Vrije Universiteit Brussel

FACULTY OF SCIENCE
Department of Computer Science
Web & Information Systems Engineering

A Semantics-based Aspect-Oriented Approach to Adaptation Engineering

A thesis presented in fulfillment of the thesis requirement for a license degree in Applied Computer Science
by

William Van Woensel

Academic Year 2006-2007

Promotor: Prof. Dr. Ir. Geert-Jan Houben
Guiding assistant: Dr. Sven Casteleyn

Vrije Universiteit Brussel

FACULTEIT VAN DE WETENSCHAPPEN
Departement Computerwetenschappen
Web & Information Systems Engineering

A Semantics-based Aspect-Oriented Approach to Adaptation Engineering

Proefschrift ingediend met het oog op het behalen van de graad van Licentiaat in de Toegepaste Informatica door

William Van Woensel

Academiejaar 2006-2007

Promotor: Prof. Dr. Ir. Geert-Jan Houben
Belegeider: Dr. Sven Casteleyn

VRIJE UNIVERSITEIT BRUSSEL • SCIENTIA VINCERE TENEBRAS •

Abstract

In the modern Web, a broad range of users are accessing Web applications in increasingly diverse ways. Therefore, the need emerged to personalize and adapt these Web applications to the specific user. This meant tailoring content access, presentation and functionality to the user’s context, e.g. location, Web client (device), preferences, age, gender, etc. However, this proves difficult in practice: design concerns, resulting from adding adaptation support to a Web application, are typically spread across the design, often leading to a complex and difficult to maintain application design. Such concerns are called cross-cutting in Aspect-Oriented Programming, where they are tackled by encapsulated them in a component called an aspect. In this research, we focus on separating adaptation engineering from regular Web engineering by applying aspect-oriented techniques. Consequently, we are able to specify adaptation separate from (regular) Web Engineering. We also show how semantic information and global (structural) information on the design can be exploited in the application of our aspect-oriented approach. This enables us to specify adaptation in a more declarative way, by *describing* what needs to be adapted rather than explicitly specifying the affected design elements. This also makes our adaptation more robust: when changing the design later on, new (or changed) elements that have to be adapted can be automatically recognized by the adaptation engine. Furthermore, by using global (structural) information on the design, we are able to specify adaptation that is in no way hard-coded to the specific application design, and thus perfectly re-usable over Web applications. Finally, because we use a domain-specific language for specifying adaptation concerns, we are able to express adaptation in a much more powerful and expressive way.

Abstract

In het moderne Web accesseert een diverse groep van gebruikers Web applicaties op steeds meer verschillende manieren. Bijgevolg verscheen de nood om deze Web applicaties te personalizeren en aan te passen aan de specifieke gebruiker. Dit omvat de toegang tot data, presentatie en functionaliteit van de applicatie afhankelijk te maken van de context van de gebruiker, bv. locatie, Web client, persoonlijke voorkeuren, leeftijd, geslacht, etc. Dit bleek echter niet vanzelfsprekend in de praktijk: design concerns, resulterend van het toevoegen van adaptatie ondersteuning aan een Web applicatie, zijn typisch verspreid over het design en leidden meestal tot een complex en moeilijk te onderhouden applicatie ontwerp. Zulke concerns worden cross-cutting genoemd in Aspect-Oriented Programming, waar dit probleem opgelost wordt door de concerns te encapsuleren in zogenaamde aspect componenten. In dit onderzoek concentreren we ons op het scheiden van adaptation engineering van het "normale" Web design door het toepassen van aspect-georiënteerde technieken. We tonen ook aan hoe semantische informatie en globale (structurele) design informatie kan gebruikt worden in de toepassing van onze aspect-georiënteerde aanpak. Dit stelt ons in staat om adaptatie op een meer declaratieve manier aan te pakken, door te *beschrijven* wat er moet aangepast worden in plaats van expliciet de nodige elementen te specifiëren die aangepast moeten worden. Dit maakt onze adaptatie ook meer robuust: als het ontwerp later veranderd wordt, kunnen de nieuwe (of veranderde) elementen die moeten aangepast worden aan de gebruiker automatisch herkend worden. Door globale (structurele) design informatie te gebruiken, zijn we tevens in staat om adaptatie te definiëren die onafhankelijk is van het specifieke applicatie ontwerp, en bijgevolg perfect herbruikbaar over verschillende Web applicaties. Uiteindelijk, omdat we een domein-specifieke taal gebruiken om adaptation concerns neer te schrijven, zijn we in staan om adaptatie op een meer krachtige en expressieve manier uit te drukken.

Acknowledgements

I would like to thank:

Prof. Dr. Ir. Geert-Jan Houben, my promoter.

Dr. Sven Casteleyn, for his guidance and support while writing this text.

The researchers at the WISE lab and at the Information Systems group at the Technische Universiteit Eindhoven.

Table of Contents

List of Figures

Chapter 1: Introduction

1.1 Research context

Since its foundation by Tim Berners-Lee and Robert Cailliau[1], the World Wide Web has evolved from being a research environment for professionals, with little need for content presentation, security or programming facilities, to an omni-present and multi-purpose network. Significant advances in mobile information technology have paralleled this evolution. Wireless network communication has been catching up with traditional Internet communication, thanks to great increases in bandwidth and availability (e.g. "hotspots"). Handheld and portable devices (e.g. pda's, mobile phones, portable game-consoles) have become Web clients in their own right, due to more economical power usage and increases in performance. Moreover, an overall price drop of computer and network technology has lead to an increase in the number and diversity of people accessing the Web. All these factors have contributed to the Internet becoming the most popular and important contemporary communication medium, resulting in a broad range of organizations (e.g. governments, businesses and academic institutions) having a presence on the Web, be it just for presentation purposes or to provide (part of) their services via a Web application.

As the Web expanded to include a broader range of users and functionality, the need emerged to personalize and adapt these Web applications to the specific user. This meant tailoring content access, presentation and functionality to the user's context, e.g. location, Web client (device), preferences, age, gender, etc. For this purpose, a user model can be kept that contains relevant information about the current user. This information can be entered manually by the user or gathered automatically by the system, by analyzing the user's interaction with the Web site.

By adding adaptation support to the Web application, extra design concerns arise in the Web application design called *adaptation concerns*. A single adaptation concern can lead to several adaptation requirements: for example, supporting age-restriction could include deleting links to elements that are unsuited for the user and/or putting emphasis on links to elements that are more suited, adjusting the layout to suit the user's age, etc. To implement these requirements various techniques can be used, such as showing/hiding elements, changing page headers, switching between predefined layouts, changing link appearance, etc.

Alongside the increasing need for adaptation has been the emergence of the Semantic Web, where semantics of content is made explicit by means of ontologies and metadata. Built on top of the eXtensible Markup Language (XML[2]), the Resource Description Framework (RDF[3]) uses metadata to describe resources, thus making their semantics explicit and machine-readable. The Web Ontology Language (OWL[4]) is built on top of RDF, and allows the user to better define the resources used in an RDF document, and the relationships between them. This metadata could be exploited to allow for more fine-grained and powerful adaptation, by inferring valuable information implicitly present in the conceptual model.

[1] http://en.wikipedia.org/wiki/WWW
[2] http://www.w3.org/XML/
[3] http://www.w3.org/RDF/
[4] http://www.w3.org/2004/OWL/

1.2 Problem statement

It is not hard to imagine that the design of an adaptation concern is typically spread across the Web application design. For example, to support age-restriction, the current user's age has to be taken into account each time a link is displayed, and subsequently some action (e.g. change the link's color or remove it) has to be taken at runtime, depending on the element the link points to. Another example is device-dependency, where the display of the Web site has to be adapted to the device the visitor uses to access the Web site. This includes adjusting the display of each media-element (e.g. video, picture) according to the connection speed and available screen-size. Such concerns are said to "cross-cut" the standard Web design, because they influence elements all over the Web application design. Considering this cross-cutting nature, ad-hoc design of adaptation concerns together with the rest of the application becoming exceedingly complex, and leads to a design that is difficult to develop and maintain. It can also be noted this is in contrast with the separation of concerns, underlying each conceptual design method.

Existing methods that tackle adaptation concerns in Web design models (mostly by adding adaptation conditions to elements) suffer from the same problem. For example, event-based approaches (e.g. [5], [9], [11]) are a proven way to model adaptation in Web applications. However, they lead to event handlers being spread over the design: in the AHA! system [5], rules have to be manually attached to each attribute that is relevant to the adaptation concern. Other approaches include adding conditions to the data extraction query of an element (e.g. Hera-S, Torii [12]) to make the instantiation of the element dynamic, but they still include spreading the adaptation concern over the different element queries.

To exemplify this, consider the following fragment from a Hera-S Application Model (AM). It contains several Navigational Units, which represent resources from the application domain and define how to navigate between them. Each of these elements contains a query responsible for instantiating the element at runtime (see 2.4). We want to add support for device-dependency by not displaying the director's picture when the visitor is using a small-screen device (e.g. mobile phone, pda, etc). Furthermore, we want to support age-restriction by constraining these resources according to the user's age. For this purpose, we can rewrite the element queries, which results in the follow code (non-altered parts are omitted, adaptation conditions are put in bold):

```
:DirectorUnit a am:NavigationUnit ;
    ...
    am:hasAttribute [
        rdfs:label "Photo" ;
        am:hasQuery
        "SELECT Ph
        FROM {$D} rdf:type {imdb:Director};
                imdb:hasMainPhoto {Ph},
                {} rdf:type {cm:CurrentUser};
                cm:userDevice {Dev}
        WHERE NOT Dev LIKE 'pda'"
    ] ;
    am:hasRelationship [
        rdfs:label "Movies directed" ;
        am:refersTo :MovieUnit ;
        am:hasQuery
            "SELECT M
            FROM {$D} rdf:type {imdb:Director};
```

```
                imdb:directorFilmography {M} imdb:hasAgeRating {}
                imdb:minAllowedAge {Min},
                {} rdf:type {cm:CurrentUser};
                cm:age {Age}
          WHERE Age >= Min"
    ] .
```

In the above example, the "Photo" attribute query in the DirectorUnit element is expanded to include an adaptation condition, which states that the query should not return any resources if the user's client device is a pda. In other words, the instantiation of the element has been made dependent on the truth value of the adaptation condition (which is woven into the element query). The "Movies directed" relationship query is also extended with an adaptation condition, stating that the user's age should be equal to or higher than the minimum allowed age for the movie.

This example clearly shows that by using an expressive and versatile extraction language like SeRQL, arbitrary adaptation conditions can easily be woven into any selection query in the AM. However, there are several drawbacks to this approach, resulting from the inherent way adaptation is specified in the global Web application design:

- **Adaptation is being implemented locally**: In the above AM fragment, we restricted the view of pictures of directors to non-pda users. However, device-dependency is only fully supported if all pictures in the entire Application Model are restricted in the same way. Although certainly possible with this approach, it requires adding the same adaptation conditions to the queries of all attributes that represent a picture in the AM; this forces the designer to manually identify and alter all these attributes. Clearly, this drawback results from the cross-cutting nature of adaptation concerns.
- **Adaptation is being hard-coded**: Each picture query has to be altered to include the necessary adaptation conditions. Therefore, the same identification of relevant elements and changes to their respective queries (see above) has to be made every time the AM is changed, for example when a new concept is added (e.g. games, songs, books etc).
- **Global (structural) information is not usable**: Adaptation conditions are limited to one particular element query, and cannot take into account structural information present in the AM. For example, expressing that pictures can only be shown to small-screen users if the containing Unit is relatively small (e.g. only contains five other elements) is impossible with this approach, because AM element queries can only query content or context data (furthermore, SeRQL does not contain a construct that enables you to count elements directly).
- **Semantic information is ignored**: Valuable semantic data about domain concepts is available in the Domain Model. For example, using media type information could save the designer from having to find all picture elements manually, which is obviously labor-intensive and leaves room for human error.

It can also be seen that by manually changing element queries for each adaptation concern, the adaptation engineering is being intertwined with the regular Web engineering; this clearly complicates the overall design process. Obviously, adaptation engineering requires a more systematic and high-level approach, one that can deal with all these problems.

1.3 Approach

An observation similar to our problem statement has already been made in the programming community, where it was found that many design concerns could not be clearly captured by existing programming techniques. Consequently, their implementation had to be spread across the program code. A clear example is logging of program execution, where the implementing code is typically spread over the entire application. Such concerns are called cross-cutting. Gregor Kiczales introduced Aspect-Oriented Programming (AOP) in [2] as a paradigm to solve this problem, which consisted of using components called aspects to encapsulate these concerns, and thus separating them from the rest of the program code. We propose to use the same paradigm to separate adaptation concerns from the standard Web design cycle, thus tackling the cross-cutting nature of these concerns. We will use Hera-S to illustrate our approach.

In our problem statement, we mentioned that valuable semantic information is currently being ignored when adding adaptation to Hera-S. A significant part of our approach will therefore consist of using such metadata to perform more fine-grained and complex adaptation. This means using semantic information about domain concepts to help select specific elements on which to perform adaptation. Information on the global structure of the Web application design (such as the number of elements contained inside another element, navigation from one element to another, etc) can also be used for this purpose. Using such (implicit) data will allow us to specify adaptation in a declarative way, which avoids us having to explicitly specify the AM elements affected by the adaptation concern. It also tackles the hard-coded drawback mentioned in our problem statement (i.e. making the adaptation specification more robust). Furthermore, by using global design information, adaptation specifications can be made independent of the specific application design.

The next chapter handles background and related work. Chapter 3 extensively covers our custom-made aspect language, while chapter 4 discusses a case-study of our approach. Finally, chapter five contains a conclusion and future work.

Chapter 2: Background and Related Work

2.1 Web

In 1945, Vanavar Bush proposed the "Memex"[5] as a response to the increasing quantity of scientific literature. He described it as a mechanical desk linked to an extensive archive of microfilms, displaying books, writings or any given document, that was able to automatically follow references from any given page to the specific page referenced. Although practically unfeasible, it gave rise to the idea of non-linear text. Later, in the mid 1960s, Ted Nelson coined the term hypertext[6] for traversing text in a non-linear way, using links to navigate between the different text pages. The early 1980s saw a number of experimental hypertext and hypermedia[7] programs; however, none of these systems achieved widespread success or name recognition with consumers (the exception being Apple Computer's HyperCard[8] application, released in 1987).

Tim Berners-Lee, while being an independent contractor at CERN from June to December 1980, proposed a project based on the concept of hypertext, to facilitate sharing and updating information among researchers. During his time there, he built a prototype system named ENQUIRE. In March 1989, Tim Berners-Lee wrote Information Management: A Proposal, in which he referenced ENQUIRE and described a more elaborate information management system; with help from Robert Cailliau, he published a more formal proposal for the World Wide Web on November 12, 1990. The successful combination of hypertext and Internet technology is viewed by many as the reason for the World Wide Web's breakthrough.

The early Web was primarily meant for researchers and had little facilities for security, content presentation or programming. Because of this, the HyperText Markup Language (HTML[9]) began not as a formal specification, but more as a collection of tools to solve an immediate problem, namely the communication and dissemination of ongoing research among Tim Berners-Lee and a group of his colleagues. However, as the need for improved layout and graphical support grew, new features were added to HTML to improve the general appearance of Web sites. Because of the increased popularity of the Web, this ad-hoc adding of features became unfeasible and more formal language specifications were required. Consequently, Berners-Lee founded the World Wide Web Consortium (W3C[10]) at the Massachusetts Institute of Technology in 1994, which comprised various companies that were willing to improve the quality of the Web by creating Web language standards and recommendations.

[5] http://en.wikipedia.org/wiki/Vannevar_Bush
[6] http://en.wikipedia.org/wiki/Ted_Nelson
[7] The term "hypermedia" is used when links are being represented by media other than (or in addition to) text (e.g. pictures, locations on the screen, etc).
[8] http://en.wikipedia.org/wiki/HyperCard
[9] http://en.wikipedia.org/wiki/HTML
[10] http://www.w3.org/

2.2 Semantic Web

The Extensible Markup Language (XML) became a W3C Recommendation on February 10th 1998, and focused on describing data rather than merely providing facilities for displaying it. XML allowed users to define a proprietary document markup (i.e. element tags and structures in which they can appear) to fit their own specific requirements. In other words, it enabled everyone to design their own document format, and to subsequently create documents in that format. Using XML Document Type Definitions (DTDs), document structure and content could be constrained by specifying a list of legal elements. XML Schema provides even a more powerful means to define element structure, by supporting data types other than strings. The following is a simple fragment of XML marked up using a custom language:

```
<pet>
    <canBe>
        <dog />
    </canBe>
    <canBe>
        <goldfish />
    </canBe>
</pet>

<person>
    <name>Sonia</name>
    <ownsAnimal>
        <dog>
            <name>Bonnie</name>
        </dog>
    </ownsAnimal>
</person>
```

The ability to add metadata (i.e. information about data) in the form of custom XML elements enables an application, written with an understanding of these elements and the way they are structured, to properly interpret the fragment. In the above example, this means that the application has to know that the `person` element corresponds to a real-world person, that `ownsAnimal` means that this person owns an animal, that `name` gives the animal's name when it is a child element of an animal, etc. Because XML elements and structure have no intrinsic meaning of their own, this information has to be explicitly provided by the programmer; the element names provided above might as well have been random strings. XML-based reasoning tools are therefore not able to provide generic reasoning support, and XML Schemas and DTD's are more of a "format" to constrain element structure and content than a true knowledge representation.

It is obvious that specifying domain-specific data properties (like `ownsAnimal`) is unavoidable when building a reasoner tool. However, relationships like `canBe`, which corresponds to a general is-a relationship, could be moved to a higher level of abstraction. As an example, imagine a single person wants to find all lonely people to increase his/her chances of finding a date. He/she could assume that, if a person owns a pet, that person is not lonely. A generic reasoner, programmed with an explicit knowledge of what people and pets are, could deduce from the statement `"person ownsAnimal dog"` that the specified instance of person isn't lonely, because `dog` is-a `pet`. In this setting, the application domain (and what information is needed) still has to be defined by the programmer, but extracting the answer from a dataset (conforming to the specified domain) can be done by using generic tools.

This observation resulted in the development of the Resource Description Framework (RDF); a specification of the RDF data model and XML syntax were released as a W3C Recommendation in 1999. RDF Schema (RDFS[11]) allows people to explicitly define application domains as ontologies using subclasses, subproperties, types, domains and ranges. OWL (Web Ontology Language) provides an extension of RDFS, by supporting cardinality constraints, more property characteristics (e.g. inverse, transitive, etc), logical conjunction/disjunction/negation of classes, etc.

2.3 Adaptation/Adaptive hypermedia

Since Ted Nelson coined the term hypertext in the mid 1960s, there has been extensive research in the field of non-linear text. Adaptive hypermedia is a research field combining work from both hypermedia and user-modeling research. Its goal is to design and develop hypermedia systems that adapt to the needs of a specific user, based on a model that comprises the user's needs, preferences and characteristics.

According to Brusilovsky [3], 1996 can be considered a turning point in adaptive hypermedia research. This is mostly due to the rapid increase in the use of the World Wide Web at that time; because of its widely diverse audience, it had a clear demand for user-dependent behavior. This is illustrated by the fact that almost all papers before 1996 discuss classic pre-Web hypermedia, while the majority of papers published since 1996 are devoted to Web-based hypermedia adaptation. Traditionally, adaptation decisions were based on user characteristics (e.g. age, gender, sexual orientation, ethnicity, etc) [3]. Since 1996 however, this situation has changed; Kobsa et al. [4] suggested extending user adaptation to include usage and environment data. Usage data is data specific to user interaction, while environment data takes the user's location and client device, browser, platform, etc into account. It can be observed that the need for adaptation to the user's environment grew parallel with the emergence of various hand-held technologies, which became full-fledged Web clients.

The techniques used in adaptation have also evolved since 1996. The original taxonomy by [3] classified adaptive hypermedia methods according to what was being adapted, and considered two distinct areas in this setting: content level adaptation (including presentation adaptation) and link level adaptation. Since then, several extensions have been made to the taxonomy. Multiple variants on link hiding (link hiding is an adaptation technique in link level adaptation, where a link is "hidden" when it is not considered desirable for the user) were introduced by [5]: link disabling (e.g. "graying out" a link) and link removal. Extensions were also made to text adaptation, as a subset of content level adaptation. Besides inserting or removing text fragments, altering text fragments, stretch-text (i.e. expanding and collapsing additional text within a page) and sorting fragments (e.g. for relevance ranking), the "dimming" of text fragments was proposed by [6], in which de-emphasis can be put on text fragments by changing their color or brightness. This kind of text adaptation is called canned text adaptation, and is considered a part of more general text adaptation. Natural language adaptation [7] has also emerged as a subset of text adaptation; it enables computers to interact with people the way people interact with each other. Natural language systems aim to generate and understand natural language, and to engage in dialogue with the user.

Besides making minor extensions to existing techniques, there was also a need for more significant changes to Brusilovski's classification [3]. Adaptation of modality was introduced

[11] http://www.w3.org/TR/2004/REC-rdf-schema-20040210/

as high-level content adaptation by Kobsa [4]. When several media (e.g. pictures, video, animation, etc) can present the same content, choices have to be made about which type of media will be used; this can be done by taking the application context (i.e. which node the user is currently visiting) and the user model into account. The rise of recommender systems (systems that try to predict items that a user may be interested in, based on information in the user's model) divided link level adaptation into two subsets: adapting links that are already present on the page (e.g. link annotation, sorting according to relevance) and generating new links. The latter can be further subdivided into three categories: discovering new useful links between documents and adding them permanently to the set of existing links, adaptive similarity-based navigation and dynamic recommendation of (added) relevant links.

As for the future, Brusilovski [3] proposes several new directions in which adaptive hypermedia could advance significantly: integration of adaptation in various existing application systems, open corpus adaptive hypermedia (automatic generation of links between documents) and adaptation support for handheld/mobile devices.

2.4 Hera-S

In conceptual modeling for Web design methods, three models are typically considered (e.g. OO-H [16], WSDM [11], UWE [15]): a Domain Model, which describes the structure of the content data, a Navigational Model, which describes hypermedia navigation over the content data and a Presentation Model that describes presentation generation. Such design methods, which discern data, navigation and presentation design, allow the designer to focus on one design issue at a time, thus reducing the inherent complexity of designing (large) Web applications.

The Hera-S design methodology concentrates on designing adaptive and personalized Web Information Systems (WIS), and uses conceptual modeling to achieve this. It comprises a Domain Model (DM), which is an abstract representation of the content data, an Application Model (AM), which defines navigational nodes (called units) grouping relevant content data and the relationships between them, and a Presentation Model (PM), describing the layout of the presentation generation. The end result of this modeling phase is a hypermedia-based view on the underlying content data.

Key feature of Hera-S is the fact that it is completely based on the Resource Description Framework (RDF), and that it uses Sesame[12], a powerful and popular open source RDF framework for this purpose. All of the aforementioned models are written in RDF: the DM is an RDFS/OWL specification serving as an ontology for the content data, the content data, AM and PM are written in RDF and the meta models for the AM and PM are specified using RDFS. These languages are part of the W3C Semantic Web[13] initiative, and make the semantics of data explicit by modeling them in a domain-specific ontology. They also allow easily using heterogeneous data sources as content data.

[12] http://www.openrdf.org
[13] http://www.w3.org/2001/sw/

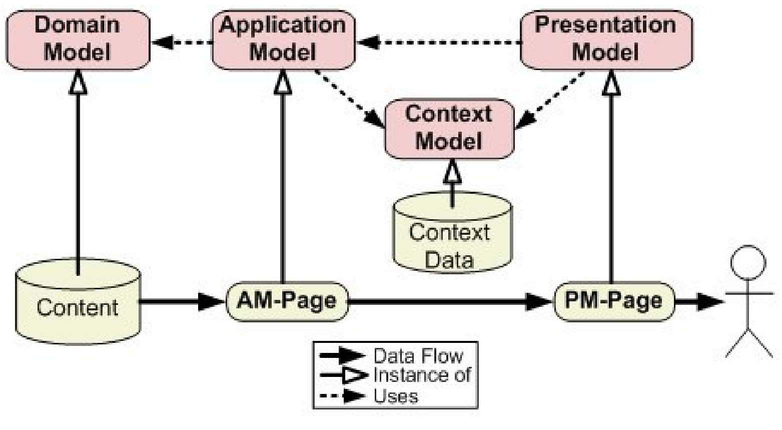


Figure 1. Hera-S architecture.

Another distinctive characteristic of Hera-S is its explicit focus on adaptation and personalization. Consequently, Hera-S also maintains a *Context Model* which keeps user specific information. An overview of the Hera-S architecture can be found in Figure 1.

In the Application Model, elements use SeRQL[14] (Sesame RDF Query Language) as a query language to retrieve relevant information from the content data. This data conforms to the Domain Model and is stored as a Sesame Repository. After executing these queries, the AM is instantiated with the results, eliciting so-called Application Model Pages (AMP's). From these AMP's the actual Web pages are generated using the Presentation Model. In e.g. [18], it was demonstrated that both proprietary and external engines can be used for this task.

The basic AM components are (Navigational) Units, Relationships and Attributes. Units group elements that will be shown together to the user. They typically correspond to a concept of the DM (e.g. movie, director, etc), but they can also refer to several domain concepts. Relationships represent a browsable link between two Units, and are based on semantic relations between underlying domain concepts (e.g. director of a movie). Units can contain Attributes and other Units called Subunits. Subunits can be compared to Relationships because they both point to a target Unit; however, Subunits allow the developer to directly include the information from the target Unit inside another Unit, instead of merely providing a link to the information. Attributes represent literal values, corresponding to strings or other URI-referable media (e.g. pictures, videos). Mostly they conform to a data-type property belonging to the corresponding concept of the containing Unit (e.g. movie title).

Each element contained inside a Unit keeps a SeRQL query, responsible for retrieving relevant data from the content data and using it to instantiate the AM element. Note that this approach allows for fine-grained and flexible adaptation to take place at the model level, by adding conditions to the WHERE clause of such a query. This approach is comparable to [12], where they describe a delivery specification language to allow for user-specific data extraction. Information can be queried from both the content and context data repositories. More AM elements exist, such as update queries (that allow updates of the context model), Form Units, guided tours, scripts, Web services, etc. Hera-S also supports frame-based navigation (cfr. frames in HTML). For more details, we refer to [19].

[14] Sesame RDF Query Language. http://www.openrdf.org/.

To illustrate the structure of a simple AM, let us consider a small example in the context of IMDB[15]. We want to present a movie by showing its title and directors, and a director by his/her name, picture and movies they directed. The resulting AM (written in Turtle[16]) would look like this:

```
:MovieUnit a am:NavigationalUnit ;
    am:hasInput [ a am:Variable ;
        am:varName “M”;
        am:varType imdb:Movie
    ];
    am:hasAttribute [
        rdfs:label “Title” ;
        am:hasQuery
        “SELECT T
        FROM {$M} rdf:type {imdb:Movie};
                rdfs:label {T}”
    ] ;
    am:hasUnit [
        rdfs:label “Director” ;
        am:refersTo :DirectorUnit ;
        am:hasQuery
        “SELECT D
        FROM {$M} rdf:type {imdb:Movie};
                imdb:movieDirector {D} rdf:type {imdb:Director}”
    ] .

:DirectorUnit a am:NavigationalUnit ;
    am:hasInput [ a am:Variable ;
        am:varName “D” ;
        am:varType imdb: Director
    ];
    am:hasAttribute [
        rdfs:label "Name" ;
        am:hasQuery
        “SELECT L
        FROM {$D} rdf:type {imdb:Director};
                rdfs:label {L}”
    ] ;

    am:hasAttribute [
        rdfs:label "Photo" ;
        am:hasQuery
        “SELECT Ph
        FROM {$D} rdf:type {imdb:Director};
                imdb:hasMainPhoto {Ph}”
    ] ;
    am:hasRelationship [
        rdfs:label “Movies directed” ;
        am:refersTo :MovieUnit ;
        am:hasQuery
        “SELECT M
        FROM {$D} rdf:type {imdb:Director};
                imdb:directorFilmography {M}”
    ] .
```

[15] http://www.imdb.com
[16] http://www.dajobe.org/2004/01/turtle/

The example contains two Units: MovieUnit and DirectorUnit. The MovieUnit contains an Attribute that represents the movie title and a Subunit, which represents the Director(s) of that movie. Note that we used a Subunit and not a Relationship, because we want the director's information to be shown together with the rest of the movie details. DirectorUnit contains two Attributes, one representing the director's name and the other the director's main photo, and a Relationship to his/her directed movies. We chose to use a Relationship here, because it would be superfluous to show the full details of all directed movies when the user is only interested in the director's details.

2.5 Other Adaptation Approaches

Adaptation in hypermedia is mostly achieved by adding adaptation conditions to the regular hypermedia design. Depending on the satisfaction of these conditions (mostly based on context-dependent information), content will be either shown or hidden, or their presentation changed. The remainder of this section is structured as follows. First, several approaches are discussed which follow the ECA (Event-Condition-Action) paradigm and uses rules to implement adaptation. Subsequently, we discuss an approach that adds conditions to data-extraction queries to personalize the Web site generation process. Finally, two existing approaches that use aspect-orientation to include adaptation are discussed.

2.5.1 AHA!

A representative example is AHA! [5], a generic adaptive hypermedia platform that is mainly meant to be used as a software platform by researchers to support their research in adaptive hypermedia. AHA! allows presentation style adaptation, conditional inclusion of objects and adaptive link destinations.

The conceptual model consists of concepts and attributes that have rules associated with them. Every time a concept is requested, the rules corresponding to the "access" attribute of the requested concept will be executed. These rules can change other attribute values, leading to a possible cascade of rule executions (because an attribute value update is also considered an event). Note that this approach (albeit implicitly) corresponds to the Event-Condition-Action (ECA) paradigm; e.g. when the rule associated with the "access" attribute is executed, it adds 1 to the "visited" attribute value (event), of which the associated rule states that when the value > 0 (condition), the "suitability" attribute of another concept can be set to true (action).

Depending on the value of the "showability" attribute of the requested concept (after the rule executions), it will be decided which page (as a concept can have several pages associated with it) will be shown. Page fragments can be inserted conditionally, by specifying an associated condition. Objects (i.e. concepts) can also be inserted conditionally into a page. This happens by first executing the "access" attribute rule of the concept, and then using the value of the "showability" attribute of the concept to determine which page will be included.

The presentation of a fragment can be changed by taking into account the value of the "visibility" attribute of a specified concept, and matching it to a predefined CSS class. When annotating links, an attribute (e.g. "suitability" or "visited") of the target concept can be used to determine the link annotation (e.g. suitable, neutral, not suitable) using colors, icons, etc. Adapting the link destination can be done in two ways (corresponding to "forward" and "backward" reasoning, respectively). By using conditional inclusion of fragments, the destination of a link can be made dependent on the associated conditions of the fragments (and thus having the correct link before the page is shown, i.e. forward reasoning). By using the "showability" attribute of the target concept, it can be decided which page associated with the concept is to be shown when the link is clicked (thus only making the decision after the page is shown, i.e. backward reasoning).

2.5.2 ASL

WSDM also uses an event-based language [21] to express adaptation concerns in their navigational model, called the Adaptation Specification Language (ASL). This language consists of condition-rule pairs: when a certain condition holds, the specified rule will be executed. However, ASL can still be considered to follow the ECA paradigm; because the functions used in a condition are dependent on user access patterns (e.g. numberOfVisits, numberOfTraversings), they have to be evaluated every time the associated event occurs (e.g. node is visited, link is traversed). A condition can be specified for just one element or for multiple elements using the `forEach` construct. In the rules, nodes can be added or deleted, links can be created or removed between nodes, etc.

2.5.3 PRML

It is noted in [9] that, although many web design methods include similar adaptation techniques that solve similar problems, their implementations can differ widely and their adaptation specifications are therefore not portable. This paper proposes a generic personalization specification language, which can be mapped to any web design method that distinguishes a Domain Model (abstract representation of the content data), a Navigation Model (defines de structure and behavior of navigation over the content data) and a Presentation Model (defines the layout of the generated hypermedia presentation) in the design process, which is mostly the case. They propose a generic language called Personalization Rules Modeling Language (PRML) for this purpose. Rules are defined in this language using the ECA [10] paradigm, and the language supports both navigational and content adaptation.

2.5.4 Torii

Another example is the Torii tool [12], developed within the W3I3 project for supporting data-intensive Web applications. Its conceptual model is based on the Entity-Relationship model, consisting of entities, attributes and relationships. The model is written in an XML-based conceptual modeling language, and later mapped to a DOM representation. A description of user-specific properties is kept in a "profile", and each user has an instance of a profile associated with him/her. Besides containing user-specific information such as e-mail, address, etc, each user can also be associated with a "group"; based on which group the user is in, he/she will get a different view of the site (called a "site view"), both in presentation style and content.

For more fine-grained adaptation a "delivery specification language" called ToriiDL [17] is used in their conceptual model, which allows the developer to specify user-specific data extractions (e.g. to retrieve the value of an attribute element). Based on the truth value of the specified condition (using both context and content dependent information) it will be determined what the user will (not) see. Differently put, they execute a query that only extracts relevant information for the specific user, based on information from context and content data. User classification (into groups) and user model updates are executed using rules (following the ECA paradigm); each of these are triggered by an event (like logging in or visiting a page) and contain an action that will be executed when the event is triggered and if the specified condition is satisfied. These events can also be used to "publish" information to users, e.g. to recommend certain books when the user visits a certain web page about a related book.

2.5.5 UWE extension

In [13], adaptation is modeled by applying aspect-orientation to a Web design method. For this purpose, they propose a lightweight extension to UWE (UML Web Engineering). Only navigational adaptation is considered, and three techniques within this category: adaptive link hiding, adaptive link annotation and adaptive link generation. They view an adaptation technique as an aspect in its own right, instead of considering an adaptation requirement as an aspect. Their method also consists of exhaustively entering each desired element in the pointcut part of an aspect (i.e. the part that selects relevant elements from the design); they do speak briefly of including wildcards as future work.

2.5.6 HAL

Aspect-orientation was also used to express adaptation in a previous version of the Hera design methodology [1]. We based ourselves (albeit loosely) on this approach. They used the High-level Adaptation transformation Language (HAL) to express pointcuts and advices on a Hera application model. A generic query mechanism was used to express pointcuts; contrary to the above mentioned approach, model elements didn't have to be manually entered one by one. Advices were applied to application model elements by attaching conditions to them; depending on the satisfaction of these conditions (based on context information) the element would be either hidden or shown.

As opposed to our approach, the implementation of the aspects was done on the instance level. At each page request, instance data was retrieved (based on the application model specification); this data was then filtered by mapping the adaptation conditions to rules, and subsequently executing these rules during data retrieval (using a third-party rule-based adaptation engine called the GAC [14]). However, this approach resulted in too much data being retrieved: data that would be filtered out anyway by the adaptation rule(s) was still retrieved at every page request, yielding unnecessary overhead. Moreover, it only supported filtering of data; in a pure filtering approach, adding/replacing/removing elements at the model level isn't possible (it is mentioned in [1] that this could have been bypassed by creating a feed-back loop with the Hera engine, but that it would have lead to more loss of performance).

2.6 Aspect-Orientation

In software engineering, the Aspect-Oriented Programming paradigm (AOP) helps programmers in the separation of concerns (more specifically cross-cutting concerns) as a next step in modularization. Separation of concerns means breaking down a program into distinct parts that overlap in functionality as little as possible. All existing programming methodologies (e.g. procedural and object-oriented programming) support some form of separation and encapsulation of concerns into single components (e.g. procedures, packages, classes and methods); however, some concerns cannot be encapsulated using these paradigms. These concerns are called cross-cutting, because they cut across different components in a single program. Logging of program execution is a typical example of a cross-cutting concern, because it necessarily affects every logged part of a system. Gregor Kiczales and this team at Xerox PARC introduced AOP in [2], and also developed the first and most popular AOP language called AspectJ[17], emphasizing simplicity and usability for end-users. On the other hand, IBM's research team offered the more powerful (but less usable) HyperJ[18] and the Concern Manipulation Environment[19], which have not seen wide usage.

All AOP languages contain some expressions to encapsulate cross-cutting concerns in a single location; therefore, the difference between AOP languages lies more in the power, safety and usability of the provided constructs. For example, "interceptors" specify which methods should be intercepted, and express a limited form of cross-cutting without much support for type-safety or debugging. AspectJ has a number of such expressions and encapsulates them in a special Java class called an aspect. An aspect can alter the behaviour of the base code (i.e. the non-aspect part of the program) by applying an advice (additional behaviour) at various join-points (specific points in a program), which are specified in a pointcut. In other words, a pointcut specifies at which join-points the advice should be applied. An aspect can also make structural changes to other classes, like adding members or parents.

Examples of join points are method executions and field references. The developer can write a pointcut to match specific join points, e.g. all field-set operations on specific fields, and then specify in the advice what code to run when the specified fields are being set by the client code.[20]

2.7 Comparison with existing approaches

An aspect-oriented approach to adaptation engineering has already been attempted in [13]. However, their method has several drawbacks (see 2.5.5). For one, they consider an adaptation technique to be an aspect in its own right; later it will be shown that we consider an adaptation requirement as an aspect (which can then be further specified using adaptation techniques). Our approach can thus be considered to use a higher level of abstraction. Aspect-orientation was also used to express adaptation in a previous version of Hera (see 2.5.6). The implementation of the aspects was done on the instance level (by filtering the retrieved data using the specified conditions), which lead to a significant overhead when retrieving data. Our approach consists of implementing aspects on the model level (by adding them to the SeRQL queries), and therefore the returned data already conforms to the given conditions. Moreover, this means that changes can be made to the model without any significant overhead.

[17] http://en.wikipedia.org/wiki/AspectJ
[18] http://www.alphaworks.ibm.com/tech/hyperj
[19] http://www.research.ibm.com/cme/
[20] http://en.wikipedia.org/wiki/Aspect_oriented

Chapter 3: Aspect language

As mentioned in the introduction of this thesis, adaptation concerns are typically spread across the Web application design. In the programming community, it was also observed that some design concerns could not be localized to a single procedure or object. In order to maintain good abstraction and modularization, aspect-oriented programming [2] was introduced, where these (cross-cutting) design concerns are encapsulated in a component called an aspect. An aspect consists of a pointcut and an advice. The pointcut selects the location(s) where the cross-cutting concern should be applied to, while the advice specifies which action should be taken (i.e. what piece of code should be injected), thus avoiding the implementation of the concern to be unnecessarily duplicated or scattered across the code.

This paradigm could also be applied to adaptation engineering; a piece of code, i.e. the advice, is duplicated (or scattered) in multiple places in the Web design, i.e. the pointcut. In our case, the pointcut will consist of a query over the Application Model, selecting the elements to which the adaptation concern applies, while the advice will typically consist of inserting adaptation conditions into places specified by the pointcut. However, we will not limit ourselves to adding conditions; arbitrary transformations of AM elements will also be possible, where elements can be added, replaced or removed based on a context- or user-dependent condition.

For this purpose, we have developed our own custom-made Aspect Language called SeAL (**Se**mantics-based Aspect-Oriented **A**daptation **L**anguage) to provide for generic adaptation support in Hera-S. The most important benefits of using a custom-made language (as opposed to a general-purpose language) are reduced complexity and ease of use: SeAL constructs take into account the structure of an AM as dictated by the AM Metamodel, thus allowing for a more high-level approach to querying these models. Moreover, a (compact) domain-specific language may be easier to understand and use than a complex and extensive general-purpose language. In the following sections, we discuss the available SeAL constructs and their semantics; how to realize adaptation concerns in this language will be the subject of the next section. After discussing the available constructs, we motivate (and defend) our choice for a custom-made language more extensively.

3.1 Pointcut

Pointcuts select the elements in the AM relevant to the adaptation concern, i.e. to which elements the advice should be applied to. By default all elements in the AM are selected (i.e. units, subunits, attributes, relationships, queries and form elements); all subsequent language constructs restrict/condition this selection in some way. The components of the pointcut language are therefore conditions that place restrictions on AM element properties, and can use information from the AM for this purpose (e.g. element type, name, (string) value, property value, etc) while some conditions can query additional information from the DM and CM. The following subsections consider these two cases.

3.1.1 Application Model

The following AM element information can be used to restrain the element selection: type, name, (string) property value, aggregation (an element `contains` another element, or is `containedIn` another element) or an aggregate function (placing a numerical condition on the amount of contained elements, specified using `count`). String pattern-matching is allowed when specifying an element’s name, type or property value. The nesting of conditions is also possible, allowing for a more fine-grained selection of elements. For example, this enables you to select all elements contained within another element that satisfies certain conditions. All supported conditions are summarized below:

1. `hasType “<type>”`: selects all elements of the specified type;
2. `hasName “<name>”`: selects all elements with the specified name;
3. `containedIn (<conditions>)`: selects all elements contained within other elements that satisfy the nested conditions;
4. `contains (<conditions>)`: selects all elements containing other elements that satisfy the nested conditions;
5. `contains ((<numericCondition>)? count (<conditions>) (<numericCondition>)?)`: selects all elements containing a certain number of other elements (specified by the given range) that satisfy the nested conditions;
6. `from (<conditions>)`: selects all Relationships that originate (i.e. navigate) from the elements conforming to the nested conditions;
7. `to (<conditions>)`: selects all Relationships that target (i.e. navigate to) the elements conforming to the nested conditions;
8. `navigateFrom (<conditions>)`: selects all elements that a Relationship targets, navigating from the elements satisfying the nested conditions;
9. `navigateTo (<conditions>)`: selects all elements where a Relationship originates from, navigating to the elements satisfying the nested conditions;
10. `hasInput (“<binding>” | [<dmExpression>] | [<SeRQL> “<dmQuery>”])`: selects all elements that have a variable of the given type. This type can be specified either explicitly by using a string compare or by referencing the Domain Model (see 3.1.2);
11. `hasLabel “<label>”`: selects all elements containing the specified label;
12. `hasSource “<source>”`: selects all elements that have the specified source Unit (i.e. the element the target Unit will be loaded into in case a Relationship is followed, cfr. frames in HTML);
13. `hasTarget “<target>”`: selects all elements with the specified “target”, i.e. the value of the “refersTo” property in the AM;
14. `hasOption “<option>”`: selects all choice form-elements with the specified option;
15. `hasQueryLang “<queryLang>”`: selects all query objects with the specified query language.

Note that conditions 7 and 8 allow you to traverse the navigational structure defined by the Relationships: for example, `navigateFrom (hasType unit and hasName ”*movie*”)` allows you to select all Units that a Relationship navigates to, which originates *from* a Unit with name “*movie*”. Therefore, it allows you to restrict the element selection using global (navigational) information on the AM, by constraining the allowed navigation towards the selected Unit. As can be seen from condition 4, the number of elements that are contained within another element can also be counted. This allows you to select elements that only contain a certain amount of elements, which is particularly interesting in the context of adaptation: for example, in case the visitor uses a small-screen device (like a pda), all Subunits that have more than five Attributes could be selected and subsequently replaced by Relationships to the Unit. Note that the two aforementioned conditions implement one of our goals formulated in our approach (see 1.3), namely to use global information on the AM to achieve a more systematic and higher-level approach to adaptation engineering.

Conditions may be negated or combined using logical conjuction and disjunction, and parenthesis may be used to alter default logical operator precedence. Some construct-specific shortcuts are also available (see first example below, where a comma is used to indicate logical disjunction in the `hasType` construct). Typical examples include:

- `hasType unit, subUnit;` (selects all elements of type Unit or Subunit);
- `hasName “*movie*” ignoreCase;` (selects all elements that have a name that contains “movie” while ignoring case);
- `hasLabel “Main*”;` (selects all elements that have a label that starts with “Main”);
- `hasType subUnit and contains (2 < count(hasType attribute) < 5);` (selects all Subunits that have between 2 and 5 Attributes specified);
- `hasType attribute and containedIn (hasType unit and contains (hasType attribute and hasLabel “Title”));` (selects all Attributes in a Unit that contains an Attribute with label “Title”);
- `hasType relationship and from (hasType subUnit) and to (hasType unit);` (selects all Relationships originating from a Unit that occurs as a Subunit and that targets a Unit).

It can be observed in the last example that Subunit types can also be used to refer to their corresponding Unit (i.e. the Unit the Subunit refers to). Indeed, a Subunit can never contain a Relationship in a valid AM, while the corresponding Unit obviously can. Also, information from the Subunit element itself (e.g. label, element query) as well as the corresponding Unit (e.g. Unit name, type, contained elements, etc) can be constrained whenever a Subunit type is specified. Although this may seem confusing at first, note that Subunits do not allow you to explicitly navigate towards a Unit (like Relationships), but actually represent an in-page occurrence of the Unit. Therefore, it can be conceptually valid to also consider the corresponding Unit when selecting a Subunit. Moreover, it provides for an easy way to select Units that occur as Subunits; in practice, this seems to be particularly interesting with regards to adaptation. For example, one could easily restrict the number of elements in a Unit that occurs somewhere as a Subunit, in case the visitor uses a small-screen device like a pda:

```
hasType subUnit and contains (2 < count(hasType attribute) < 5);
```

Additionally, we consider implementing more explicit `represents` and `representedBy` constructs as future work.

As shown in the first example, the selection of elements can be restricted according to the element’s type using the `hasType` construct. Element types include unit, subUnit, relationship, attribute, query and form-element. However, some of these types can be further restricted: for example, a Unit can be either a Navigational Unit or a Form Unit, and a Subunit can be aggregated inside a Unit as either a set of Subunits, a tour of Subunits (as a so called “guided tour”) or as a single Subunit. Ideally, such types would be expressed using a single keyword. However, this would result in an exponential growth in keywords if further subtyping is possible, and leads to long and complex keywords: for example, `subUnitSetNav` would indicate all sets of Subunits that refer to a Navigational Unit. Therefore, it was chosen to use a tree-like structure to express subtypes in a more concise and expressive way. Typical examples include:

- `hasType unit > form, subUnit > set`; (selects all Form Units, and Subunits that occur as a set);
- `hasType subUnit > set | tour`; (selects Subunits that occur inside a Unit as either a set of Subunits or as a tour of Subunits).

If a further specification is not given, all elements of the given supertype are returned by default. As a shortcut, these so called “type-trees” can be seperated using comma’s (corresponding to a logical disjunction, as can be seen in the first example). Inside the type tree, logical disjunction is expressed using the “|” symbol (to avoid ambiguity with the comma’s seperating the type trees, as can be seen in the second example).

SeAL also supports calls to the native underlying query language (in our case SeRQL) using the `<SeRQL>` construct. This support has been provided so that the expert user can still specify intricate queries that exploit the full expressiveness of SeRQL (and of which the semantics are not expressible in SeAL). For the full pointcut language reference, we refer to Appendix A.

3.1.2 Domain Model

Until now, only information from the Application Model has been considered when restricting elements. Naturally, this follows from the fact that we are selecting elements from the AM, and using AM information for this purpose is an obvious first step. However, as stated in our approach (see 1.3), one of the goals was to exploit semantic information to allow for a more systematic and high-level adaptation process. This information is readily present in the Domain Model; therefore, we provide for a hook in SeAL that allows the user to query for this semantic information using SeRQL queries. For example, to select all Subunits whose corresponding domain concept (e.g. person, movie) is connected to an age rating:

```
hasType subUnit and hasInput in (<SeRQL> SELECT I FROM
{imdb:hasAgeRating} rdfs:domain {I})
```

This condition selects elements with (an) input variable(s) that appear(s) in the domain of the `imdb:hasAgeRating` property. In this example, it is clear that the selection of elements can only occur at runtime, namely when querying the DM; it will only be known at runtime (and not at specification time, as is the case with pointcut conditions that only use information from the AM) which elements will or will not have an “hasAgeRating” property. It should be noted that it is also possible to specify the above pointcut using only information from the AM:

```
hasType subUnit and contains (hasType attribute and hasLabel
“*age*rating*” ignoreCase);
```

However, it is obvious from this example that it places a burden on the adaptation engineer: he/she has to make sure that an Attribute with a specific label is present at the places where the advice has to be executed, and therefore he/she has to know the places where adaptation is needed. This clearly implies that the designer has to design the AM is such a way to make sure that the above aspect would work. Using semantic information allows a more declerative approach, and alleviates this burden from the designer. It is also much more robust: when adding resources to the Web application on (and therefore new Units describing these resources), the adaptation pointcut can directly identify these new elements via their concepts in the DM, without the designer having to manually find them.

In practice however, it seemed that using SeRQL queries to extract semantic information had some significant drawbacks. In OWL DL, it is common to use unions of classes to specify domains or ranges of properties; for example, to specify that the property "hasAgeRating" can have either a Movie *or* a Game as its domain, a union containing both these classes has to be defined as the property's domain (if these domains are defined seperately, the overall domain will be considered the intersection of the seperately specified domains). Because the elements of a union are enumerated using an RDF collection, specifying a SeRQL query for a property domain or range becomes very difficult, if not impossible:

```
SELECT I FROM {imdb:hasAgeRating} rdfs:domain {} owl:unionOf {}
rdf:first {I}
```

In this example, only the first element of the collection is considered; the developer has to make a separate query to try all elements from the collection. There is no known construct (or method) in SeRQL that allows you to do this in one query, or to even know the number of elements beforehand. Therefore, it was decided to develop a (compact) extension to SeAL that allows you to reference the domain and range of a property in a more concise and high-level way.

Also, we argue in one of the following sections (see 3.3) that an important reason for choosing a custom-made language to specify pointcuts was that we can exploit the fact that only AM RDF repositories (i.e. repositories that conform to the AM Metamodel) are being queried. In this SeAL extension, we can additionally exploit the fact that we will mostly be querying the domain and range of properties in our DM queries: for instance, in our age-restriction example, we have to query whether the Unit's concept is in the domain of the "hasAgeRating" property. In SeRQL this would be:

```
SELECT node2 FROM {imdb:hasAgeRating} rdfs:domain {<node2>}
```

As mentioned before, this does not take into account possible collections used in the domain; moreover, we can assume that in our case, mostly the domain and range properties of resources will be queried. Consequently, we write down the above example in our SeAL extension as follows:

```
SELECT node1 FROM {<node1>} {imdb:hasAgeRating} {<node2>}
```

Every subject (i.e. node1) in the triple is queried as (part of) the domain of the property, while every object (i.e. node2) is queried as (part of) the range of the property, taking into account the fact that the domain or range can contain a collection. The only exception is the `rdf:type` property, which is handled in the same way as it would be in SeRQL. As can be seen from the example, the syntax of this language extension roughly corresponds to that of a simple SeRQL expression. Like in SeRQL, branching (when specifying multiple properties of a single subject) and chaining (when the object of a triple is the subject of another triple) of nodes are possible, while only typed variables (and of course the variable in the SELECT clause) and URI references may occur in a path expression. Some basic inferencing is also supported: subclasses and subproperties are automatically taken into account when querying the domain or range of a property.

As with an ordinary SeRQL query, the values found for the variable in the SELECT clause will be returned. These are then compared to the binding variable types of the elements selected in the pointcut expression (i.e. the concepts corresponding to the elements). Elements that contain a variable whose type occurs in the results will satisfy the condition.

As is the case when selecting elements from the AM, users may want to query the DM in a more extensive way. Therefore, a hook is provided to enter SeRQL queries that are executed directly on the Domain Model, by using the `<SeRQL>` construct. It has also been mentioned previously that when multiple domains (or ranges) are defined seperately, the resulting domain will be considered as the intersection of the seperately specified domains. This means that if a resource occurs as the subject of this property, it has to be an instance of each of the classes in the separate domains at the same time. These semantics are not supported by the SeAL extension. For the full reference of this extension, we refer to Appendix C.

3.1.3 Instance level

An obvious next step is to use information from the content data as well. Querying the Domain Model allows you to check for relationships that could exist, while querying the content data allows you to check for relationships that actually do exist. This can be interesting when, for example, your dataset is incomplete and only contains age-rating information for a limited amount of movies. Instead of querying the DM for the possible existence of the "hasAgeRating" property for the "movie" resource type, it could be checked whether the specific resource used as input for the Unit element at runtime (obtained from executing a Relationship or Subunit query linking to the Unit, see 2.4) actually participates in such a relationship. Obviously, this would imply executing each aspect (of which the pointcut uses instance information) at runtime and upon page-request, because the element queries have to be executed before the specific input resource for the Unit can be known. Because of time constraints, support for this was not implemented; this could be done in future work.

3.2 Advice

While a pointcut selects the elements the adaptation requirement applies to, an advice contains the actions implementing the adaptation requirement. These actions can include adding conditions to an element (making the instantiation of the element dynamic) and/or conditionally adding, removing or replacing elements from the Application Model.

3.2.1 Adding conditions

Adding conditions to elements is a typical way of adapting. Depending on the truth value of these conditions the element will be either shown or hidden, and by referencing the Context Model in such conditions, the Web application can be made dependent on the current user or context. In Hera-S, this can be done by adding conditions to the associated SeRQL query of an AM element; after instantiating the AM by executing these queries (resulting in so called AM Pages, see 2.4), only element instances for which the query conditions returned true will be shown.

In SeAL, such a condition can be specified in the advice by using the "."-operator to navigate through the respective models, resulting in so-called navigation paths. Navigation paths consist of predicate names, which can be used to navigate through the content/context data towards a specific object value. This value can then be compared to a constant or another navigation path value. A navigation path always starts with a namespace suffix, denoting the model which is currently being navigated: "cm:" selects the Context Model, while other namespace-suffices specify ontologies present in the Domain Model (in our case “imdb:” is used). When navigating through the Context Model, the current user node will always be used as a starting point; in the Domain Model, the (content data) resources corresponding to the AM elements selected in the pointcut (i.e. the resources resulting from the execution of the element queries) will be used for this purpose. An example is given below:

> `addCondition cm:age >= 18` (adds a condition to the elements selected in the pointcut, stating that the elements will only be shown if the user is 18 or above, i.e. he/she is not a minor)

A better example is to actually compare the current age of the user with the minimum age of an age-restricted object. The following example does this by comparing object values resulting from different navigation paths:

> `addCondition imdb:hasAgeRating.minAllowedAge >= cm:age` (adds a condition to the selected elements, stating that the age of the current user should be higher than or equal to the minimum allowed age)

Note that in the left part of the comparison, navigation starts from the selected element concepts, while in the right part navigation through the Context Model is specified for the current user. Conditions can be combined using logical conjunction, disjunction and negation, and parenthesis can be used to alter standard operator precedence. However, his notation is insufficiently expressive in some cases: consider the following example, where we do not want to show elements for which the current user has given a low rating. Using only the constructs introduced above, this would result in:

> `addCondition cm:givenRating.about != imdb or cm:givenRating.rating > 5` (adds a condition to the elements selected in the pointcut, stating that either the user has not yet given a rating to the resource, or that the user gave a rating over 5)

However, it can be clearly seen that above condition is flawed, because it states that the user either never rated the resource or that a rating (for any element) over five was given: the rating could belong to a completely different resource! To express conditions where one resource has to be linked to another, the following notation is used:

> `addCondition not cm:givenRating.about = imdb; ratingValue < 5` (this condition states that either the user has not yet entered a rating to the resource, or that the given rating was not below 5 for the selected resources)

The semantics of this construct correspond to a “branch” in SeRQL, i.e. a property can be stated about a resource, after which another property about the same resource can be given, without explicitly repeating the resource as the subject of the property (in SeRQL, this is also denoted by a semi-colon). It can also be seen from the above example that it is not mandatory to specify property names after a namespace: this allows you to directly reference the output resources of the query. For example, the above condition says that if the current user has given a rating to the resources specified by the query (e.g. movies an actor was in), that rating must be higher than 5.

Also, it is implied in the example that only one navigation path value has to satisfy the comparison, i.e. that only one value from `cm:givenRating.about` has to equal the output resource(s) from the query. In some cases, this may not be preferable: for example, when you only want to show movies of which *all* the features (e.g. Dolby DTS, HD, etc) are supported by the user's hardware:

```
addCondition  imdb:moviePropertySupport  =  cm:hasMovieEquipment.
imdb:moviePropertySupport
```

In this example, all values resulting from the navigation path on the left hand side have to satisfy the comparison (for at least one navigation path value on the right hand side). Adding support for this is future work. Also, it can be seen that in the second navigation path, a separate namespace is specified for the `imdb:moviePropertySupport` predicate. Obviously, it is possible that several namespace prefixes are used in the Domain Model; therefore, we allow different namespaces to be specified for each property. Note the starting point for the navigation path always depends on the *first* namespace, i.e. if we had put "imdb:" as the first namespace in the second navigation path instead, the starting point would have been the output resources from the element query instead of the current user node.

To conclude, it is noted that the operands of the comparison can occur in any order, e.g. the first example from this subsection can also be written as

```
addCondition 18 <= cm:age
```

Or indeed, any arbitrary comparison can be made (containing integers, real numbers or strings):

```
addCondition 5 = 4;
```

3.2.2 Adding/removing/replacing elements

New elements can be added to the elements selected in the pointcut, existing elements selected in the pointcut can be removed, or (parts of) the elements can be replaced if a certain condition (specified in the same way as above) is fulfilled. Because the conditions are dependent on content- or context-data, these advices (containing an addition, removal or replacement) have to be executed at runtime *upon page-request* (because context and content information can be updated after visiting another page, by executing update queries), while the advices adding conditions to elements only have to be performed at design-time. Note that this kind of adaptation is not possible in the current condition-based approach in Hera-S, which can only add conditions to elements. In the below example, the elements selected in the pointcut are deleted if the given condition is satisfied:

`if (cm:age < 18) delete;` (simply deletes the elements selected in the pointcut if the current user is a minor, i.e. cm:age < 18)

To add elements, add<*Something*> statements can be used where <*Something*> can be any of the AM element types (the same kind of type specification is used as in the pointcuts, see 3.1.1). Part of a SeRQL construct query can also be given to add elements (the designer is responsible for the validity), which will be directly added to the FROM clause of a SeRQL construct query (or several, depending on how many elements were selected in the pointcut), with the first node (corresponding to one of the selected elements) already present. These queries are subsequently executed on the AM. Both cases are illustrated below:

- `if (cm:bandwith >= 1000) {`
  `addElement attribute (containing label “Trailer”, query “SELECT T FROM {$variable} rdf:type {imdb:Movie}; imdb:movieTrailer {T}”);`
  `}` (adds an Attribute to the elements selected in the pointcut (movies), showing the movie trailer with label “Trailer” and the corresponding query, if the user’s bandwidth is above 1000 Kbps)
- `if (cm:localTime > 2200 and cm:localTime < 600) {`
  `addElement “ams:hasAttribute {} rdfs:label “Night”; hasQuery ‘SELECT L FROM {$V} rdf:type {imdb:NightImage}; imdb:location {L}’”; }` (adjusts the display of the Web site to the visitor’s local time, by adding an Attribute that represents a night-time image)

When replacing elements, only the explicitly specified parts of the elements are replaced; other parts are copied. The type of an element (see second example below), its value and/or its name (see first example) can be replaced. In the latter two cases, pattern matching symbols can be used to match part of the string to be replaced, and the parts matched by the pattern matching symbol will simply be copied. Typical examples include:

-
  ```
  if (cm:userDevice = “pda”) {
        replace hasTarget value “Big*Unit”
        by hasTarget value “Small*Unit”;
  ```
  `}` (if the user uses a pda, let Relationships and Subunits refer to a smaller version of the Unit)

-
  ```
  if (cm:userDevice = “pda”) {
        replace hasSubUnit by
        hasRelationship;
  ```
  `}` (replaces the contained Subunit elements by Relationship elements; the RDF properties of the particular Subunits, e.g. label, query, etc, are left unchanged)

Multiple actions can belong to a single condition, and conditions can also be nested. This way, multiple conditional changes can be specified using a single aspect:

```
if (cm:userDevice.type = “pda”) {
      replace hasSubUnit by hasRelationship;
      addElement attribute (containing label “Adjusted to device”,
      query   “SELECT   L   FROM   {$V}   rdf:type   {imdb:PDAImage};
      imdb:location {L}”);
      if (cm:userDevice.screenSize < 100) {
            replace hasTarget value “Big*Unit”
            by hasTarget value “VerySmall*Unit”;
      } else {
            replace hasTarget value “Big*Unit”
            by hasTarget value “Small*Unit”;
      }
}
```

This example first checks whether the visitor uses a pda: if so, the Subunit elements (contained inside the elements selected in the pointcut) are replaced by Relationships to the corresponding Units, to avoid crowding the smaller pda screens. It indicates this by adding an Attribute that displays an image representing a pda-device. Furthermore, smaller versions of the Units are now targeted; if the screen is very small, even smaller versions will be targeted. For a full reference on the advice part of SeAL, we refer to Appendix B.

3.2.3 Undoing advice actions

Obviously, it has to be possible to undo advice actions (e.g. when an existing aspect has to be altered/extended, or is not required anymore). As can be seen from the subsections above, there are two broad categories of advice actions that can be executed: adding conditions to element queries and adding/removing/replacing AM elements. Adding conditions to AM element queries is done at design time, and the resulting queries apply to all users at all times. This is in contrast to adding/removing/replacing elements, where advices are executed every time a page is requested by a specific user (because the advice conditions are dependent on volatile context and content information). This results in a copy being made of the requested AM part, and subsequently executing the advice on it. Undoing actions is therefore not an issue, because they are not executed on the original AM, but on a copy specifically meant for the user. However, this is not the case when conditions are added to AM queries, because these changes are made to the original, global AM. Intuitively, there are two ways to undo transformations on the AM caused by adding conditions to element queries:

- By combining conditions on queries in a specific way, so conditions of the advice that has to be undone can be "taken" out of the queries;
- By keeping an original copy of the AM (without any aspects executed on it), and re-executing aspects that weren't undone.

Clearly, the second option seems to be the most simple and straight-forward solution. By adding support for this to the Hera-S engine, any aspect could thus be easily undone.

3.3 Domain-specific vs. general-purpose language

We have already briefly mentioned some of our reasons to choose a domain-specific language over a general-purpose language at the beginning of chapter 3. We could have used a general-purpose query language to specify pointcuts over the Application Model (like SeRQL), which is versatile and expressive enough to execute queries on any RDF repository. However, this approach does not take advantage of the fact that the structure of the RDF repository is already known beforehand, because it is dictated by the AM Metamodel. Also, domain-experts may prefer to learn a more compact and domain-specific language like SeAL than a complex and extensive language like SeRQL. Moreover, the advantages of domain-specific languages have been well documented [20].

To prove our point for the pointcut part, we need some practical adaptation examples, that clearly show that a pointcut expression in SeAL is far more understandable and concise than a query with similar semantics in SeRQL. For example, imagine we do not want to show photo's in Subunits to small-screen pda users (photo's that occur at the top level of the Unit are still shown), to avoid overly crowding their screen. To do this, we have to select all Attributes present in a Unit, which occurs itself as a Subunit in another Unit. In SeAL, this would become:

```
hasType attribute and containedIn (hasType subUnit);
```

While in SeRQL, this would result in:

```
SELECT V4 FROM {V5} <http://wwwis.win.tue.nl/~hera/Hera-S/am-
metamodel.owl#refersTo> {V6} <http://wwwis.win.tue.nl/~hera/Hera-
S/am-metamodel.owl#hasAttribute> {V4} WHERE V5 IN (SELECT V3 FROM
{V2} <http://wwwis.win.tue.nl/~hera/Hera-S/am-
metamodel.owl#hasSetUnit> {V3} UNION SELECT V3 FROM {V2}
<http://wwwis.win.tue.nl/~hera/Hera-S/am-metamodel.owl#hasUnit> {V3}
UNION SELECT V3 FROM {V2} <http://wwwis.win.tue.nl/~hera/Hera-S/am-
metamodel.owl#hasTourUnit> {V3})
```

As a second example, suppose we do not want to show Subunits present inside other Subunits to pda-users, again to avoid crowding their screen. Consequently, all Subunits present in a (Navigational) Unit that itself occurs as a Subunit, have to be selected. In SeAL, this would result in:

```
hasType subUnit containedIn (hasType subUnit > nav);
```

In SeRQL, this would become:

```
SELECT V1 FROM {V0} <http://wwwis.win.tue.nl/~hera/Hera-S/am-
metamodel.owl#hasUnit> {V1} UNION SELECT V1 FROM {V0}
http://wwwis.win.tue.nl/~hera/Hera-S/am-metamodel.owl#hasSetUnit>
{V1}) INTERSECT (SELECT V4 FROM {V5}
<http://wwwis.win.tue.nl/~hera/Hera-S/am-metamodel.owl#refersTo> {V6}
<http://wwwis.win.tue.nl/~hera/Hera-S/am-metamodel.owl#hasUnit> {V4}
WHERE V5 IN (SELECT V3 FROM {V2} <http://wwwis.win.tue.nl/~hera/Hera-
S/am-metamodel.owl#hasUnit> {V3} UNION SELECT V3 FROM {V2}
<http://wwwis.win.tue.nl/~hera/Hera-S/am-metamodel.owl#hasSetUnit>
{V3}) UNION SELECT V4 FROM {V5} <http://wwwis.win.tue.nl/~hera/Hera-
S/am-metamodel.owl#refersTo> {V7}
<http://wwwis.win.tue.nl/~hera/Hera-S/am-metamodel.owl#hasSetUnit>
{V4} WHERE V5 IN (SELECT V3 FROM {V2}
http://wwwis.win.tue.nl/~hera/Hera-S/am-metamodel.owl#hasUnit> {V3}
UNION SELECT V3 FROM {V2} <http://wwwis.win.tue.nl/~hera/Hera-S/am-
metamodel.owl#hasSetUnit> {V3})
```

It is clear from these two examples that the SeAL expression is much shorter and clearer, and to understand the SeRQL expression you need a significant knowledge of the AM Metamodel and the SeRQL query language. As a final example, suppose we want to select all Subunits in a (Navigational) Unit that contains another Subunit with label “Free movies”. In SeAL, this would become:

```
hasType subUnit and containedIn (hasType unit > nav and contains
(hasType subUnit and hasLabel “Free movies!”));
```

In SeRQL:

```
(SELECT V1 FROM {V0} <http://wwwis.win.tue.nl/~hera/Hera-S/am-
metamodel.owl#hasUnit> {V1} UNION SELECT V1 FROM {V0}
<http://wwwis.win.tue.nl/~hera/Hera-S/am-metamodel.owl#hasSetUnit>
{V1} UNION SELECT V1 FROM {V0} <http://wwwis.win.tue.nl/~hera/Hera-
S/am-metamodel.owl#hasTourUnit> {V1}) INTERSECT SELECT V12 FROM {V12}
<http://wwwis.win.tue.nl/~hera/Hera-S/am-metamodel.owl#refersTo>
{V15} <http://wwwis.win.tue.nl/~hera/Hera-S/am-
metamodel.owl#hasRelationship> {V14}
<http://wwwis.win.tue.nl/~hera/Hera-S/am-metamodel.owl#refersTo>
{V13} WHERE V13 IN (SELECT V2 FROM {V2}
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
{<http://wwwis.win.tue.nl/~hera/Hera-S/am-
metamodel.owl#NavigationalUnit>} INTERSECT (SELECT V6 FROM {V6}
<http://wwwis.win.tue.nl/~hera/Hera-S/am-metamodel.owl#hasUnit> {V7}
WHERE V7 IN (SELECT V4 FROM {V3} <http://wwwis.win.tue.nl/~hera/Hera-
S/am-metamodel.owl#hasUnit> {V4} UNION SELECT V4 FROM {V3}
<http://wwwis.win.tue.nl/~hera/Hera-S/am-metamodel.owl#hasSetUnit>
{V4} UNION SELECT V4 FROM {V3} <http://wwwis.win.tue.nl/~hera/Hera-
S/am-metamodel.owl#hasTourUnit> {V4}) INTERSECT SELECT V5 FROM {V5}
<http://wwwis.win.tue.nl/~hera/Hera-S/am-metamodel.owl#refersTo>
{<http://www.vub.ac.be/imdb-am#UserRatingUnit>}) UNION SELECT V8 FROM
{V8} <http://wwwis.win.tue.nl/~hera/Hera-S/am-
metamodel.owl#hasSetUnit> {V9} WHERE V9 IN (SELECT V4 FROM {V3}
<http://wwwis.win.tue.nl/~hera/Hera-S/am-metamodel.owl#hasUnit> {V4}
UNION SELECT V4 FROM {V3} <http://wwwis.win.tue.nl/~hera/Hera-S/am-
metamodel.owl#hasSetUnit> {V4} UNION SELECT V4 FROM {V3}
<http://wwwis.win.tue.nl/~hera/Hera-S/am-metamodel.owl#hasTourUnit>
{V4}) INTERSECT SELECT V5 FROM {V5}
<http://wwwis.win.tue.nl/~hera/Hera-S/am-metamodel.owl#refersTo>
{<http://www.vub.ac.be/imdb-am#UserRatingUnit>}) UNION SELECT V10
FROM {V10} <http://wwwis.win.tue.nl/~hera/Hera-S/am-
metamodel.owl#hasTourUnit> {V11} WHERE V11 IN (SELECT V4 FROM {V3}
<http://wwwis.win.tue.nl/~hera/Hera-S/am-metamodel.owl#hasUnit> {V4}
UNION SELECT V4 FROM {V3} <http://wwwis.win.tue.nl/~hera/Hera-S/am-
metamodel.owl#hasSetUnit> {V4} UNION SELECT V4 FROM {V3}
<http://wwwis.win.tue.nl/~hera/Hera-S/am-metamodel.owl#hasTourUnit>
{V4}) INTERSECT SELECT V5 FROM {V5} <http://www.w3.org/2000/01/rdf-
schema#label> {"Free movies"})))
```

This clearly illustrates that the SeRQL query becomes prohibitively large and complex when nesting of two levels or more occurs in the corresponding SeAL expression, while semantically speaking the example is still relatively simple. This complexity arises from the fact that the language does not take into account the peculiarities of the AM Metamodel, and therefore concepts like "element contained inside other element" or "element contains other elements" will always result in complex SeRQL queries, independent of the specific AM that is being queried. This is in contrast to expressions in our custom-made language, where these concepts can be clearly captured by domain-specific constructs like `containedIn` and `contains`. Also, we mentioned an aggregation function in 3.1.1 that can count the number of elements contained inside a Unit; note that SeRQL does not contain a construct that allows us to do this (our implementation does this indirectly by counting the number of query results, see 3.4.1). There are also performance issues that inherently occur in SeRQL with the nesting of queries (see 3.4).

The same observation can be made for advices; instead of developing a custom language to express adaptation actions and conditions, we could have used a general-purpose query language to specify the advice part of an aspect. For example, we could have used SeRQL to define construct queries where the relevant elements (i.e. the elements to which the actions should be applied to) are selected in the FROM clause, and the CONSTRUCT clause contains the actions to be performed. However, the benefits of using a domain-specific language for selecting elements have already been discussed above. Additionally, several problems arise when executing advice actions in this manner, as explained below.

For example, to add adaptation conditions to an element (see 3.2.1), the new element query (i.e. with the added conditions) would have to be entered manually in the CONSTRUCT clause and then connected to a selected element; so if more than one element was selected, a separate construct query would have to be made for each element. Subsequently, to remove the old query (i.e. the query without the conditions), a separate construct query containing the old query in the construct clause would have to be made for each element as well. It can also be seen that the adaptation engineer needs sufficient knowledge of the AM and the SeRQL query language to write such queries. Adding elements using SeRQL would pose no additional problems (as mentioned in 3.2.2, it is already possible to do this in SeAL by specifying part of a SeRQL construct query), and neither would deleting elements. However, replacing elements would also require two construct queries for each selected element; one to add the changes to the element and one to remove the replaced properties of the element. Furthermore, it would not be possible to use pattern-matching to copy part of a string value.

Although this section certainly motivates our choice of developing a custom-made language to enter pointcut-advice pairs, note that in our advice implementation we still use SeRQL as an underlying query language, in order to obtain the elements from the AM that satisfy the conditions and subsequently execute the advice actions on these elements (see 3.4.1).

3.4 Implementation

Our implementation of SeAL uses SeRQL as an underlying query language, which enables us to extract the elements from the AM that satisfy the pointcut conditions. This can be done by translating each pointcut condition to a separate SeRQL query, which can extract the elements from the AM that satisfy the condition. Our first implementation consisted of simply mapping the logical connectors to their WHERE clause counterparts, nesting each condition query as an operand in the WHERE clause of a single query. However, we found that nested queries had serious performance problems in SeRQL: all nested queries are executed each time a matching value is found in the parent query, even when there are not shared variables between them (opposed to e.g. SQL, where optimizations are available). This lead to a dramatic loss of performance, because nested queries had to be used each time multiple conditions were specified. Therefore, our following approach mapped the logical connectors to set combinatory operators (union, intersect, minus) instead, and combined the condition queries into one big SeRQL construct. However, when nesting occurred in the conditions themselves (e.g. `containedIn`, `containing`) queries still had to be nested in the separate queries, and as a result nesting levels of 4-5 were not feasible. Implementing a count function or referencing the DM (i.e. combining the returned values with the other SeRQL constructs) was also not possible in this approach.

Consequently, our approach now consists of directly executing each separate condition query on the Application Model. The results are then put into Vector objects and the logical connectors, combining the conditions, are mapped to equivalent Vector methods (e.g. logical conjunction corresponds to `retainAll`). The count function can thus be implemented by executing the query corresponding to the nested condition, and then counting the resulting values. References to the Domain Model can be resolved by executing the specified query (or SeAL expression, see 3.1.2) on the DM, and subsequently comparing the returned values to the ones obtained in the pointcut expression. Furthermore, the performance loss resulting from nested queries are effectively avoided, by manually executing each nested condition query only once.

The Application Model, Domain Model and instance data are stored as Sesame repositories, which is a powerful, versatile and popular open-source RDF framework. This also motivates our use of SeRQL (Sesame RDF Query Language) as a general-purpose language underlying SeAL. We decided to use JavaCC[21] (a parser/scanner generator for Java) to automatically generate a parser for SeAL. We chose JavaCC because it is a popular parser generator for

[21] https://javacc.dev.java.net/

Java, and thus stability and availability of documentation is more or less guaranteed. It also contains a tool JJTree that can automatically generate Abstract Syntax Tree classes (AST) and instantiate them during the parsing process. An AST is an abstract representation of the syntax of the given expression, and each AST class corresponds to a production rule in the parser specification. An instance of an AST provides for a powerful means of parsing the expression and retrieving semantics from the syntax. JJTree also supports the Visitor design pattern, which is used extensively throughout our pointcut implementation.

As already mentioned, the pointcut conditions are translated to separate SeRQL constructs, which are subsequently executed on the Application Model (or Domain Model). To encapsulate such SeRQL constructs, a separate package called `serql` was developed. A class diagram of this package can be found in Appendix D.

The implementation of the advice uses a similar approach as the pointcut part: the JavaCC parser generator was used to construct a parser for the language, and the JJTree tool was used to automatically generate AST classes. The Visitor design pattern was implemented to evaluate the advice expressions, using the relevant SeRQL packages to alter the Application Model. To enable adding conditions to elements, the `serql` package was extended to support SeRQL query rewriting.

3.4.1 Architecture

Below follows a summary of the implementation architecture. The classes that were automatically generated by JavaCC are not mentioned, except for the ones that have been extended significantly (these are indicated). For more details on these generated classes, we refer to the JavaCC documentation[22].

QueryFactory

An object of this class creates (and possibly executes) queries on an RDF repository. These can include queries that are too complex to put in client code, or are frequently used by several different classes. It also contains getter methods for frequently used instances of serql package classes (e.g. “rdf:type”, “rdfs:label” instances of the `serql.Property` class). This object’s state and behaviour cannot be changed by external objects; however, because it is used in several other objects, multiple instances of this class could accidently be created. Therefore, this class implements the Singleton design pattern.

Subclasses are *AMQueryFactory*, *DMFactory* and *CMFactory*, which extend QueryFactory with methods specific for the Application Model, Domain Model and Context Model, respectively. For example, AMQueryFactory contains a method that returns the type of an AM element, based on the blank node or URI reference of that element, or a method that deletes an element together with its contained elements.

TypeEnum

This Java enumeration type contains all possible AM element types (e.g. unit, subunit, query, etc) and subtypes (e.g. navigational unit, form unit, onload query, etc).

[22] https://javacc.dev.java.net/doc/ docindex.html

Type

This generic class represents a type specification (called a “type-tree”). This specification can consist of just one top-level type or a top-level type together with several subtypes. To implement this, each Type instance keeps a TypeEnum instance (to identify its own type) and two other Type objects, one as sibling type and one as child type. The latter simply represents a subtype specification, while the sibling type represents a disjunction (or union) in the type tree: for example, `superType1 > subType1 | subType2` states that the specified type can be either `subType1` or `subType2`; `subType1` is the child type of `superType1`, while `subType2` is the sibling of `subType1`. Such type additions can be propagated through the type tree; when `subType2` (which is of the same level as `subType1` in the type tree) is added in the above example, it will first be added to `superType1` which will delegate the request to `subType1` to add the subtype as a sibling. Mostly, a certain type can only have a limited number of subtypes and sibling types; for this purpose, Type keeps an array of allowed sibling types and subtypes, which are checked every type a subtype or sibling type is added to the Type instance. These allowed types are obviously application-dependent and have to be initialized by a Type subclass. Moreover, a separate TypeEnum enumeration has to be made for each application domain to enumerate the domain-specific types.

AMType is a subclass of Type, representing an AM element type; this also allows us to do the necessary type checking in the *AMType* subclass instead of in the parser specification (significantly reducing complexity in the latter). For example, `query > onload | conditional` represents all AM query elements that are either of type onload or conditional. AMType sets the allowed sibling types and subtypes for each AM element type; in the above example, `query` would have no allowed sibling types and as allowed subtypes `onload`, `conditional` and `update`, while these subtypes would have each other as allowed sibling types and no allowed subtypes. It also checks whether a type is allowed as a top-level type (`query` in the above example). AMType also contains specific methods for use in the Application Model. For example, `toPointcutQuery(…)` returns a SeRQL construct that selects AM elements of a type corresponding to the represented type tree, `toAddQuery(…)` creates a construct query that adds an element of the corresponding type to the Application Model, etc.

HasElCount (AST class)

This class represents the count aggregate function. For example:

```
contains (2 < count (hasType attribute) < 5)
```

This expression selects only elements that contain between two and five Attributes. An object of this class keeps a Vector object of `NumCondition` instances for this purpose (see below).

NumCondition (AST class)

An instance of this class represents a numeric condition on the count aggregate function. It keeps a Comparator object (see below), an instance of Position and an integer value. Position is an enumeration class, and keeps whether the integer value occurred before or after the comparator; this is needed to later compare the result of the count function (i.e. the number of contained elements) to the given integer(s).

Comparator (AST class)

An object of this class represents a numeric comparator (<, > , <=, >=, = or !=). It contains a method that compares two integer values. It also contains a method to check whether the number 0 satisfies the comparison (in the above example, 0 would be included if the first comparison was omitted); this information is needed later on.

AMParserVisitor

An object of this class implements the Visitor design pattern. Consequently, it contains corresponding "visit" methods for each AST class. Each separate condition is translated to a SeRQL query that extracts all AM elements satisfying the condition. The results of these query executions are then combined using Vector methods, corresponding to the specified logical operators (e.g. logical "and" corresponds to an intersection of sets, implemented using `retainAll`). As an example, consider the following pointcut expression:

```
hasType attribute and hasLabel "Title";
```

During the parsing process, an AST instance is created corresponding to the expression syntax. The root node of this AST is subsequently visited by the AMParserVisitor object. In our case, the tree would look like:

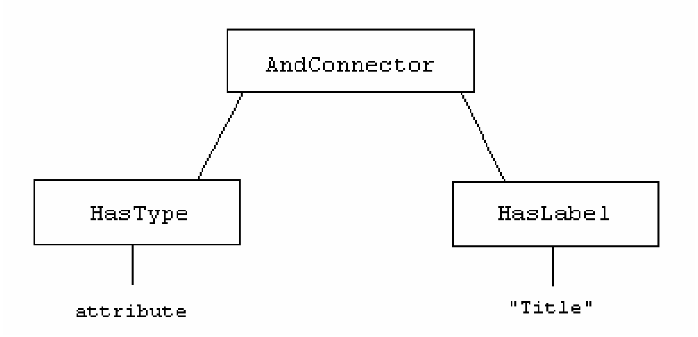


Figure 2. Abstract syntax tree.

Firstly, the visit method for the `AndConnector` node is called, which delegates the call to the visit method for its first child, namely `HasType`. There, the condition is translated to its corresponding SeRQL query, namely

```
SELECT V1 FROM {V0} ams:hasAttribute {V1}
```

This query is executed and its results returned to the `AndConnector` visit method, which puts them in a Vector object. Then the method for the second child is called, namely `HasLabel`, leading to the following query:

```
SELECT V0 FROM {V0} rdfs:hasLabel {V1} WHERE V1 LIKE "Title"
```

The query is executed and the results are added to the Vector object using the `retainAll(Collection c)` method, which retains only the elements in the Vector that are contained in the Collection c. The Vector is then returned from the visitor object, containing only the AM elements satisfying both conditions.

DMParserVisitor

In case the Domain Model is referenced in the `hasInput` condition (which places a constraint on the input variable type), an instance of this class is initialized with all the variable types present in the Application Model, together with the corresponding elements. If a SeRQL query was given to be directly executed on the DM (indicated by the `<SeRQL>` keyword), the given variable types are compared to the results of this query (only the first variable in the SELECT clause will be considered); elements containing a variable whose type occurs in the results, are considered to satisfy the condition and are returned by the DMParserVisitor object.

If an expression in our custom language was given, the visitor object is called on the root of the corresponding AST instance. As previously mentioned, the SeAL language extension supports `rdf:type` statements, with semantics equal to SeRQL. Therefore an instance of *DMParserPreVisitor* (see below) is first created, which looks for `rdf:type` statements and simply replaces the subject variable by the object URI type. After this, the paths and branches in the given expression are translated to the corresponding SeRQL query: for example,

```
{I} ams:hasAgeRating {J}
```

would result in:

```
SELECT “dummy” FROM {ams:hasAgeRating} rdfs:domain {I};
                                       rdfs:range {J}
```

As already mentioned, some basic inferencing is also supported. After taking into account possible subclasses of `V0` and `V1` and subproperties of ams:hasAgeRating, the above query becomes (note that the query below is valid because each class is also a subclass of itself, idem for properties):

```
SELECT “dummy” FROM {V2} rdfs:domain {V0} rdfs:subClassOf {I};
                        rdfs:range {V1} rdfs:subClassOf {J};
                        rdfs:subPropertyOf {ams:hasAgeRating}
```

This query is then merged with a query kept locally by the visitor object, that keeps all SeRQL paths and branches corresponding to the given SeAL expression. After all paths and branches in the expression have been handled this way, every occurrence of the select variable in the local query is replaced by an AM variable type and subsequently executed. In case the query returns any results, it means that the elements corresponding to the variable type satisfy the condition, and are added to a Vector object. This Vector object is returned after all AM variable types have been handled this way.

As previously mentioned, `unionOf` constructs for domains/ranges are also supported. Therefore, the visitor object first checks whether the domain/range of the given property corresponds to a collection or a single class. In case of a single class, the aforementioned approach is used. If a URI was given as a domain/range, the latter case can be reduced to finding an element in the collection that is equal to the given URI, by executing queries of the form (subclasses and subproperties have not been taken into account):

```
SELECT   V0   FROM   {<http://www.vub.ac.be/imdb-schema#hasAgeRating>}
rdfs:domain  {}  owl:unionOf  {}  rdf:first  {};  rdf:rest  {}  rdf:first
{V0}
```

This process is repeated until a result is returned that equals the given URI (or no results are returned). The paths in this query are then added to the query kept locally by the object. Note that in case the domain/range was specified as a variable, every element in the collection has to be considered (because they cannot be compared to a specific URI), and a logical disjunction of the paths is needed in the local query (the variable can match the first element, *or* the second, etc). This is done by replacing the local query by a union, to which the disjunctive queries are then added.

DMParserPreVisitor

An object of this class is a preprocessor for the DMParserVisitor. It looks for `rdf:type` statements and replaces the subject variable by the object URI type. If the subject and object of the predicate do not correspond to a variable and URI respectively, an exception is thrown. It also throws an exception if the SELECT clause variable is not bound in the given expression.

ElSpec (AST class)

An instance of this class specifies (part of) an element to be added to the Application Model. Such an element has a type (specified by a Type object) and can either have a value (e.g. in case of a label) or a name (e.g. in case of a Unit), or can contain other elements (also specified by ElSpec objects).

SubElSpec (AST class)

This class is used when replacing (parts of) an element; an instance of this class specifies the part of the element to be replaced, or the part by which it will be replaced. The type, value or name of an element can be replaced. In the latter two cases, it is also possible to specify wildcards to match part of the string to be replaced, and to specify whether case should be ignored in the pattern matching process.

PropPath (AST class)

An object of this class represents a navigation path, starting with a namespace-prefix and optionally containing a sequence of properties.

PropBranch (AST class)

An object of this class represents a navigation branch, containing a sequence of properties and optionally starting with a namespace-prefix.

PropSeq (AST class)

An instance of this class represents a sequence of properties. For this purpose, it keeps a Vector containing the specified property names.

AdviceParserVisitor

An object of this class implements the Visitor design pattern, and contains "visit" methods for each AST class. After an advice expression has been parsed, an AST instance is returned corresponding to the expression syntax, of which the root is subsequently visited by the AdviceParserVisitor object.

When a condition is being added to the selected elements, the corresponding element queries are first extracted from the Application Model (if they are malformed or not present, an exception is thrown). Subsequently, the advice expression is translated to SeRQL query paths and conditions and added to the extracted query. The original query from the AM is then replaced by this new query. To illustrate our approach, consider the following example:

```
imdb:p1.p2 = 2; p4 = 3 or cm:p5 = 4
```

Firstly, the property sequence `p1.p2` is translated to the corresponding SeRQL path, namely `{V1} p1 {V2} p2 {V3}`. The condition `= 3` is then mapped onto its WHERE clause counterpart, namely `V3 = 3`. Then, `p4` is added as a branch to the previously obtained path, resulting in `{V1} p1 {V2} p2 {V3}; p4 {V4}`. The same process is repeated with the second OR operand.

Finally, the obtained paths have to be added to the selected element queries. As already mentioned, every time the `imdb:` namespace prefix is used (or any prefix different from `cm:`), the navigation path will start with the output nodes (i.e. SELECT values) from the selected queries. Therefore, the start node of each of the obtained paths has to be replaced by the SELECT values of the selected queries. In case there is only one select value (e.g. variable `S`), this results in the first path becoming: `{`**`S`**`} p1 {V2} p2 {V3}; p4 {V4}`. Because of the logical disjunction, both paths (resulting from translating the operands) have to be made optional in the SeRQL query (if one path is not matched, the query can thus still return results), and the path conditions have to be combined using a logical OR. This results in the following query (added parts are put in bold):

```
SELECT S FROM {S} rdf:type {imdb:someType},
      [{S} p1 {V2} p2 {V3}; p4 {V4}],
      [{} rdf:type {cm:CurrentUser}; p5 {V5}]
WHERE (V3 = 2 AND V4 = 3) OR V5 = 4
```

In case of a conditional change to the Application Model (i.e. adding/removing/replacing elements), the given condition is first evaluated w.r.t. the content data. The process of translating the condition to the corresponding SeRQL query is similar to the one explained above. If any results are returned by the resulting query, the specified actions are executed.

When adding an element to the AM, the given element specification is translated to the corresponding SeRQL construct query, which is subsequently executed. In case plain SeRQL was given, it is added to the FROM clause of a construct query and connected to a selected element; if more than one element was selected, separate construct queries are created and executed for each selected element. When deleting the selected elements, both the elements they contain and "containing" relationships (e.g. in case the Attribute `node2` was selected in the pointcut, the `node1 ams:hasAttribute node2` property also has to be removed). Replacing parts of elements (e.g. type, type of contained element, value, name, etc) is much more complicated, and for details on the implementation we refer to the code.

Chapter 4: Case study

To carry out a case-study that explores the possible benefits of our approach (as compared to the current condition-based approach), we first needed the Hera-S design models necessary to automatically generate a basic, navigation-based Web application in which adaptation can be specified. In this case-study, we will concentrate on the Appliction Model, and to a lesser extent the Domain Model (when exploiting the semantics of the content data). These models were written in the context of the Internet Movie DataBase[23] (IMDB) and Amazon.com[24]. Also, we developed a Wrapper tool to automatically obtain a significant dataset from these Web sites, to serve as content data underlying the Web application. Subsequently, support for various adaptation concerns was added to the application design, using both our aspect-oriented approach and the condition-based approach currently used in Hera-S.

By taking into account various criteria such as the effort needed to implement an arbitrary adaptation concern and the re-usability and robustness of the resulting code, we were able to effectively compare both approaches. Furthermore, the examples illustrate the achievement of our goals (as stated in 1.3), namely to use aspect-orientation to handle cross-cutting concerns, and to exploit semantic and global (structural) information to achieve a more declarative approach to adaptation specification.

The created AM and DM are summarized below. The development of the Wrapper tool is shortly discussed thereafter, and test-results are given. Finally, we add support for several adaptation concerns in the Application Model, using the current condition-based approach and our aspect-oriented approach.

4.1 Application Model

The Application Model describes hypermedia navigation over the content data. It does this by defining Navigational (or Form) Units, which contain Attributes, Relationships and other Units (called Subunits). Attributes represent the actual data; they contain a query that extracts the relevent data item from the dataset (e.g. movie title, year, etc), while Relationships allow navigation between two Units. Form Units also contain form elements (e.g. input fields, choice/radio buttons, etc) and thus allow for user interaction. For more information on the Application Model, we refer back to the introduction on Hera-S (see 2.4).

Our Application Model was written in the context of IMDB and Amazon.com, and can be fed to the Hera-S engine (together with the other models) to generate a full-fledged Web application. It contains Navigational Units representing movies, actors, directors, etc. for this purpose. Most of the major concepts are represented by two different Units; a “short” Unit that contains a summary of the movie or person, and a more extensive one that covers all the details, i.e. representing a more elaborated view on the information. The former is used as a Subunit in other Units (e.g. to represent the cast, director or writer of a movie). These short Units typically contain a Relationship to the larger Unit, enabling the user to choose when he/she wants to see a person’s full information. The AM also contains several Form Units, which enables the user to search for a specific movie (by doing either a simple or advanced

[23] http://www.imdb.com
[24] http://www.amazon.com

search) and to register or login to the Web application. The final AM contains a total of 22 Units, and is currently available at http://wilma.vub.ac.be/~wvwoense/am.rdf.

4.2 Domain Model

The Domain Model represents the conceptual design in the Hera methodology, and is used as a schema for the obtained dataset. Because the model and dataset will be used later on for testing our advice language, it is important that it contains sufficient support for adaptation; e.g. movie age ratings, hardware support (i.e. movie sound, color), popularity rating, language, showtimes, etc. Hardware support is particularly interesting in this context: a user could input his/her hardware configuration (e.g. sound system, Plasma TV, etc), and movies would be shown/ordered according to their support of these hardware properties. This was done more extensively for games: each game can be played on certain software platforms on certain game consoles; each game console can support one or several of these platforms, and software platforms can be compatible (making each game playable on one platform also playable on the other). This allows for an even more fine-grained adaptation to the user's hardware. Movie showtimes are equally interesting: if the user inputs his/her home address or town and work hours, movies could be ordered according to the showtimes best fitting their free time and shown nearest to the user's home. The resulting DM is currently available on http://wilma.vub.ac.be/~wvwoense/DM.owl, and contains 24 classes and 97 properties (47 of which are datatype properties).

4.3 Context Model

The Context Model (CM) is maintained to allow for user- and context-based adaptation, and is used as a schema for the actual context data. The CM models all the necessary user information, consisting of both user information (e.g. name, gender, age, etc) and usage information (information derived from interaction with the Web site, e.g. number of times a link was followed, a specific concept was viewed, etc). The Context Model is currently available on http://wilma.vub.ac.be /~wvwoense/CM.owl.

4.4 Wrapper tool

To obtain a significantly large dataset, a software tool was needed that could automatically extract information from IMDB and Amazon.com, and save it as RDF triples consistent with the schema defined by the Domain Model. Because the retrieval of the dataset was not the main goal of the thesis, the available time span for its development was limited. Therefore, a relatively simple approach was used to extract the relevant information from the HTML pages. A brief discussion of the implementation of the Wrapper tool is given below. Test results are given afterwards.

4.4.1 Implementation

The javax.swing.text.html.HTMLDocument package and javax.swing.text.html. HTMLEditorKit.ParserCallback/Parser and HTMLDocumentLoader classes were used for the implementation of the Wrapper tool. An instance of HTMLDocument represents the element structure of a HTML page, making it available for navigation after the parsing process[25]. In this approach, an instance of (a subclass of) ParserCallback is obtained from the HTMLDocument instance which implements all the required parser events. This object is then given to a Parser object, which will invoke the corresponding method of the ParserCallback instance at each parser event, enabling the ParserCallback object to build the corresponding element structure in the HTMLDocument instance. HTML elements can be obtained from the HTMLDocument instance by using iterators to navigate to the desired elements. There are two classes available for this purpose: HTMLDocument.Iterator en ElementIterator.

4.4.2 Test

We have run an instance of the Wrapper tool for 10 hours, resulting in 296233 RDF statements, consisting of 23626 movies and 1481 persons. Because only the top-selling items are listed on Amazon.com, the videogame result set will always be limited to 100. The consistency of the dataset has been validated using online validators (WonderWeb OWL Ontology Validator[26] en Pellet OWL Reasoner - Online demo[27]). Because some constraints are not checked by these validators (e.g. cardinality constraints), some ad-hoc code was also used to test whether the dataset conformed to the Domain Model. The dataset can be found on http://wilma.vub.ac.be/~wvwoense/imdbRep.rar.

4.5 Adding support for adaptation concerns

This section describes adding support for several adaptation concerns to our basic, navigation-based Web application design. We used both our aspect-oriented approach and the current condition-based approach to implement each adaptation concern, and subsequently compared the effort required by the designer and the re-usability and robustness of the resulting code.

Firstly, we extended our basic (navigation-based) Web application design to restrict the view of non-registered users. Using our generic mechanism, this support was provided by a single aspect:

```
pointcut: hasType relationship and from (hasType subUnit) and to
(hasType unit);
advice: addCondition cm:isRegistered = true;
```

This pointcut selects all Relationships (i.e. links between Units) that originate from a Subunit and target a Unit. The advice adds to these Relationships the condition that the (current) user should be registered. Thus, the above adaptation aspect will present the non-registered user with a restricted view on the Web application: only top-level links (i.e. those appearing in Units) will be shown, while any link that originates from a Subunit and targets a top-level Unit, and thus typically presents an elaborated view on the particular concepts represented in the Subunit, will be hidden. Note that this still allows a non-registered user to follow useful (top-level) links, such as a link to a registration form or even a search form: when the search results are represented as Subunits, non-registered users are still able to see the short version, while only registered users would be able to view the full information. In the current Hera-S

[25] A (more efficient) alternative would have been to extract the needed information during parsing, by writing a subclass of ParserCallback that implements the different parser events. However, experience tells us that this would have resulted in much more complex code.
[26]. http://phoebus.cs.man.ac.uk:9999/OWL/Validator
[27] http://www.mindswap.org/2003/pellet/demo.shtml

approach, this adaptation concern resulted in three Relationship queries to be manually changed, resulting in a query of the form (added adaptation condition is put in bold):

```
"SELECT M
FROM {$M} rdf:type {imdb:Movie}, {} cm:isRegistered {R}"
```

Although in our AM this resulted in only a limited amount of queries having to be manually altered, it can also be seen that the adaptation aspect can be easily re-used in a different design, because it is in no way hard-coded to the specific AM.

The previous example basically restricts the visibility of certain links according to a certain property of the user. Often, it is preferable to automatically redirect the visitor to e.g. a registration form when such a link is followed, while in the previous example this behavior is dependent on the implementation of Hera-S (i.e. what Hera-S does when a Relationship query does not return any results; the link could lead to an error page, or could simply be hidden). An aspect stating that the visitor should be automatically redirected to a registration form is given below:

```
pointcut: hasType relationship and from (hasType subUnit) and to
(hasType unit);
advice: if (cm:isRegistered = false) {
              replace hasTarget value by hasTarget value
              “imdb:RegistrationForm” ;
        }
```

The pointcut remains unchanged, selecting all Relationships originating from a Subunit and targeting a Unit. The advice then replaces the value of the “hasTarget” attribute to “imdb:RegistrationUnit” (if the user is not registered), causing all selected links to be redirected to the registration Unit. It can be seen that this aspect does not simply add a condition to the selected Relationships, as was the case in the previous example. Therefore, this kind of behavior cannot be achieved by the current Hera approach (or indeed any condition-based approach, which rely solely on adding conditions to elements), because specific AM element properties (other than their queries) have to be altered. Note that the aspect can still be re-used over different Web applications, because it is not hard-coded to our specific AM.

Subsequently, we restricted the view of under-aged users by only allowing adults to see age-restricted information. To achieve this in our aspect-oriented approach, we exploited metadata present in the Domain Model:

```
pointcut: hasType subUnit and hasInput in
(<SeRQL> SELECT I FROM {imdb:hasAgeRating} rdfs:domain {I}) ;
advice: addCondition imdb:hasAgeRating.minAllowedAge <= cm:age;
```

The pointcut selects all Subunits that have as input a certain concept which has an “hasAgeRating” property specified in the Domain Model (the latter part is represented by the native SeRQL expression). The advice then adds a condition to these Subunits, denoting that they should only be shown if the age of the user (specified in the Context Model) is higher than the minimum allowed age (specified in the Domain Model) for that resource. It can be seen that in this example, no specific AM elements have been specified in the pointcut. Only at runtime will it be determined which elements have a ‘hasAgeRating’ property specified (in the Domain Model), and the corresponding elements will be selected from the AM. This clearly illustrates that the burden of identifying the place(s) in the AM where a certain adaptation needs to be performed is alleviated from the application engineer. Instead, these places are specified in a declarative way, using semantic metadata present in the Domain Model. Using the condition-based approach, the same results were achieved by manually changing eight Subunit queries to queries of the form (added adaptation conditions are put in bold):

```
"SELECT M
FROM {$P} imdb:actorFilmography {} imdb:castMovie {M},
{} rdf:type {cm:CurrentUser}; cm:age {Age},
{M} imdb:hasAgeRating {} imdb:minAllowedAge {MinAllowed}
WHERE Age >= MinAllowed
```

In addition to being less labor-intensive, it can also be seen that the aspect-oriented approach is much more robust: even when (later on) adding new resources to the Web application (imagine for example that IMBD decides to add songs to its collection), and therefore adding new Units and Subunits describing these resources, the adaptation aspect will subsequently also identify these new resources and their corresponding Subunits, and restrict their access/visibility accordingly.

Until now, we have adapted the Web site without the explicit knowledge (or wishes!) of the user. To remedy this, we decided to add adaptation support for certain user preferences, by allowing the user to specify whether he/she wants spoilers (i.e. plot outline) when viewing movie information. This resulted in the following aspect:

```
pointcut: hasLabel “Plot outline”;
advice: addCondition cm:preferences.showSpoilers = true;
```

Although this aspect implements the adaptation concern as expected, it can be seen that this can just as easily be done using the current condition-based approach (added adaptation conditions are put in bold):

```
"SELECT P
FROM {$M} imdb:moviePlotOutline {P}, {} rdf:type {cm:CurrentUser};
cm:preferences {} cm:viewSpoilers {allowSpoilers}
WHERE allowSpoilers = true
```

It can also be seen that the aspect is hard-coded and localized: if there were other movie Units present in the AM (e.g. which represent an elaborated version), the developer would have to make sure their plot Attributes have the same labels (or account for all possible labels in the pointcut, by using a logical OR), else the pointcut would only select one of the attributes. However, there are still some advantages to using our aspect-oriented approach in this case; by specifying all adaptation concerns as aspects, they can more easily be undone (see 3.2.3). Also, it provides a better overview of what adaptation concerns already exist (by separating them from the “regular” Web design), and can still significantly reduce complexity in the AM if many of these “simple” (and possibly overlapping) adaptation concerns occur.

Finally, we decided to provide support for pda-users by replacing information on large pages (e.g. with more than 5 Attributes specified) by links which point to smaller, dedicated pages showing the same information. Consequently, we exploited information on the global structure of the AM:

```
pointcut: hasType subUnit and contains (count (type attribute) >= 5);
advice: if (cm:userDevice = "pda")
            replace hasType subUnit by hasType relationship;
```

The pointcut selects all Subunits which contain five or more Attributes (“large” Subunits). The advice then replaces these Subunits (actually their in-page occurrence) by Relationships that link to dedicated pages representing this particular information. Again, the advice cannot be implemented using the current Hera-S approach, because specific AM element properties have to be changed. Note that the pointcut exploits characteristics of the entire AM, not just one single element, in this case the amount of Attributes. Because it exploits such implicit information, it relieves the burden from the developer of identifying the places in the AM where adaptation needs to be performed. Because it describes which elements have to be adapted instead of explicitly specifying them, it is also much more robust. Finally, it can also be seen that this aspect is perfectly portable across Web applications, because it is in no way hard-coded to our specific AM.

Chapter 5: Conclusion

5.1 Summary

In this thesis, we pursued a further separation of concerns in Web application design, by separating adaptation engineering from the regular Web design process. To achieve this, we first observed that adaptation is typically a cross-cutting design concern, and therefore aspect-oriented techniques can be applied to separate the adaptation concerns from the rest of the application design. Furthermore, we observed that semantic information (readily available in the Domain Model) and global (structural) information could be used to allow for more robust, expressive and fine-grained adaptation. We then recognized that this implicit information would also allow for a more declarative approach, by *describing* what needs to be adapted instead of explicitly specifying the affected design elements. It was also noted that by using global design information, adaptation can be specified independent from the application context, and thus be re-used over different Web applications. Finally, we postulated that by using a domain-specific language for specifying adaptation concerns (as opposed to a general-purpose language), that we would be able to express adaptation in a much more powerful, concise and expressive way.

Based on these observations, we proposed a custom-made language called SeAL, and defended several key design decisions. For instance, we discussed our reasons to extend SeAL with a facility that simplifies querying the Domain Model, and why to use so-called "type-trees" to specify Application Model element types. More importantly, we extensively discussed our decision to develop a custom-made language over using an existing general-purpose query language to express aspects. Subsequently, our implementation was discussed, including our general approach and software architecture.

Finally, we conducted a case-study that compared the existing Hera-S adaptation approach with our aspect-oriented (and semantics-based) approach, and demonstrated that our approach allows for easier, more compact and powerful adaptation specification. The examples in the case-study also illustrated that our goals as stated in the introduction (see 1.3) had been fulfilled; by applying aspect-oriented techniques, adaptation engineering could be successfully separated from the regular Web design. Furthermore, by using semantic metadata and global design information, adaptation specifications could be made less labor-intensive, more expressive and more robust. We also showed that several adaptation requirements are simply not expressible in the current Hera-S approach (which only allows adding conditions to elements). Moreover, it confirmed that adaptation specifications based on implicit global design information be easily be re-used across Web applications.

5.2 Future work

As mentioned in 3.1.3, including instance level information when evaluating pointcuts is a logical next step in the development of SeAL. This would allow us to test for semantic relationships between resources that actually exist in the content data, instead of relationships that could exist (as dictated by the Domain Model). For example, this can be interesting when your dataset is incomplete and only contains age-rating information for a limited amount of movies. Instead of querying the DM for the possible existence of the "hasAgeRating" property for the given resource type, it could be checked whether the input resource for a Unit element at runtime actually participates in such a relationship. It is clear that this implies executing each aspect (whose pointcut uses instance information) at runtime and upon page-request, because the element queries have to be executed before the specific input resource for the Unit can be known.

In advice conditions (see 3.2.1), navigation path values can be compared to a (single) literal value or other navigation path values. In our implementation, only one (actually, at least one) value resulting from the navigation path has to satisfy this comparison. In some cases however, it is necessary that *all* values should satisfy the comparison. For example, when you only want to show movies that are fully supported by the user's hardware:

```
addCondition              imdb:moviePropertySupport              =
cm:hasMovieEquipment.moviePropertySupport
```

In this example, all navigation path values from `imdb:moviePropertySupport` should satisfy the comparison at least once, i.e. each movie property has to be supported by the user's hardware. This implies a universal quantification of variables in the WHERE clause of the resulting element query, which can be written as the negation of an existential quantification of the inverse condition (i.e. if the condition holds for all values, there does not exist a value for which it does not hold):

```
SELECT M
FROM {$M} rdf:type {imdb:Movie}
WHERE NOT EXISTS
    (SELECT P1
     FROM {$M} imdb:moviePropertySupport {P1}
     WHERE P1 != ANY (SELECT P2
                      FROM {} rdf:type {cm:CurrentUser};
                              cm:hasMovieEquipment {}
                              imdb:moviePropertySupport {P2}))
```

This could be indicated by using `all` and `some` (= default case, where only one value has to satisfy the comparison) keywords at the relevant side(s) of the comparison. However, this can pose a problem with the open-world assumption of the Semantic Web: saying that there does not exist a value for wich the condition does not hold, implies that we have full knowledge of the outside world (i.e. all possible values for which the condition can hold).

As mentioned in 3.1.1, future work is also supporting explicit constructs to refer to the corresponding Unit of a Subunit and vice-versa, namely `represents` and `representedBy`, respectively. Finally, there are several special cases in the AM Metamodel which SeAL does not yet support: e.g. nested FormUnits, scripts, webservices, conditional queries, etc. Adding support for these elements is future work. Also, it is clear that adaptation has only been specified in the Application Model; adaptation specifications could be extended to include the Presentation Model. The possibilities (and limitations) of such an extension have not yet been researched.

Appendix A: SeAL specification (pointcut part)

```
(0) : <pointcut>          ::= "pointcut:" <pointcutExp> ";"

(1) : <pointcutExp>       ::= (<conditions> | "<SeRQL>" <valueStr>)

(2) : <conditions>        ::= <aCondition> ("and" <aCondition>)*

(3) : <andCondition>      ::= <orCondition> ("or" <orCondition>)*

(4) : <orCondition>       ::= ("not")? <condition>

(5) : <condition>         ::= "(" <conditions> ")" | <hasType> | <hasName> |
<hasBinding> | <hasText> | <hasOption> | <hasLabel> | <hasSource> |
<hasTarget> | <containedIn> | <containingEl> | <hasQueryLang> | <from> |
<to> | <navigateFrom> | <navigateTo>

(6) : <hasType>           ::= "hasType" <typeTree> ("," <typeTree>)*

(7) : <typeTree>          ::= <typeKind> ("|" <typeKind> | ">" <typeKind>)*

(8) : <typeKind>          ::= "all" | "unit" | "subUnit" | "form" | "nav" |
"tour" | "single" | "set" | "relationship" | "attribute" | "query" |
"onLoad" | "conditional" | "update" | "formEl" | "button" | "textInput" |
"choiceInput"

(9) : <hasName>            ::= "hasName" <value>

(10): <hasBinding>        ::= "hasInput" (<value> | "in" <dmExp> | "in"
<dmExpMap>)

(11): <hasText>           ::= "hasText" <value>

(12): <hasOption>         ::= "hasOption" <value>

(13): <hasLabel>          ::= "hasLabel" <value>

(14): <hasSource>         ::= "hasSource" <value>

(15): <hasTarget>        ::= "hasTarget" <value>

(16): <containedIn>       ::= "containedIn" "(" <condition> ")"

(17): <containingEl>      ::= "contains" "(" (<conditions> | <hasElCount>)
")"

(18): <hasElCount>        ::= <preNumCondition> "count" "(" <conditions> ")"
(<postNumCondition>)? | (<preNumCondition>)? "count" "(" <conditions> ")"
<postNumCondition>

(19): <hasQueryLang>      ::= "hasQueryLang" <value>

(20): <from>              ::= "from" "(" <conditions> ")"

(21): <to>                ::= "to" "(" <conditions> ")"

(22): <navigateFrom>      ::= "navigateFrom" "(" <conditions> ")"

(23): <navigateTo>        ::= "navigateTo" "(" <conditions> ")"
```

```
(24): <value>            ::= <valueStr> ("ignoreCase")?

(25): <preNumCondition>  ::= <numValue> <numComp>

(26): <postNumCondition> ::= <numComp> <numValue>

(27): <valueStr>         ::= "\"" (~["\""])+ "\""

(28): <dmExpMap>         ::= "[" "<SeRQL>" (~["]"])+ "]"

(29): <dmExp>            ::= "[" (~["]"])+ "]"

(30): <numComp>          ::= "<" | "<=" | ">" | ">=" | "==" | "!="

(31): <numValue>         ::= (["0"-"9"])+

Note: spaces and tabs ("\t") are ignored.
```

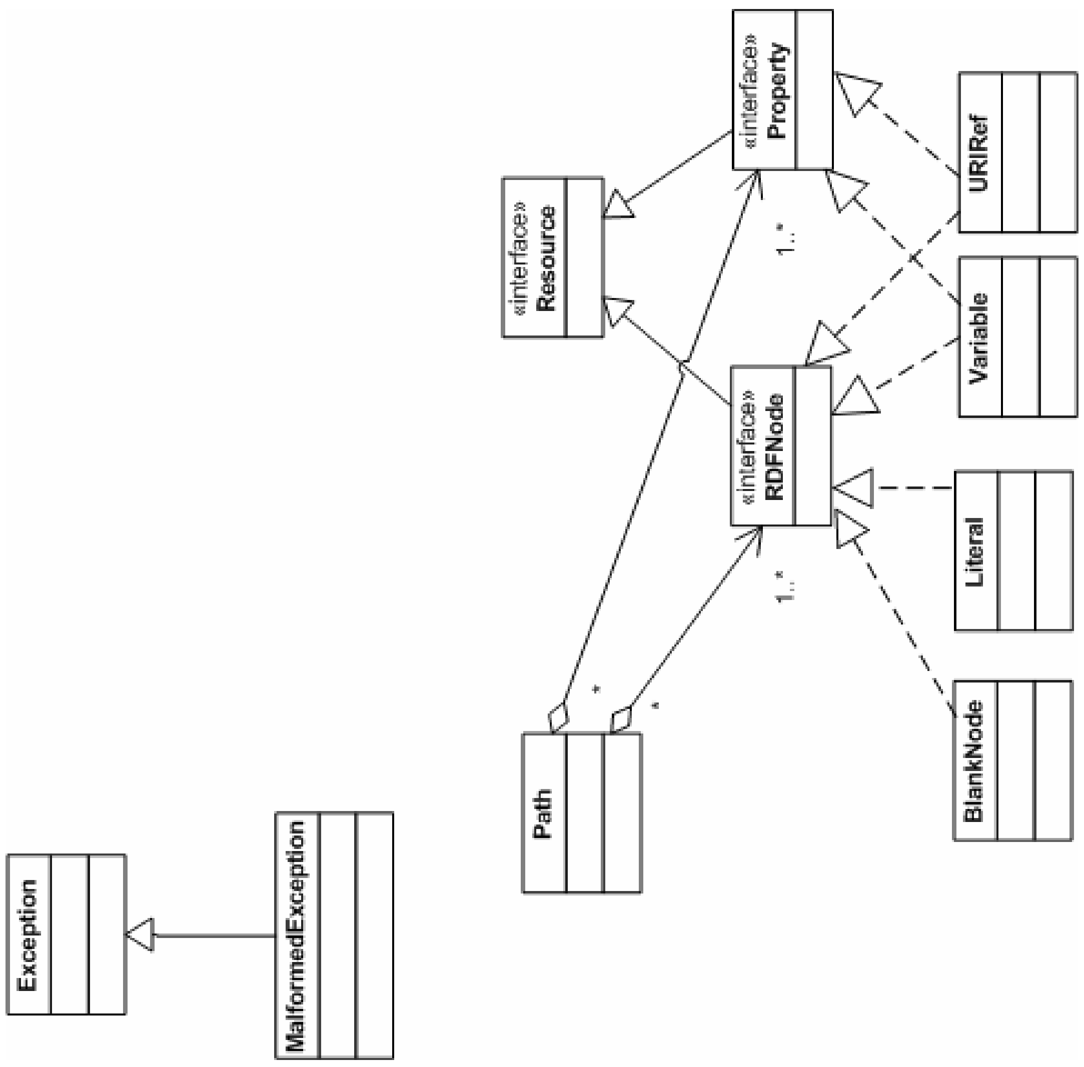
Appendix B: SeAL specification (advice part)

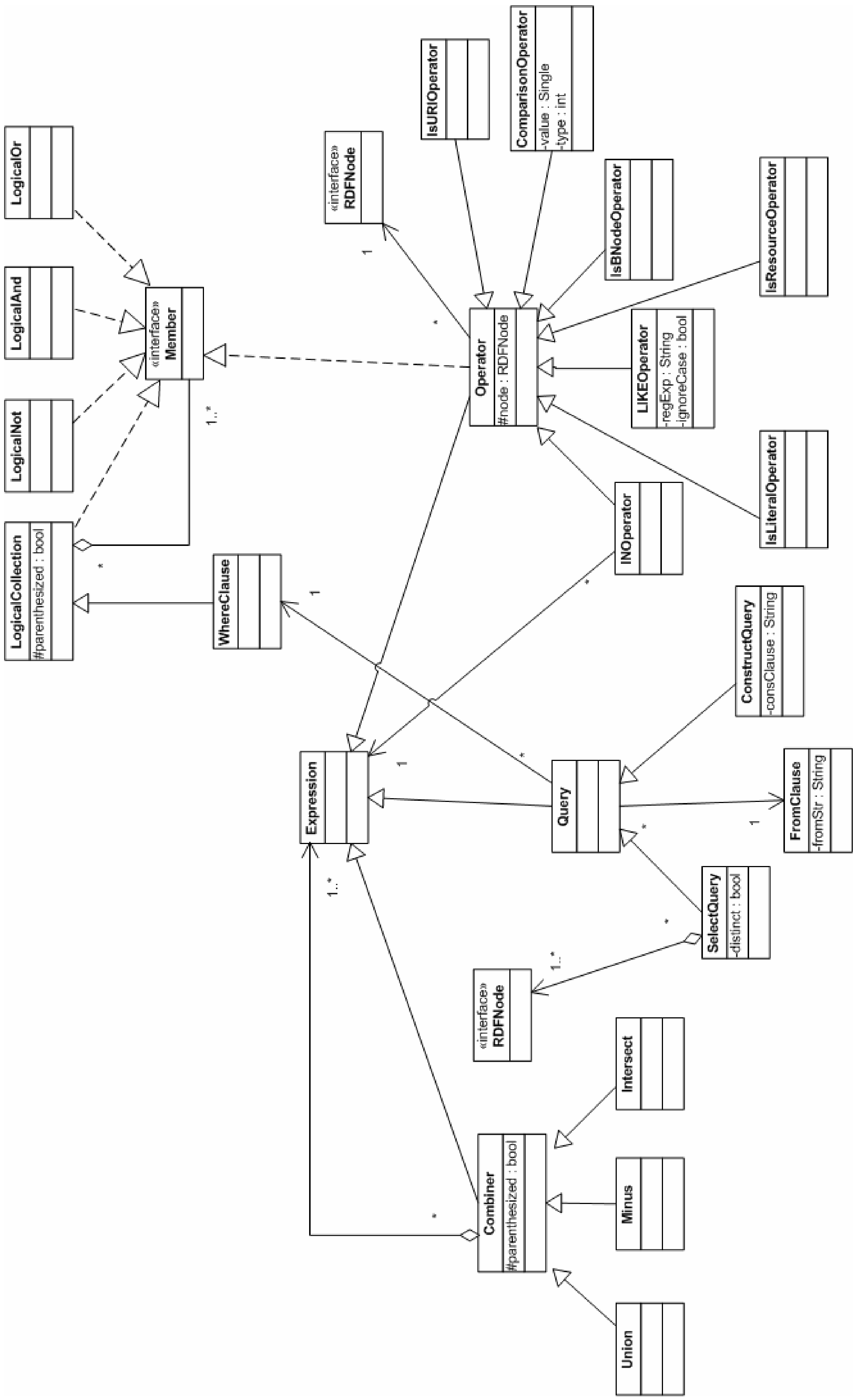
```
(0) : <advice>          ::= "advice:" ((<addCondition> | <condChange> |
<action>) ";")+

(1) : <addCondition>    ::= "addCondition" <conditions>

(2) : <condChange>      ::= <ifCond> (<elseIfCond>)* (<elseCond>)?

(3) : <ifCond>         ::= "if" "(" <conditions> ")" "{" (<action> |
<condChange>)+ "}"

(4) : <elseIfCond>     ::= "else" "if" "(" <conditions> ")" "{" (<action>
| <condChange>)+ "}"

(5) : <elseCond>       ::= "else" "{" (<action> | <condChange>)+ "}"

(6) : <action>          ::= <addEl> | "delete" | <replEl>

(7) : <conditions>      ::= <aCond> ("and" <aCond>)*

(8) : <aCond>           ::= <oCond> ("or" <oCond>)*

(9) : <oCond>           ::= ("not")? <condition>

(10): <condition>       ::= <firstPropComp> (";" <nextPropComp>)*

(11): <firstPropComp>   ::= (<propPath> | <floatNumber> | <intNumber> |
<stringValue>) (<comparator> (<floatNumber> | <intNumber> | <stringValue> |
<condition>))?

(12): <nextPropComp>    ::= (<propBranch> | <floatNumber> | <intNumber> |
<stringValue>) (<comparator> (<stringValue> | <floatNumber> | <intNumber> |
<condition>))?

(13): <propPath>       ::= (<value> ":" <propSeq> | <value>)

(14): <propBranch>     ::= (<value> ":" <propSeq> | <propSeq>)

(15): <propSeq>        ::= <value> ("." (<value> ":")? <value>)*

(16): <stringValue>    ::= <valueStr> ("ignoreCase")?

(17): <valueStr>       ::= "\"" (~["\""])+ "\""

(18): <floatNumber>    ::= (["0"-"9"])+ ("." (["0"-"9"])*)?

(19): <intNumber>      ::= (["0"-"9"])+

(20): <addEl>          ::= "addElement" (<valueStr> | <elSpec>)

(21): <elSpec>         ::= <typeTree> (<valueStr> | (("hasName"
<valueStr>)? "(" "containing" <elDef> ")"))

(22): <elDef>          ::= <elSpec> ("," <elSpec>)*

(23): <typeTree>       ::= <typeKind> (">" <typeKind>)*

(24): <typeKind>       ::= "all" | "unit" | "subUnit" | "form" | "nav" |
"tour" | "single" | "set" | "relationship" | "attribute" | "query" |
"onLoad" | "conditional" | "update" | "standard" | "formEl" | "button" |
"textInput" | "choiceInput" | "source" | "target" | "label"

(25): <replEl>          ::= "replace" <replOperand> "by" <replOperand>

(26): <replOperand>     ::= <subElSpec> ("," <subElSpec>)*

(27): <subElSpec>       ::= <typeTree> ((<valueStr> | "hasName" <valueStr>)
("ignoreCase")?)?

Note: spaces and tabs ("\t") are ignored.
```

Appendix C: SeAL extension for Domain Model

```
(1): <expression> ::= "select" <name> "from" <subExp> ("," <subExp>)*

(2): <subExp>     ::= <path> (";" <branch>)*

(3): <path>       ::= <rdfNode> (<rdfProp> <rdfNode>)+

(4): <branch>     ::= (<rdfProp> <rdfNode>)+

(5): <rdfNode>    ::= "{" <rdfVar> | <rdfURIRef> "}"

(6): <rdfProp>    ::= <rdfVar> | <rdfURIRef>

(7): <rdfVar>     ::= <name>

(8): <rdfURIRef>  ::= "<" <name> ">"

(9): <name>       ::= (~["{","}","<",">"," ","\t","\r","\n"])+

Note: spaces and tabs ("\t") are ignored.
```

Appendix D: Class diagram of serql package




LogicalOr
LogicalAnd
LogicalNot
«interface» Member
LogicalCollection
#parenthesized : bool
WhereClause
«interface» RDFNode
IsURIOperator
ComparisonOperator
-value : Single
-type : int
IsBNodeOperator
IsResourceOperator
Operator
#node : RDFNode
LIKEOperator
-regExp : String
-ignoreCase : bool
INOperator
IsLiteralOperator
ConstructQuery
-consClause : String
Expression
Query
FromClause
-fromStr : String
SelectQuery
-distinct : bool
«interface» RDFNode
Intersect
Combiner
#parenthesized : bool
Minus
Union
1
*
1..*
1
*
1
*
1
1
*
*
1..*
1..*
*

Bibliography

[1] Sven Casteleyn, Zoltán Fiala, Geert-Jan Houben, Kees van der Sluijs. From Adaptation Engineering to Aspect-Oriented Context-dependency. In Proceedings of the 15th international conference on World Wide Web.
[2] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. Aspect-oriented programming. In Proceedings of the Eleventh European Conference on Object-Oriented programming.

[3] P. Brusilovsky. Adaptive hypermedia. User Modeling and User-Adapted Interaction, 11(1-2):87–100, 2001.

[4] Kobsa, A., Koenemann, J. and Pohl, W.: 1999, Personalized hypermedia presentation techniques for improving online customer relationships. Technical report No. 66 GMD, German National Research Center for Information Technology, St. Augustin, Germany.

[5] De Bra, P. and Calvi L.: 1998, AHA! An open Adaptive Hypermedia Architecture. The New Review of Hypermedia and Multimedia 4, 115-139.

[6] Hothi, J. and Hall, W.: 1998, An Evaluation of Adapted Hypermedia Techniques Using Static User Modeling. In Proceedings of Second Adaptive Hypertext and Hypermedia Workshop at the Ninth ACM International Hypertext Conference Hypertext'98, Pittsburgh, PA. Computing Science Reports 98/12, Eindhoven University of Technology, pp. 45-50.

[7] Zukerman, I. and Litman,D.: 2001, Natural language processing and user modeling: synergies and limitations. User Modeling and User Adapted Interaction 11(1-2), 129-158.

[8] Brusilovsky, P. (1996). Methods and techniques of adaptive hypermedia. In User Modeling and User-Adapted Interaction, 6 (2-3), pages 87-129, Springer Science+Business Media B.V.

[9] Irene Garrigós, Jaime Gómez, Peter Barna, Geert-Jan Houben. A Reusable Personalization Model in Web Application Design. Proc. ICWE 2005 Workshop on Web Information Systems Modeling (WISM2005), at ICWE2005, International Conference on Web Engineering, Sydney, Australia, 25 July 2005, p. 40-49, 2005, published by University of Wollongong.

[10] Dayal U. Active Database Management Systems. In Proceedings 3rd International Conference On Data and Knowledge Bases, pp 150-169, 1988.

[11] De Troyer, O. Audience-driven web design in Information modelling in the new millennium, IDEA GroupPublishing, ISBN 1-878289-77-2 (2001)

[12] Ceri, S., Fraternali, P., Maurino, A., Paraboschi, S.. One-To-One Personalization of Data-Intensive Web Sites. In WebDB Workshop (1999).

[13] Bausmeister, H., Knapp, A., Koch, N., Zhang, G. Modelling Adaptivity with Aspects. In Proceedings ICWE2005, Sydney, Australia, pp. 406-416 (2005)

[14] Fiala, Z., Houben G.J. A Generic Transcoding Tool for Making Web Applications Adaptive. In Proceedings of the CAiSE'05 FORUM, Porto, Portugal, 2005, 15-20.

[15] Koch N., Kraus A. The expressive Power of UML-based Web Engineering. Proc. of IWWOST´02, CYTED, pp. 105-119, 2002.

[16] J. G´omez, C. Cachero, and O. Pastor. Conceptual Modelling of Device-Independent Web Applications. IEEE Multimedia Special Issue on Web Engineering, pages 26–39, 04 2001.

[17] S. Ceri, P. Fraternali, and S . Paraboschi. Specification of W3I3 models. Technical Report W3I3PAP2, W3I3 Esprit Project n. 28771, Feb. 1999.

[18] Fiala, Z., Frasincar, F., Hinz, M., Houben, G.J., Barna, P., Meissner, K. Engineering the Presentation Layer of Adaptable Web Information Systems. In Proceedings of the International Conference on Web Engineering, Munich, Germany, pp. 459-472 (2004)

[19] van der Sluijs, K., Houben, G.J., Broekstra, J., Casteleyn, S. Hera-S - Web Design Using Sesame. In Proceedings of the 6th International Conference on Web Engineering, pp. 337-344, Palo Alto,California, USA (2006)

[20] Van Deursen, A., Klint, P. and Visser, J.. Domain-Specific Languages: An Annotated Bibliography. In SIGPLAN Notices 35(6), ACM Press, pp. 26-36 (2000)

[21] Casteleyn, S., De Troyer, O., Brockmans, S. Design Time Support for Adaptive Behaviour in Web Sites. In Proceedings of the 18th ACM Symposium on Applied Computing, pp. 1222 - 1228, Melbourne, USA (2003)