The *type* of a class node is $rdfs$:$Class$. The set of superclasses (classes from which the current class node is inheriting properties) is given by $subClassOf$. RDFS allows multiple inheritance for classes because $rdfs$:$subClassOf$ (as any other property) can be repeated on a particular class. The *extent* of a class node is the set of all instances of this class.

Schema property	Result
type	$rdfs$:$Class$
$subClassOf$	S with $S \subset C$
extent	R' with $R' \subset R$

Table 5.3: Schema properties for class nodes.

Each node representing a property has five schema properties as shown in Table 5.4. The *type* of a property node is rdf:$Property$. The set of superproperties (properties which the current property is specializing) is given by $subPropertyOf$. Note that the domain or range of a superproperty should be superclasses for the current property's domain or range, respectively. The *domain* and *range* return sets of classes that represent the domain and

the range, respectively, of the property node. The *extent* of a property node is the set of resource pairs linked by the current property: this set of pairs is a subset of the Cartesian product between the associated domain and range extents.

Schema property	Result
type	rdf:$Property$
subPropertyOf	S with $S \subset P$
domain	D with $D \subset C$
range	R with $R \subset C$
extent	E with $E \subset \cap_{d \in domain} extent(d) \times \cap_{r \in range} extent(r)$

Table 5.4: Schema properties for property nodes.

One should note that we assume in the data model that there can be several edges having the same *name* but linking different pairs of resources. All these properties can be seen as "instances" (abusing the term "instance" previously referring to resource instances of a particular class) of the property node with the *id* value equal to their common name.

In absence of a schema, all RDF properties have type rdf:$Property$, domain R, and range $R \cup L$. In this way one can define the *extent* of an RDF property even if the property is not explicitly defined in a schema. In a schemaless RDF graph all resources are assumed to be of type $rdfs$:$Resource$.

5.3.5 Complete Models

The RDF Semantics [Hayes, 2004] defines the RDF-closure and RDFS-closure of a certain model M by adding new triples to the model M according to a collection of given inference rules. We refer to the original model M as the extensional data and to the newly generated triples as the intensional data. There are two inference rules for RDF-closure and nine inference rules for RDFS-closure. The inference rules for RDF-closure add for all properties in the model the rdf:*type* property (pointing to rdf:$Property$). Examples of inference rules for RDFS-closure are the transitivity of $rdfs$:$subClassOf$, the transitivity of $rdfs$:$subPropertyOf$, and the rdf:*type* inference for an rdf:*type* edge that follows after an $rdfs$:$subClassOf$ edge. One should note that the resulting output of applying these inference rules may trigger other rules. Nevertheless the rules will terminate for any RDF input model M, as there is only a finite number of triples that can be formed with the finite vocabulary of M.

Definition 7 *An RDF model M is complete if it contains both its RDF-closure and RDFS-closure.*

In the proposed data model we consider complete models and we neglect reification and the properties $rdfs$:$seeAlso$, $rdfs$:$isDefinedBy$, $rdfs$:$comment$, and $rdfs$:$label$ without loosing generality.

5.4 Basic RAL Operators

The purpose of defining RAL is twofold: to provide a reference mathematical study for RDF query languages and to enable algebraic manipulations for RDF query optimization. RAL is an algebra for RDF defined from a database perspective, some of its operators being inspired by their relational algebra counterparts. We used a similar approach in developing XAL [Frasincar et al., 2002a], an algebra for XML query optimization.

During the presentation of RAL operators we will use the RDF data from the example in Figure 5.2 as input for the operators. It is assumed that all operators know about the complete RDF model as it was defined in Definition 7. That means that they all have the complete knowledge (both extensional and intensional data) present in the given model. Variants of the proposed operators can be defined using the suffix "^" which will make the operators neglect the intensional data, i.e., data derived by applying RDF(S) inference rules to the input model is neglected (similar to RQL's "strict interpretation").

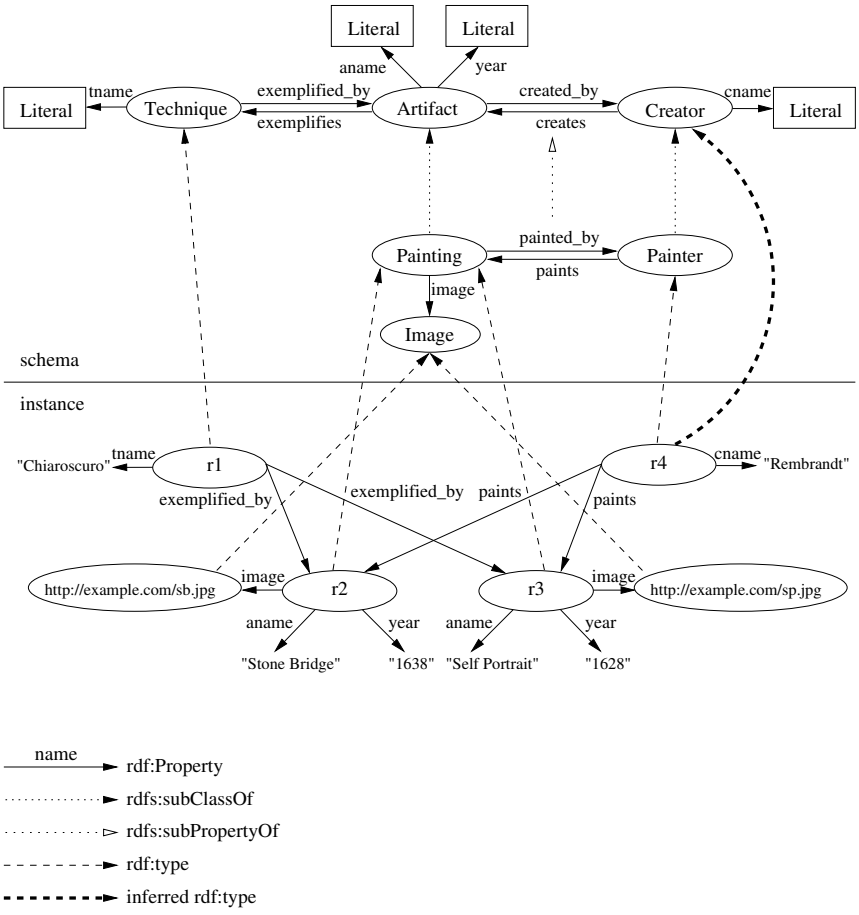


Figure 5.2: Example schema and instance.

Figure 5.2 is an excerpt from the RDF schema and RDF instance of some Web data describing different painting techniques. For reasons of simplicity we consider only one

painting technique ("*Chiaroscuro*"), one painter ("*Rembrandt*"), and two paintings of the same painter ("*StoneBridge*" and "*SelfPortrait*"). The figure does not present the RDFS primitives *rdfs*:*Resource*, *rdf*:*Property*, *rdfs*:*Class*, and *rdfs*:*Literal* from which all the resources and literals are derived. In order to simplify Figure 5.2 we chose to present only the extensional data and just one intensional data element given by the inferred edge *rdf*:*type* between $r4$ and *Creator*. For the same reasons we omit from the figure edges representing the inverse properties *exemplifies* and *painted_by* between instances (e.g., the edge labeled *painted_by* between $r2$ and $r4$) that are nevertheless part of the data model.

We define RDF collections to be sets of nodes (resources/literals). A collection is denoted as $\{e_1, e_2, ...e_n\}$ where $e_1, e_2, ...e_n$ are the nodes in the collection. A node, a unique element in the RDF graph, is also a unique element in a collection that contains it. The collections (sets) of nodes are closed under all operators, which implies that RAL expressions can be easily composed. The collection concept is similar to the monad concept from mathematics [Wadler, 1992]. A monad is defined over a certain type M. In contrast to the monad, RAL collections are more liberal in the sense that they are not restricted to a particular type M. A RAL collection can contain both literals and resources of different types. A monad is defined as a triple of functions $(map^M, unit^M, join^M)$. RAL also has the *map* operation defined and the monad *join* operation is equivalent to RAL's *union* operation. In RAL there is no *unit* operation as the singleton collection $\{n\}$ is written in the same way as the single node n. Based on the similarities between monads and RAL collections, one can reuse the three monad laws (left unit law, right unit law, and associativity law) as equivalence rules in RAL (see the first three RAL laws from Section 5.6). The fact that RAL collections are not ordered enables the commutativity law of some binary operations (e.g., Law 11 from Section 5.6). In comparison with the relational algebra, RAL is more powerful as binary operations like union do not have to meet the "compatibility" condition from the relational algebra.

RAL operators come in three flavors: extraction operators retrieve the needed resources from the input RDF model, loop operators support repetition, and construction operators build the resulting RDF model. The RAL philosophy is based on the fact that the collection of nodes represents a collection of graph components that contain these nodes. Using the extraction operators a subgraph of the original graph is selected. The construction operators build a new model by creating nodes/edges as well as reusing old nodes (possibly without some edges) and old edges.

The general form of the operators is

$$o[f](x_1, x_2, \ldots x_n : expression)$$

Informally, this form represents the following. For each binding of x to a tuple from the input collections, $f(x)$ is computed. A tuple is formed by taking one element from each input collection: $x_1, x_2, \ldots x_n$. Note that $x1, x2, \ldots x_n$ are algebra expressions that return collections. f is a function that may use basic/derived properties or one of the proposed operators. Based on the semantics of operator o a partial result for the application of o to

$f(x)$ is computed for each binding x. The operator result is obtained by combining (through set union) all partial results. All unary operators use this implicit union mechanism, the *map* operator, to compute the result. In the operator's general form, the function f is optional. For readability reasons we use for binary operators the infix notation.

RAL operators are defined to work on any RDF description, with or without an explicit schema. Note that implicitly there is always a default schema based on the following RDFS primitives: $rdfs$:$Resource$, rdf:$Property$, $rdfs$:$Class$, and $rdfs$:$Literal$. These RDFS primitives can be used to retrieve a particular schema in case that such information is not known in advance. Once the application schema is known, one can formulate queries to return instances from the input model.

5.4.1 Extraction Operators

The extraction operators retrieve the resources/literals of interest from the input collection of nodes. If the operator is not defined on nodes that represent literals, these nodes are simply neglected.

In the examples that illustrate the operators we will use expressions that return collections of resources from the example RDF model m of Figure 5.2. The expression c represents the collection (set) of all resources present in model m.

Projection

$$\pi[re\_name](e : expression)$$

The input of the projection is a collection of nodes (specified by the expression e) and the projection operator computes the values (objects) of the properties with a name given by the regular expression re_name over strings. The symbol # represents the wildcard that matches any string.

Example 1 $\pi[exemplified\_by](r1)$ *returns the collection of artifacts that exemplify the painting technique* $r1$ *from the input model (depicted in Figure 5.2):* $r2$ *and* $r3$.

Example 2 $\pi[(P|p)aint[s]\#](r4)$ *returns the collection of paintings painted by* $r4$*:* $r2$ *and* $r3$.

Example 3 $\pi[rdf{:}type](r4)$ *returns the collection of resources representing a type of* $r4$*:* $Painter$, $Creator$, *and* rdf:$Resource$.

Selection

$$\sigma[condition](e : expression)$$

In a selection the *condition* is a Boolean function that uses as constants URIs and/or strings. The operators allowed in the *condition* are RAL operators, the usual comparison operators $(=, >=, <=, <, >, <>)$, and logical operators (and, or, not). The input of the

selection is a collection of nodes and the operator selects only the nodes that fulfill the *condition*.

Example 4 *$\sigma[\pi[tname] = \text{“}Chiaroscuro\text{”}](c)$ is a selection operation applied to the collection c of all resources in the input model. The expression returns the resource(s) representing the painting technique with the name “Chiaroscuro” (i.e., $r1$).*

Example 5 *$\sigma[\pi[rdf{:}type] = Creator](\{r3, r4\})$ returns resources from the input model with the value of rdf:type being Creator: $r4$, since $r4$ is a resource of type Painter and Painter is a subclass of Creator.*

Example 6 *$\sigma\hat{}[\pi[rdf{:}type] = Creator](\{r3, r4\})$ (different from the selection in the previous example, as “^” implies the use of only the extensional data) returns the empty collection, as the inferred rdf:type of $r4$ (i.e., Creator) from the input model will not be available to the operator.*

Cartesian Product

$$(x : expression) \times (y : expression)$$

The Cartesian product takes as input two collections of nodes on which it performs the set-theoretical Cartesian product. Each pair of nodes is used to build an anonymous resource that has all the properties of the original resources. Thus, this newly built resource will have all the types of the original two resources (RDF multiple classification of resources). The final output is the collection of all those anonymous resources.

Example 7 *$\sigma[\pi[rdf{:}type] = Technique](c) \times \sigma[\pi[rdf{:}type] = Painter](c)$ where c represents the collection of all resources in the input model, returns one anonymous resource having all the properties of the only technique $r1$ and the only painter $r4$. As a consequence this anonymous resource has both types Technique and Painter.*

Join

$$(x : expression) \bowtie [condition] \ (y : expression)$$

The join expression is defined to be a Cartesian product followed by a selection, so equivalent to

$$\sigma[condition](x \times y)$$

The expression has as input two collections of resources that have their elements paired only if they fulfill the *condition* (referring to the left and right operands). Anonymous resources are built for each such pair. The output is the collection of all those anonymous resources.

Example 8 *$(t := \sigma[\pi[rdf{:}type] = Technique](c)) \bowtie [\pi[exemplified\_by](t) = \pi[paints](p)]$ $(p := \sigma[\pi[rdf{:}type] = Painter](c))$ where c represents the collection of all resources in the input model, returns an anonymous resource having all the properties of $r1$ and $r4$. Note that in this expression $r1$ and $r4$ are paired because there is a painting (e.g., $r2$) that exemplifies $r1$ and is painted by $r4$.*

Union

$$(x : expression) \cup (y : expression)$$

The union operator combines two input collections of nodes reflecting the set-theoretical union.

Difference

$$(x : expression) - (y : expression)$$

The difference operator returns the nodes present in the first input collection but not in the second input collection.

Intersection

$$(x : expression) \cap (y : expression)$$

The intersection operator returns the nodes present in both input collections.

5.4.2 Loop Operators

Loop operators are used in RAL to control the repetitive application of a function or operator. They express repetition at input and/or function/operator level.

Map

$$map[f](e : expression)$$

The map operator is defined as

$$\cup(f(e_1), f(e_2), ...f(e_n))$$

if the collection e contains the elements $e_1, e_2, ...e_n$. So, the map operator expresses repetition at input level. The results of applying the function/operator f to each element in the input collection are combined (through set union) to obtain the final result. All unary extraction operators have an implicit map operator associated with them.

Example 9 *$map[id](c)$ where c represents the collection of all resources in the input model, computes the labels of all the non-blank nodes in the input model, i.e., the labels of all resources having an id property.*

Kleene Star

$$*[f](e : expression)$$

The Kleene star operator is defined as

$$e \cup f(e) \cup ...f(f(...(f(f(e)))...)) \cup ...$$

So, the Kleene star operator expresses repetition at function/operator level. It repeats the application of the function/operator f on the given input for possibly an infinite number of times. For each iteration the result is obtained by combining (through set union) the output of applying the function/operator on the input with the input. If after an iteration the result is the same as the input, a fixed point is reached and the repetition stops. In order to ensure termination, a variant of this operator that specifies the number of iterations n is defined below:

$$*[f, n](e : expression)$$

Note that the map operator does not include the input in the result, while the Kleene star operator does.

Example 10 *$map[id](*[\pi[rdfs{:}subClassOf]](Painting))$ gives the id of all ancestor classes in the type hierarchy starting with Painting. For our example the result will contain three labels denoting the types Painting, Artifact, and rdfs:Resource. If there would have been loops made by the rdfs:subClassOf property in the input model, the above example would still have terminated. The fact that the input model has a finite number of classes implies that at a certain moment a fixed point is reached (we obtain the same output collection as for the previous iteration) and thus the Kleene star operator terminates.*

5.4.3 Construction Operators

Querying an RDF model implies not only extracting interesting nodes from the input model but also constructing an output model by deleting nodes/edges from the extracted graph and by creating new nodes/edges.

Before actually committing a construction operation, the RDF constraints are checked on the output model. If these constraints are not met, the operation aborts. Examples of RDF constraints are: resource identifiers have to be unique, the value of *rdf:type* cannot be a literal, literals cannot have properties, etc.

Create Node

$$cnode[type, id]()$$

The create node operator possibly adds a new node to the graph. The input collection is not used in the operator semantics. The type of the new node, specified by *type*, is a resource of type $rdfs{:}Class$. The *id* is a resource identifier if the node represents a

resource, or a string if the node represents a literal. The *id* is used as input in the system's new id generator (*nig*) skolem function. This function returns the unique *nodeID*. The *nodeID* is equal to *id* if *id* is given or it is a new unique identifier if *id* is empty. In the first case an old node identifier is returned if *id* is already used as a *nodeID* in the data model. In the second case a blank node is assigned a new *nodeID*. Note that the function *nig* is injective. As a side effect of this operator, an edge representing the *type* property is added between the newly created resource and its associated type resource. The create node operator returns the created node (a collection containing one node).

Example 11 *cnode*[*Painter*]() *creates a blank node of type Painter, while cnode*[*Literal*, "*Caravagio*"]() *creates a Literal node representing the string* "*Caravagio*".

Create Edge

$$cedge[name, subject](object : expression)$$

The create edge operator possibly adds new edges (properties) to the graph. The name (label) of the edges, as specified by *name*, is the id of a resource of type *rdf*:*Property* (the id of a property resource). The *subject* and the *object* must have types complying with the domain and the range of the property resource indicated by *name*. If there is already an edge between *subject* and *object* with the label given by *name* then there is no need to create a new edge. Recall that the RDF semantics doesn't allow the presence of two edges that share the same label between the same two nodes.

The *subject* is one node (or singleton collection) in the graph. The *object* can be a collection of nodes. Note that in the above description *object* denotes a node from the input collection. The edges are created between the *subject* node and the *object* node(s). The create edge operator returns the subject node (a collection containing one node). This operation can be generalized after introducing variables in RAL as shown in Subsection 5.5.1.

Example 12 *If* $n1$ *and* $n2$ *are the two nodes constructed in Example 11,* $n1$ *denoting the blank node and* $n2$ *denoting the literal node,* $cedge[name, n1](n2)$ *creates an edge labeled name between the nodes* $n1$ *and* $n2$.

Delete Node

$$dnode(e : expression)$$

The delete node operation deletes nodes from the graph. The input collection gives the nodes that are removed. The operation returns the empty collection. As a side effect the edges connected to these nodes as subject or object are also deleted.

Example 13 $dnode(\{r2, r3\})$ *deletes the nodes* $r2$ *and* $r3$, *and all the edges connected to* $r2$ *or* $r3$. *For the given model this implies the elimination of the two resources representing paintings and their associated edges.*

Delete Edge

$$dedge[re\_name, subject](object : expression)$$

The delete edge operation deletes edges from the graph. The edges that are deleted have to start in the *subject* node and to end in one of the nodes from the *object* collection. The name (label) of the edges to be deleted is given by the regular expression *re_name*, a regular expression over strings. If the *subject* and/or the *object* expressions are empty the edges to be deleted are identified by the remaining input arguments. The operation returns the *subject* input.

Example 14 *$dedge(\#, r1)(\{r2, r3\})$ deletes the edges between $r1$ and $r2$, and between $r1$ and $r3$, irrespective of their name. In the concrete example the information that two paintings exemplify the painting technique ($r1$) is removed.*

5.5 Additional RAL Features

5.5.1 Variables

A variable is a substitute for a collection of nodes (possibly) resulting from an evaluation of an algebraic expression. A variable thus serves as a shortcut of such an expression that can be used in more complex algebraic expressions. There are several reasons for introducing variables. First, as we already saw in the definition of the join operator, the join's selection condition may need a reference mechanism for the two operands (input collections). Second, variables can be very useful in expressing complex expressions in which a collection is used repeatedly. The third reason is related to the fact that query languages like RQL give their results in terms of a table that has as columns variables and as rows bindings of these variables. If one would like to use RAL to implement RQL expressions this compatibility feature should be met.

Example 15 *$y := \pi[paints](x := r4)$ instantiates x with $r4$ and y with $r2$ and $r3$. If one wants to export these variables, the result will be a table, similar to a table returned by RQL, with two columns x and y, and two rows: the first row contains $r4$ and $r2$ and the second row contains $r4$ and $r3$.*

The last reason for having variables is the fact that it has a nice application for the construction operators. If the extracted nodes are bound to variables, these variables can be elegantly used in the construction part of RAL. The create edge operation can be extended by allowing a collection of nodes not only in the *object* part but also in the *subject* part by representing both parts with variables. The semantics of this construction operator is that for each variable binding an edge will be created between the corresponding nodes.

Example 16 *Consider the variable bindings from the previous example $y := \pi[paints](x := r4)$. The expression $cedge[peind, x](y)$ will add two edges with the label peind (the French translation of paints) to the model, one between $r4$ and $r2$, and one between $r4$ and $r3$.*

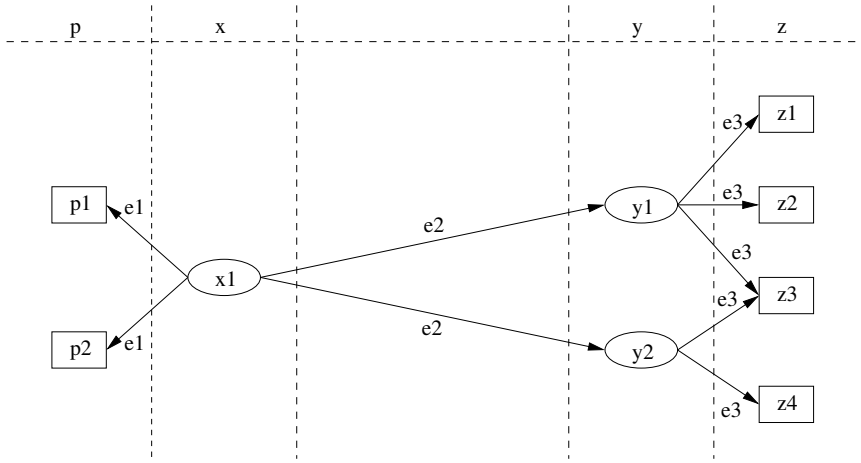


Figure 5.3: Variable bindings.

As shown in the previous example the value of the inner variable (x) is associated with two values of the outer variable (y). The two pairs ($r4$,$r2$) and ($r4$,$r3$) created by the projection operator can be seen as two 2-tuples similar to those from the relational model.

Generalizing this we can say that n-1 nested projections create a set of sets of sets ... of sets (n times) of variable bindings or in other words they generate n-tuples.

Example 17 *To illustrate the above consider the tuple bindings for the following expression operating over the RDF graph depicted in Figure 5.3:* $z := \pi[e3](y := \pi[e2](x := x1))$. *The resulting bindings are the following 3-tuples:* ($x1$,$y1$,$z1$), ($x1$,$y1$,$z2$), ($x1$,$y1$,$z3$), ($x1$,$y2$,$z3$), *and* ($x1$,$y2$,$z4$).

Note that by generating these tuple bindings we possibly generate duplicates at the variable level (the variable x is bound five times to the same value $x1$ in the above example). These duplicates are removed prior to applying variable bindings as input for an operator in order to assure "duplicate-free" collections.

In order to be able to compare results with RDF query languages that use as their output tables of tuples, RAL provides a mechanism to export tuple bindings. This is achieved simply by specifying the variable names participating in the tuple, separated by ",". For instance x,y,z exports the five tuples from the previous example. Note that if we export only one variable, say x, there will still be five 1-tuples (five times $x1$), i.e., export does not remove duplicates.

So far we discussed only variables which were bound during the multiple application of the projection operator, i.e., they occurred on the same path in the graph. These variables are dependent in the sense that the value of the next variable(s) depends on the binding of the previous ones. There might be, however, variables that do not depend on each other, i.e., they do not appear on the same path in the graph. In case of exporting independent variables export performs a cross product of their bindings.

Example 18 *The variable p from* $p := \pi[e1](x1)$ *is independent from the variables introduced in the previous example. Exporting* p, y, z *results in the following tuple bindings:* $(p1,y1,z1)$, $(p1,y1,z2)$, $(p1,y1,z3)$, $(p1,y2,z3)$, $(p1,y2,z4)$, $(p2,y1,z1)$, $(p2,y1,z2)$, $(p2,y1,z3)$, $(p2,y2,z3)$, *and* $(p2,y2,z4)$.

5.5.2 Additional Operators

Sort

$$\Sigma[value\_expression(e)](e : expression)$$

The sort operator orders alphabetically a collection based on *value_expression*. This *value_expression* is an expression that returns a collection of strings (literals or URI references). The *value_expression* is applied for each node in the input collection and the original nodes are ordered alphabetically based on the computed values.

Note that RAL collections are sets, i.e., they are not ordered. Nevertheless it is useful to be able to output ordered collections, as a last operator to be possibly used in a RAL expression.

Example 19 $\Sigma[\pi[name]](\pi[paints](r4))$ *orders alphabetically the resources representing* $r4$*'s paintings based on their names.*

5.5.3 RQL and RAL

RQL [Karvounarakis et al., 2002] is the most advanced RDF(S) query language to date and RAL was designed taking into consideration RQL's power of expression. RQL path expressions from the `FROM` clause and RQL conditions from the `WHERE` clause can easily be converted in RAL expressions using RAL operators. The vice versa conversion is not always possible as there are RAL expressions (e.g., expressions with construction operators) that are not expressible in RQL. Unlike RAL, RQL is not a closed query language; it takes as input an RDF graph and it returns a table of variable bindings. Since this table does not represent an RDF graph (just values of some variables) it cannot be used again as input for the next query. As a consequence, views are not supported. Nevertheless, RQL offers some degree of nesting queries in the `FROM` and `WHERE` clauses.

Example 20 *Find the name of all painting techniques and the name of the painters who used these techniques.* In RQL this query looks as follows:

```
SELECT Xtn, Zcn
FROM {X:Technique}exemplified_by.painted_by{Z}.cname{Zcn},
     {M}tname{Xtn}
WHERE X=M
```

In our concrete example this query returns two identical rows. The pair *Chiaroscuro*, *Rembrandt* appears twice as a result since there are two paintings ($r2$ and $r3$) that exemplify the *Chiaroscuro* technique and are painted by *Rembrandt* ($r4$).

The following RAL program exports the same variable bindings of Xtn and Zcn as the above RQL query:

$z := \pi[painted\_by](\pi[exemplified\_by](x := \sigma[\pi[rdf{:}type] = Technique](c)))$; $Xtn := \pi[tname](x)$; $Zcn := \pi[cname](z)$; Xtn, Zcn

Instead of just outputting variable values in a table-like fashion the construction operators of RAL allow for constructing a full-fledged RDF graph. For instance the following expression connects all painters from the previous query to the techniques they were using by adding a *ptechnique* edge: $cedge[ptechnique, z](x)$.

5.6 RAL Equivalence Laws

One of the advantages of using an algebra expression for a query is the ability to rewrite this expression in a form that satisfies certain needs. For example an automatic translator from RQL to RAL can use RAL equivalence laws to rewrite algebra expressions for query optimization purposes.

The proposed set of equivalence laws is inspired by the monad laws [Wadler, 1992], and the relational algebra's equivalence laws [Ullman, 1989]. In [Beeri and Kornatzky, 1993] it was shown how relational equivalence laws can be reused (redefined) in an object-oriented context.

Law 1 (Left unit) *If e_1 is of unit type (singleton collection), i.e., $e_1 = \{n\}$, then*

$$e_2(e_1) = e_2(n)$$

Law 2 (Right unit) *If e_2 is the identity function, i.e., $e_2(e) = e$, then*

$$e_2(e_1) = e_1$$

Law 3 (Empty collection) *If e_2 is the empty function, i.e., $e_2(e) = ()$, then*

$$e_2(e_1) = ()$$

Law 4 (Decomposition of $\bowtie$)

$$e_1 \bowtie [condition]\ e_2 = \sigma[condition](e_1 \times e_2)$$

Law 5 (Decomposition of π) *If name is a regular expression that can be decomposed in several regular expressions $name_1$, ... $name_n$ then*

$$\pi[name](e) = \pi[name_1](e) \cup ...\pi[name_n](e)$$

Law 6 (Cascading of σ)

$$\sigma[c_1 \wedge ...c_n](e) = \sigma[c_1](...(\sigma[c_n](e))...)$$

Law 7 (Commutativity of σ)

$$\sigma[c_1](\sigma[c_2](e)) = \sigma[c_2](\sigma[c_1](e))$$

Law 8 (Commutativity of σ with π) *If the condition c involves solely nodes that have incoming edges named by the regular expression name, then*

$$\pi[name](\sigma[c(\pi[name])](e)) = \sigma[c](\pi[name](e))$$

Law 9 (Commutativity of σ with $\times$) *If the condition c involves solely nodes from e_1, then*

$$\sigma[c](e_1 \times e_2) = \sigma[c](e_1) \times e_2$$

Law 10 (Commutativity of σ with $\cup$, $\cap$, $-$) *If θ is one of the operators $\cup$, $\cap$, and $-$, then*

$$\sigma[c](e_1 \ \theta \ e_2) = \sigma[c](e_1) \ \theta \ \sigma[c](e_2)$$

Law 11 (Commutativity of $\cup$, $\cap$, $\times$) *If θ is one of the operators $\cup$, $\cap$, and $\times$ then*

$$e_1 \ \theta \ e_2 = e_2 \ \theta \ e_1$$

Law 12 (Commutativity of π with $\times$) *If name is a regular expression that can be decomposed in two regular expressions $name_1$ and $name_2$, and if $name_1$ involves solely nodes in e_1, and $name_2$ involves solely nodes in e_2, then*

$$\pi[name](e_1 \times e_2) = \pi[name_1](e_1) \times \pi[name_2](e_2)$$

Law 13 (Commutativity of π with $\cup$)

$$\pi[name](e_1 \cup e_2) = \pi[name](e_1) \cup \pi[name](e_2)$$

Law 14 (Associativity of $\cup$, $\cap$, $\times$) *If θ is one of the operators $\cup$, $\cap$, and $\times$ then*

$$(e_1 \ \theta \ e_2) \ \theta \ e_3 = e_1 \ \theta \ (e_2 \ \theta \ e_3)$$

In order to illustrate the usefulness of the above laws for query optimization we use an example. The query optimization heuristics is based on pushing the selections/projections down as far as possible and applying the most restrictive selections first as it was done similarly in the relational algebra context. The example schema is given in Figure 5.4. It is a slightly modified example compared to the one from Figure 5.2 in the sense that the properties between concepts are replaced by literal (value) properties that function as concept identifier locators. This new example comes from a Web data integration exercise

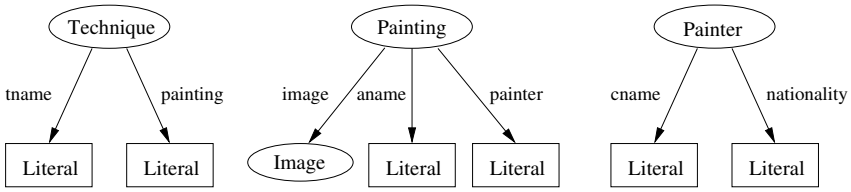


Figure 5.4: Example schema.

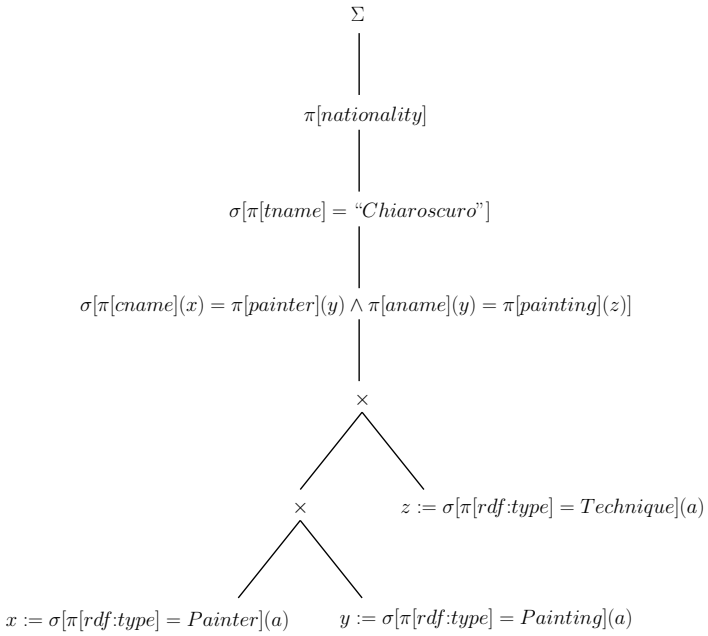


Figure 5.5: First (initial) query tree.

in which different schemas need to be merged "by value". We chose this schema example as it better (compared with the example from Figure 5.2) illustrates the proposed query optimization.

The query under investigation is: *Return in alphabetical order the nationalities of the painters that used the Chiaroscuro painting technique.* A query parser will produce the initial query tree given in Figure 5.5. In all query trees a represents the collection of all resources in the input model classified under the schema from Figure 5.4.

A query execution module will process a node in a query tree as soon as the operands are available. Such a node will be replaced by the collection that results from executing the node's associated expression. The execution terminates when the root node is processed. The final query result is the collection obtained from processing the root node.

In the example, during the execution of the initial query tree a very large Cartesian product between all painters, paintings, and techniques is generated. By pushing the selections down (using Law 6, Law 7, and Law 9) one can get the query tree in Figure 5.6.

A further improvement is obtained by applying the most restrictive selections first (using Law 7, Law 9, Law 11, and Law 14). The resulting query tree is given in Figure 5.7.

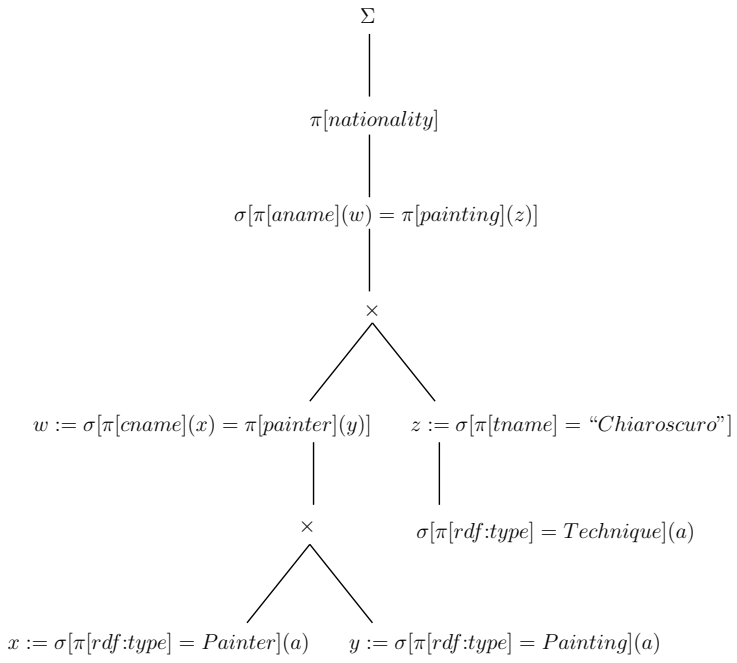


Figure 5.6: Second query tree.

So, with the aid of RAL laws three equivalent query trees were obtained.

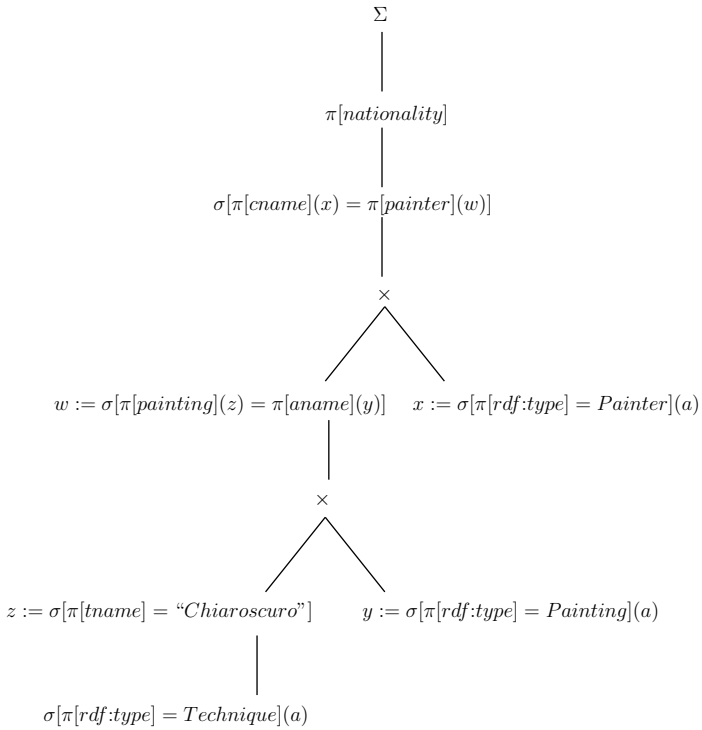


Figure 5.7: Third query tree.

In order to better understand why it is more efficient to execute the last query tree, we will give a quantitative dimension to our example. Suppose that the instance of the proposed schema example has 5 painting techniques, 100 painters, and 1000 paintings. Only 100 of all paintings use the Chiaroscuro painting technique. Let's compute now the number of elements generated by the Cartesian products for each query tree. For the first query tree we have 100 x 1000 + 5 x 100 x 1000 = 600,000 elements, for the second query tree 100 x 1000 (painters are matched to their paintings) + 1000 x 1 (paintings are matched to the Chiaroscuro painting technique) = 101,000 elements, and for the last query tree 1 x 1000 (paintings are matched to the Chiaroscuro painting technique) + 100 x 100 (paintings that use the Chiaroscuro technique are matched to their painters) = 11,000 elements. The most efficient to execute is the last query tree as its Cartesian products produce the smallest number of elements.

5.7 Conclusions

RAL is an RDF algebra defined to support the formal specification of an RDF query language. It presents a set of operations to be used in both the extraction and construction parts of a formally defined RDF query language. It is one of the first RDF algebras developed from a database perspective. Compared with existing RDF query languages, the construction phase is not neglected and is part of the language specification.

Besides being a reference language for RDF query languages, RAL can also be used for RDF query optimization. Based on RAL equivalence laws we propose a heuristic algorithm for RDF query optimization inspired by the one found in relational algebra (i.e., pushing the selections/projections down as far as possible and applying the most restrictive selections first).

As future work we will analyze the expressive power of RAL with respect to existing RDF query languages. Comparing the expressive power of RAL to that of other algebras, like relational algebra or object algebra, gives some insight into the real strength of the language, but the true test is the comparison with other existing query languages for RDF.

We would like to further investigate optimization laws that enable algebraic manipulations for query optimization. The lack of order (between resources) in RDF models and RAL collections, as well as the simplicity and composability of RAL operators (similar to the relational algebra ones) seem to foster the definition of RAL optimization laws. A translator from a popular RDF query language (e.g., RQL) to RAL and a RAL engine will enable us to experiment with different aspects of RDF query optimization.

Chapter 6

Data Visualization in Hera

A common and natural representation for RDF data is a directed labeled graph. Although there are tools to edit and/or browse RDF graph representations, we found their architecture rigid and not easily amenable to producing effective visual representations, especially for large RDF graphs. We discuss here how GViz, a general purpose graph visualisation tool, allows the easy construction and fine-tuning of various visual exploratory scenarios for RDF data. GViz's extended ability of customizing the visualization's icons showed to be very useful in the context of RDF graph structures visualisation. We demonstrate our approach by applying the developed visualization techniques for the RDF data models used in the Hera methodology. Based on the proposed visualization techniques one can answer complex questions about this data and have an effective insight into its structure.

6.1 Introduction

RDF is intended to describe the Web metadata so that the Web content is not only machine readable but also machine understandable. In this way one can better support the interoperability of Web applications. RDF Schema (RDFS) is used to describe different RDF vocabularies (schemas), i.e., the classes and properties of a particular application domain. An instantiation of these classes and properties form an RDF instance. It is important to note that both an RDF schema and an RDF instance have RDF graph representations.

Realizing the advantages that RDF offers, in the last couple of years, many tools were built in order to support the browsing and editing of RDF data. Among these tools we mention Protégé [Noy et al., 2001], OntoEdit [Sure et al., 2003], and RDF Instance Creator (RIC) [Grove, 2002]. Most of the text-based environments are unable to cope with large amounts of data in the sense of presenting them in a way that is easy to understand and navigate [Card et al., 1999]. The RDF data we have to deal with describes a large number of Web resources, and can thus easily reach tens of thousands of instances and attributes. We advocate the use of visual tools for browsing RDF data, as visual presentation and nav-

igation enables users to effectively understand the complex structure and interrelationships of such data. Existing visualization tools for RDF data are: IsaViz [Pietriga, 2002], OntoRAMA [Eklund et al., 2002], and the Protégé visualization plugins like OntoViz [Sintek, 2004] and Jambalaya [Storey et al., 2002].

The most popular textual RDF browser/editor is Protégé [Noy et al., 2001]. The generic modeling primitives of Protégé enable the export of the built model in different data formats among which is also RDF/XML. Protégé distinguishes between schema and instance information, allowing for an incremental view of the instances based on the selected schema elements. One of the disadvantages of Protégé is that it displays the information in a hierarchical way, i.e., using a tree layout [Sugiyama et al., 1981], which makes it difficult to grasp the inherent graph structure of RDF data.

In this chapter, we advocate the use of a highly customizable, interactive visualization system for the understanding of different RDF data structures. We implemented an RDF data format plugin for GViz [Telea et al., 2002], a general purpose visual environment for browsing and editing graph data. The largest advantage that GViz provides in comparison with other RDF visualization tools is the fact that it is easily and fully customizable. GViz was architected with the specific goal in mind of allowing users to define new operations for data processing, visualization, and interaction to support application specific scenarios. GViz also integrates a number of standard operations for manipulation and visualization of relational data, such as data viewers, graph layout tools, and data format support. This combination of features has enabled us to produce, in a short time, customized visualization scenarios for answering several questions about RDF data. We demonstrate our approach to RDF data visualization by using a real dataset example of considerable size.

In the next section, we describe the real-world dataset we use, and show the results obtained when visualizing it with several existing RDF tools. Our visualization tool, GViz, is presented in Section 6.3. Section 6.4 presents several visualization scenarios we built with GViz for the used RDF dataset, and details various lessons learned when building and using such visualizations. Finally, Section 6.5 concludes the chapter proposing future directions for visualizing RDF information.

6.2 Related Work

Throughout this chapter, we will use an example based on real data made available by the Rijksmuseum in Amsterdam, the largest art and history museum in the Netherlands. In the example there is a museum schema used to classify different artists and their artifacts. The museum instance describes more than 1000 artists and artifacts.

Figure 6.1 depicts the museum schema in Protégé. As can be noticed from this figure such a text-based representation cannot nicely depict the structure of a large amount of data. More exactly, a text-based display is very effective for data mining, i.e., posing targeted queries on a dataset once one knows what structure one is looking for. However, text-based displays are not effective for data understanding, i.e., making sense of a given (large) dataset of which the global structure is unknown to the user.

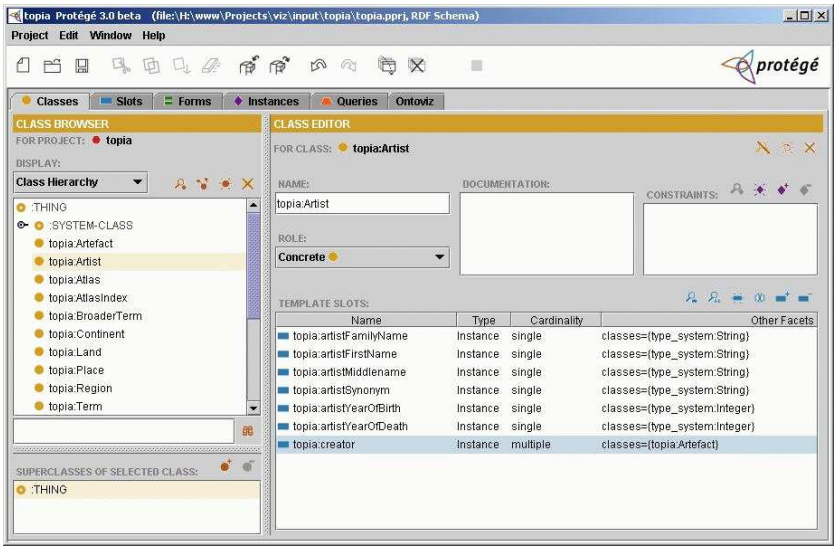


Figure 6.1: Museum schema in Protégé (text-based).

In order to alleviate the above limitation, Protégé offers a number of built-in visualization plugins. Figure 6.2 shows the graph representation generated by the OntoViz plugin for two classes from the museum schema. The weak point of OntoViz is the fact that it is not able to produce good layouts for graphs that have more than 10 nodes.

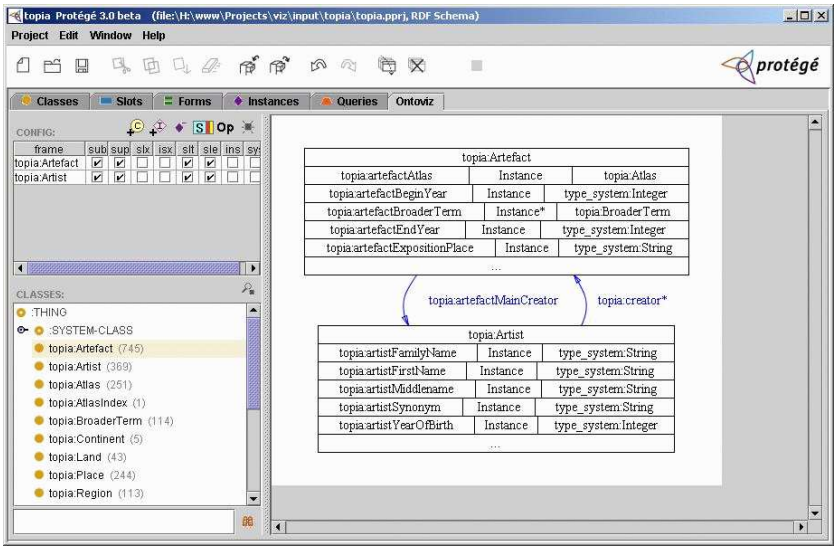


Figure 6.2: Museum schema in Protégé (with OntoViz plugin).

IsaViz [Pietriga, 2002] is a visual tool for browsing/editing RDF models. IsaViz uses AT&T's GraphViz package [Gansner et al., 2002; North, 2002] for the graph layout. Figure 6.3 shows the same museum schema using IsaViz. The layout produced by the tool is much better than the one generated with OntoViz. However, the directed acyclic graph layout used [Sugiyama et al., 1981] becomes ineffective when the dataset at hand has

roughly more than hundred nodes, as can be seen from Figure 6.3. IsaViz has a 2.5D GUI with zooming capabilities and provides numerous operations like text-based search, copy-and-paste, editing of the geometry of nodes and arcs, and graph navigation.

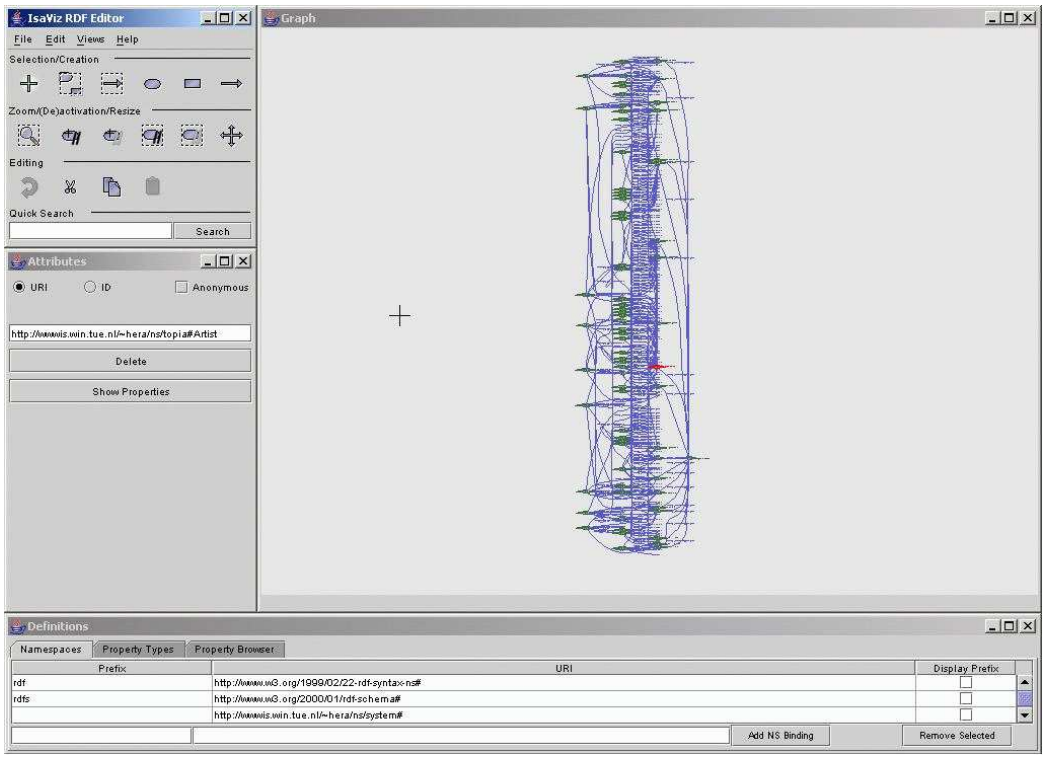


Figure 6.3: Museum schema in IsaViz.

For all these reasons, we believe that IsaViz is a state of the art tool for browsing/editing RDF models. However, its rigid architecture makes it difficult to define application-dependent operations others than the standard ones currently provided by the tool. Experience in several communities interested in visualizing relational data in general, such as software engineering and web engineering, and our own experience with RDF data in particular, has shown that tool customization is extremely important. Indeed, there is no "silver bullet" or best way to visualize large graph-like datasets. The questions to be answered, the data structure and size, and the user preferences all determine the "visualization scenario", i.e., the kind of (interactive) operations the users may want to perform to get insight in the data and answers to their questions. It is not that each separate application domain demands a specific visualization scenario. Users of the same domain and/or even the same dataset within the same domain may require different scenarios. Building such scenarios often is responsible for a large part of the complete time spent in understanding a given dataset [Telea, 2004]. This clearly requires the visualization tool in use to be highly (and easily) customizable.

6.3 GViz

In our attempt to understand RDF data through visual representations, an existing tool was used. We implemented an RDF data format plugin for GViz [Telea et al., 2002], a general purpose visual environment for browsing and editing graph data. The largest advantage that GViz provides in comparison with other RDF visualization tools is the fact that it is easily and quickly customizable. One can seamlessly define new operations to support application specific scenarios, making thus the tool more amenable for the user needs. In the past, GViz was successfully used in the reverse engineering domain, in order to define application specific visualization scenarios. Figure 6.4 presents the architecture of GViz based on four components: selection, mapping, editing, and visualization. In the next section we describe the data model used in GViz. Next, we outline the operation model describing the tasks that can be defined on the graph data. We finish the description of the GViz architecture with the visualization component which we illustrate using the museum schema dataset. The GViz core implementation is done in C++ while the user interface and scripting layer were implemented in Tcl [Raines, 1998] to take advantage of the run-time scripting and weak typing flexibility that this language provides. All the GViz customization code developed for the RDF visualization scenarios presented in this chapter was done in Tcl.

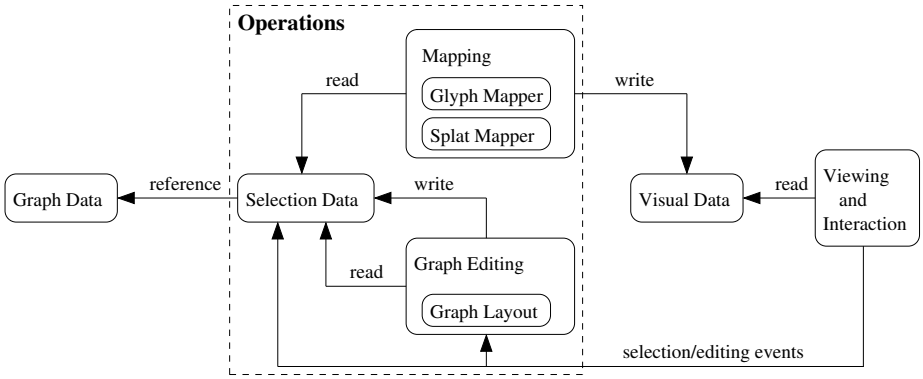


Figure 6.4: GViz architecture.

6.3.1 Data Model

The data model of GViz consists of three elements:

- *graph data*: this is the RDF graph model, i.e., a labeled directed multi-graph in which no edges between the same two nodes are allowed to share the same label. Nodes stand for RDF resources/literals and edges denote properties. Each node has a type attribute which states if the node is a *NResource* (named resource), an *AResource* (anonymous resource), or a *Literal*. The label associated to a node/edge is given by the value attribute. The labels for NResource nodes and edges are URIs. The

label for Literals is their associated string. The value of an AResource is an internal identifier with no RDF semantics. Note that the type and value attributes are GViz specific attributes that should not be confused with their RDF counterparts. Since GViz's standard data model is an arbitrary attributed graph, with any number of (name, value) type of attributes per node and edge, the RDF data model is directly accommodated by the tool.

- *selection data*: selections are subsets of nodes and edges in the graph data. Selections are used in GViz to specify the inputs and outputs of its operations; their use is detailed in Section 6.3.2.
- *visual data*: this is the information that GViz ultimately displays and allows the user to interact with. Since GViz allows customizing the mapping operation, i.e., the way graph data is used to produce visual data, the latter may assume different look-and-feel appearances. Section 6.4 illustrates this in the context of our application.

6.3.2 Operation Model

As shown in Figure 6.4, the operation model of GViz has three operation types: *selection*, *graph editing*, and *mapping*. Selection operations allow users to specify subsets of interest from the whole input graph. In the RDF visualization scenarios that we built with GViz, we defined different complex selections based on the attributes of the input model. These selections can perform tasks like: "extract the schema from an input set of RDF(S) data (which mixes schema and instance elements)". Custom selections are almost always needed when visualizing relational data, since a) the user doesn't usually want to look at too many data elements at the same time, and b) different subsets of the input data may have different semantics, thus have to be visualized in different ways. A basic example of the latter assertion is the schema extraction selection mentioned above.

Graph editing operations enable the modification, creation, and deletion of nodes/edges and/or their attributes. For our RDF visualization scenarios, we did not create or delete nodes or edges. However, we did create new data attributes, as follows. One of the key features of GViz is that it separates the graph layout, i.e., computing 2D or 3D geometrical positions that specify where to draw nodes and edges, from the graph mapping, i.e., specifying how to draw nodes and edges. The graph layout is defined as a graph editing operation which computes position attributes. Among the different layouts that GViz supports we mention the spring embedder, the directed (tree), the 3D stacked layout, and the nested layout [Telea et al., 2002]. Although based on the same GraphViz package as IsaViz, the layouts of GViz are relatively more effective, as the user can customize their behavior in detail via several parameters.

Mapping operations, or briefly mappers, associate nodes/edges (containing also their layout information) to visual data. The latter is implemented using the Open Inventor 3D toolkit, which delivers high quality, efficient rendering and interaction with large 2D and 3D geometric datasets [Wernecke, 1993]. GViz implements two mappers: the glyph mapper and the splat mapper.

The glyph mapper associates to every node/edge in the input selection a graphical icon (the glyph) and positions the glyphs based on the corresponding node/edge layout attributes. Essentially, the glyph mapper produces the "classical" kind of graph drawings, e.g., similar to those output by IsaViz. However, in contrast to many graph visualization tools, the glyph mapper in GViz allows full customization of the way the nodes and edges are drawn. The user can specify, for example, shapes, sizes, and colors for every separate node and edge, if desired, by writing a small Tcl script of 10 to 20 lines of code on the average. We used this feature extensively to produce our RDF visualizations described in Section 6.4. The splat mapper produces a continuous two-dimensional splat field for the input selection. For every 2D point, the field value is proportional to the density of nodes per unit area at that point. Essentially, the splat mapper shows high values where the graph layout used has placed many nodes, and low values where there are few nodes. Given that a reasonably good layout will cluster highly interconnected nodes together, the splat mapper offers a quick and easy way to visually find the clusters in the input graph (Figure 6.9, Section 6.4). For more details on this layout, see [van Liere and de Leeuw, 2003].

A final way to customize the visualizations in GViz is to associate custom interaction to the mappers. These are provided in the form of Tcl callback scripts that are called by the tool whenever the user interactively selects some node or edge glyph with the mouse, in the respective mapper windows. These scripts can initiate any desired operation using the selected elements as arguments, for example showing some attributes of the selected arguments. Examples of this mechanism are discussed in Section 6.4.

As explained above, GViz allows users to easily define new operations. For the incremental view of RDF(S) data, we defined operations as: extract schema, select classes and their corresponding instances, select instances and their attributes. As for the glyph mappers, these operations have been implemented as Tcl scripts of 10 to 25 lines of code. The usage of the custom selection, layout, and mapping operations for visualizing RDF(S) data is detailed in the remainder of this chapter.

6.3.3 Visualization

Figure 6.5 presents the museum data schema in GViz. We use here a radial tree layout, also available in the GraphViz package, instead of the directed tree layout illustrated in Figure 6.3 for IsaViz. As a consequence, the structure of the schema is easier to understand now.

In the above picture the edges with the label *rdf*:*type* are depicted in blue. There are two red nodes to which these blue edges connect, one with the label *rdfs*:*Class* and the other with the label *rdf*:*Property*, shown near the nodes as balloon pop-up texts. We chose to depict the property nodes (laid out in a large circular arc around the upper-left red node) in orange and the class nodes (laid out in a smaller circle arc around the lower-right red node) in green. As it can be noticed from the picture there are a lot of orange nodes which is in accordance with the property-centric approach for defining RDFS schemas. In order to express richer domain models we extended the RDFS primitives with the cardinality

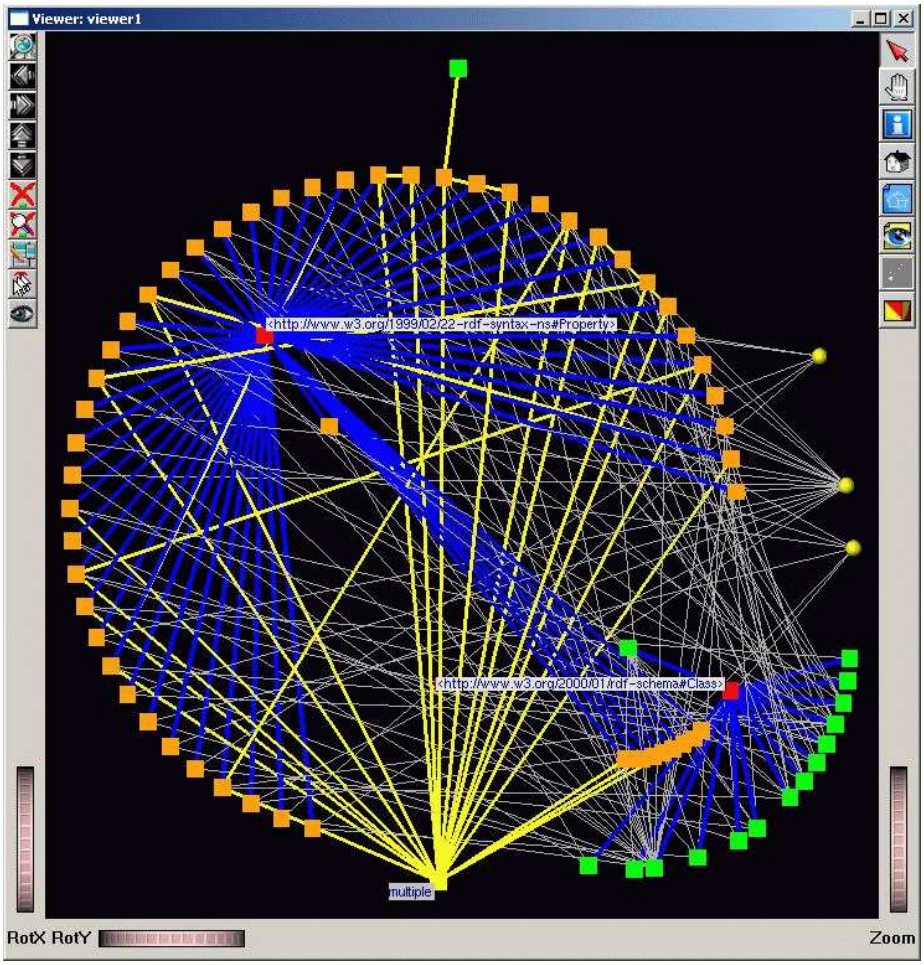


Figure 6.5: Museum schema in GViz (2D).

of properties, the inverse of properties, and a media type system. These extensions are showed in yellow edges and yellow spheres (positioned at the right end of the image). The yellow edges that connect to orange nodes represent the inverse of a property. The yellow edges that connect an orange node with the yellow rectangle labeled *multiple* (positioned at the middle of the figure bottom) state that this property has cardinality one-to-many. The default cardinality is one-to-one. Note that there are not many-to-many properties as we had previously decomposed these properties in two one-to-many properties. The three yellow spheres represent the media types: *String*, *Integer*, and *Image*. The light gray thin edges denote the domain and the range of properties. Note that only range edges can have a media node at one of its ends. As these edges are a) not so important for the user and b) quite numerous and quite hard to lay out without many overlaps, we chose to represent them in a visually inconspicuous way, i.e., make them thin and using a background-like, light gray color.

The tailoring of the graph visualization presented above is only one example. One can define some other visualizations depending on ones needs. Figure 6.6 presents a 3D view of

the same museum schema example. Here, we used a spring embedder layout, also available from the GraphViz package, to position all schema nodes in a 2D plane. Next, we designed a custom operation that selects the two $rdfs{:}Class$ and $rdf{:}Property$ nodes and offsets them away from the 2D layout plane, in opposite directions. This creates a 3D layout, which allows the user to better distinguish the different kinds of edges. For example, the edges labeled $rdf{:}type$ (colored in blue) are now clearly separated, as they reach out of the 2D plane to the offset nodes.

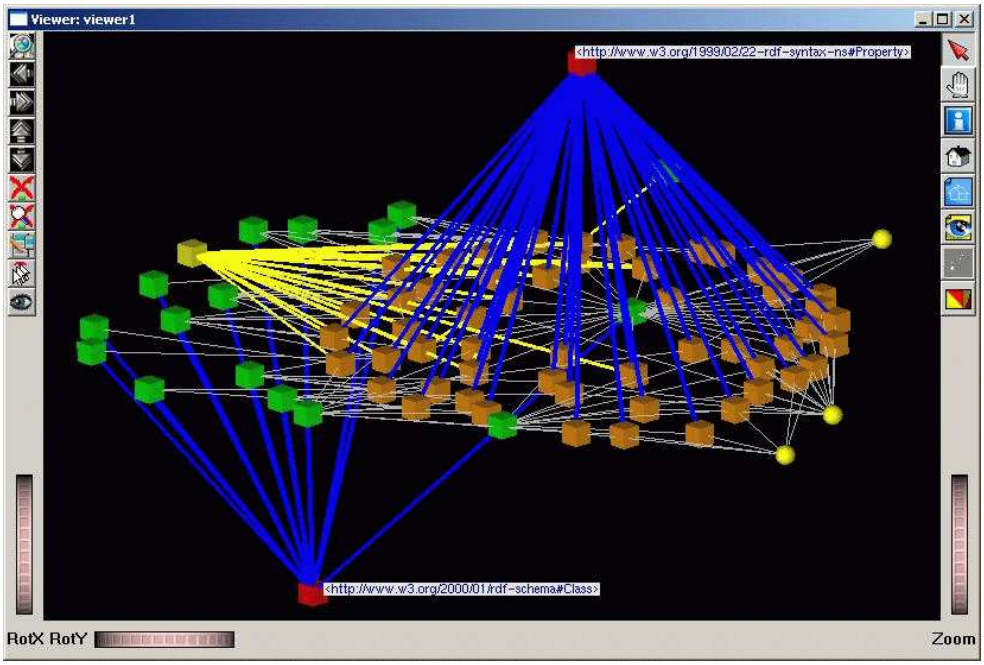


Figure 6.6: Museum schema in GViz (3D).

6.4 Applications

In order to better understand the context in which we developed our visualization applications we now briefly describe the Hera project [Vdovjak et al., 2003]. Hera is a methodology for designing and developing Web Information Systems (WISs) on the Semantic Web. All the system specifications are represented in RDFS. For the scope of this chapter it is important to have a look at two of these specifications: the conceptual model (domain model) and the application model (navigation model).

The conceptual model describes the types of the instances that need to be presented. An example of the conceptual model we already saw in Figure 6.5. A conceptual model is composed of concepts and concept properties. There are two kinds of concept properties: relationships (properties between concepts) and attributes (properties that refer to media types).

The application model defines the navigation over the data, i.e., a view on the conceptual model that contains appropriate navigation links. The application model is an overlay model over the conceptual model, a feature exploited in the definition of the transformations of the conceptual model instances into application model instances. An application model is composed of slices and slice properties. A slice contains concept attributes (not necessarily from the same concept) as well as other slices. There are two kinds of slice properties: compositional properties (aggregations) and navigational properties (links). The owner relationship is used to associate a slice to a concept. Each slice has a title attribute related to it.

A conceptual model instance and an application model instance are represented in RDF (which should be valid according to the corresponding RDFS specifications, i.e., the conceptual model and the application model, respectively). In the WIS application it is only the application model instance that will be visible to the user.

We consider now four types of RDF(S)-related visualization scenarios that are relevant in the support of the WIS application designer:

- conceptual model visualization
- conceptual model instance visualization
- application model visualization
- application model instance visualization

In Section 6.3.3 we already showed how one can visualize conceptual models. A second similar scenario for the conceptual model visualization is described next.

6.4.1 Conceptual Model Visualization

The conceptual model visualization enables one to better understand the structure of the application's domain. It answers questions like: what are the concepts?, what are the properties?, what are the relationships between concepts and properties? what are the most referenced concepts?, what are the most referenced media types?, etc.

Figure 6.7 shows the extracted schema from an RDF file that contains both the schema and its associated instance. The extraction is done by a custom selection operation, as described in Section 6.3.2. The picture is very similar to the one from Figure 6.5. However, there are two differences between this picture and the one from Figure 6.5. First, we now use a different layout, i.e., a spring embedder instead of a radial tree. Secondly, we now depict also the direction of the edges. The edges are fading out towards the start node. A direction effect is created: the edges get brighter as they approach the end node. We found this representation of the edge direction much more effective than the arrow representation when visualizing large graphs, as the drawing of arrows produces too much visual clutter in this case. Moreover, the edge fading glyph is faster to render than an arrow glyph, as it involves a single (shaded) line primitive.

From Figure 6.7 we can deduce that the most used media type is *String* (the text-based descriptions are the most popular for this domain specification) and the most referenced concept is the *Artifact* (it has the most relationships). Each artifact is classified by some museum terms (e.g., *Self Portraits*). There is a hierarchy of museum terms, terms are grouped in broader terms (e.g., *Portraits*), and broader terms are grouped in top terms (e.g., *Paintings*).

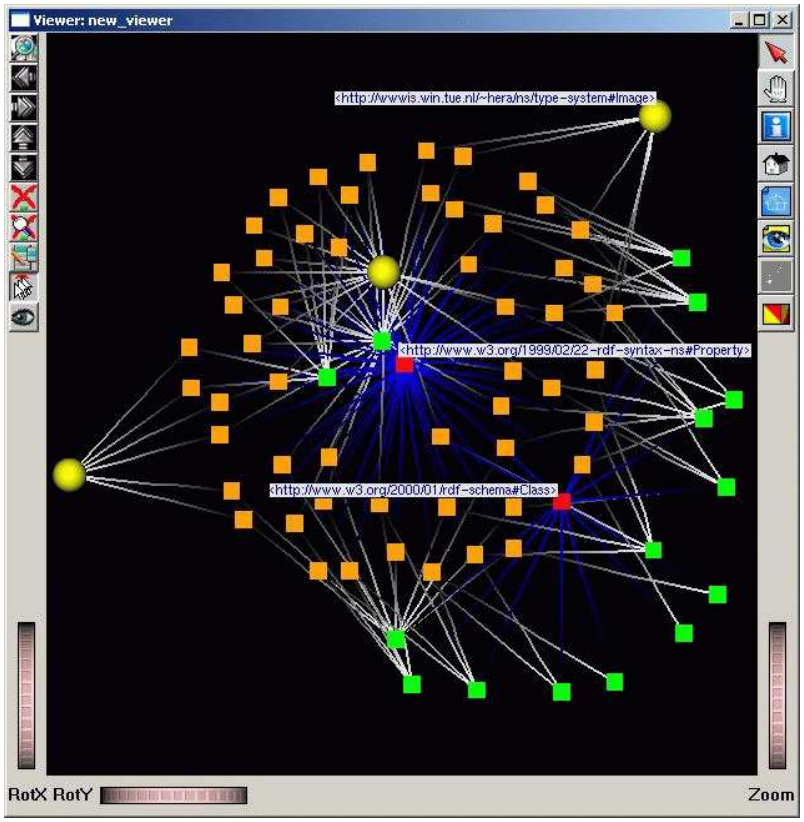


Figure 6.7: Museum extracted conceptual model.

6.4.2 Conceptual Model Instance Visualization

The conceptual model instance visualization answers questions like: what are the instances of a certain concept?, what are the relations between two selected concept instances?, what are the most referenced instances?, what are the attributes of a selected instance?, etc.

In most of the encountered situations, there are (much) more concept instances than concepts. For example, our museum dataset contains tens of thousands of instances. It is easy to imagine other applications where this number goes up to hundreds of thousands, or even more. Drawing all these instances simultaneously is neither efficient nor effective. Indeed, no graph layout we were able to test could produce an understandable image of an arbitrary, relatively tightly connected graph with tens of thousands of nodes in a

reasonable amount of time (e.g., tens of seconds). In order to keep the instance visualization manageable, we decided for an incremental view scenario on the RDF(S) data. First, the user selects the subpart of the schema for which he wants the corresponding instances to be visualized. Next, we use a custom interaction script (Section 6.3.2) of about ten lines of code to separately visualize the instances of the selected items. For example, when the user selects the Artist and Artifact concepts from Figure 6.7, the GViz tool automatically shows the instances of these concepts and their relations in another window, using a spring embedder layout (Figure 6.8). In Figure 6.8 we used a custom glyph mapper to depict the artifacts with blue rectangles and the artists with green rectangles. The relations between these instances are represented by fading white edges. One can note that there are more artifacts than artists, as expected.

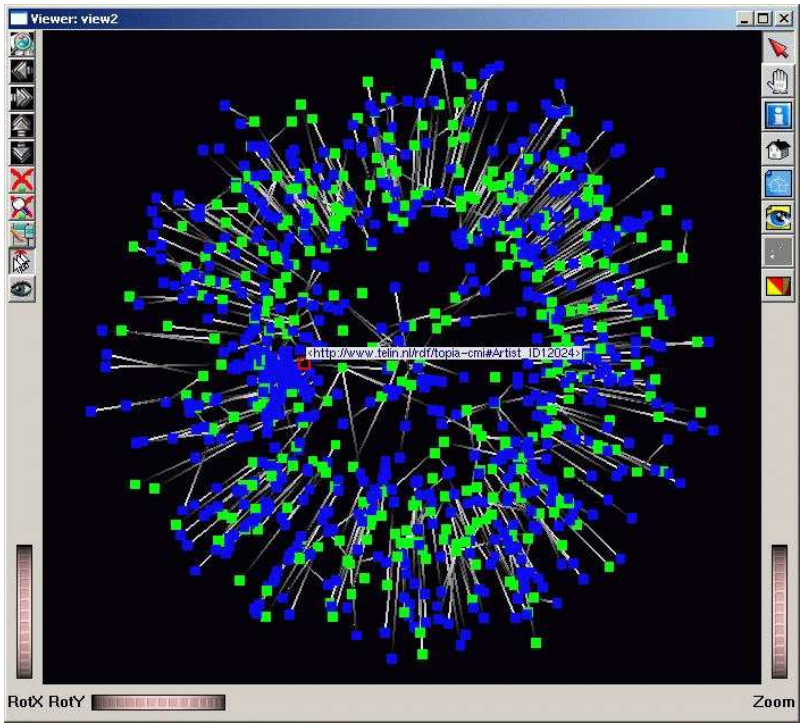


Figure 6.8: Artists/artifacts properties in the conceptual model instance.

Figure 6.9 shows the same selected data (artists and artifacts) but using a splat mapper instead of the classical glyph mapper. The scalar density function (splat field) is constructed as outlined in Section 6.3.2. We visualize the splat field using a blue-to-red colormap that associates cold hues (blue) to low values and warm hues (yellow, red) to medium and high values. Figure 6.9 (left) shows the splat field as seen from above. Figure 6.9 (right) shows the same splat field, this time visualized using an elevation plot that shows high density areas also by offsetting these points in the Z (vertical) direction. A red/yellow color in Figure 6.9 (left and right) or a high elevation point in Figure 6.9 (right) indicate that there are a lot of relations for a particular instance or group of instances. In this way one can notice from Figure 6.9 which are the artists with the most artifacts. The

artists with the most artifacts are the unknown artists (potter, goldsmith, bronzesmith, etc.) that show up as the singular peak in the left of Figure 6.9 (left). On the average, these artists have several tens (up to 60) artifacts. They are followed by Rembrandt and the unknown painters, who show up as the other two higher peaks to the right of Figure 6.9 (left). This can be explained by the fact that in the 17th century, for which the Rijksmuseum has a special focus, there were a lot of artifacts done by unknown artists.

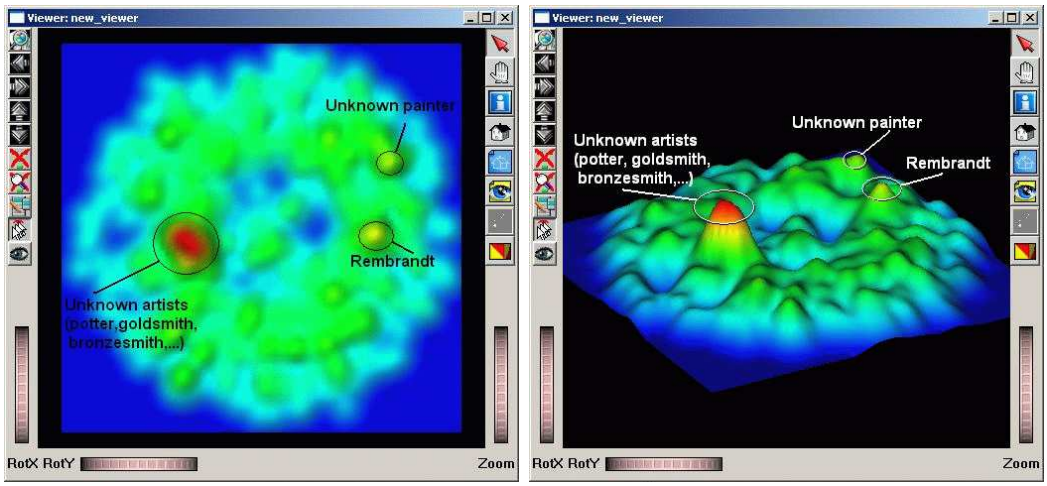


Figure 6.9: Conceptual model instance splatting (left: 2D; right: elevation plot).

We have further customized our visualization scenario, as follows. When the user selects one instance of Figure 6.8, we use a custom interaction script on the mapper of Figure 6.8 to pop up another window to display the instance attributes. The selected instance is shown as having the balloon pop-up label in Figure 6.8. Figure 6.10 shows the attributes of the selected instance, in this case Rembrandt.

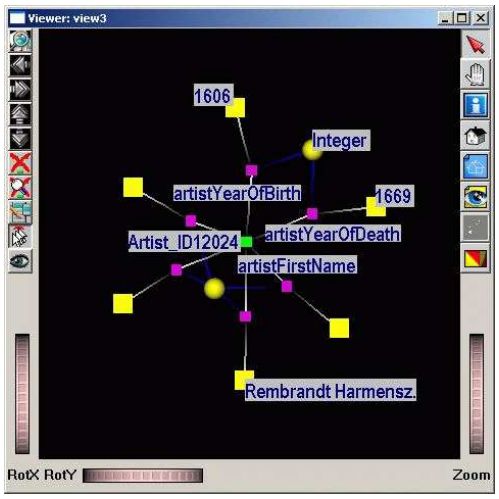


Figure 6.10: Attributes of a selected concept instance.

6.4.3 Application Model Visualisation

The application model visualization enables one to better understand the navigation structure of a hypermedia presentation. It answers questions like: what are the application model slices?, what are their links?, what are the slice owners?, what are the slice titles?, what slices are navigation hubs?, what are possible navigation paths from a certain slice?, etc.

Figure 6.11 depicts the application model for the museum example. We chose to present here the top-level slices (slices that correspond to web pages) and the links between them in order to decrease the complexity of the picture. A new glyph shape was designed in order to represent the pizza slice shape for slices (as defined in the application model's graphical representation language). The blue thick edges represent links between slices. Each slice has associated with it two attributes. We use a custom layout to place these nodes right above the top of the slice node. The slice nodes themselves are laid out using the spring embedder already discussed before. The two attributes of each slice are visualized by using two custom square glyphs, as follows: the yellow glyph (left) stands for the name of the slice and the green glyph (right) denotes the concept owner of the slice (remember that the concept owner is a concept from the conceptual model). In the center of the picture is the *Slice.artefact.main* slice which has the most links associated with it, i.e., it is a navigation hub. The figure also shows the designer's choice to present the museum information based of the terms hierarchy: top terms, broader terms, and terms.

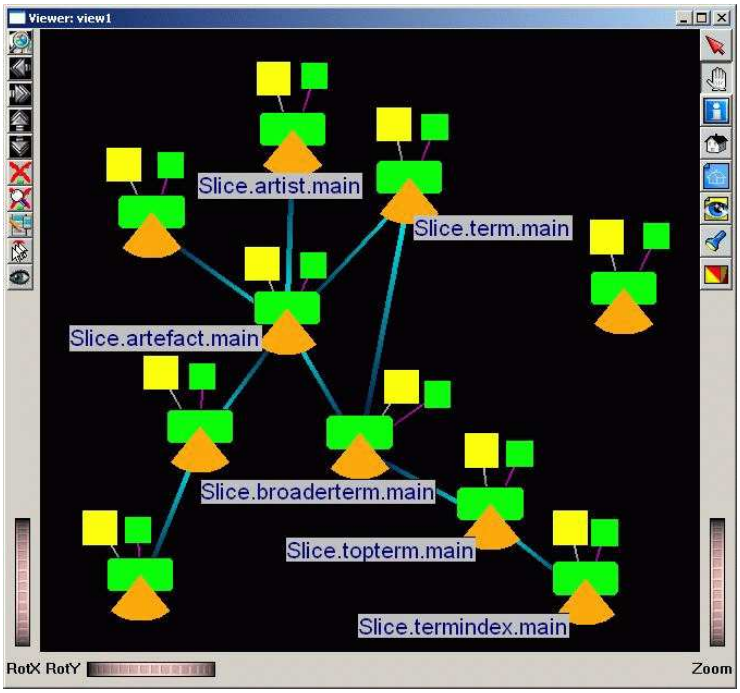


Figure 6.11: Museum application model.

6.4.4 Application Model Instance Visualisation

The application model instance visualization answers questions like: what are the instances of a certain slice?, what are the slice instances reachable from a certain slice instance?, what are the most referenced slice instances?, what are the attributes of a selected slice instance?, etc.

As there are more slice instances than slices, in order to keep the visualization manageable we used the same visualization scenario as for conceptual model instances, i.e., to use incremental views. The user can select from the mapper in Figure 6.11 the slices for which he wants the corresponding instances to be visualized. For example, after selecting the *Slice.topterm.main*, *Slice.broaderterm.main*, and *Slice.term.main* slices from Figure 6.11, we use the same mechanism of a custom interaction script (Section 6.3.2) to pop up another window that shows the instances of these slices and their associated links. Figures 6.12 and 6.13 show the corresponding slice instances, as described below.

For the visualizations in Figures 6.12 and 6.13, we use yellow sphere glyphs for nodes labeled *Slice.topterm.main*, green sphere glyphs for nodes labeled *Slice.broaderterm.main*, and blue rectangle glyphs for nodes labeled *Slice.term.main*. The chosen colors and shapes are motivated by the need to produce an expressive, easy to understand picture when presenting a large number of instances coming from three slices linked in a hierarchical way, as follows. We did give up the pizza slice glyph for these visualizations as we found out that this glyph produces too much visual clutter for large graphs. Next, we chose colors of increasing brightness (blue, green, and yellow) to display items of increasing importance (terms, broader terms, and top terms, respectively). The size of the glyphs used for these items also reflects their importance (the top term glyphs are the largest, whereas the term glyphs are the smallest). A final significant cue is the shape: the more important top and broader terms are drawn as 3D shaded spheres, whereas the less important terms are drawn as 2D flat squares. For the edges connecting these glyphs in the visualization, we used a varying color and size scheme that varies both line color and line thickness along the edge between the end nodes' colors and sizes respectively. Summing up, the combination of above choices produces a visualization where the overall structure of top terms and broader terms "pops" into the foreground, whereas the less important terms and their links "fade" into the background. As a comparison, we were unable to get the same clear view of the structure by just varying the layout parameters and using the same glyph for all nodes.

After selecting the slice instance corresponding to the *Paintings* top term, we obtain in Figure 6.12 the broader term slice instances accessible after one step, showed in red. By this, we mean the terms that a user of the web site (whose design our dataset captures) can access after one navigation step. This translates to nodes which are directly connected (via an edge) to the selected slice instance in our RDF dataset.

In Figure 6.13 we visualize the term slice instances accessible from the same Paintings top term instance slice after two steps, also drawn in red. These correspond to web pages that the user of the web site can access after two navigation steps. An example for the second step is the navigation from the broader term *Portraits*.

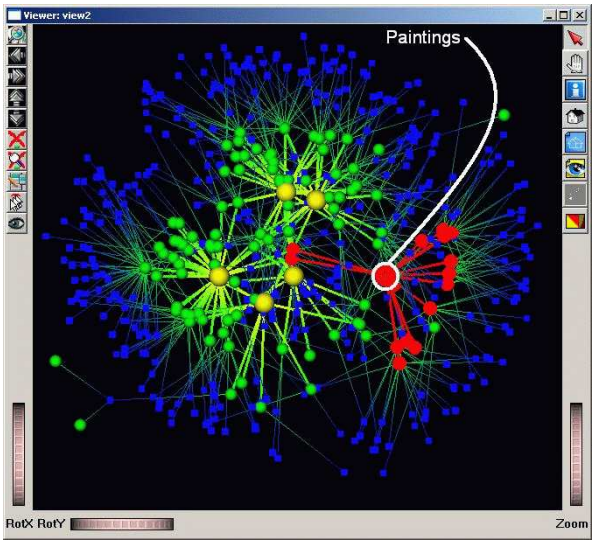


Figure 6.12: Broader term slice instances accessible from the Paintings slice instance.

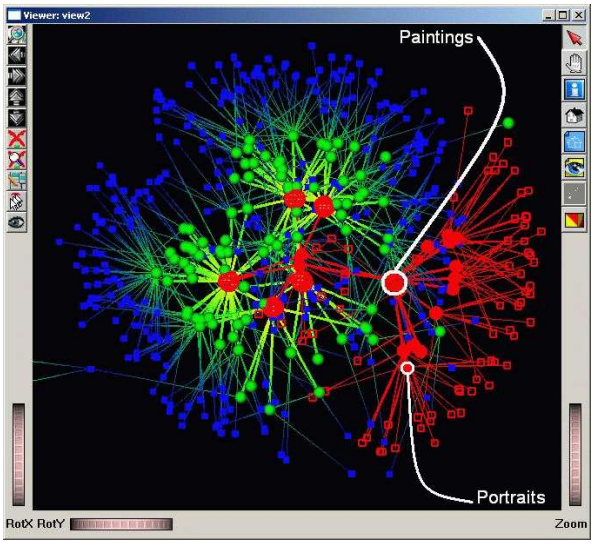


Figure 6.13: Term slice instances accessible from the Paintings slice instance.

6.5 Conclusions

In this chapter we have shown how a general purpose graph visualization tool, GViz, can be used for the visualization of large RDF graphs produced from real-world data. All experiments were performed in the context of the Hera project, a project that investigates the designing and developing of Web Information Systems on the Semantic Web. The visualization of large amounts of RDF input data and RDF design specifications enabled us to answer complex questions about this data and to give an effective insight into its structure.

Several ingredients were crucial for obtaining these results. First, the amount of customizability of the GViz tool (layouts, selections, node and edge drawing, choice between glyph and splat mappers, and custom user interaction) was absolutely necessary to produce the desired visualization scenarios. We found all these elements to be necessary to create the desired results. We have actually experimented with customizing just the layout but not the glyphs and/or the interaction. In all cases, the results were not flexible enough to give the users the desired look-and-feel that would make the scenario effective for answering the relevant questions. Secondly, the script-based customization mechanism of GViz allowed a user experienced with Tcl scripting to produce the scenarios described here (which were imagined by a different user, inexperienced with Tcl) in a matter of minutes. Thirdly, we found that using several visual cues (shape, color, size, and shading) together to enhance a single attribute, as for example described in Section 6.4.4, is much more effective than using a single cue. Finally, we mention that none of the investigated RDF visualization tools (Section 6.2) showed the high degree of customization of GViz needed for our scenarios.

In the future, we would like to explore the GViz 3D visualization capabilities for RDF data, possibly getting an even better insight into the data structure. Another research direction would be to use GViz in conjunction with a popular RDF query language (like RQL for example). Our purpose is here twofold: to use the RDF query language as a selection operation implementation for GViz when visualizing RDF data and to support the RDF query language with the visualization of the input and resulted set of RDF data. Finally, as it is planned in the Hera project to use OWL instead of RDF for the future input data/design specifications we would like also to conduct visualization experiments on the more semantically rich OWL data.

Chapter 7

Concluding Remarks

Realizing the benefits that Semantic Web technologies offer, many traditional WIS design methodologies as well as newly proposed WIS design methodologies use Semantic Web technologies for modeling WIS. Hera is a new design methodology that targets SWIS design. It distinguishes two phases: the data collection phase and the presentation generation phase. The first phase makes available data coming from different, possibly heterogeneous data sources. The second phase produces Web presentations tailored to the user and its browsing platform. In this dissertation we have presented the presentation generation phase of Hera. Section 7.1 sums up our results. Section 7.2 suggests future research directions based on our results.

7.1 Conclusions

At the beginning of the dissertation, in Chapter 1, we did ask five research questions. In the rest of this section we will summarize our findings and give the answers to the five research questions that we came up with in the different chapters of this dissertation.

Question 1: How to design the presentation generation for SWIS?
In Chapter 2 we studied several (S)WIS design methodologies. Among the found characteristics of these methodologies we mention: data integration, user interaction, presentation modeling, presentation personalization etc. At the current moment SWIS design methodologies are at their infancy, as we found no SWIS design methodology that has (all) the previously identified characteristics. In order to fill this gap, in Chapter 3 we proposed Hera, a SWIS methodology that has many of the features identified in Chapter 2. This dissertation concentrates on the presentation generation phase of the Hera methodology.

The presentation generation phase of the Hera methodology identifies the following design steps:

- **conceptual design**: constructs the conceptual model (CM), a uniform representation of the application's data. It defines the concepts and the concept relationships that are specific to the application's domain.

- **application design**: constructs the application model (AM), the navigational structure over the application's data. It defines slices and slice relationships. A slice is a meaningful presentation unit of some data. There are two types of slice relationships: slice navigation, used for links between slices, and slice aggregation, used for embedding a slice into another slice.

- **presentation design**: constructs the presentation model (PM), the look-and-feel specifications of the presentation. It defines regions and region relationships. As for slices, there are two types of region relationships: region navigation, used for links between regions, and region aggregation, used for embedding a region into another region. Regions have associated layout (positioning of inner regions inside a region) and style (fonts, colors etc.) information.

For the model representations we used RDF, the foundation language of the Semantic Web. Some of the advantages of using RDF as a representation language are: it is able to describe semi-structured Web data, it enables the reuse of previously defined vocabularies (e.g., the CC/PP UAProf vocabulary to represent user preferences and device capabilities), it allows the exchange of data between applications in a uniform format etc.

Question 2: How can we support adaptation during the design of the presentation generation for SWIS?

In Chapter 3 we have identified two types of adaptation that can be supported in the presentation generation phase of Hera:

- **static adaptation**: adaptation performed before the user starts browsing the presentation. The static adaptation is specified by appearance conditions, for elements in the CM, AM, and PM. These conditions use data from a user profile (UP) which stores the static user preferences and device capabilities. Elements for CM, AM, and PM that have the associated conditions not satisfied are removed from the specifications.

- **dynamic adaptation**: adaptation performed during the user browsing before each page is generated. The dynamic adaptation uses AM queries in order to update the User Session (US). US stores dynamic data, i.e., data created during user browsing based on user's input. The presentation generation phase uses data from US in a similar way as the CM.

Based on these two types of adaptation, in Chapter 3, we have presented two variants of Hera's presentation generation phase: the static variant and the dynamic variant. In the specifications of the dynamic variant we use (Se)RQL one of the most expressive RDF query languages. At the current moment W3C started to work at the RDF query language called SPARQL [Prud'hommeaux and Seaborne, 2005]. When this language will become more mature we plan to use it also in Hera as a replacement for (Se)RQL.

Question 3: What CASE tools can support the design of the presentation generation for SWIS?
One of the characteristics of SWIS design methodologies identified in Chapter 2 was the tool support. SWIS design methodologies are not well supported by CASE tools. In order to better sustain the design activities proposed in the presentation generation phase of the Hera methodology we have implemented a CASE tool, the Hera Presentation Generator (HPG), which is described in Chapter 4. It integrates several tools built in the last couple of years in the Hera project: builders for CM, AM, and PM, a prototyping tool based on previously defined models, and a data transformations visualization tool.

There are two variants defined for HPG: HPG-XSLT, which corresponds to the static variant of the Hera presentation generation phase, and HPG-Java, which corresponds to the dynamic variant of the Hera presentation generation phase. HPG-XSLT implements the data transformations using XSLT stylesheets, and HPG-Java implements the data transformations in Java using Jena and Sesame libraries. HPG-Java exploits more of the RDF model semantics than HPG-XSLT. Nevertheless, HPG-Java lost the declarativity, simplicity, and reuse capabilities of the XSLT transformation templates. A distributed architecture of the HPG based on Web Services is also provided.

Question 4: How can one realize query optimization inside a SWIS?
The dynamic variant of Hera's presentation generation phase uses RDF queries. The execution time of these RDF queries by a query engine is an important factor in the SWIS response time to a user request. In Chapter 5 we have proposed RAL, an RDF algebra that can be used for RDF query optimization. It defines a data model and a set of operators. The collections (sets) of nodes are closed under all operators. There are two types of operators: extraction operators, which retrieve nodes of interest from an input collection, and construction operators that build an output model possibly using also the extracted nodes. Some of the extraction operators were inspired by the ones found in relational algebra.

RAL operators satisfy equivalence laws, some of them resembling the relational algebra equivalence laws. We propose a heuristic algorithm for RDF query optimization similar to the one given in the relational algebra (i.e., pushing the selections/projections down as far as possible, and applying the most restrictive selections first). A translator from an RDF query language like (Se)RQL to RAL and a RAL engine that implements this query optimization algorithm can be a replacement of the currently used RDF query engines that do not support query optimization.

Question 5: What are suitable visualization techniques for the data used by a SWIS?
All Hera models and their instances are described in RDF. Model instances and to some extent even models do form large graphs for which visualization techniques can be useful to get a better insight into the model properties. In Chapter 6 it is described how one can apply a general-purpose graph visualization engine, GViz, for the visualization of the models used in the presentation generation phase of Hera. We did visualize conceptual models, conceptual model instances, application models, and application model instances.

One of the main advantages of GViz compared with other graph visualization tools is its customization facilities. We did define for example specific glyphs and layouts, for the visualization of the Hera models. Based on the script-based customization mechanism we were able to produce in a matter of minutes model-specific visualization scenarios.

7.2 Future Research

This dissertation describes the presentation generation phase of a SWIS design methodology. There are several directions in which this work can be extended.

One research direction is to extend the presented methodology with new steps and models. The following extensions are possible:

- To extend the proposed SWIS design methodology with a requirements gathering phase. Previous work done for WIS design methodologies based on use case specifications, user interaction specifications (OOHDM [Guell et al., 2000]), and task modeling (WSDM [De Troyer and Casteleyn, 2004]) can be useful for this purpose. By devising an interaction digram or a task model one can automatically generate the navigation structure of the application eliminating the design effort of building AMs. The generated AM can be further customized by the designer.

- To extend the dynamic adaptation in such a way that a SWIS produces adaptive hypermedia. We have published some ideas in this direction in [Vdovjak et al., 2003]. Based on these ideas one could dynamically build the User Model, Domain model, and Adaptation Model of AHAM. In this way one could also reuse existing adaptive hypermedia engines (like AHA! [De Bra et al., 2000]) inside an adaptive SWIS.

- To define a declarative RDF transformation language to be used for the data transformation specifications in the proposed methodology. We envisage that this language will be based on templates similar to XSLT but applied to the RDF context.

- To extend the Web service-oriented architecture of the HPG with new services like a data query service, a data integration service, or a service able to generate adaptive hypermedia presentations.

- To use richer Semantic Web languages for model specifications. The RDF extensions that we added for the conceptual model vocabulary (like the inverse and cardinality of relationships) are already part of the OWL language. In this way the applications built with the Hera methodology would have an even higher degree of interoperability with other applications.

Another research direction is related to RAL. The following research activities are possible:

- To build a translator from (Se)RQL or SPARQL to RAL and a RAL engine that implements our query optimization heuristics. The experiences that we could gain by using these two tools can help us in refining RAL so that it better meets practical needs.

- To explore new equivalence laws possibly involving also the construction operators. In addition, one could also use for query optimization the semantic equivalence laws that are valid in specific RDF models.

Another possible direction is related to the RDF graph visualization techniques. Some possible research activities are:

- To explore 3D visualization of RDF data, possibly getting an even better insight into its structure. The previous experiences with using GViz [Telea et al., 2002] for the 3D visualization of the graphs involved with software (re)engineering might be also useful in the RDF context.

- To use RDF graph visualization in conjunction with an RDF query language (like SPARQL) for depicting graphically the input and output sets of an RDF query. This will help for example the user to visually identify which input nodes and edges were used in the output of a query.

- To conduct visualization experiments with other Semantic Web languages like OWL. As indicated previously we plan to user richer (than RDF) Semantic Web languages for model specifications. As such it will be interesting to visualize the Hera models represented in these languages.

Bibliography

Aduna, BV (2005). openrdf.org ... home of sesame. http://www.openrdf.org/.

AIFB, University of Karlsruhe (2004). Karlsruhe ontology and semantic web tool suite. http://kaon.semanticweb.org/.

Allsopp, D., Beautement, P., Carson, J., and Kirton, M. (2002). Towards semantic interoperability in agent-based coalition command systems. In *The First Semantic Web Working Symposium*. IOS Press.

Anthemion Software (2004). wxwidgets. http://www.wxwidgets.org/.

Apache Software Foundation (2004). Xalan-java. http://xml.apache.org/xalan-j/.

Apache Software Foundation (2005a). Apache tomcat. http://jakarta.apache.org/tomcat/.

Apache Software Foundation (2005b). Webservices - axis. http://ws.apache.org/axis/java/user-guide.html.

Ayars, J., Bulterman, D., Cohen, A., Day, K., Hodge, E., Hoschka, P., Hyche, E., Jourdan, M., Kim, M., Kubota, K., Lanphier, R., Layaida, N., Michel, T., Newman, D., van Ossenbruggen, J., Rutledge, L., Saccocio, B., Schmitz, P., and ten Kate, W. (2005). Synchronized multimedia integration language (smil 2.0) - [second edition]. W3C Recommendation 07 January 2005. http://www.w3.org/TR/SMIL/.

Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D. L., Patel-Schneider, P. F., and Stein, L. A. (2004). Owl web ontology language reference. W3C Recommendation 10 February 2004. http://www.w3.org/TR/owl-ref/.

Beckett, D. (2003). Redland rdf application framework. http://www.redland.opensource.ac.uk.

Beckett, D. (2004). Rdf/xml syntax specification (revised). W3C Recommendation 10 February 2004. http://www.w3.org/TR/rdf-syntax-grammar/.

Beeri, C. and Kornatzky, Y. (1993). Algebraic optimization of object-oriented query languages. *Theoretical Computer Science*, 116(1&2):59–94.

Berglund, A., Boag, S., Chamberlin, D., Fernandez, M. F., Kay, M., Robie, J., and Simeon, J. (2005). Xml path language (xpath) 2.0. W3C Working Draft 04 April 2005. `http://www.w3.org/TR/xpath20/`.

Berners-Lee, T. (1998). What the semantic web can represent. W3C 1998. `http://www.w3.org/DesignIssues/RDFnot.html`.

Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific American*, 284(5):34–43. `http://www.scientificamerican.com/2001/0501issue/0501berners-lee.html`.

Biron, P. V. and Malhotra, A. (2001). Xml schema part 2: Datatypes. W3C Recommendation 02 May 2001. `http://www.w3.org/TR/xmlschema-2/`.

Boag, S., Chamberlin, D., Fernandez, M. F., Florescu, D., Robie, J., and Simeon, J. (2005). Xquery 1.0: An xml query language. W3C Working Draft 04 April 2005. `http://www.w3.org/TR/xquery/`.

Bos, B., Celik, T., Hickson, I., and Lie, H. W. (2004). Cascading style sheets, level 2 revision 1 css 2.1 specification. W3C Candidate Recommendation 25 February 2004. `http://www.w3.org/TR/CSS21/`.

Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte, S., and Winer, D. (2000). Simple object access protocol (soap) 1.1. W3C Note 08 May 2000.

Brambilla, M., Ceri, S., Comai, S., Fraternali, P., and Manolescu, I. (2002). Model-driven specification of web services composition and integration with data-intensive web applications. *IEEE Data Engineering Bulletin*, 25(4):53–59.

Brickley, D. and Guha, R. (2004). Rdf vocabulary description language 1.0: Rdf schema. W3C Recommendation 10 February 2004. `http://www.w3.org/TR/rdf-schema/`.

Brusilovsky, P. (2001). Adaptive hypermedia. *User Modeling and User-Adapted Interaction*, 11(1-2):87–110.

Bush, V. (1945). As we may think. *The Atlantic Monthly*, 176(1):101–108.

Card, S. K., Mackinlay, J. D., and Shneiderman, B. (1999). *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann.

Casteleyn, S. and De Troyer, O. (2001). Structuring web sites using audience class hierarchies. In *Conceptual Modeling for New Information Systems Technologies (ER 2001 Workshops)*, volume 2465, pages 1222–1228. Springer.

Casteleyn, S., De Troyer, O., and Brockmans, S. (2003). Design time support for adaptive behaviour in web sites. In *18th ACM Symposium on Applied Computing (SAC 2004)*, pages 1222–1228. ACM.

Casteleyn, S., Garrigos, I., and De Troyer, O. (2004). Using adaptive techniques to validate and correct an audience driven design of web sites. In *Web Engineering - 4th International Conference (ICWE 2004)*, volume 3140 of *Lecture Notes in Computer Science*, pages 55–59. Springer.

Catell, R. G. G., Barry, K. D., Berler, M., Eastman, J., Jordan, D., Russell, C., Schadow, O., Stanienda, T., and Velez, F. (2000). *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann.

Ceri, S., Fraternali, P., and Bongio, A. (2000). Web modeling language (webml): a modeling language for designing web sites. *Computer Networks, Ninth International World Wide Web Conference*, 33:137–157.

Ceri, S., Fraternali, P., Bongio, A., Brambilla, M., Comai, S., and Matera, M. (2003). *Designing Data-Intensive Web Applications*. Morgan Kaufmann.

Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S. (2001). Web services description language (wsdl) 1.1. W3C Note 15 March 2001.

Clark, J. (1999). Xsl transformations (xslt) version 1.0. W3C Recommendation 16 November 1999. `http://www.w3.org/TR/xslt`.

Connolly, D., van Harmelen, F., Horrocks, I., McGuinness, D. L., Patel-Schneider, P. F., and Stein, L. A. (2001). Daml+oil (march 2001) reference description. W3C Note 18 December 2001. `http://www.w3.org/TR/daml+oil-reference`.

De Bra, P., Aerts, A., Houben, G. J., and Wu, H. (2000). Making general-purpose adaptive hypermedia work. In *WebNet 2000 World Conference on the WWW and Internet (WebNet 2000)*, pages 117–123. AACE.

De Bra, P., Houben, G. J., and Wu, H. (1999). Aham: A dexter-based reference model for adaptive hypermedia. In *10th ACM conference on Hypertext and Hypermedia (Hypertext'99)*, pages 147–156. ACM.

De Troyer, O. and Casteleyn, S. (2004). Designing localized web sites. In *5th International Conference on Web Information Systems Engineering (WISE 2004)*, volume 3306, pages 547–558. Springer.

De Troyer, O. and Leune, C. (1998). Wsdm: A user-centered design method for web sites. *Computer Networks, Seventh International World Wide Web Conference*, 30:85–94.

Decker, S., Brickley, D., Saarela, J., and Angele, J. (1998). A query and inference service for rdf. In *The W3C Query Languages Workshop*. `http://www.w3.org/TandS/QL/QL98/pp/queryservice.html`.

Decker, S., Melnik, S., Van Harmelen, F., Fensel, D., Klein, M., Broekstra, J., Erdmann, M., and Horrocks, I. (2000). The semantic web: The roles of xml and rdf. *IEEE Internet Computing*, 4(5):63–74.

Diaz, A., Isakowitz, T., Maiorana, V., and Gilabert, G. (1995). Rmc: A tool to design www applications. In *Fourth International World Wide Web Conference (WWW 4)*.

Diaz, A., Isakowitz, T., Maiorana, V., and Gilabert, G. (1997). Extending the capabilities of rmm: Russian dolls and hypertext. In *30th Hawaii International Conference on System Sciences (HICSS-30)*, volume 6, pages 177–186. IEEE Computer Society.

Dubinko, M., Klotz, L. L., Merrick, R., and Raman, T. V. (2003). Xforms 1.0. W3C Recommendation 14 October 2003. `http://www.w3.org/TR/xforms/`.

Eklund, P. W., Roberts, N., and Green, S. (2002). Ontorama: Browsing rdf ontologies using a hyperbolic-style browser. In *1st International Symposium on Cyber Worlds (CW 2002)*, pages 405–411. IEEE Computer Society.

Fernandez, M. F., Florescu, D., Levy, A. Y., and Suciu, D. (2000). Declarative specification of web sites with strudel. *VLDB Journal*, 9(1):38–55.

Fiala, Z., Frasincar, F., Hinz, M., Houben, G. J., Barna, P., and Meissner, K. (2004). Engineering the presentation layer of adaptable web information systems. In *Web Engineering - 4th International Conference (ICWE 2004)*, volume 3140 of *Lecture Notes in Computer Science*, pages 459–472. Springer.

Fiala, Z., Hinz, M., Meissner, K., and Wehner, F. (2003). A component-based approach for adaptive, dynamic web documents. *Journal of Web Engineering*, 2(1-2):58–73.

Frasincar, F., Barna, P., Houben, G. J., and Fiala, Z. (2004a). Adaptation and reuse in designing web information systems. In *International Conference on Information Technology: Coding and Computing (ITCC 2004)*, pages 387–291. IEEE Computer Society.

Frasincar, F. and Houben, G. J. (2001). Xml-based automatic web presentation generation. In *WebNet 2001 World Conference on the WWW and Internet (WebNet 2001)*, pages 372–377. AACE.

Frasincar, F. and Houben, G. J. (2002). Hypermedia presentation adaptation on the semantic web. In *Adaptive Hypermedia and Adaptive Web-Based Systems (AH 2002)*, volume 2347 of *Lecture Notes in Computer Science*, pages 133–142. Springer.

Frasincar, F., Houben, G. J., and Pau, C. (2002a). Xal: an algebra for xml query optimization. In *Database Technologies 2002, Thirteenth Australasian Database Conference (ADC 2002)*, volume 5 of *Conferences in Research and Practice in Information Technology*, pages 49–56. Australian Computer Society Inc.

Frasincar, F., Houben, G. J., and Vdovjak, R. (2001). An rmm-based methodology for hypermedia presentation design. In *Advances in Databases and Information Systems (ADBIS 2001)*, volume 2151 of *Lecture Notes in Computer Science*, pages 323–337. Springer.

Frasincar, F., Houben, G. J., and Vdovjak, R. (2002b). Specification framework for engineering adaptive web applications. In *The Eleventh International World Wide Web Conference, WWW 2002, Web Engineering Track (WWW 2002)*. `http://www2002.org/CDROM/alternate/682/index.html`.

Frasincar, F., Houben, G. J., Vdovjak, R., and Barna, P. (2002c). Ral: an algebra for querying rdf. In *3rd International Conference on Web Information Systems Engineering (WISE 2002)*, pages 173–181. IEEE Computer Society.

Frasincar, F., Houben, G. J., Vdovjak, R., and Barna, P. (2004b). Ral: An algebra for querying rdf. *World Wide Web Journal*, 7(1):83–109.

Frasincar, F., Telea, A., and Houben, G. J. (2005). *Visualizing the Semantic Web*, chapter 9: Adapting graph visualization techniques for the visualization of RDF data. Springer.

Gansner, E., Koutsofios, E., and North, S. (2002). Drawing graphs with dot. `http://www.graphviz.org/Documentation/dotguide.pdf`.

Gomez, J. and Cachero, C. (2003). *Information Modeling for Internet Applications*, chapter OO-H Method: extending UML to model web interfaces, pages 144–173. Idea Group Publishing.

Grant, J. and Beckett, D. (2004). Rdf test cases. W3C Recommendation 10 February 2004. `http://www.w3.org/TR/rdf-testcases/`.

Grove, M. (2002). Rdf instance creator. `http://www.mindswap.org/~mhgrove/RIC/RIC.shtml`.

Gruber, T. R. (1993). A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220.

Guell, N., Schwabe, D., and Vilain, P. (2000). Modeling interactions and navigation in web applications. In *Conceptual Modeling for E-Business and the Web (ER 2000) Workshops*, volume 1921 of *Lecture Notes in Computer Science*, pages 115–127. Springer.

Guha, R. V. (2000). Rdfdb query language. `http://www.guha.com/rdfdb/query.html`.

Guha, R. V., Lassila, O., Miller, E., and Brickley, D. (1998). Enabling inferencing. In *The W3C Query Languages Workshop*. `http://www.w3.org/TandS/QL/QL98/pp/enabling.html`.

Halpin, T. (1995). *Model-Based Design and Evaluation of Interactive Applications*. Prentice-Hall.

Hayes, P. (2004). Rdf semantics. W3C Recommendation 10 February 2004. `http://www.w3.org/TR/rdf-mt`.

Hester, A. M., Borges, R., and Ierusalimschy, R. (1997). Xi brazilian software engineering symposium (sbes 1997). In *CGILua: A Multi-paradigmatic Tool for Creating Dynamic WWW Pages*.

Hewlett-Packard Development Company, LP (2005). Jena - a semantic web framework for java. `http://jena.sourceforge.net/`.

Houben, G. J., Barna, P., Frasincar, F., and Vdovjak, R. (2003). Hera: Development of semantic web information systems. In *Web Engineering - 3th International Conference (ICWE 2003)*, volume 2722 of *Lecture Notes in Computer Science*, pages 529–538.

Houben, G. J., Frasincar, F., Barna, P., and Vdovjak, R. (2004). Engineering the presentation layer of adaptable web information systems. In *Web Engineering - 4th International Conference (ICWE 2004)*, volume 3140 of *Lecture Notes in Computer Science*, pages 60–73. Springer.

Ierusalimschy, R., de Figueiredo, L. H., and Filho, W. C. (1996). Lua-an extensible extension language. *Software: Practice and Experience*, 26(6):635–652.

Intellidimension Inc (2002). Rdfql query language reference. `http://www.intellidimension.com/RDFGateway/Docs/querying.asp`.

Isakowitz, T., Stohr, E. A., and Balasubramanian, P. (1995). Rmm: A methodology for structured hypermedia design. *Communications of the ACM*, 38(8):34–44.

Isakowitz, T., Bieber, M., and Vitali, F. (1998). Web information systems. *Communications of the ACM*, 41(1):78–80.

Jacyntho, M. D., Schwabe, D., and Rossi, G. (2002). A software architecture for structuring complex web applications. *Journal of Web Engineering*, 2(1-2):37–60.

Jin, Y., Xu, S., and Decker, S. (2001). Ontowebber: Model-driven ontology-based web site management. In *1st International Semantic Web Working Symposium (SWWS 2001)*, pages 529–547. Stanford University.

Jin, Y., Xu, S., and Decker, S. (2002). Managing web sites with ontowebber. In *8th International Conference on Extending Database Technology (EDBT 2002)*, volume 2287 of *Lecture Notes in Computer Science*, pages 766–768. Springer.

Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., and Scholl, M. (2002). Rql: a declarative query language for rdf. In *Eleventh International World Wide Web Conference (WWW2002)*, pages 592–603. ACM.

Kay, M. (2005a). Saxon (the xslt and xquery processor). `http://saxon.sourceforge.net`.

Kay, M. (2005b). Xsl transformations (xslt) version 2.0. W3C Working Draft 11 February 2005. `http://www.w3.org/TR/xslt20/`.

Kifer, M., Lausen, G., and Wu, J. (1995). Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(1):741–843.

Klapsing, R. and Neumann, G. (2000). Applying the resource description framework to web engineering. In *First International Conference on Electronic Commerce and Web Technologies (EC-Web 2000)*, pages 229–238. Springer.

Klapsing, R., Neumann, G., and Conen, W. (2001). Semantics in web engineering: Applying the resource description framework. *IEEE MultiMedia*, 8(2):62–68.

Klyne, G. and Carroll, J. J. (2004). Resource description framework (rdf): Concepts and abstract syntax. W3C Recommendation 10 February 2004. `http://www.w3.org/TR/rdf-concepts/`.

Klyne, G., Reynolds, F., Woodrow, C., Hidetaka, O., Hjelm, J., Butler, M. H., and Tran, L. (2004). Composite capability/preference profiles (cc/pp): Structure and vocabularies 1.0. W3C Recommendation 15 January 2004.

Koch, N., Kraus, A., and Hennicker, R. (2001). The authoring process of the uml-based web engineering approach. In *First International Workshop on Web-Oriented Software Technology (IWWOST 2001)*.

Kokkelink, S. (2001). Transforming rdf with rdfpath. Working Draft. `http://zoe.mathematik.Uni-Osnabrueck.DE/QAT/Transform/RDFTransform.pdf`.

Lassila, O. (2001). Enabling semantic web programming by integrating rdf and common lisp. In *The First Semantic Web Working Symposium (SWWS 2001)*, pages 403–410. Stanford.

Lassila, O. and Swick, R. R. (1999). Resource description framework (rdf) model and syntax specification. W3C Recommendation 22 February 1999. `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222`.

Lei, Y., Motta, E., and Domingue, J. (2003). Design of customized web applications with ontoweaver. In *International Conference on Knowledge Capture (K-CAP 2003)*, pages 54–61. ACM.

Lima, F. and Schwabe, D. (2003a). Application modeling for the semantic web. In *1st Latin American Web Congress (LA-WEB 2003)*, pages 93–102. IEEE Computer Society.

Lima, F. and Schwabe, D. (2003b). Designing personalized web applications. In *Web Engineering, International Conference (ICWE 2003)*, volume 2722 of *Lecture Notes in Computer Science*, pages 417–426. Springer.

Lyardet, F. and Rossi, G. H. (1996). Enhancing productivity in the development of hypermedia applications. In *Workshop on Next Generation CASE Tools (NGCT 1996), CAiSE 1995*.

Maedche, A., Staab, S., Stojanovic, N., Studer, R., and Sure, Y. (2003). Semantic portal: The seal approach. In *Spinning the Semantic Web Bringing the World Wide Web to Its Full Potential [outcome of a Dagstuhl seminar]*, pages 317–359. MIT Press.

Maedche, A., Staab, S., Studer, R., Sure, Y., and Volz, R. (2002). Seal - tying up information integration and web site management by ontologies. *IEEE Data Engineering Bulletin*, 25(1):10–17.

Malhotra, A. and Sundaresan, N. (1998). Rdf query specification. In *The W3C Query Languages Workshop*. `http://www.w3.org/TandS/QL/QL98/pp/rdfquery.html`.

Marchiori, M. and Saarela, J. (1998). Query + metadata + logic = metalog. `http://www.w3.org/TandS/QL/QL98/pp/metalog.html`.

Martinez, J. M. (2003). Mpeg-7 overview. Version 9, ISO/IEC JTC1/SC29/WG11/N5525 March 2003.

McBride, B. (2001). Jena: Implementing the rdf model and syntax specification. In *Second International Workshop on the Semantic Web (SemWeb 2001)*, volume 40 of *CEUR Workshop Proceedings*, pages 23–28.

Mecca, G., Atzeni, P., Masci, A., Merialdo, P., and Sindoni, G. (1998). The araneus web-base management system. In *SIGMOD Conference*, pages 544–546.

Melnik, S. (1999). Algebraic specification for rdf models. Working Draft. `http://www-diglib.stanford.edu/diglib/ginf/WD/rdf-alg/rdf-alg.pdf`.

Melnik, S. (2001). Rdf api draft. `http://www-db.stanford.edu/~melnik/rdf/api.html`.

Miller, L. (2002). Inkling: Rdf query using squishql. `http://swordfish.rdfweb.org/rdfquery`.

Moura, S. S. D. and Schwabe, D. (2004). Interface development for hypermedia applications in the semantic web. In *1st Latin American Web Congress (LA-WEB 2004)*, pages 106–113. IEEE Computer Society.

Murugesan, S., Deshpande, Y., Hansen, S., and Ginige, A. (2001). Web engineering: A new discipline for development of web-based systems. In *Web Engineering*, volume 2016 of *Lecture Notes in Computer Science*, pages 3–13. Springer.

Neumann, G. and Nusser, S. (1993). Wafe - an x toolkit based frontend for application programs in various programming languages. In *USENIX Winter*, pages 181–192. USENIX Association.

Neumann, G. and Zdun, U. (2000). Xotcl, an object-oriented scripting language. In *The 7th USENIX Tcl/Tk Conference*, pages 163–174. USENIX Association.

North, S. C. (2002). Drawing graphs with neato. `http://www.graphviz.org/Documentation/neatoguide.pdf`.

Noy, N. F., Sintek, M., Decker, S., Crubezy, M., Fergerson, R. W., and Musen, M. A. (2001). Creating semantic web contents with protege-2000. *IEEE Intelligent Systems*, 16(2):60–71.

O'Toole, A. (2003). Web service-oriented architecture: The best solution to business integration. Cape Clear Software. `http://www.capeclear.com/clear_thinking/Web_Service_Oriented_Architecture2.pdf`.

Pastor, O., Fons, J., and Pelechano, V. (2003). Oows: A method to develop web applications from web-oriented conceptual models. In *International Workshop on Web-Oriented Software Technology (IWWOST 2003)*, pages 65–70.

Paterno, F. (2000). *Model-Based Design and Evaluation of Interactive Applications*. Springer.

Pietriga, E. (2002). Isaviz: a visual environment for browsing and authoring rdf models. The Eleventh International World Wide Web Conference (WWW 2002), Developer's day.

Prud'hommeaux, E. (2002). Algae howto. W3C. `http://www.w3.org/1999/02/26-modules/User/Algae-HOWTO.html`.

Prud'hommeaux, E. and Seaborne, A. (2005). Sparql query language for rdf. W3C Working Draft 17 February 2005.

Raines, P. (1998). *Tcl/Tk Pocket Reference*. O'Reilly & Associates.

Rossi, G., Schwabe, D., and Lyardet, F. (1999). Web application models are more than conceptual models. In *International Workshop on the World-Wide Web and Conceptual Modeling (WWWCM 1999), ER 1999*, volume 1727 of *Lecture Notes in Computer Science*, pages 239–253. Springer.

Rutledge, L., Alberink, M., Brussee, R., Pokraev, S., van Dieten, W., and Veenstra, M. (2003). Finding the story: Broader applicability of semantics and discourse for hypermedia generation. In *ACM Conference on Hypertext and Hypermedia (Hypertext 2003)*, pages 67–76. ACM.

Schewe, K.-D. and Thalheim, B. (2001). Modeling interaction and media objects. In *Natural Language Processing and Information Systems (NLDB 2000)*, volume 1959 of *Lecture Notes in Computer Science*, pages 313–324. Springer.

Schewe, K.-D. and Thalheim, B. (2004). Reasoning about web information systems using story algebras. In *Advances in Databases and Information Systems (ADBIS 2004)*, volume 3255 of *Lecture Notes in Computer Science*, pages 54–66. Springer.

Schmitz, P., Yu, J., and Santangeli, P. (1998). Timed interactive multimedia extensions for html (html+time). W3C Note 18 September 1998. `http://www.w3.org/TR/NOTE-HTMLplusTIME`.

Schwabe, D., Rossi, G., and Barbosa, S. D. J. (1996). Systematic hypermedia application design with oohdm. In *The Seventh ACM Conference on Hypertext (Hypertext 1996)*, pages 116–128. ACM.

Schwabe, D., de Almeida Pontes, R., and Moura, I. (1999). Oohdm-web: an environment for implementation of hypermedia applications in the www. *ACM SIGWEB Newsletter*, 8(2):18–34.

Schwabe, D. and Rossi, G. (1998). An object oriented approach to web-based application design. *Theory and Practice of Object Systems*, 4(4):207–225.

Schwabe, D., Szundy, G., Moura, S. S. D., and Lima, F. (2004). Design and implementation of semantic web applications. In *WWW Workshop on Application Design, Development and Implementation Issues in the Semantic Web (WE-SW 2004)*, volume 105 of *CEUR Workshop Proceedings*, pages 275–284.

Seaborne, A. (2001). Rdql - a data oriented query language for rdf models. HP Labs. `http://www.hpl.hp.com/semweb/rdql.html`.

Sintek, M. (2004). Ontoviz tab: Visualizing protégé ontologies. `http://protege.stanford.edu/plugins/ontoviz/ontoviz.html`.

Sintek, M. and Decker, S. (2002). Triple - an rdf query, inference, and transformation language. In *First International Semantic Web Conference (ISWC 2002)*, volume 2342 of *Lecture Notes in Computer Science*, pages 364–378. Springer.

Souchon, N. and Vanderdonckt, J. (2003). A review of xml-compliant user interface description languages. In *International Workshop on Design, Specification and Verification of Interactive Systems (DSV-IS 2003)*, volume 2844 of *Lecture Notes in Computer Science*, pages 377–391. Springer.

Storey, M.-A. D., Noy, N. F., Musen, M. A., Best, C., Fergerson, R. W., and Ernst, N. (2002). Ontoedit: Multifaceted inferencing for ontology engineering. In *International Conference on Intelligent User Interfaces (IUI 2002)*, pages 239–239. ACM.

Sugiyama, K., Tagawa, S., and Toda, M. (1981). Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems*, 11(2):109–125.

Sure, Y., Angele, J., and Staab, S. (2003). Ontoedit: Multifaceted inferencing for ontology engineering. *Journal on Data Semantics*, 1:128–152.

Telea, A. (2004). *Managing Corporate Information Systems Evolution and Maintenance*, chapter 9: An Open Architecture for Visual Reverse Engineering, pages 211–227. Idea Group Inc.

Telea, A., Frasincar, F., and Houben, G. J. (2003). Visualisation of rdf(s)-based information. In *Seventh International Conference on Information Visualization (IV 2003)*, pages 294–299. IEEE Computer Society.

Telea, A., Maccari, A., and Riva, C. (2002). An open toolkit for prototyping reverse engineering visualization. In *IEEE EG VisSym '02*, pages 241–250. Eurographics.

Thalheim, B. (2000). *Entity-Relationship Modeling*. Springer.

Thalheim, B. and Dusterhoft, A. (2001). Sitelang: Conceptual modeling of internet sites. In *Conceptual Modeling (ER 2001)*, volume 2224 of *Lecture Notes in Computer Science*, pages 179–192. Springer.

Thalheim, B., Schewe, K.-D., Romalis, I., Raak, T., and Fiedler, G. (2004). Website modeling and website generation. In *Web Engineering - 4th International Conference (ICWE 2004)*, volume 3140 of *Lecture Notes in Computer Science*, pages 577–578. Springer.

Thompson, H. S., Beech, D., Maloney, M., and Mendelsohn, N. (2001). Xml schema part 1: Structures. W3C Recommendation 02 May 2001. `http://www.w3.org/TR/xmlschema-1/`.

Ullman, J. D. (1989). *Principles of Database and Knowlwdge-Base Systems*, volume 1&2. Computer Science Press.

van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D. L., Patel-Schneider, P. F., and Stein, L. A. (2003). Web ontology language (owl) reference version 1.0. W3C Working Draft 21 February 2003. `http://www.w3.org/TR/owl-ref/`.

van Liere, R. and de Leeuw, W. (2003). Graphsplatting: Visualizing graphs as continuous fields. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):206–212.

van Ossenbruggen, J., Hardman, L., and Rutledge, L. (2005). Combining rdf semantics with xml document transformations. *International Journal of Web Engineering and Technology*, 2(4). To appear (guest editors: Frasincar, F., Houben, G. J., and van Ossenbruggen, J.).

Vdovjak, R. (2005). *A Model-driven Approach for Building Distributed Ontology-based Web Applications*. PhD thesis, Eindhoven University of Technology.

Vdovjak, R., Frasincar, F., Houben, G. J., and Barna, P. (2003). Engineering semantic web information systems in hera. *Journal of Web Engineering*, 2(1-2):3–26.

Wadler, P. (1992). Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493.

Wernecke, J. (1993). *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor*. Addison-Wesley.

Wielemaker, J. (2000). Swi-prolog rdf parser. `http://www.swi-prolog.org/packages/rdf2pl.html`.

Wireless Application Protocol Forum, Ltd. (2001). Wireless application group: User agent profile. 20 October 2001.

Index

Summary

Due to Web popularity many information systems have been made available through the Web, resulting in so-called Web Information Systems (WIS). Due to the complex requirements that WIS need to fulfill, the design of these systems is not a trivial task. Design methodologies provide guidelines for the construction of WIS such that the complexity of this process becomes manageable. Based on the separation-of-concerns principle some of these methodologies propose models to specify different aspects of WIS design.

Model-driven WIS design methodologies have been recently influenced by emerging technologies like the ones provided by the Semantic Web. We call WIS that use Semantic Web technologies Semantic Web Information Systems (SWIS). Hera is a SWIS design methodology that employs RDF, the foundation language of the Semantic Web, for its model representation. Using a standardized language to represent models fosters application interoperability. There are two main phases in Hera: the data collection phase, which makes available data coming from different possibly heterogeneous sources, and the presentation generation phase, which builds Web hypermedia presentations based on the previously integrated data. This dissertation concentrates on the presentation generation phase of the Hera methodology.

Chapter 1 introduces the research questions and provides an outline of the dissertation content. Chapter 2 gives an overview of existing model-driven (S)WIS design methodologies and their support tools. It also identifies a number of desired (S)WIS design methodology features that are used as a comparison criteria for the analyzed methodologies. Chapter 3 describes the presentation generation phase of Hera, a model-driven SWIS design methodology. Differently than many of the analyzed SWIS design methodologies, Hera has most of the desired features of a SWIS design methodology like data integration support, presentation personalization, and user interaction. All Hera models and their instances are described in RDF.

The presentation generation phase of the Hera methodology identifies the following design steps:

- conceptual design: constructs the conceptual model (CM), a uniform representation of the application's data. It defines the concepts and the concept relationships that are specific to the application's domain.

- application design: constructs the application model (AM), the navigational structure over the application's data. It defines slices and slice relationships. A slice is a

meaningful presentation unit of some data. There are two types of slice relationships: navigation relationships, used for links between slices, and aggregation relationships, used for embedding a slice into another slice.

- presentation design: constructs the presentation model (PM), the look-and-feel specifications of the presentation. It defines regions and region relationships. A region is an abstraction for a rectangular area on the user display where the contents of a slice are presented. As for slices, there are two types of region relationships: navigation relationships, used for links between regions, and aggregation relationships, used for embedding a region into another region. Regions have associated layout (positioning of inner regions inside a region) and style (fonts, colors, etc.) information.

There are two types of presentation adaptation supported in the presentation generation phase of Hera: static adaptation, i.e., adaptation performed before the user starts browsing, and dynamic adaptation, i.e., adaptation performed while the user is browsing. The static adaptation is based on appearance conditions for elements in the CM, AM, and PM. These conditions use data from a user profile (UP) which stores the static user preferences and device capabilities. The dynamic adaptation uses AM queries in order to update the user session (US). US stores dynamic data, i.e., data created during user browsing based on user's input. The presentation generation process uses data from US in a similar way as the CM. Based on these two types of adaptation there are two variants of Hera's presentation generation phase: the static variant and the dynamic variant.

Chapter 4 describes a CASE tool, the Hera Presentation Generator (HPG), that supports the presentation generation phase of Hera. There are two variants of the HPG, HPG-XSLT, which corresponds to the static variant of Hera's presentation generation phase, and HPG-Java, which corresponds to the dynamic variant of Hera's presentation generation phase. HPG-XSLT implements the data transformations using XSLT stylesheets, and HPG-Java implements the data transformations in Java using Jena and Sesame libraries. HPG-Java exploits more of the RDF model semantics than HPG-XSLT. A distributed architecture of the HPG based on Web Services is also provided.

Chapter 5 proposes RAL, an RDF algebra that can be used for RDF query optimization. It defines a data model and a set of operators. The collections (sets) of nodes are closed under all operators. There are two types of operators: extraction operators, which retrieve nodes of interest from an input collection, and construction operators that build an output model possibly using also the extracted nodes. The extraction operators satisfy equivalence laws resembling to the ones found in relational algebra. We propose a heuristic algorithm for RDF query optimization similar to the one given in relational algebra.

Chapter 6 shows how one can apply a general-purpose graph visualization engine, GViz, for the visualization of the models used in the presentation generation phase of Hera. Compared with other visualization tools GViz has the advantage of being easily customizable. As Hera models have the tendency to be rather large, we define visualization scenarios in order to get a better insight into the model properties. We did use GViz for the visualization of conceptual models, conceptual model instances, application models, and application model instances.

Samenvatting

Dankzij de populariteit van het Web zijn veel informatiesystemen beschikbaar via het Web, wat resulteert in zogeheten Web Informatie Systemen (WIS). Door de complexe eisen waaraan een WIS moet voldoen is het ontwerp van deze systemen geen triviale taak. Ontwerpmethoden geven richtlijnen voor de constructie van een WIS zodat de complexiteit van dit ontwerpproces beheersbaar wordt. Gebaseerd op het principe van separation-of-concerns stellen sommige van deze methoden modellen voor om de verschillende aspecten van het WIS-ontwerp te specificeren.

Model-gedreven WIS-ontwerpmethoden zijn recent beïnvloed door nieuwe technieken zoals het Semantic Web die biedt. We noemen WIS die Semantic Web-technieken gebruiken Semantic Web-informatiesystemen (SWIS). Hera is een SWIS-ontwerpmethode die gebruik maakt van RDF, de basistaal van het Semantic Web, om de modellen uit te drukken. Het gebruik van een standaardtaal voor de representatie van modellen bevordert de uitwisselbaarheid tussen toepassingen. Er worden in Hera twee belangrijke fasen onderscheiden: de data-collectie fase, die gegevens beschikbaar maakt uit verschillende, mogelijk heterogene informatiebronnen, en de presentatie-generatie fase, die Web-hypermediapresentaties genereert op basis van de eerder geïntegreerde gegevens. Dit proefschrift legt de nadruk op de presentatie-generatie fase van de Hera methode.

Hoofdstuk 1 presenteert de onderzoeksvragen en geeft een overzicht van de inhoud van het proefschrift. Hoofdstuk 2 geeft een overzicht van bestaande model-gestuurde ontwerpmethoden voor (S)WIS, en bepaalt een aantal gewenste eigenschappen voor zulke ontwerpmethoden die gebruikt worden als criteria om de methoden te vergelijken. Hoofdstuk 3 beschrijft de presentatie-generatie fase van Hera, een model-gestuurde SWIS-ontwerpmethode. Anders dan veel van de in hoofdstuk 2 beschouwde methoden, heeft Hera de meeste van de in hoofdstuk 2 beschouwde eigenschappen zoals de ondersteuning van de integratie van gegevens, de personalisatie van de presentatie en de interactie met de gebruiker. Alle modellen van Hera en hun instanties worden beschreven in RDF.

De presentatie-generatie fase in Hera onderscheidt de volgende ontwerpstappen:

- conceptueel ontwerp: bouwt een conceptueel model (CM), een uniforme representatie van de gegevens die in de toepassing gebruikt worden. Deze stap definieert de concepten en de verbanden ertussen die in het toepassingsdomein relevant zijn.

- toepassingsontwerp: bouwt een toepassingsmodel (AM), een navigatiestructuur over de gegevens in het CM. Deze structuur definieert “slices” en verbanden daartussen.

Een slice is een eenheid van presentatie die bepaalde gegevens aan de gebruiker toont. Tussen slices bestaan twee soorten verbanden: navigatie verbanden om verwijzingen (hyperlinks) uit te drukken, en aggregatie verbanden om een slice binnen een andere op te nemen.

- presentatieontwerp: bouwt het presentatiemodel (PM), de specificaties van de "look-and-feel" van de uiteindelijke presentatie. Het PM definieert gebieden en verbanden daartussen. Een gebied is een abstractie van een rechthoek op het scherm waarbinnen de inhoud van een slice wordt weergegeven. Zoals voor slices bestaan er twee soorten verbanden: navigatie verbanden om doorverwijzingen uit te drukken, en aggregatie verbanden om een gebied in een ander op te nemen. Een gebied heeft informatie over opmaak (de positionering van deelgebieden binnen het gebied) en stijlinformatie (lettertypen, kleuren, etc.).

De presentatie kan in deze fase van Hera op twee manieren worden aangepast: statisch, d.w.z. voordat de gebruiker begint te bladeren, en dynamisch, d.w.z. tijdens het bladeren. Statische aanpassing wordt bepaald door weergavevoorwaarden die aan elementen in het CM, het AM en het PM gesteld kunnen worden. Deze voorwaarden maken gebruik van gegevens uit een gebruikersprofiel (UP) waarin de statische gebruikersvoorkeuren en de mogelijkheden van de weergaveapparatuur worden opgeslagen. Dynamische aanpassing ondervraagt het AM en houdt een gebruikerssessie (US) bij. De US bevat dynamische gegevens, d.w.z. gegevens die tijdens het bladeren uit de invoer van de gebruiker worden afgeleid. Het presentatie-generatie proces gebruikt de gegevens uit het US op eenzelfde manier als die uit het CM. Wegens deze twee soorten adaptatie zijn er twee varianten van de presentatiefase in Hera: de statische en de dynamische.

Hoofdstuk 4 beschrijft een CASE-tool, Hera Presentation Generator (HPG), die de presentatiefase van Hera ondersteunt. Er bestaan twee varianten van HPG: HPG-XSLT, die overeenkomt met de statische variant van de fase, en HPG-Java, die met de dynamische variant overeenkomt. HPG-XSLT voert de data-transformaties uit met behulp van XSLT-stylesheets, terwijl HPG-Java ze in Java uitvoert, met behulp van de Jena- en Sesame-bibliotheken. HPG-Java benut de semantiek van de RDF-modellen beter dan HPG-XSLT. Verder wordt er een gedistribueerde architectuur voor HPG gegeven, gebaseerd op Web Services.

Hoofdstuk 5 presenteert RAL, een algebra voor RDF die gebruikt kan worden voor de optimalisatie van queries (vragen). Een datamodel wordt gedefinieerd en een verzameling operatoren op het datamodel. De verzamelingen knopen zijn gesloten onder alle operatoren. Er zijn twee soorten operatoren: extractie-operatoren, die relevante knopen uit een invoer-verzameling halen, en constructie-operatoren, die een uitvoermodel opbouwen, mogelijk met gebruikmaking van de opgevraagde knopen. De extractie-operatoren voldoen aan equivalenties vergelijkbaar met die van de relationele algebra. We stellen een heuristisch algoritme voor query-optimalisatie voor dat lijkt op dat voor de relationele algebra.

Hoofdstuk 6 laat zien hoe GViz, een algemeen hulpmiddel voor graafvisualisatie, ingezet kan worden om de modellen te presenteren die bij de presentatie-generatie fase van Hera

worden ingezet. In vergelijking met andere hulpmiddelen is GViz gemakkelijk instelbaar. Omdat Hera-modellen behoorlijk groot kunnen worden, gebruiken we visualisatie-scenario's om een beter inzicht te krijgen in de eigenschappen van modellen. GViz is toegepast op conceptuele modellen, toepassingsmodellen, en instanties van beide.

Curriculum Vitae

Flavius Frasincar was born on 14th November 1971 in Bucharest, Romania. After completing his pre-university education at "Gheorghe Lazar" high-school, profile mathematics-physics, he started in the same year to study at "Politehnica" University of Bucharest, Romania. During the six years of faculty study, he received four three-months scholarships: one at Tampere University of Technology, Finland, one a at University of Sunderland, United Kingdom, and two at Eindhoven University of Technology, the Netherlands. He graduated in 1997 with a master in Computer Science.

After graduation he worked as a teaching assistant (courses Programming Techniques, Introduction to Object-Oriented Programming, and Parallel Algorithms) at the same university for one year. In 2000 he joined the software technology program of Stan Ackermans Institute at Eindhoven University of Technology. After two years he received a Professional Doctorate in Engineering (PDEng). He started his PhD research in 2000 in Databases and Hypermedia Group, Department of Mathematics and Computer Science at Eindhoven University of Technology. From 2004 he is also part-time assistant professor (courses Databases, Java Programming, and Web Information Systems) in the same group. His research interests are: Web information systems, Semantic Web, Web query languages, Web data visualization, databases, and software engineering.

SIKS Dissertatiereeks

====
1998
====

1998-01 Johan van den Akker (CWI)
DEGAS - An Active, Temporal Database of Autonomous Objects
1998-02 Floris Wiesman (UM)
Information Retrieval by Graphically Browsing Meta-Information
1998-03 Ans Steuten (TUD)
A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective
1998-04 Dennis Breuker (UM)
Memory versus Search in Games
1998-05 E.W. Oskamp (RUL)
Computerondersteuning bij Straftoemeting

====
1999
====

1999-01 Mark Sloof (VU)
Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products
1999-02 Rob Potharst (EUR)
Classification using decision trees and neural nets
1999-03 Don Beal (UM)
The Nature of Minimax Search
1999-04 Jacques Penders (UM)
The practical Art of Moving Physical Objects
1999-05 Aldo de Moor (KUB)
Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems

1999-06 Niek J.E. Wijngaards (VU)
Re-design of compositional systems
1999-07 David Spelt (UT)
Verification support for object database design
1999-08 Jacques H.J. Lenting (UM)
Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation

====
2000
====

2000-01 Frank Niessink (VU)
Perspectives on Improving Software Maintenance
2000-02 Koen Holtman (TUE)
Prototyping of CMS Storage Management
2000-03 Carolien M.T. Metselaar (UVA)
Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief
2000-04 Geert de Haan (VU)
ETAG, A Formal Model of Competence Knowledge for User Interface Design
2000-05 Ruud van der Pol (UM)
Knowledge-based Query Formulation in Information Retrieval
2000-06 Rogier van Eijk (UU)
Programming Languages for Agent Communication
2000-07 Niels Peek (UU)
Decision-theoretic Planning of Clinical Patient Management
2000-08 Veerle Coup (EUR)
Sensitivity Analyis of Decision-Theoretic Networks
2000-09 Florian Waas (CWI)
Principles of Probabilistic Query Optimization
2000-10 Niels Nes (CWI)
Image Database Management System Design Considerations, Algorithms and Architecture
2000-11 Jonas Karlsson (CWI)
Scalable Distributed Data Structures for Database Management

====
2001
====

2001-01 Silja Renooij (UU)
Qualitative Approaches to Quantifying Probabilistic Networks
2001-02 Koen Hindriks (UU)
Agent Programming Languages: Programming with Mental Models
2001-03 Maarten van Someren (UvA)
Learning as problem solving
2001-04 Evgueni Smirnov (UM)
Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets
2001-05 Jacco van Ossenbruggen (VU)
Processing Structured Hypermedia: A Matter of Style
2001-06 Martijn van Welie (VU)
Task-based User Interface Design
2001-07 Bastiaan Schonhage (VU)
Diva: Architectural Perspectives on Information Visualization
2001-08 Pascal van Eck (VU)
A Compositional Semantic Structure for Multi-Agent Systems Dynamics
2001-09 Pieter Jan 't Hoen (RUL)
Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes
2001-10 Maarten Sierhuis (UvA)
Modeling and Simulating Work Practice
BRAHMS: a multiagent modeling and simulation language for work practice analysis and design
2001-11 Tom M. van Engers (VUA)
Knowledge Management:
The Role of Mental Models in Business Systems Design

====
2002
====

2002-01 Nico Lassing (VU)
Architecture-Level Modifiability Analysis
2002-02 Roelof van Zwol (UT)
Modelling and searching web-based document collections
2002-03 Henk Ernst Blok (UT)
Database Optimization Aspects for Information Retrieval

2002-04 Juan Roberto Castelo Valdueza (UU)
The Discrete Acyclic Digraph Markov Model in Data Mining
2002-05 Radu Serban (VU)
The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents
2002-06 Laurens Mommers (UL)
Applied legal epistemology; Building a knowledge-based ontology of the legal domain
2002-07 Peter Boncz (CWI)
Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications
2002-08 Jaap Gordijn (VU)
Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas
2002-09 Willem-Jan van den Heuvel(KUB)
Integrating Modern Business Applications with Objectified Legacy Systems
2002-10 Brian Sheppard (UM)
Towards Perfect Play of Scrabble
2002-11 Wouter C.A. Wijngaards (VU)
Agent Based Modelling of Dynamics: Biological and Organisational Applications
2002-12 Albrecht Schmidt (Uva)
Processing XML in Database Systems
2002-13 Hongjing Wu (TUE)
A Reference Architecture for Adaptive Hypermedia Applications
2002-14 Wieke de Vries (UU)
Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems
2002-15 Rik Eshuis (UT)
Semantics and Verification of UML Activity Diagrams for Workflow Modelling
2002-16 Pieter van Langen (VU)
The Anatomy of Design: Foundations, Models and Applications
2002-17 Stefan Manegold (UVA)
Understanding, Modeling, and Improving Main-Memory Database Performance

====
2003
====

2003-01 Heiner Stuckenschmidt (VU)
Ontology-Based Information Sharing in Weakly Structured Environments
2003-02 Jan Broersen (VU)
Modal Action Logics for Reasoning About Reactive Systems
2003-03 Martijn Schuemie (TUD)
Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy
2003-04 Milan Petkovic (UT)
Content-Based Video Retrieval Supported by Database Technology
2003-05 Jos Lehmann (UVA)
Causation in Artificial Intelligence and Law - A modelling approach
2003-06 Boris van Schooten (UT)
Development and specification of virtual environments
2003-07 Machiel Jansen (UvA)
Formal Explorations of Knowledge Intensive Tasks
2003-08 Yongping Ran (UM)
Repair Based Scheduling
2003-09 Rens Kortmann (UM)
The resolution of visually guided behaviour
2003-10 Andreas Lincke (UvT)
Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture
2003-11 Simon Keizer (UT)
Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks
2003-12 Roeland Ordelman (UT)
Dutch speech recognition in multimedia information retrieval
2003-13 Jeroen Donkers (UM)
Nosce Hostem - Searching with Opponent Models
2003-14 Stijn Hoppenbrouwers (KUN)
Freezing Language: Conceptualisation Processes across ICT-Supported Organisations
2003-15 Mathijs de Weerdt (TUD)
Plan Merging in Multi-Agent Systems
2003-16 Menzo Windhouwer (CWI)
Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses

2003-17 David Jansen (UT)
Extensions of Statecharts with Probability, Time, and Stochastic Timing
2003-18 Levente Kocsis (UM)
Learning Search Decisions

====
2004
====

2004-01 Virginia Dignum (UU)
A Model for Organizational Interaction: Based on Agents, Founded in Logic
2004-02 Lai Xu (UvT)
Monitoring Multi-party Contracts for E-business
2004-03 Perry Groot (VU)
A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving
2004-04 Chris van Aart (UVA)
Organizational Principles for Multi-Agent Architectures
2004-05 Viara Popova (EUR)
Knowledge discovery and monotonicity
2004-06 Bart-Jan Hommes (TUD)
The Evaluation of Business Process Modeling Techniques
2004-07 Elise Boltjes (UM)
Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes
2004-08 Joop Verbeek(UM)
Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politiele gegevensuitwisseling en digitale expertise
2004-09 Martin Caminada (VU)
For the Sake of the Argument; explorations into argument-based reasoning
2004-10 Suzanne Kabel (UVA)
Knowledge-rich indexing of learning-objects
2004-11 Michel Klein (VU)
Change Management for Distributed Ontologies
2004-12 The Duy Bui (UT)
Creating emotions and facial expressions for embodied agents
2004-13 Wojciech Jamroga (UT)
Using Multiple Models of Reality: On Agents who Know how to Play

2004-14 Paul Harrenstein (UU)
Logic in Conflict. Logical Explorations in Strategic Equilibrium
2004-15 Arno Knobbe (UU)
Multi-Relational Data Mining
2004-16 Federico Divina (VU)
Hybrid Genetic Relational Search for Inductive Learning
2004-17 Mark Winands (UM)
Informed Search in Complex Games
2004-18 Vania Bessa Machado (UvA)
Supporting the Construction of Qualitative Knowledge Models
2004-19 Thijs Westerveld (UT)
Using generative probabilistic models for multimedia retrieval
2004-20 Madelon Evers (Nyenrode)
Learning from Design: facilitating multidisciplinary design teams

====
2005
====

2005-01 Floor Verdenius (UVA)
Methodological Aspects of Designing Induction-Based Applications
2005-02 Erik van der Werf (UM))
AI techniques for the game of Go
2005-03 Franc Grootjen (RUN)
A Pragmatic Approach to the Conceptualisation of Language
2005-04 Nirvana Meratnia (UT)
Towards Database Support for Moving Object data
2005-05 Gabriel Infante-Lopez (UVA)
Two-Level Probabilistic Grammars for Natural Language Parsing
2005-06 Pieter Spronck (UM)
Adaptive Game AI