

MASTER

Authoring of adaptation in the GRAPPLE project

Ploum, E.L.M.

Award date:
2009

[Link to publication](#)

Disclaimer

This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

Authoring of Adaptation in the GRAPPLE Project

by
E.L.M. Ploum

Supervisors:

Prof. dr. P.M.E. De Bra
Ir. K.A.M. van der Sluijs

Exam Committee:

Prof. dr. P.M.E. De Bra
Ir. K.A.M. van der Sluijs
Dr. M. Voorhoeve

Eindhoven, August 2009

Preface

"Authoring of Adaptation in the GRAPPLE project": it could have been the title of a master thesis about authoring adaptive applications for the umpteenth Adaptive Hypermedia System. To some extent, indeed it is, since this thesis is about the authoring process of adaptive applications. It is my very pleasure, however, to be able to say that in this thesis we describe a new dimension that has been added to this authoring process by the GRAPPLE project. The merit of this new dimension lies in the fact that it enables adaptive applications to be specified in a generic way, which, in turn, allows adaptive applications to become interoperable.

Since I have entered the GRAPPLE project when it had been running for exactly a year, when I started my graduation process there were many developments in progress but not so many were thoroughly documented and completely decided upon. We could thus say that I suffered from a genuine 'cold start', but fortunately there were three people that have been always willing to help me, answer all my (numerous) questions and discuss the issues that I –sometimes mistakenly- perceived. Therefore, I would like to thank Paul De Bra for his guidance and ability to relate everything together. I also would like to thank Kees van der Sluijs very much, who was my tutor in its broadest sense, who has had an enormous amount of patience with me, and who helped me to put everything in perspective over and over again. And finally I would like to thank David Smits, my roommate and GALE-genius, for his advice and ideas on more than just my technological questions. Lastly, I would also like to thank all the members of the Information Systems group for the great time that I have had with them for the last seven months.

Eva Ploum
Veghel, August 2009

Table of Contents

PREFACE	2
TABLE OF CONTENTS	3
LIST OF ABBREVIATIONS	6
1 INTRODUCTION	7
1.1 Adaptive Hypermedia Systems.....	8
1.2 Components of an Adaptive Hypermedia System.....	11
1.3 Authoring for Adaptive Hypermedia Systems.....	12
1.4 Goals of this thesis.....	13
2 BACKGROUND MATERIAL	14
2.1 The GRAPPLE Project.....	14
2.2 Authoring in GRAPPLE.....	14
2.2.1 Domain Model.....	14
2.2.2 User Model.....	14
2.2.3 Concept Relationship Types.....	15
2.2.4 Conceptual Adaptation Model.....	16
2.2.5 GRAPPLE Authoring Process.....	16
2.3 Running example.....	19
2.3.1 Domain Model of the Example Application.....	19
2.3.2 Definition of the Behaviour of the Milky Way Example Application.....	21
2.3.3 How the Milky Way example application should look like.....	23
2.4 Thesis overview.....	24
3 ENGINE INDEPENDENT APPLICATION SPECIFICATION	25
3.1 Generic Adaptation Language.....	25
3.1.1 GAL Query Language.....	27
3.2 UM Specification Rules.....	28
3.3 DM Specification.....	29
3.4 UM Specification.....	30
3.4.1 UM Specification Rules for the Milky Way Application.....	32
3.5 GAL Specification.....	32
3.5.1 The Guided Tour Units.....	33
4 GRAPPLE AUTHORING MODELS	35

4.1	Domain Models in GRAPPLE.....	35
4.1.1	Types of Domain Models.....	35
4.1.2	Decoupling Application Dependent and Application Independent information	36
4.1.3	The Layout Domain Model	36
4.1.4	DM Format.....	36
4.2	Concept Relationship Types in GRAPPLE.....	37
4.2.1	Classes of CRTs.....	38
4.2.2	Predefined CRTs	40
4.2.3	CRT Format.....	46
4.3	Conceptual Adaptation Model.....	48
4.4	Three Approaches to Defining the Page Structure	49
4.4.1	Manual Approach	49
4.4.2	Automated Approach.....	50
4.4.3	Template Based Approach.....	53
5	TRANSLATION OF AUTHORING MODELS INTO ENGINE INDEPENDENT SPECIFICATIONS	58
5.1	Bases of the DM, UM and GAL specifications	58
5.1.1	DM specification	58
5.1.2	UM specification	58
5.1.3	GAL specification.....	59
5.2	Translating CRTs into GAL and User Model specifications.....	65
5.2.1	Auxiliary CRTs in the Engine Independent Application Specification	66
6	GALE.....	68
6.1	Architecture of GALE	68
6.1.1	Adaptation Engine	68
6.1.2	Eventbus.....	70
6.1.3	DM Service	70
6.1.4	UM Service	70
6.2	Fetching an Attribute Value in GALE.....	70
6.3	GDOM File.....	71
6.3.1	Layout Definition in GDOM.....	72
6.4	GDOM file of the Milky Way Example Application.....	73
7	COMPILING GAL INTO GALE ENGINE FORMAT	77
7.1	GDOM File.....	77
7.1.1	DM Specification.....	77
7.1.2	UM Specification.....	78
7.1.3	GAL Specification	80
7.2	Page Structure Specifications in GDOM	81

8 FUTURE WORK AND CONCLUSIONS 82

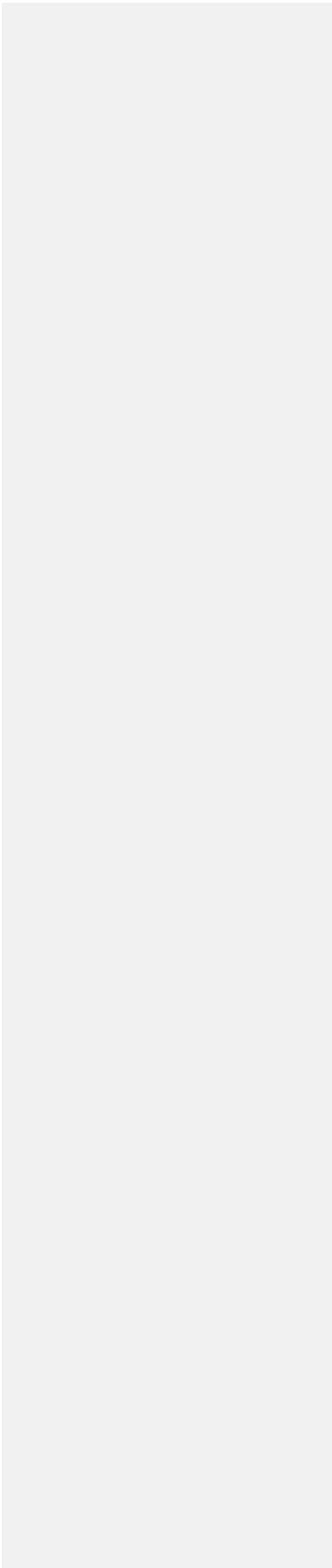
REFERENCES 83

APPENDIX A 85

APPENDIX B 91

APPENDIX C 96

APPENDIX D 98



List of Abbreviations

GRAPPLE	Generic Responsive Adaptive Personalized Learning Environment
DM	Domain Model
UM	User Model
CAM	Conceptual Adaptation Model
CRT	Concept Relationship Type
GAL	Generic Adaptation Language
GAT	GRAPPLE Authoring Tools
GALE	GRAPPLE Adaptive Learning Environment
ALE	Adaptive Learning Environment
AHS	Adaptive Hypermedia System
LMS	Learning Management System
TEL	Technology Enhanced Learning

1 Introduction

An Adaptive Hypermedia System (AHS) is a hypermedia system¹ that can adapt the presentation and the content of pages to the user. This adaptation is done by means of a User Model (UM) in which information about the user that is relevant for the adaptation is stored. By tailoring, or personalising, (the presentation of) the information to the user, its utility is increased [Ste2009].

Since personalisation of information is especially well-known in education, the 'Adaptive Learning Environment' (ALE) is the type of AHS that the GRAPPLE² project –which is strongly related to this thesis– puts its focus on. An example of how personalisation can be applied to education is adaptation to students' learning styles. Different learning styles exist and there are indications that students learn better if they are taught according to their preferred learning style [Sta2006].

In order to be able to offer adaptive courses –also called 'adaptive applications'– to students, first they must be developed by an author. For wide adoption of what is called 'Technology Enhanced Learning' (TEL), of which adaptive courses are part, normal high school teachers should thus be able to do the required authoring, and therefore usually special authoring tools are provided for this. These authoring tools enable the author to create high level models of the course, thus protecting him/her from the complexity that adaptive applications can easily bring about. In particular, the author usually only defines the course's units of information and associates adaptation behaviour to these units, such as in which order the units must be studied. The authoring environment will then translate this high level description of the application into a series more low level.

The set of descriptions of the application that is produced by the authoring tools, however, is still only a description. To actually use the adaptive course it must be run on an *Adaptation Engine* (AE). An adaptation engine, though, requires a collection of rules that tells the AE how to behave rather than a set of descriptions of the final application. The authoring models must thus be translated into something that the AE understands, and this is exactly what causes that most adaptive applications are tied to a specific authoring tool and adaptation engine. When an AHS is developed, very often an authoring environment that is tailored to the AHS is developed along, that is, the authoring tools then produce output in a format that is already readable for the AHS's Adaptation Engine (see Figure 1, top). This obviously jeopardises the interoperability of both the authoring tools, which produce system-specific output, and the adaptive application, which can only be run on a predetermined AHS. Since there are so many different authoring tools and AHSs, this lack of interoperability has become a burden on the adoption of TEL. Even though there is a trend to personalise learning material, TEL can not really take off because one often needs different AHSs for different adaptive courses, and the willingness to support multiple systems is only limited.

To solve this problem of interoperability, the GRAPPLE project introduces a generic intermediate language for the expression of adaptive applications' navigational structure: Generic Adaptation Language (GAL). If we would use the analogy of baking a pie for the authoring process of an adaptive application, the navigational structure of an adaptive application would be like the pie's characteristics, like its taste and texture. Authoring tools of traditional AHS would directly translate the high level authoring models, a description of the pie, into the pie's ingredients. In GRAPPLE, however, an Engine Independent Compiler translates the authoring models of the application into an Engine Independent Application Specification, which is written in GAL, that functions as a kind of *recipe* for the application. An adaptation engine can then use this recipe to select its own ingredients and bake its own version of the application, by having a compiler that translates the Engine Independent Application Specification into rules that the Adaptation Engine understands. The introduction of this intermediate step that describes the application in a generic way, thus enables the creation of adaptive applications with any authoring tool that produces GAL and their execution on any adaptation engine that can process GAL (see Figure 1, bottom). This would require a single effort to write compilers for the translation from authoring tools to GAL and from GAL to adaptation engines.

In this thesis we investigate the authoring process from GRAPPLE's authoring tools to an adaptation engine. The main focus thereby will be on the expressivity of and the requirements for the output format of the GRAPPLE Authoring Tools, the intermediate language (GAL) and the input format of the GRAPPLE Adaptation Engine. Furthermore, we will examine how these three languages can be translated into each other as a preparation for the development of the compilers between these languages that will be built in the near future.

We will now first give a brief introduction to the field of Adaptive Hypermedia, and then present the goals of this thesis. Then the GRAPPLE project is introduced, after which we give an overview of the structure of the remainder of this thesis.

¹ Further reading on Hypermedia is available through the course 2ID65, at <http://www.wis.win.tue.nl/sakai>

² Generic Responsive Adaptive Personalised Learning Environment

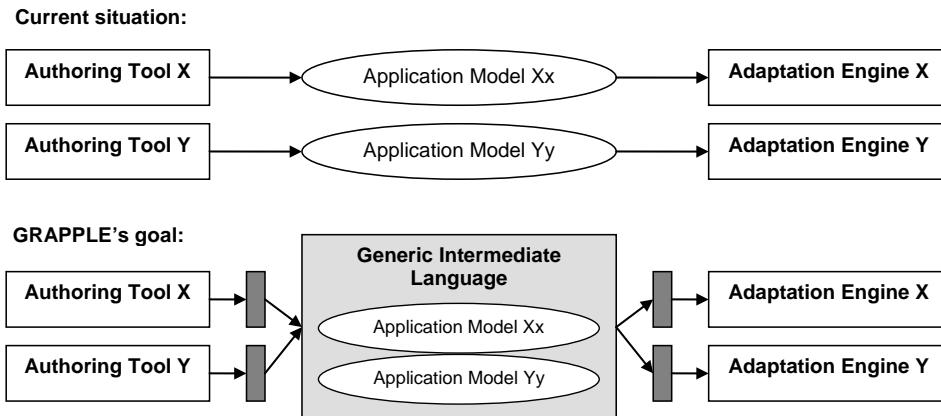


Figure 1: Interoperability of Adaptive Applications by the introduction of a Generic Intermediate Language (and the addition of compilers)

1.1 Adaptive Hypermedia Systems

In this section we briefly give a general introduction to Adaptive Hypermedia Systems. An adaptive hypermedia system has three key features according to Brusilovsky [Bru1996]:

- It is a hypertext or hypermedia system
- It uses a user model in which some information about the user is stored
- It “adapts various visible aspects of the system to the user”

In practice this means that a collection of resources (text, images, simulations, etc.) is somehow structured into logical information units, usually referred to as ‘concepts’, which are accessed by the user through ‘pages’ in a browser (even when the user is presented with a simulation, we refer to this with ‘page’). The exact content, structure and presentation of a page when it is requested is determined by the adaptation engine of the system and is based on the information in the user model.

Navigation between the concepts/pages is possible by hyperlinks, which are usually highlighted objects on a page that cause the AHS to access another concept (and thus another page) when clicked (see Figure 2). The destination of a hyperlink can be hardcoded or vary depending on some condition.

The collection of hyperlinks, the adaptive behaviour of the application and the structure of the pages will together be referred to as the ‘navigational structure’ of the application. The navigational structure of an adaptive application is what provides the unique learning experience to users.

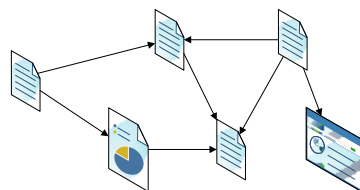


Figure 2: Possible link structure between pages in a hypermedia system

An important remark that must be made is that ‘adaptive’ is *not* the same as ‘automatic’ or ‘adaptable’. An *automatic* system decides which action to take based on fixed rules that specify when to choose which alternative and hold for all users. A simple search facility that selects all books from a database searches all elements for a certain keyword would be an example of an automatic system. An *adaptable* system has multiple alternatives for its rules, but which rule is selected is decided by some external force (e.g. the user or an administrator) [Kap1993], [Vas1996]. It would for example show scientific or fiction books about ‘X’ based on whether the user is searching in the ‘Education mode’ or in the ‘Literature mode’. Finally, an *adaptive* system (of the first order) is capable of *observing* the situation for a particular user and deciding *on its own*

which rule is most suitable to use for that instance of the user model [Höb1995]. An adaptive system would observe whether the user shows more interest to scientific or to fiction book, and decides based on that which mode to use for a search.

There are many examples of AHS; some well-known ones are:

- Interbook (see Figure 3), by Peter Brusilovsky et.al. [Bru1998],
- AHA! (see Figure 4), by Paul De Bra et. al. [Bra1998],
- SQL-Tutor (see Figure 5), by Tanja Mitrovic et.al. [Mit1998], and
- ELM-ART (see Figure 6), by Gerhard Weber et.al. [Web1996].

As the figures illustrate, both the functionality of and the representation of information by the systems differs, but they all aim to teach a body of information to the student and do that adaptively.

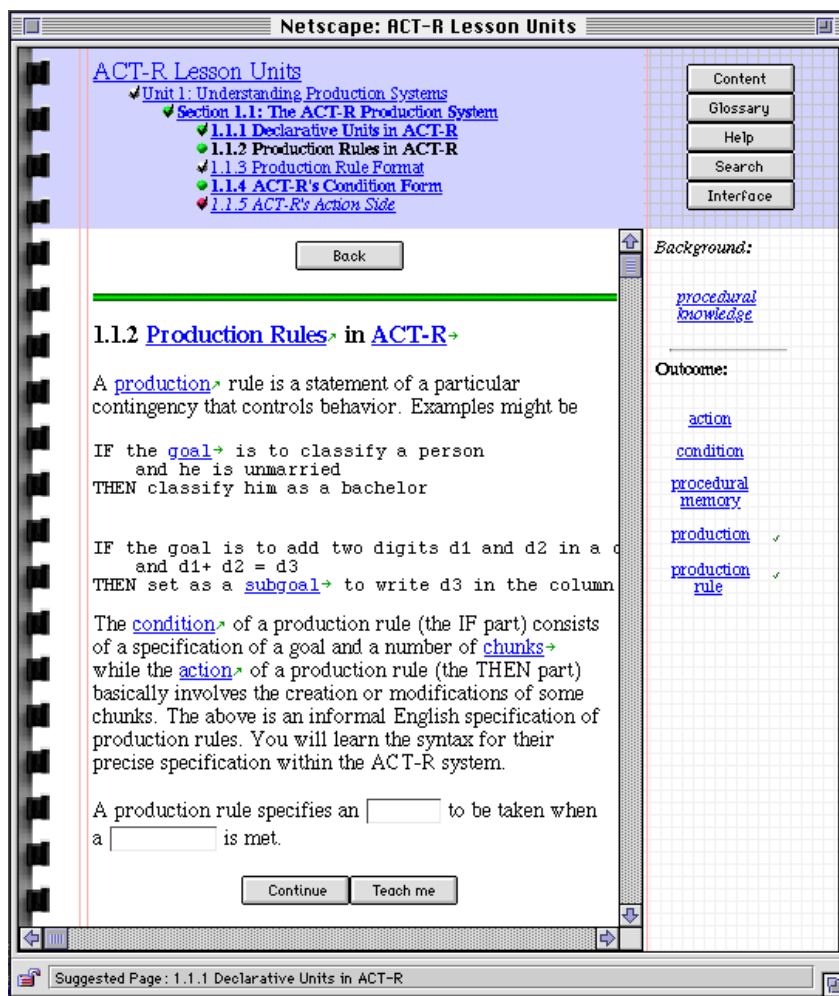


Figure 3: Screenshot from Interbook

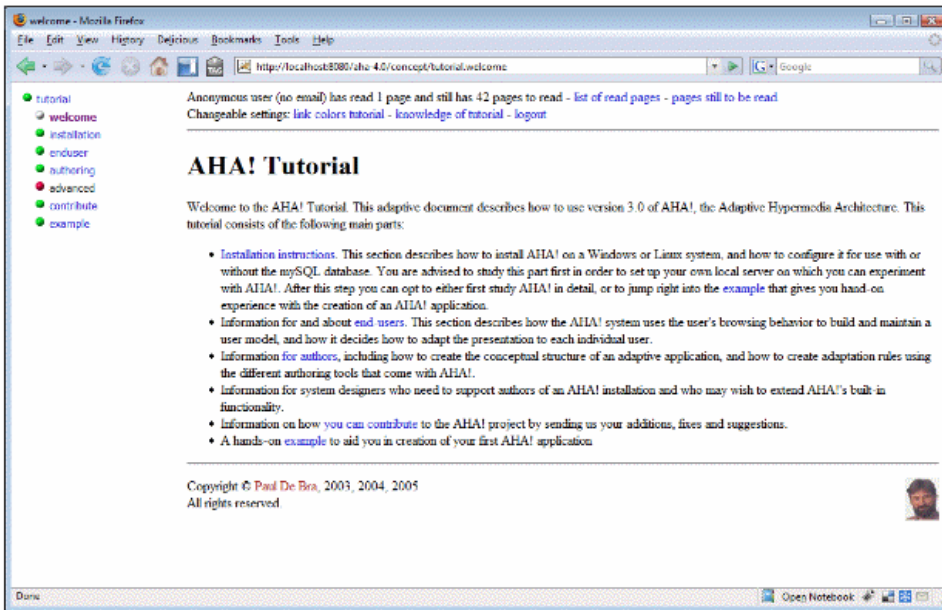


Figure 4: Screenshot from AHA!



Figure 5: Screenshot from SQL-Tutor

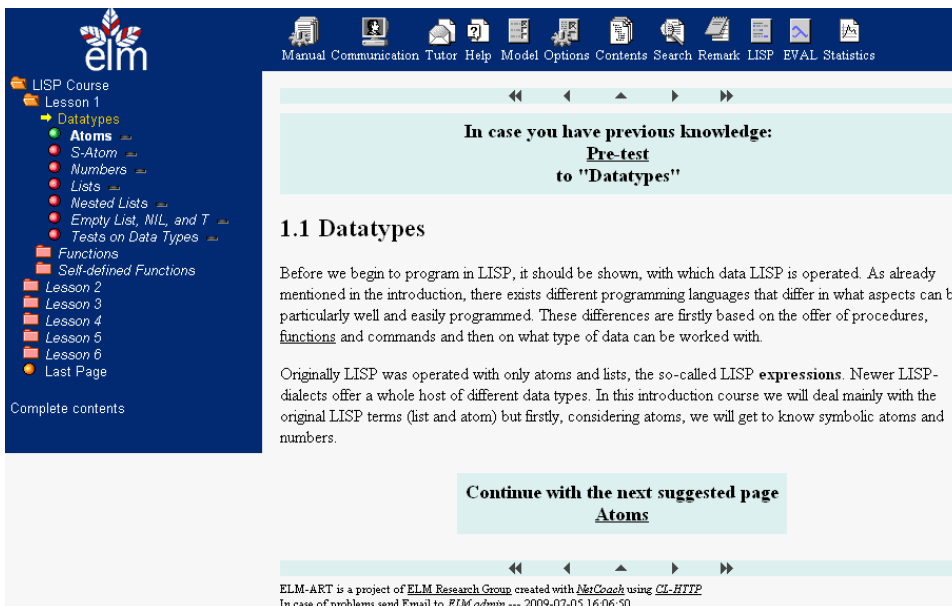


Figure 6: Screenshot from ELM-ART

1.2 Components of an Adaptive Hypermedia System

Although the implementation per Adaptive Hypermedia System differs, all AHSs require the presence of some general components. In order to be able to abstract from the details of these components per AHS, the AHAM reference architecture [Bra1999] was introduced (see Figure 7). According to this framework, in general an AHS consists of at least the following components:

- Domain Model (DM)
- User Model (UM)
- Adaptation Model (AM)
- Adaptation Rules
- Adaptation Engine

The Domain Model carries domain specific information of the adaptive application; usually it is a sort of ontology or taxonomy³ that consists of *concepts*, which are logical units of information, and *concept relationships*, which describe the semantic relationships between concepts. In this thesis, we will also refer to the latter with 'DM relationships' or 'semantic relationships'. Furthermore we explicitly assume that they only describe the domain-related relationships, and that thus no application dependent information is expressed by them.

The User Model contains three types of information: application independent user information, contextual information and information that relates the user to the DM concepts. The part of the UM that relates the user to the DM concepts is usually an overlay model of the DM, meaning that the concepts are present in the UM as well with some extra attributes added (e.g. number of times that the user visited the concept).

In adaptive applications, each user action that is of interest to the system is somehow registered in the User Model. Usually not the very actions themselves are registered, but rather information that is derived from them (e.g. not the fact that the user clicked a particular link is stored, but the fact that (s)he accessed a particular

³ Neither 'ontology' nor 'taxonomy' are in fact terms that exactly apply here. The DM is a model of the domain according to the insights of the author only. An ontology would suggest that there is a clear hierarchy that is generally agreed upon by experts [Vos2007], while a taxonomy suggests that the concept hierarchy of the domain model is inherent to the domain (i.e. there is a natural hierarchy) [Ree2003]. Neither has to be the case for the DM as we use it here; the bottom line is that the information is somehow logically structured according to the domain.

concept). The author decides, restricted by the capabilities of the AHS, which actions are of interest; this may go as far as counting the user's eye blinks, but it is important to keep in mind what information is useful and what is not.

There are two techniques for registering user information which are distinguished by the moment on which the interaction information is logged and how it is stored. *Forward reasoning* immediately draws conclusions from user actions and stores them in the User Model, e.g. from clicking on a page that represents concept X it is concluded that knowledge about X has increased by 25%, so the knowledge value in the UM is increased by that amount. Information from the UM is immediately available in case of forward reasoning. In *Backward reasoning* the user's actions are stored without drawing conclusions from them yet. For adaptations it is then first decided which information of the UM is needed, and then the event log is searched for events that where of influence on the attribute of interest (and then value is computed). In this thesis we assume that forward reasoning is used.

The Adaptation Rules describe behaviour of the adaptive application (not *the* behaviour), that is, they describe possible behaviour that the application can conduct, not the complete behaviour of the application. In particular, an adaptation rule describes how its input concepts together with the state of the User Model influence the output concepts. The AHAM reference architecture distinguishes *generic adaptation rules*, which "associate adaptation behaviour with concept relationships (or other structures)", and *specific adaptation rules*, which describe the adaptation behaviour between specific, directly-addressed, concepts [Wu2002].

The Adaptation Model of an adaptive application is defined by Wu as its "set of generic and specific adaptation rules" [Wu2002]. Another way to describe it is the structure of adaptation rules that is imposed on the DM. Starting from the above description of adaptation rules, this means that the AM indeed describes *the* behaviour of the application; in particular it describes how the UM is updated and the adaptation of the application.

Finally, the Adaptation Engine interprets the rules of the Adaptation Model thereby providing the user with the actual adaptive application. An example of how an adaptation process work might work is: when a user requests access to a concept, first the appropriate resource for the page of the concept is selected. If the resource contains any adaptive parts, this is handled. Lastly, the structure and presentation of the page is finalised.

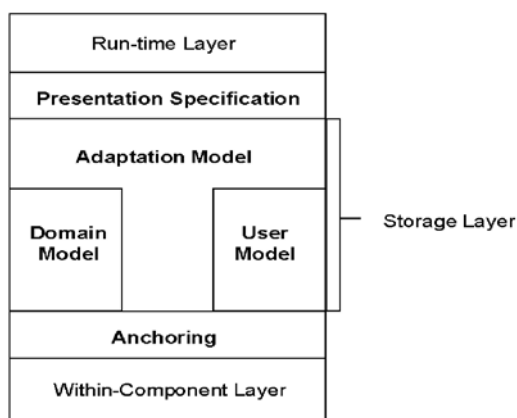


Figure 7: AHAM Reference Model

1.3 Authoring for Adaptive Hypermedia Systems

Most Adaptive Hypermedia Systems have their own authoring environment for the creation of adaptive applications. This enables authors create model the application in a relatively easy way. Most authoring environments have a similar approach to the design process: the author should first Domain Model for the application in which the concepts and concept relationships are specified. Furthermore, the resources that describe the domain must be associated with the concepts. There are several methods for relating the conceptual structure of the application to the resources. The most popular alternatives are to explicitly assign resources to concepts on the one hand, and on the other hand the usage of a Resource Model from which the most suitable resource for a concept is selected at runtime.

Finally, the author should often specify some *concept hierarchy* that indicates the (partial) ordering among the concepts. The concept hierarchy is often considered the adaptive hypermedia equivalent of a book's table of

contents; in Figure 3 the a view on the concept hierarchy is depicted on the top of the page, and in Figure 6 it is depicted on the left of the page. Note, that adaptive applications often only show the relevant part of the concept hierarchy on pages, rather than the entire hierarchy; in GRAPPLE we call this view on the concept hierarchy the *treeview*.

The author should also define some Adaptation Model (usually in the form of learning strategies) onto the DM. This can be done in several ways, depending on the specific authoring tool. Some tools, like the Graph Author, require the author to design a learning strategy themselves and provide basic components to implement it. In case of the Graph Author the author selects a Concept Relationship Type (CRT) –a pedagogical relationships that comprises one or more related adaptation rules- and then either uses it to connect specific concepts to it or to associate concept relationship with the CRT. An example would be to associate the DM relationship 'is-a' with a CRT about knowledge propagation, which would result in the behaviour that if the user studies a child concept of an 'is-a' relationship, some of the knowledge thus acquired would propagate to the more general concept's parent.

Another way to create an AM for the DM is the approach that Christea takes in My Online Teacher (MOT) [McD2009]: the author constructs a 'goal map' based on the DM, which is then used to create adaptive lessons. The adaptation behaviour is thus not directly specified by the author, but rather derived from the structure of the lessons.

The adaptation rules that are used by the application are usually predefined. Most authoring tools, though, also allow the author to define such rules by themselves. Since this requires knowledge about the working principles of the AHS, this is normally only done by advanced authors.

Optionally, depending on the authoring environment, the looks and feels of the application can be defined in a Presentation Model.

1.4 Goals of this thesis

The nature of my graduation project has been rather practical. Instead of the classical approach in which first a problem is stated, then a method to solve the problem is proposed and finally the methods and the resulting solution are discussed, we have in this case taken a bottom-up approach. This means that after the status quo of the GRAPPLE project (at the time) was studied, we started to work on an example application. The work on this application raised many issues, and the process of discovering and solving those issues has been the main goal of my graduation project. As a result of this approach the nature of this thesis is rather descriptive. It is not so much a report of my activities during the graduation process, but rather a description of the results of the work that was done.

The goals of my graduation project are:

1. Exploring and documenting the GRAPPLE authoring process,
2. Exploring, improving and documenting the expressivity of the models that are used in the authoring process: output models of the GRAPPLE Authoring Tools (GAT), GAL and the GRAPPLE Adaptation Engine format (GDOM),
3. Exploring, documenting and suggesting improvements for the functionality of the GRAPPLE Adaptation Engine,
4. Preparations for compilers from the final output model of the GRAPPLE Authoring Tools to GAL, and
5. Preparations for compilers from GAL to the GRAPPLE Adaptation Engine format (GDOM)

The parts of the goals with an exploratory nature, have been achieved by the development of the example adaptive application. This application was written by hand in the output of the GAT authoring models, in GAL and in the GRAPPLE Adaptation Engine format, in order to provide detailed insight in the authoring process and to get hands on experience with them. Together with a thorough study of the (relevant) project material that has been developed so far, and through active participation in the considerations of the (relevant) project partners, this has provided the foundation for this thesis.

2 Background Material

In this chapter we first give an introduction to the GRAPPLE project and the authoring process in GRAPPLE. Then we describe the Milky Way example application, which was written as a part of this thesis. Finally, a brief thesis overview is given.

2.1 The GRAPPLE Project

As we mentioned already, the context of this thesis is the GRAPPLE project. We therefore briefly introduce the project, before turning to the description of the authoring process.

GRAPPLE is an abbreviation for 'Generic Responsive Adaptive Personalised Learning Environment', which refers to the core goal of the project: development of a generic and therefore portable learning environment that can be adapted to the user. Ideally a user should be able to carry his/her User Model along from one adaptive course to another. The great advantage of this would be that the user's UM becomes very rich which enables the ALEs to adapt to the user better and better.

The GRAPPLE consortium wants to achieve this goal on the one hand by the integration of Learning Management Systems (LMS) with Adaptive Hypermedia Systems, such that personalised learning management and personalised learning can be offered together (that is, within the same application or service), and on the other hand by proposing a generic intermediate language which enables the decoupling of adaptive applications' navigational structures from the AHS they run on and thereby enabling their interoperability. The former sub goal would allow AHSs to communicate with LMS, such that an AHS can share useful information with the LMS that it is integrated in. The latter sub goal would enable any adaptive application to be run on any AHS (within any LMS). Note that such an application could then again benefit from any knowledge that the LMS has about the user (which possibly originated from other adaptive courses).

The project is an international cooperation between 15 universities and companies from 9 countries. The most important partner within the scope of this thesis, apart from the Eindhoven University of Technology, is the University of Warwick which takes care of the GRAPPLE authoring tools. The Generic Adaptation Language (GAL), the generic intermediate language, and the GRAPPLE Adaptive Learning Environment (GALE) are developed in Eindhoven.

2.2 Authoring in GRAPPLE

Since the authoring process of adaptive courses for GRAPPLE is the focus of this thesis, we will in this section describe this process from the very beginning, in which an author starts creating authoring models, to the end, in which the application is actually run on the GALE Adaptation Engine. First, however, we give a brief introduction to the four authoring models that the author creates with the GRAPPLE Authoring Tools (GAT).

2.2.1 Domain Model

The Domain Model (DM) contains concepts and concept relationships that together express the structure of the application's domain. Furthermore, the DM contains the information resources that describe the domain, which are in GRAPPLE explicitly associated to the concepts.

There are basically two types of information entities that can be associated with a concept: *resources* and *facts*. The former is the URI of or the URL to a file containing information that somehow describes the concept, like a picture, a simulation or a textual description. The latter is indeed simply a fact about the concept, like for example "the Sun's diameter is 1.392.000 km"⁵. Both types of information are referred to as *properties* of the concept.

For each concept in the DM, facts are simply stored in string form, and resources have their URIs stored. The properties can be referred to with their *name*: e.g. "image" for a property that stores a URI of an image of the concept or "diameter" for a property that stores the string "Diameter: 1.392.000 km". Finally, resource properties can (optionally) also have an associated label that describes the resource.

2.2.2 User Model

The User Model (UM) is an overlay model of the DM –which means that it contains all concepts from the DM– and is intended for the storage of user specific information per concept. This is information like the number of

⁵ Source: <http://en.wikipedia.org/wiki/Sun>

times that a user visited a concept, his/her knowledge of the concept or whether the concept is suitable for the user to study or not. It may also store other user dependent information like the user's password, his/her e-mail address or the type of device that the user is currently viewing the application with.

During the authoring process of the adaptive application, the infrastructure for the storage of this information is created. This is done by the creation of (*UM*) *attributes*, which are elements for the storage of (user) data that changes at runtime due to updates by the Adaptation Engine. To put it differently: data that is unchangeable (at runtime) is always stored in *properties* and all data of which the value may differ from time to time is stored in *attributes*.

Attributes contain more information than properties: besides a name, an attribute also has a data type, properties that indicate the attribute is 'persistent' 'public' and 'readOnly', and a default value. The data type indicates the type of the data that will be stored in the attribute at runtime, e.g. it could store that 'knowledge' is stored as a Real. The 'persistent' property indicates whether the value of the attribute is at runtime to be remembered by the system or that it must calculate the value when it is needed⁶, e.g. the number of visits that the user has paid to a concept is usually remembered and thus 'persistent', and the attribute that indicates whether the concept is suitable to study for the user is usually calculated at the time when it is needed and thus not 'persistent'. The 'public' property indicates whether the value of the attribute is also available to other applications than the application that the attribute was created for. The 'readOnly' attribute indicates whether at runtime the AE is allowed to update the value of the attribute or not; when the attribute value is 'borrowed' from another application (where the attribute thus must have been declared to be public), this may be the case. Finally the default value of the attribute is related to the 'persistent' property: if the attribute is persistent, the default value indicates the initial value of the attribute, that is, the value of the attribute when the user is newly introduced to the application. If the attribute is not persistent, the default value contains the expression from which the attribute's value can be calculated when it is needed.

Authors do not have to create attributes (and their data type, default value and properties) for the DM concepts by themselves, rather this is automatically done by the compiler when the author uses *Concept Relationship Types* (see next section). In theory an author may even be unaware of the existence of attributes.

2.2.3 Concept Relationship Types

Concept Relationship Types (CRTs) describe adaptive behaviour in generic, application independent terms. This means that they must describe how concepts would influence each other without specifying those concepts. It also means that the CRT should be applicable to any application, and that thus no application specific information may be explicitly referred to and no specific concept hierarchy may be assumed. CRTs can thus briefly be said to be 'abstract descriptions of behaviour'.

Technically, CRTs consist of a piece of code that describes the behaviour and which uses *placeholders* for the concepts. A CRT could thus for example express that it involves two unspecified concepts, ?A and ?B, and that the "knowledge of concept ?A is the knowledge of concept ?B times three", without specifying what concept ?A and ?B are. Because the CRT assumes that the concepts that are used to implement the CRT have certain UM attributes, e.g. ?A and ?B are assumed to both have a 'knowledge' attribute, it also contains a brief definition of which UM attributes it expects the concepts that use this CRT to have.

Because it is not a trivial task to specify behaviour in such generic terms, the adaptation behaviour of the application that authors specify in the Conceptual Adaptation Model is mostly based on *predefined* CRTs, i.e. CRTs that have been pre-programmed in the authoring tool. Authors can, however, define their own CRTs with the CRT tool; probably this can mostly be done by modifying existing CRTs.

In CRTs it is sometimes necessary to refer to the 'role' that one concept fulfils for the other, without actually referring to the DM relationship that exists between those concepts –sometimes there may not even exist a DM relationship between the concepts. We, for example, all know that a leaf relates to a branch like how a nail relates to a finger, but we may use different semantic relationships to express those relations (e.g. 'hangs on' and 'sits on' respectively). Yet in a CRT, which must be application independent, we must be able to refer to these relationships. To solve this problem, we recognise the existence of special CRTs that do not specify any behaviour, but rather specify such a 'role'. These CRTs can be applied to concept relationships (note that this differs from normal CRTs which are instantiated by concepts), such that we can in the CRTs refer to them by simply referring to their role.

The most important *Auxiliary* CRT is the 'parent' CRT; the 'parent' CRT indicates which concepts are more general than other concepts, and the concept hierarchy of the application is derived from this CRT. Examples of typical names of concept relationships that could have the 'parent' role are 'is-a', 'hasType' and 'subClassOf'. Whenever, in this thesis, we refer to 'parent concepts' or 'child concepts' we refer to the parents and children according to the 'parent' concept relationship, i.e. according to the concept hierarchy. The CRTs use the placeholder "\$hasParent" to indicate that the concept relationship with the 'parent' role for that context

⁶ Indeed, similar to the notion of 'persistence' that we know in the field of databases.

should be substituted.

Another important Auxiliary CRT is the 'semantic parent' CRT, which indicates that a DM relationship points at the semantic parent of a concept. It may be the case that the concept hierarchy is based on a classification that deviates from the parent relationships that would be natural from a semantic point of view, as is the case in the Milky Way application (see Section 2.3). In the Milky Way application the parent relationship is mostly based on types, e.g. Venus is-a Planet, while the semantic relationship would be based on the structure of the solar system, that is, Venus isPlanetOf Sun.

2.2.4 Conceptual Adaptation Model

The Conceptual Adaptation Model (CAM) specifies the adaptation behaviour of an adaptive application. It is in fact a collection of instantiated CRTs, that is, it binds the placeholders in the CRTs to concepts from the DM. Rather than the behaviour that is expressed in CRTs, the behaviour that is specified in the CAM is thus application specific.

The CRTs are instantiated with concepts as follows: the CAM tool displays the DM to the author and allows for the selection of CRTs. When a CRT is selected, one or more *sockets* appear on the screen to which the author can drag concepts from the DM. How the concepts that are dragged into these sockets are bound to the placeholders of the CRT, is defined in the CRT. By the description of the CRT, it is ensured that the author has a clear idea of what the sockets represent and how the CRT's behaviour is applied to the concepts that are dragged into them. If the author would, for example, use a CRT that will count the number of visits to a concept at runtime, the socket that would appear on the screen could display "count the user's visits to these concepts".

A particularly important part of the adaptation that is handled in the CAM tool is *resource selection*. When a concept has multiple resources associated with it in the DM, it may be the case that there are multiple candidate resources to fulfil the same role. An example would be that the page that represents a concept should contain one image while there are multiple image files associated with the concept. Resource selection is in this case used to allow the author to specify which resource is selected under which condition, and furthermore it allows him/her to address this entire selection construction by a (simple) name. This is done by storing the resource selection query in a *resource attribute*; addressing the resource attribute will always return the right value under the right condition.

It is in particular for resource selection that (resource) properties (in the DM) are named: rather than referring to the URI of the resource in the selection query, one refers to the value of the *property of that stores the resource*. This allows the author to specify for example that if the user views the page through a mobile device then show "small_picture" and otherwise the "big_picture". If the author later on decides to show other images, (s)he only has to change the URI stored in the concept's "small_picture" property.

It is not clear yet how the resource selection mechanism of the authoring tools exactly will work, but for the larger part of this thesis we can abstract from this. In case it is indeed of importance to consider the details of this functionality, the various alternatives will be discussed.

2.2.5 GRAPPLE Authoring Process

The authoring process within the GRAPPLE framework is illustrated in Figure 9. At the top of the figure, the author and the four components of the GRAPPLE Authoring Tools are depicted. The GRAPPLE Authoring Tools (GAT) is a generic set of authoring tools for the creation of adaptive applications. The GAT consists of four components the Domain Modelling tool (DM tool), the Concept Relationship Type (definition) tool (CRT tool), the User Modelling tool (UM tool) and the Conceptual Adaptation Modelling tool (CAM tool). Together they produce the authoring models that describe the adaptive application. Of these tools, the author will use the DM tool and CAM tool in particular for the (basic) modelling of the application. Figure 8 illustrates how the Domain Model and (predefined) CRTs are used to produce the Conceptual Adaptation Model.

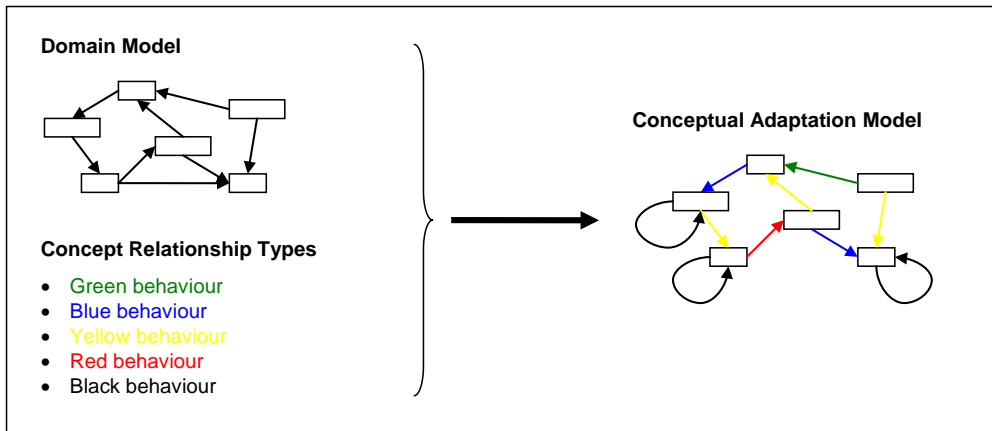


Figure 8: How Domain Model and Concept Relationship Types relate to the Conceptual Adaptation Model

Their (probable) infrequency of use is the reason that in Figure 9 the arrows of the CRT tool and UM tool are dotted: only advanced authors will actually use the CRT tool to define CRTs on their own. Mostly, they will use the GAT's predefined CRTs; the CRT tool is merely for defining CRTs.

Similarly, the UM tool will also be used in particular by advanced users. The User Model (UM) is for the major part implicitly defined by the CRTs that the author uses, and the Engine Independent Compiler can thus in principle derive all information for the UM specification from the DM and the set of CRTs that is used by the author. The UM tool is therefore only used for relating the User Model for the application to the User Model of the LMS, for example to use test results to indicate a user's knowledge of a concept. Indeed this is a functionality that is probably needed by quite advanced authors only.

When the author is finished modelling, the CAM contains all information that is needed for the next stage of the authoring process (see Figure 9), that is, the CAM literally includes the DM(s), the UM and the CRTs that are used in the CAM [D3.3]. It was a design decision to do this, but it would be perfectly arguable as well to have the UM and the CRTs and DM(s) that are used for the application as input for the engine independent compiler directly.

In the next stage of the authoring process, the authoring models are processed by the Engine Independent Compiler. If engine specific information is found during the compilation, this information is directly forwarded to the GALE Adaptation Engine. In principle, however, especially when the *GRAPPLE* Authoring Tools are used, no engine dependent information should be present in the authoring models.

The engine independent information, which is normally everything, is translated into the *Engine Independent Application Specification*, which consists of a GAL specification, a DM specification and a UM specification. The former describes the navigational structure of the application, that is, it specifies page elements like the title, images and links and the adaptation aspects like events that occur on access to the concept. The DM specification merely is an RDF translation of the original DM tool output file; this is done to enable the GAL specification to use the DM information. The UM specification is not just the RDF translation of the original UM tool output file. Rather it is augmented with constructions that are added by instantiating the CRTs. It thus not only stores the advanced information that was produced by the UM tool, but also the UM attributes and all rules that are created by the CRTs.

In the last stage of the authoring process, the GAL, DM and UM specifications are fed into an engine specific compiler, which is in this case the GALE AE compiler. This compiler translates the generic specifications into a format that can be processed by the adaptation engine, in this case a GDOM file and files that describe the structure of the concepts'. Since the final input for the adaptation engine was derived from a *generic* format, the resulting application might differ somewhat depending on the adaptation engine that was used. Note, however, that it is exactly because of this generic intermediate format that the authoring tools as well as the adaptation engine may be replaced by functionally equivalent others.

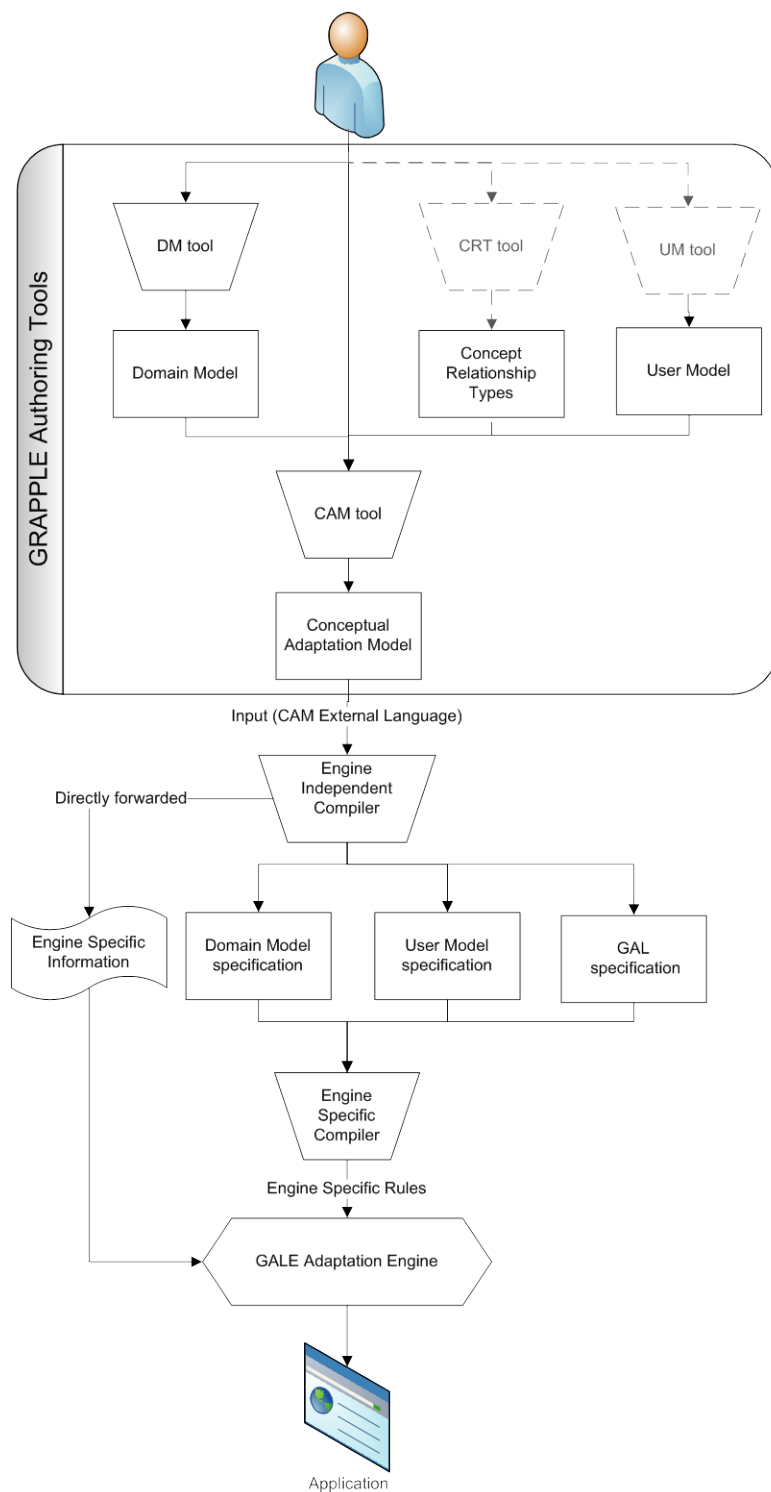


Figure 9: GRAPPLE Authoring Process

2.3 Running example

Throughout this thesis we will illustrate much of the work that was done using an example application about the Milky Way. It is a simple adaptive application that we think of as representative of what an average author, say, a high school teacher, should be able to produce with the final version of the GRAPPLE Authoring Tools.

The example was written in the output languages of the GRAPPLE Authoring Tool, its Engine Independent Application Specification was written in GAL, and it was written in GDOM, the most important input format for the GALE Adaptation Engine. Although all formats have their own optimal method of expressing this application, the goal throughout the project has been to express the application such that the formats could be translated into each other in a straightforward way by compilers.

To ensure that the reader gets a clear view of what we express in these various formats, we use this chapter to describe the Milky Way application in detail. First, we will describe the Domain Model, and then we describe the desired pedagogical aspects of the application. Finally we illustrate how the application should look when it is finally used by the user.

2.3.1 Domain Model of the Example Application

The domain of the Milky Way example application is mainly our solar system and some celestial objects. The concept 'Milkyway' is considered to be the starting concept of the application. After login, the user will by default arrive at the page of this concept, but on a user's very first visit, this concept's page shows an introductory text instead of information about the Milkyway.

The *concept hierarchy* of an application can be compared to the table of contents of a book; it is a logical outline of the content of the application. Note, however, that it is characteristic for many adaptive hypermedia documents that there is no linear order imposed on the content, that is, there is usually no need to read the content in the order as specified by the concept hierarchy.

The concept hierarchy of the Milky Way application is as follows:

- Milkyway
 - Star
 - Sun
 - Eta Carinae
 - Nebula
 - Planet
 - Mercury
 - Venus
 - Earth
 - Mars
 - Jupiter
 - Saturn
 - Uranus
 - Neptune
 - Moon
 - Moon (Earth)
 - Phobos
 - Deimos
 - Io
 - Europa
 - Ganymede
 - Callisto
 - Titan
 - Miranda
 - Ariel
 - Umbriel
 - Titania
 - Oberon
 - Triton

Note, that the concept hierarchy basically is a list of 5 general, or abstract, concepts (Milkyway, Star, Nebula, Planet and Moon) under which instantiations of those concepts are listed (e.g. Sun and Eta Carinae are instances of the concept Star).

Figure 10 illustrates the Domain Model of the Milky Way example application: the concepts are mostly related by concept relationships that indicate to which abstract concept an instantiated concept belongs, and by concept relationships that indicate which objects rotate around or belong to which other objects.

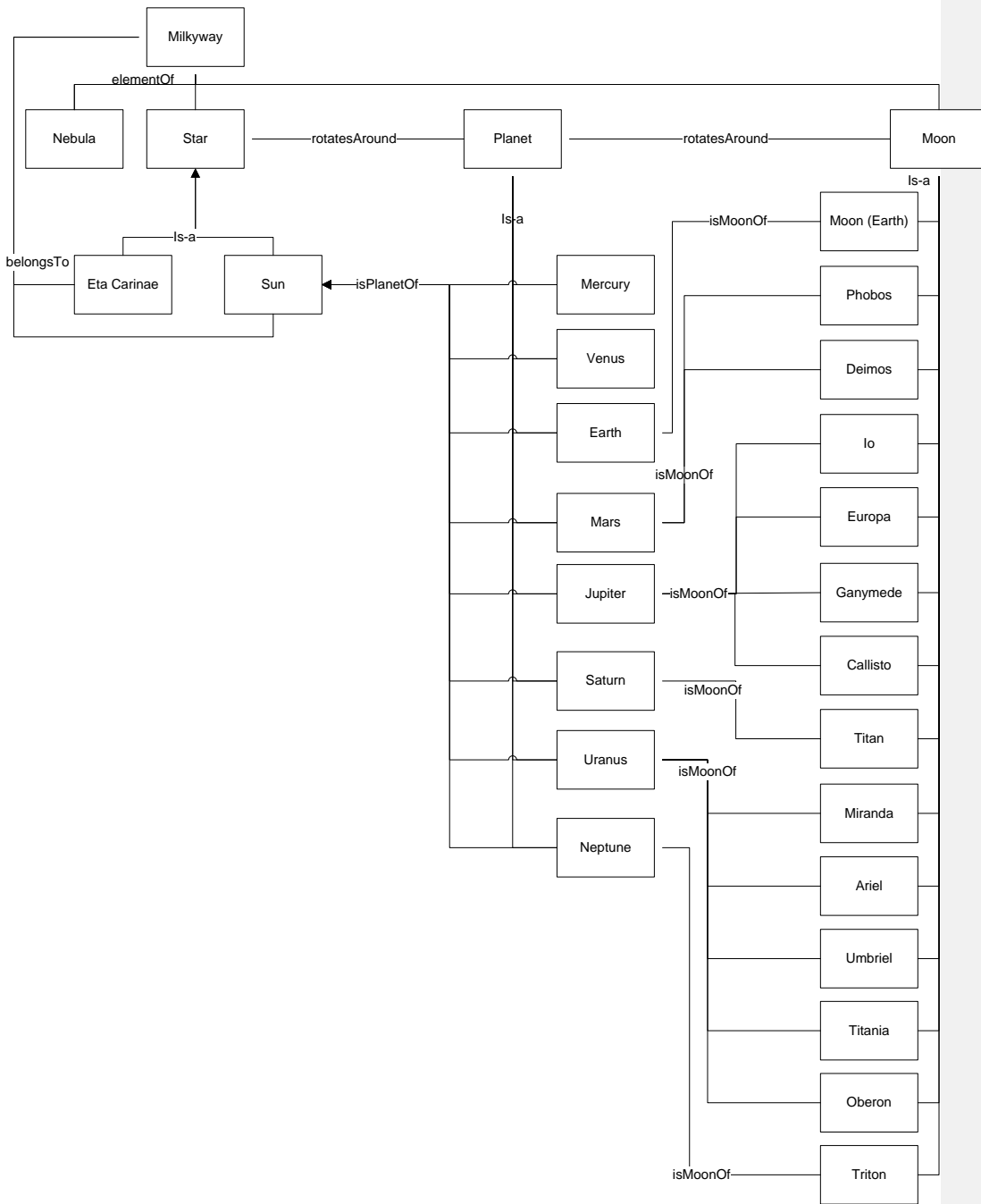


Figure 10: Domain Model of the Milky Way example application

2.3.2 Definition of the Behaviour of the Milky Way Example Application

In this section we briefly define behaviour of the Milky Way example application and the names of the CRTs that express this behaviour and which we will introduce in this thesis.

The behaviour of the Milky Way example application can be summarised as follows:

- The number of visits that a user pays to a concept is counted. The CRT that is used to express this behaviour is called *Visited*.
- When a user visits a concept the user's knowledge of the concept is increased, depending on whether the concept was recommended and on whether the user has already visited the concept before. We call this behaviour a *Knowledge Update*.
- Visiting a child concept of a parent relationship (see Figure 11) causes the parent of that concept to have an increase of knowledge as well. We call this behaviour *Knowledge Propagation*.
- Some concepts are prerequisites for other concepts, and they must therefore be visited before those other concepts are recommended. The CRT that is used to express this behaviour is called *Prerequisite*.
- Before a concept is recommended, its prerequisites must be sufficiently studied. The CRT that is used to express this behaviour is called *Concept Recommendation*.
- The application has a Guided Tour that along the concepts 'Star', 'Planet' and 'Moon' and then starts with the 'Sun' concept and continues its way outwards visiting all planets with their moons (e.g. Mercury, Venus, Earth, Moon (Earth), Mars, Phobos, etc.). The CRT that is used to express this mechanism is called *Guided Tour*.

In Figure 11 the 'Prerequisite' CRT is illustrated, as well as the two special CRTs: 'parent' and 'semantic parent'. The CRTs 'Visited', 'Knowledge Update', 'Knowledge Propagation' and 'Concept Recommendation' are not depicted in the figure, since they are all unary. Drawing them as self-loops would needlessly increase the complexity of the model, while it would not add any value. The 'Guided Tour' CRT was already explained in the above, and would also only jeopardise the readability.

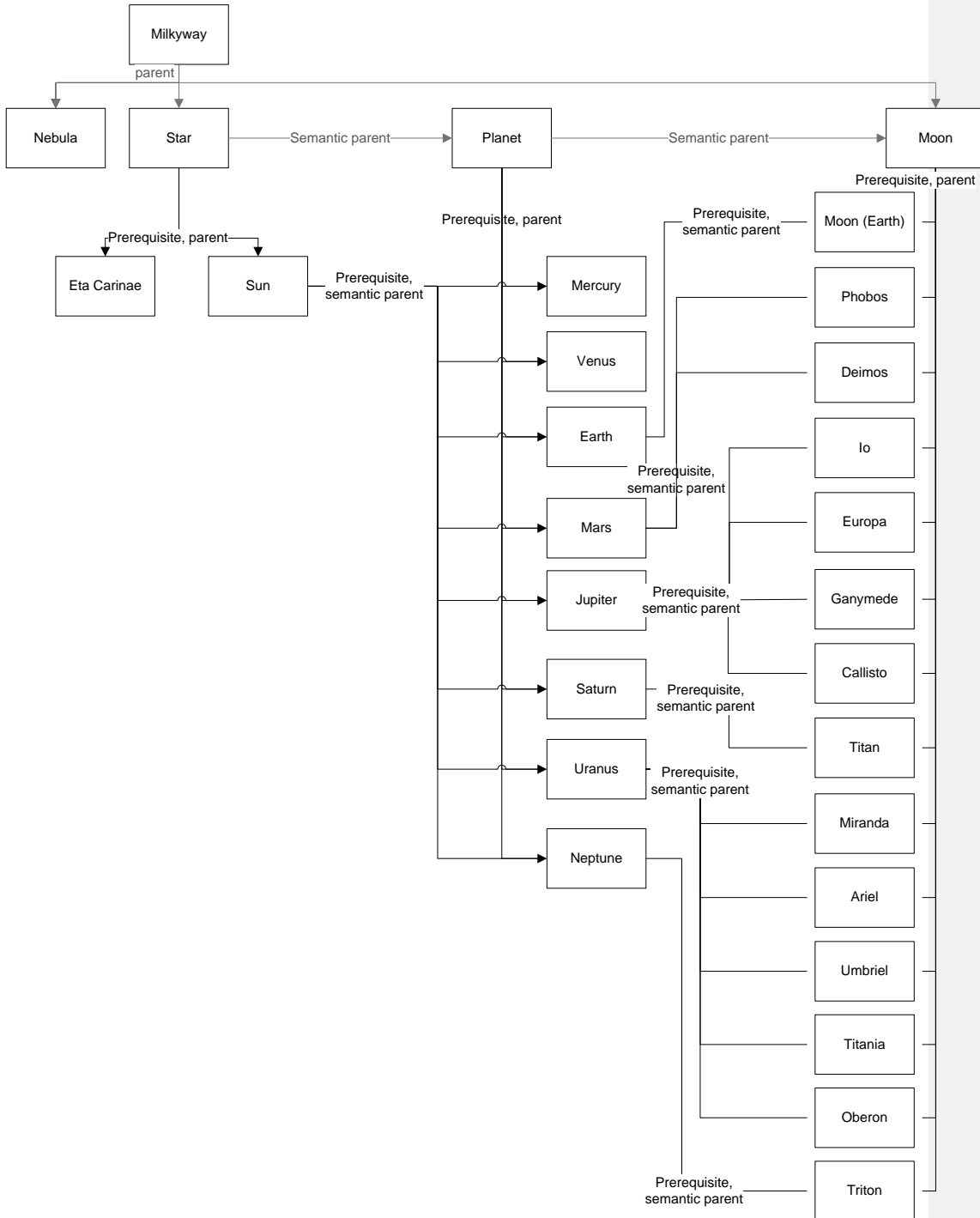


Figure 11: Partial specification of the Milky Way example application's behaviour

2.3.3 How the Milky Way example application should look like

In a browser the area of the window can be divided into several (rectangular) areas in a table-like way. Each area can have a 'view' assigned to it, and these views typically contain files that describe fragments of the page in application independent terms.

Figure 12 shows a screenshot of the Milky Way example application. The illustration can be divided into two parts: the visible part of the concept hierarchy, which is positioned on the left, and the part that represents the concept, which fills most of the screen. The left part is what we call the 'treeview', and the other part will be referred to as the '(concept) page', or 'content view'. Most of the time, however, we will simply refer to the 'page' of a concept, while we in fact mean to address the content view.

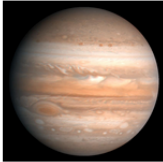
<p>Milkyway Star Nebula Planet Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune Moon</p>	<p>Anonymous user (no email) has read [onbekend] and still has [onbekend] to read - list of read pages - pages still to be read Changeable settings: link colors MilkywayViews - knowledge of MilkywayViews - password - logout</p> <hr/> <h2 style="text-align: center;">Jupiter</h2> <p>Is Planet of: Sun</p> <p>Image of: Jupiter</p>  <p>Information:</p> <p>Jupiter is the fifth planet from the Sun and the largest planet within the Solar System.[10] It is two and a half times as massive as all of the other planets in our Solar System combined. Jupiter is classified as a gas giant, along with Saturn, Uranus and Neptune. Together, these four planets are sometimes referred to as the Jovian planets.</p> <p>The following Moon(s) rotate around Jupiter:</p> <ul style="list-style-type: none">• Europa• Io• Ganymede• Callisto <hr/> <p>Start Guided Tour >></p> <p>The Guided Tour will first teach you about stars, planets and moons and then shows you the members of our solar sytem. It will be an exciting travel starting from the Sun all the way to Neptane and his moon Triton.</p> <p>Visited: 4</p>
--	--

Figure 12: Screenshot of the example application; concept 'Jupiter'

In the example application we have a (concept) page layout for both types of concepts (abstract and instantiated); all concepts of the same type thus follow the same page layout. The layout schemas are specified in Figure 13; and as can be seen, the rightmost schema indeed applies to the screenshot in Figure 12. The elements that comprise the page layout will be referred to as 'page (layout) elements'.

Header
Concept Title
Image
Information
...
Guided Tour Navigation
Visits:

Header
Concept Title
Reference to parent
Image
Information
...
Related concepts
• ...
• ...
Guided Tour Navigation
Visits:

Figure 13: Page structure of abstract concepts in Milky Way example application (left) and page structure of instantiated concepts (right); both page structures consist of 'page elements'

2.4 Thesis overview

Now that the authoring process is introduced, we can give an overview of the remainder of this thesis. In the following chapters, we will first introduce the Generic Adaptation Language and the Engine Independent Application Specification that is written in this language in Chapter 3. We start with this part because it is the core element of the GRAPPLE framework; it is the glue between the authoring tools and the adaptation engine. Then, in Chapter 4 we cover the authoring models in more detail, such that in Chapter 5 we can describe the translation from the authoring models to the Engine Independent Application Specification. Then in Chapter 6 we describe the GRAPPLE Adaptive Learning Environment (GALE), the environment in which the application will finally be run, and the format that is required by this environment. Chapter 7 describes the translation of the Engine Independent Application Specification into the GALE format (GDOM). Finally in the last chapter we present the conclusions and suggestions for future work.

3 Engine Independent Application Specification

The Engine Independent Compiler which is depicted in Figure 9 produces three specifications that together form the Engine Independent Specification of the adaptive Application: the DM specification, the UM specification and the GAL specification. These specifications are based on the authoring models that the author has modelled with the GAT. The specifications are written in two languages: Generic Adaptation Language and Generic Update Language.

Generic Adaptation Language is a RDF-based⁷ language that specifies the *navigational aspects* of an adaptive application. This means that a GAL specification specifies everything that has anything to do with the structure of the page of a concept and *direct concept access*. Basically, everything from the DM that is printed on page and all events generated by direct concept (page) access is expressed in the GAL specification.

There are, however, also parts of the application's adaptation behaviour, which are not directly triggered by the access to a concept. An update to a UM attribute, for example, may trigger another UM attribute update, but this latter update can not be expressed in GAL since it is not directly triggered by concept access. Also attribute values that only depend on the value of other attributes⁸ do not appear in the GAL specification. Since it is necessary to express these parts of the adaptation behaviour as well, we have to define UM Specification Rules. These Rules enable us to propagate attribute updates to other attributes as well; we could thus say that they enable us to *derive* information from events.

In this chapter we will first briefly introduce the reader to GAL and the UM Specification Rules, and then we will describe the files that together form the engine independent application specification, namely the DM specification, the UM specification and the GAL specification.

3.1 Generic Adaptation Language

The Generic Adaptation Language (GAL) is a language for the generic expression of the navigational structure of adaptive applications [D1.1]. The navigational structure of an application is comprised by both the structure of the page and the navigation between pages; presentational aspects like link colours and other issues that are usually handled through cascading style sheets are explicitly *not* part of an application's navigational structure.

A GAL specification of an adaptive application consists of a collection of *Units*. GAL Units are containers for elements that are semantically related to each other; a page that represents a concept is an example of what would typically be expressed by a Unit. Note that a concept may have several different representations, and as such there may be multiple Units for each concept. For now, however, we only consider Units that represent concepts their pages.

A GAL Unit is, like GAL attributes and GAL subunits, a GAL component (see Figure 14). This basically means that it has a name, and optionally a label, an order attribute and an emphasis attribute. The names of the Units in the Milky Way example application that represent concepts from the DM all start with the concept's name followed by the postfix “_Unit”; this is a trick that we used to ensure that Units are uniquely identifiable and that their name already specifies for which concept the Unit is used. Then, labels are strings that describe the element that they are associated with. The 'order' attribute is used to express a partial order on GAL components; in principle GAL components are unordered, except for the components for which an order attribute is specified. The 'emphasis' attribute is a categorical variable that indicates whether, and if so how much (low, medium or high), a component *has emphasis*; it is left to the Adaptation Engine to interpret what the consequences of the levels of 'emphasis' are. A Unit furthermore may contain zero or more input variables, GAL attributes, links, and subunits.

Input variables are mainly used to inform Units that are used by multiple concepts about which concept should be used as an input for the Unit, but since all Units are used by a single concept only in our Milky Way example application input variables are not relevant for this thesis.

GAL attributes are very important elements for Units: each GAL attribute represents something that must be displayed on screen. They should thus not be confused with resource attributes or UM attributes, although these will often be the objects that in fact are to be displayed (by these GAL attributes). An example of how a GAL attribute could look can be found in Code Fragment 1. The GAL attribute's name is the name that is used for reference to the attribute, the attribute's label is a string that normally describes what the attribute represents and the attribute's value is a literal value or a query that expresses the value of the attribute.

⁷ More information on RDF: <http://www.w3.org/TR/2002/WD-rdf-primer-20020319/>

⁸ In database terms we could call this a *view* on several UM attributes

```
gal:hasAttribute [
  gal:name attribute_name;
  gal:label attribute_label;
  gal:value attribute_value;
];
```

Code Fragment 1: 'gal:hasAttribute'-construction

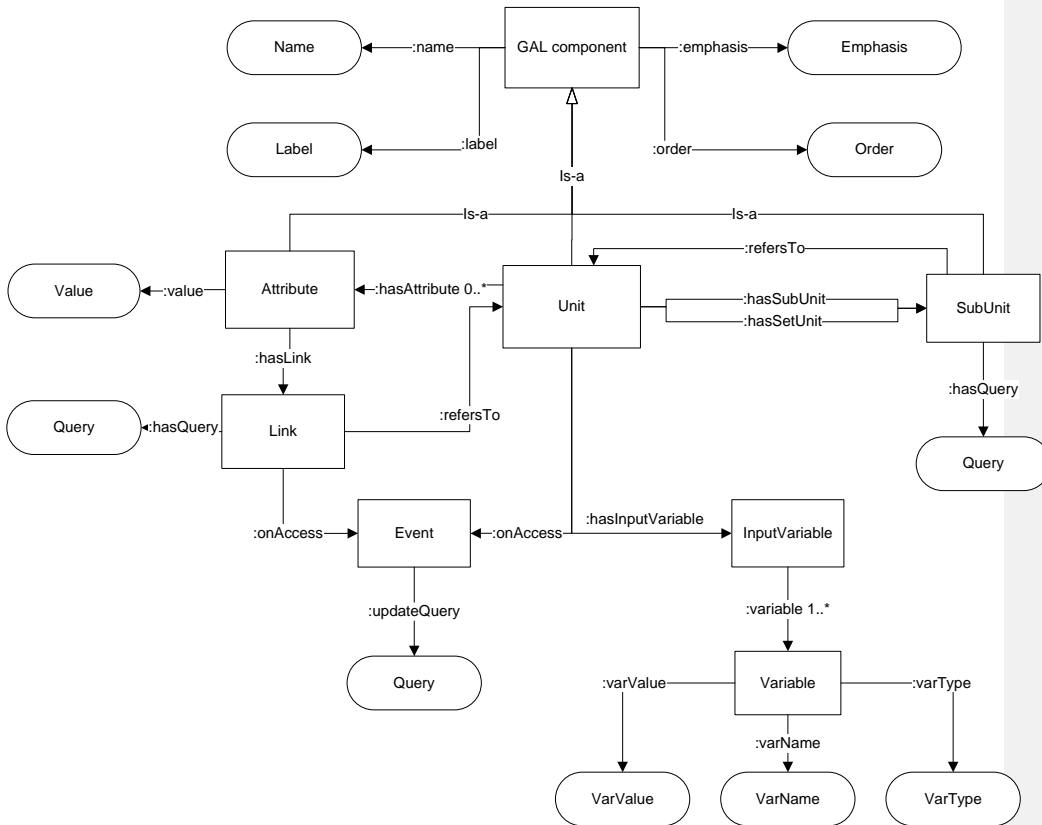


Figure 14: Meta-model of the Generic Adaptation Language

The GAL link construction allows for navigation to other Units, which is in the scope of this thesis equivalent to navigation to other concept pages. A GAL link is connected to a GAL attribute value which functions as the anchor of the hyperlink. The label is not part of the anchor. Code Fragment 2 is a 'gal:hasLink'-construction as it is mostly used in the Milky Way example application. Note, that the Unit name also defines the concept, so there is –in theory- no need to separately specify the target concept with the 'gal:hasQuery'-construct. For reasons of compilation, though, we maintain the query anyway. Links to Units that do not represent a concept, though, require an additional clause that specifies the link's target concept. Since the name of a Unit and the concept name do not coincide, the link's target is usually defined by a query of which the Where-clause specifies the target concept's title and of which the Select-clause appends the postfix "_Unit" to the target concept's name.

```
gal:hasLink [
  gal:refersTo Unit_name;
  gal:hasQuery Query_for_concept;
];
```

Code Fragment 2: 'gal:hasLink'-construction
 Authoring of Adaptation in the GRAPPLE Project

Another useful GAL component is the `setUnit`, a special case of which is the `subUnit`. The `setUnit` can be used to present a `subUnit` with all different bindings of its query at once. The structure of the `subUnit` is thus repeated for each possible binding. A `subUnit` without a `setUnit` can also have different bindings for a query, but only one binding at a time can be shown. Both `setUnits` and `subUnits` are created by adding a `gal:setUnit`-block or a `gal:subUnit`-block to a normal GAL Unit and defining the content as if it were a normal Unit. Furthermore the query that is used for binding must be added by a `gal:hasQuery`-statement.

The `gal:hasCondition`-construction is not depicted in Figure 14; the reason for this is that it is a quite simple construct but since it can be used in combination with GAL attributes as well as `subUnits` and queries it would needlessly complicate the figure if it were to be added. The construction is merely only an if-then-else statement for which the code of the then-block and the else-block is not executed but rather included in the specification.

Finally, GAL has a (restricted) notion of events. Events contain update queries which are executed whenever the element that they are attached to is accessed. Both links and Units can have associated events; the former's queries are executed when the link is followed and the latter's queries are executed when the Unit (i.e. the associated concept) is accessed. The update queries can only update attributes of the associated UM specification (see Figure 9).

3.1.1 GAL Query Language

In this section we briefly explain an important component of GAL: the GAL Query Language. It should be noted in advance, though, that there has not been made any decision on the definitive query language yet, so the currently used language is by no means final.

A GAL query consists of a `Select`-clause, followed by a `Where`-clause. If a query is used to fill out a value, it is surrounded by a `gal:hasQuery`-construction; if it is part of a `gal:hasCondition`-construction this is not necessary. An empty `Select`-clause in the if-clause of a `gal:hasCondition`-construction is interpreted as `false`. The `From`-clause is omitted, since queries always relate to the UM specification.

The query language works similar to RDF: queries are triples containing a subject, predicate and object, one of which is not bound. In GAL Query Language bound variables are prefixed with a dollar sign ('\$'), and unbound variables are prefixed with a question mark ('?'). The only bound variables that are normally used are the concept by which the Unit is currently used –which is bound to the input variable-, the current user and the placeholders for the DM relationships that fulfil a certain role –e.g. `$hasParent`-. The variable in the `Select`-clause is bounded to all possible values. As an example consider concept `$Concept` which has two related resources, namely Resource A and Resource B. Both resources are connected to the concept with the relationship `hasResource`. The query that selects both resources of `$Concept` is:

```
gal:hasQuery [
  Select ?resource
  Where $Concept :hasResource ?resource.
];
```

Code Fragment 3: A basic GAL query

Since queries can be more complex than only triples with an unbound variable, there are three simple rules that must be followed in GAL Query Language:

1. A selection criterion is ended with a dot ('.') or by closing the `gal:hasQuery`-construction.
2. Multiple selection criteria in the `Where`-clause are connected with logical 'and'.
3. Each line of a selection criterion has at least a predicate and an object.

The subject:

 - a. Is specified explicitly, like in Code Fragment 3, if it is the first line of a selection criterion
 - b. Is the subject of the previous line, if that line has a semicolon (;) at the end
 - c. Is the object of the previous line if that line has an open end (i.e. no dot or semicolon)

As an example of a complex query, consider the case in which the aforementioned resources have a label: Resource A has label 'beginner' and Resource B has label 'advanced'. We now query for a resource that is advanced:

```
gal:hasQuery [
  Select ?resource
  Where $Concept :hasResource ?resource.
         ?resource :hasLabel "advanced".
];
```

Code Fragment 4: A more complex GAL query

A shorter alternative for this query removes the dot at the end of the selection criterion's first line and the reference to "?resource" on the second line, thus obtaining:

```
gal:hasQuery [  
  Select ?resource  
  Where $Concept :hasResource ?resource  
        :hasLabel "advanced".  
];
```

Code Fragment 5: Alternative notation for Code Fragment 4

Finally, update queries work similar as normal queries, except that they are in a 'gal:hasUpdateQuery'-construction rather than a 'gal:hasQuery'-construction and the Select-clause is replaced by an Update-clause. The Update-clause specifies how the selected attribute should be updated by a statement of the form: *selected_attribute := new_value*;. Note that "new_value" may be a constant as well as (an expression using) the value of an attribute that is also selected in the Where-clause.

Consider the following example of an update query in Code Fragment 6: the query that assigns the *name* of the resource with the *label* "advanced" to the concept's variable "currentResource".

Note, that the second line of the first selection criterion has "?resource" as its subject, since the previous line has an open end, and that the third line of the same criterion also has "?resource" as its subject, because it uses the subject of the second line (denoted by the semicolon at the end of line 2) –which was "?resource".

```
gal:hasUpdateQuery [  
  Update ?currentResource := ?resourceName;  
  Where $Concept :hasResource ?resource  
        :hasLabel "advanced";  
        :hasName ?resourceName.  
        $Concept :hasVariable ?currentResource.  
];
```

Code Fragment 6: GAL (complex) update query

3.2 UM Specification Rules

The UM Specification Rules will have to take care of updates to the User Model which are not directly triggered by concept access. The rationale behind the introduction of UM Specification Rules has been the fact that it is not possible in GAL to consider updates to attributes as being events. The reason for this, is that in GAL updates can only originate from the current concept, and UM attribute updates that are triggered by other UM attributes rather than concept access often apply to other concepts than the currently visited one.

Since at the time of this writing the need for these Rules has only been acknowledged very recently, there are currently not many ideas yet regarding the particulars of the language in which these Rules are to be defined. So far, it has only been known that the UM Specification Rules will have to deal with UM attribute values that depend on the values of other UM attributes, and that they will have an 'If-Then'-like structure. The Rules will be triggered by the updates of the independent UM attributes⁹ and which specify the value of the dependent ones. For an attribute 'A' of which the value is equal to the sum of 'B' and 'C', Rules like the following should be defined:

- If (update B) then (A := B + C), and
- If (update C) then (A := B + C)

In this thesis, however, we will sometimes use a shorthand notation that directly expresses that value of 'A' as being the sum of 'B' and 'C', by simply writing: $A = B + C$. Note, that this expression easily translated into rules like the two in the above, by adding for each independent UM attribute a rule that states that if the independent UM attribute is updated the expression as in the shorthand notation with the '=' replaced by ':=' is evaluated. Indeed we will assume in the remainder of this thesis that only the expression in shorthand notation will suffice because the Engine Specific Compilers will know how to deal with this.

⁹ General expression form: $Dependent_Variable = f(Independent_Variable_{i=1...i=n})$, f being a function

3.3 DM Specification

The DM specification that is produced by the Engine Independent Compiler (see Figure 9), is the least complicated one of the three files that together form the application's specification. As already noted in 2.2, it is merely a translation from the DM tool's output file into RDF turtle format. The reason that this is done is the fact that the GAL specification is based on RDF, which therefore also expects to be querying RDF.

The RDF structure of the DM is an overlay of all concepts of the DM with the concept relationships incorporated within them. An example concept from the specification can be found in Code Fragment 7.

After the name of the concept, which identifies the start of a new element in the DM specification, it is declared that the element is a concept by the statement "rdf:type dm:concept;". This is done to ensure that all concepts that originate from the DM can always be distinguished from elements that may be added to the DM specification by CRTs later on.

The next element is the concept's title, which is marked by the predicate "dm:title".

Furthermore, the concept's resources are in a special block in which the name, URI and label of the resource are specified. A concept (factual) property, which is not depicted in Code Fragment 7, would be denoted by a similar construction as for resources, except for that the label would be omitted and the name of the construction would be "dm:fact".

Also, a description could optionally be added, which would be marked by the predicate "dm:description".

Finally, the concept relationships of which the concept is an input concept, are denoted with their name prefixed with "mw:" as predicates, 'mw:' being the shorthand for the application's namespace –which is in this case 'gale://grapple-project.org/milkyway/'.

```
gale://grapple-project.org/milkyway/etacarinae [
  rdf:type dm:concept;
  dm:title "Eta Carinae";
  dm:resource [
    dm:name "image";
    dm:value gale://grapple-project.org/milkyway/img/img_etacarinae.jpg;
    dm:label "Image of: ";
  ];
  dm:resource [
    dm:name "info";
    dm:value gale://grapple-project.org/milkyway/resource/etacarinae_text.jpg;
    dm:label "Information: ";
  ];
  mw:isa gale://grapple-project.org/milkyway/star;
  mw:belongsTo gale://grapple-project.org/milkyway/milkyway;
];
```

Code Fragment 7: DM information about 'Eta Carinae' represented in the DM specification

The DM specification of the example application has one additional entity¹⁰ besides the concepts from the original DM (see Code Fragment 8). This entity is part of the Guided Tour-construction and is added as a consequence from the author using the 'Guided Tour CRT'. The "Guided Tour" entity contains the concepts of the guided tour in the order that the author used to relate them; the 'position' predicate is used to express the latter.

```
gale://grapple-project.org/milkyway/guidedtour [
  tour:concept [
    tour:concept_name gale://grapple-project.org/milkyway/star;
    tour:position "1";
  ];
  tour:concept [
    tour:concept_name gale://grapple-project.org/milkyway/planet;
    tour:position "2";
  ];
];
```

¹⁰ The reason that we refer to 'entity' rather than 'concept' is that this unit of information is not part of the original DM. In fact it is part of the mechanism of the Guided Tour.

```

tour:concept [
  tour:concept_name gale://grapple-project.org/milkyway/moon;
  tour:position "3";
];

tour:concept [
  tour:concept_name gale://grapple-project.org/milkyway/sun;
  tour:position "4";
];

tour:concept [
  tour:concept_name gale://grapple-project.org/milkyway/mercury;
  tour:position "5";
];

...
];

```

Code Fragment 8: Guided Tour entity

3.4 UM Specification

The UM specification contains the user-specific data in *UM attributes* for all concepts next to the data that was added by the author through the UM tool. Note, that the UM attributes are added based on the information from the CRTs that are instantiated by the Engine Independent Compiler.

The UM specification for the Milky Way example application contains each concept of the original DM and an additional concept for the Guided Tour. All DM concepts possess four UM attributes: 'visited', 'knowledge', 'suitability' and 'changed'. Note, that it is because certain CRTs were applied to these concepts that they possess these attributes and not for any other reason.

Code Fragment 9 is the UM information that belongs to 'Eta Carinae', the same concept of which the DM information can be found in Code Fragment 7. Each of the four UM attributes is generated by a CRT that uses that attribute in its code. The "suitability" attribute, however, is an exception compared to the other automatically generated attributes: its value is not the default value that is specified in each of the CRTs that use this attribute, but rather a value that has been specified by the code-part of a certain CRT. The particulars of how the value could get there, will be explained in the next chapter. For now, it should merely be noted that the value is specified using UM Specification Rule Expressions.

For each of the four attributes apart from the name and value, four other properties are specified: the attribute's type, and properties that indicate whether the concept is *persistent*, *public* and *readOnly*. The type of the attribute is specified to ensure that the rules that are specified in the CRTs' code-parts always have attributes of the correct type at their disposal. The 'persistent' property indicates whether the value of this attribute is to be stored at runtime or whether its value is to be calculated when it is needed. The 'visited' attribute is a typical example of an attribute for which the value would be stored, since it is used to count the user's visits to the concept. The 'suitability' attribute is a typical example of an attribute for which the value should be calculated, since it is something of which the value depends on the situation: in some cases a concept might be suitable for the user and in others it might not be so.

```

gale://grapple-project.org/milkyway/etacarinae [

  um:attribute [
    um:name "visited";
    dt:type Integer;
    um:value "0";
    um:persistent "true";
    um:public "true";
    um:readOnly "false";
  ];

  um:attribute [
    um:name "knowledge";
    dt:type Real;
    um:value "0";
    um:persistent "true";
    um:public "true";
    um:readOnly "false";
  ];
];

```

```

um:attribute [
  um:name "suitability";
  dt:type Boolean;
  um:value "gale://grapple-project.org/milkyway/star#knowledge >= 50/(2+1)";
  um:persistent "false";
  um:public "false";
  um:readOnly "false";
];
];
um:attribute [
  um:name "changed";
  dt:type Real;
  um:value "0";
  um:persistent "true";
  um:public "true";
  um:readOnly "false";
];

```

Code Fragment 9: UM information about 'Eta Carinae' represented in the UM specification

Finally, there is also some user-specific data for the Guided Tour to be kept track of; therefore this entity is also represented in the UM specification. The associated code can be found in Code Fragment 10, and stores four attributes: *current_concept*, which keeps track of the current or lastly visited concept of the guided tour, *next_concept*, which holds the URI of the next concept to be visited, *active*, a Boolean that indicates whether the user is currently in the guided tour and *paused*, which indicates whether the guided tour is currently paused by the user or not. The former two of these attributes are used to keep track of the user's progress in the guided tour; the latter two attributes are used to keep track of the user's current state regarding the tour and are used to base the page's navigation upon.

```

gale://grapple-project.org/milkyway/guidedtour [
  rdf:type rdfs:class

  tour:attribute [
    tour:attribute_name "current_concept";
    dt:type Uri;
    tour:value "gale://grapple-project.org/milkyway/mercury";
    tour:persistent "true";
    tour:public "false";
    tour:readOnly "false";
  ];

  tour:attribute [
    tour:attribute_name "next_concept";
    dt:type Uri;
    tour:value "gale://grapple-project.org/milkyway/venus";
    tour:persistent "true";
    tour:public "false";
    tour:readOnly "false";
  ];

  tour:attribute [
    tour:attribute_name "active";
    dt:type Boolean;
    tour:value "false";
    tour:persistent "true";
    tour:public "false";
    tour:readOnly "false";
  ];

  tour:attribute [
    tour:attribute_name "paused";
    dt:type Boolean;
    tour:value "false";
    tour:persistent "true";
    tour:public "false";
  ];

```



```

        tour:readOnly "false";
    };
];

```

Code Fragment 10: Guided Tour entity for storage of user-specific information

3.4.1 UM Specification Rules for the Milky Way Application

In order to be able to express *knowledge propagation*, that is, the behaviour that causes knowledge to flow from concepts to their parents, some UM Specification Rules which can not easily be expressed in the shorthand notation must be added to the UM Specification. The semantics and syntax of these Rules has not yet been decided upon.

An example of how the Rule for knowledge propagation for Eta Carinae could look is specified in Code Fragment 11. It specifies that if the 'changed' attribute of the concept is updated (note the special pseudo code "IF_UPDATED" that is used for this), first the 'knowledge' attribute of the concept's parent is updated and then the 'changed' attribute of the parent is updated.

```

IF_UPDATED {Select ?changed
    Where   gale://grapple-project.org/milkyway/etacarinae um:attribute ?A
           um:name "changed"
           um:value ?changed}
THEN ( Update ?K := ?K + ?changed/(2+1)
    Where   gale://grapple-project.org/milkyway/star um:attribute ?A
           um:name "knowledge"
           um:value ?K
    AND
    Update ?parent_changed := ?changed/(2+1)
    Where   gale://grapple-project.org/milkyway/star um:attribute ?A
           um:name "changed"
           um:value ?parent_changed.
)

```

Code Fragment 11: Rule for Knowledge Propagation

3.5 GAL Specification

As well as for the UM and DM specifications, the original DM serves as a basis for the GAL specification as well. Each DM concept is used for the generation of two GAL Units: one Unit that represents the page of the concept and one that represents the concept in the treeview (see Section 2.3.3). For each of the three specifications it holds that their skeleton is based on the DM but the CRTs fill out the details.

A very important remark must be made about the description of the GAL specification that follows below: the exact contents of the Units in the specification must be specified somewhere, and currently it is not defined where or how. In particular: the Engine Independent Compiler can not on its own derive from the fact that there is X-resource in the DM specification or a Y-attribute in the UM specification that it should or should not be printed on the concept's page and that it therefore should or should not appear in the GAL specification. And it can also not know whether it must make only Units or also treeview Units. The exact content of the GAL specification must be specified, and currently we let this in the middle how this is done. One option is to derive the content from the page elements in case the automated authoring approach is used (note that this is not really feasible for the template based approach, since its possible contents are unknown), another option is to have some hard-coded default of what should be in the GAL specification in the Engine Independent Compiler, and probably many other options can be thought of. For now, we assume that the GAL specification's content specification is derived from the page elements according to the structures in Figure 13.

In Appendix A the two Units that belong to concept 'Sun', one of the concepts with the most complex page structure of the example application, can be found. The Unit starts with the name of the Unit and the declaration of the fact that it is indeed a Unit. As was explained in Section 3.1, it is in this case unnecessary to have an input variable since each Unit is only used by a single concept; therefore the concept name can be hard-coded in the Unit.

The first attribute of the Unit is the 'title' attribute; it simply fetches the concept's title from the DM specification.

The next four attributes together form a tagline that includes both the concept's parent and its semantic parent. For concept 'Sun' the predicate "mw:isa" has the parent role and the predicate "mw:belongsTo" has the semantic parent role; note that for another concept these roles may again be fulfilled by other DM

relationships. The complete tagline has the structure "*Is* <parent_concept> of: <semantic_parent_concept>", the concept references being hyperlinks; for 'Sun' this would result in "*Is* Star of: Milkyway". The first part of the tagline, which consists of the string "Is" followed by the link to the parent concept, is represented by the two attributes; the string "of: " followed by the link to the semantic parent concept is represented by the second pair of attributes. The reason that the tagline is divided over four attributes is first of all that in GAL only attribute values can be links, and therefore each of the two links needs its own attribute. Secondly, there is no way to know that the strings are related to the hyperlinks and therefore the strings can not be stored in the attributes that store the hyperlinks.

The anchor texts of the hyperlinks, the concept references, are stored in the 'gal:value' elements of the GAL attribute. The values are then associated with a 'gal:hasLink'-construction, such that they indeed become the anchors of the respective hyperlinks. Since the hyperlinks should be adapted such that the degree of recommendation of their target concepts can be indicated, the 'gal:hasLink'-construction is followed by a 'gal:hasCondition'-construction which expresses how much emphasis the (anchor of the) hyperlink should get depending on the suitability of the target concept and on whether it has already been visited before or not.

The next two attributes represent the image resource and the description resource of the concept respectively. In this case there is only one resource for both attributes, but if there would be more candidates the attribute would have specified the resource selection here.

Like the parent tagline, the related concepts tagline and the accompanying list of links to concepts can really only be derived from the template and not from the resources in the DM, which could have been the case with the title, image and description attribute. The structure of the related concepts tagline is similar to that of the parent tagline, except for its content: "*The following* <child_of_parent_concept>(s) rotate around: <concept_title>". The former place holder is in the GAL specification filled by a link to the child of the concept's parent, or to put it differently: a link to the parent of the concept's semantic child. The latter place holder is filled by the (current) concept's title. For 'Sun' the tagline would be: "*The following* Planet(s) rotate around: Sun". Again, the tagline must be split into four attributes.

The list of related concepts that follows below the related concepts tagline, is expressed by a 'gal:hasSetUnit'-construction. The construction contains a query that retrieves the concept's semantic children. Note, that for this operation we use the semantic parent relationship to the 'Sun' concept, and that we here thus use the DM relationship that has the semantic parent role for the 'Sun's' semantic children. Or, to put it differently: we fetch the concepts that have 'Sun' as a parent according to their semantic parent relationship.

The subUnit "body_Unit" to which the 'gal:hasSetUnit' refers, contains the content that must be displayed for each of the concepts that is fetched by the query. Since we want to print the titles of all concepts with a hyperlink to the concepts' pages themselves, the subUnit contains a 'gal:hasAttribute'-construction that selects the concepts' titles with an associated 'gal:hasLink'-construction and 'gal:hasCondition'-construction with the mechanism for assigning the correct emphasis to the hyperlinks' target concepts.

Then a subUnit that refers to the "Guided_Tour_Included_Unit" is included; the reason that the subUnit was included here is merely one of convenience for the reader. The "Guided_Tour_Included_Unit" as well as the "Guided_Tour_Excluded_Unit" are very lengthy, and it would probably disorient the reader if we had plainly included it in the code. Section 3.5.1 will elaborate on these Units.

The last attribute of the Unit displays the UM attribute "visited". The attribute has a label "Visited: " and its value is queried from the UM specification.

Finally, the 'gal:onAccess'-construction of the Unit expresses the updates that are directly triggered by the user's access to the concept. The first update is unconditional and consists of only an updateQuery that increases the "visited" attribute of the concept by one. The other update is indeed conditional and therefore consists of a 'gal:hasCondition'-construction with the update queries inside the then-clauses and else-clauses.

The treeview_Unit of the 'Sun' concept is identical to the treeview_Units of all other concepts. Treeview_Units are merely adapted links that offer access to the concept that they represent. These treeview_Units are components of the treeview of the entire application, and should thus be used by some other Unit. This Unit was, however, not specified since this is not a trivial task and it would go beyond the scope of this thesis to do so.

3.5.1 The Guided Tour Units

The Guided_Tour_Included_Unit and the Guided_Tour_Excluded_Unit are important parts of the Guided Tour mechanism. The former Unit is included in Units of concepts that are part of the guided tour; the latter is included in Units of concepts that are not part of the guided tour. Since the Units are quite similar, we will only explain the Guided_Tour_Included_Unit, which is the more complicated one of the two (see Appendix B). The Unit is not explained in the order in which it has been coded, but rather in the order in which end-users would normally use the hyperlinks that belong to this mechanism.

What the Guided_Tour_Included_Unit exactly prints on screen depends on different aspects. First and foremost it depends on whether the tour is 'active' or not, that is, whether the user is currently in the tour or

not. Then the 'paused' attribute is of influence, and finally there is a slight alteration if the currently visited concept happens to be the last concept of the tour.

If the tour is not active and not paused, the concept shows a hyperlink with the anchor text 'Start Guided Tour >>'. The target of this hyperlink is looked up using the "guided tour" concept in the DM specification: by simply referring to the name of the tour concept that has position "1", it is ensured that the hyperlink navigates to the first concept of the tour.

When the user is at the first (or any other) concept of the tour, three hyperlinks are available: "Next >>", "Pause" and "Exit". If "Next >>" is clicked, the application navigates to the concept that is specified in the UM attribute 'next_concept'. Furthermore, the onAccess-code related to the hyperlink is executed: the UM attribute "current_concept" is updated to name of the target concept, and the UM attribute "next_concept" is updated to the name of the concept that follows the target concept in the guided tour. Both queries first retrieve the position of the current concept, then fetch the name of the concept that is one position along and finally assign the new name as the new value to the attribute of interest. Since the "Next >>"-hyperlink is only shown if there is indeed a next concept in the guided tour, there is no danger of accidentally referring to a non-existent concept. When the last concept of the guided tour is the currently visited one, the label of the hyperlink is "Restart the Tour >>", and following it results navigation to the first concept of the guided tour and in the UM attributes "current_concept" and "next_concept" to be resetted to the names of the concepts with positions "1" and "2" respectively.

When the user clicks "Pause", the Boolean UM attribute "paused" is set to "true" and the Boolean UM attribute "active" is set to "false" by an 'gal:onAccess'-construction with two update queries. The user remains on the same page, and the hyperlinks are replaced by a hyperlink with the anchor text "Resume the Tour where you left off". The UM attributes "active" and "paused" are set to "true" and "false" by an update query respectively. When the user clicks "Exit" the same update queries as for "Restart the Tour" and "Pause" are all executed. The hyperlink navigates to the page of the current concept.

Note, that the code in Appendix B does not specify link recommendation for the Guided Tour's hyperlinks. The reason that this was not done is that it would needlessly lengthen the code while in fact it would only add the same 'gal:hasCondition'-construction to each 'gal:hasLink'-construction which has been described in the previous section already.

Finally, the Guided_Tour_Excluded_Unit, is similar to the Guided_Tour_Included_Unit, except for the fact that the "Next >>" hyperlink is replaced by a hyperlink with the anchor text "Back to Tour". Indeed, following the link does not result in any UM attribute updates.

4 GRAPPLE Authoring Models

In this chapter we will more closely examine the Domain Model, the CRTs and the Conceptual Adaptation Model and their formats. At the start of my graduation project these models were specified only very marginal; many of the descriptions presented below were completely implicit or still undiscussed then. Describing their characteristics and their relation to the authoring tools has therefore been a non-trivial matter.

Finally, note that the UM, which is used to model the more advanced aspects of the User Model, is ignored in this chapter. The reason for this is that it would add much complexity to the subject matter, while the functionality that it supports is for advanced authors only.

4.1 Domain Models in GRAPPLE

Traditionally, there has never been a very clear distinction between Domain Models as being the set of concepts that is used for the adaptive application, and Domain Models as being application independent descriptions of certain domains or topics. And another practice that has often been seen is that the adaptation aspects of an application are defined with the same tool and stored within the same file as the application's DM. As a result of these issues, DMs have often become an inseparable part of the application to the extent that the DM would be useless without the rest of the application. In both cases the core problem is that application dependent information and –supposedly- application independent information are intermingled. The authoring tool of the AHS AHA!, the Graph Author, which loads 'application files' in which the Domain Models as well as the adaptation aspects are defined is a good example of this custom.

In GRAPPLE the intermingling of the application's adaptation aspects and the application's DM is put to an end, by the introduction of the CAM tool. By having DM's imported into a separate tool for defining the adaptation aspects of the application, it is ensured by construction that there can get no adaptation information whatsoever into the DM. There is thus a strict decoupling of the semantic part and the adaptive part of an application: the Domain Model can only cover *content*, and the Conceptual Adaptation Model specifies adaptation aspects of that application.

GRAPPLE also provides the means to more clearly separate the application independent part of the DM from the application specific part, but this is more subtle. In the remainder of this section we therefore first explain something about different types of DMs and then elaborate on how the DM tool allows for the protection of application independent information.

4.1.1 Types of Domain Models

To fully appreciate the need to distinguish the application dependent part from the application independent part of the DM, it is important to be aware of the different types of DM that exist. There are basically three types of DMs (although this distinction is not made in the DM tool) which can be distinguished based on their *function*:

- **Pure Domain Models.** Pure DMs are domain models in the most literal meaning of the word. They are abstract representation of a certain topic or domain, and are entirely application independent. Concepts in pure DMs may have associated resources that describe the concept.
- **Application Domain Models.** An Application DM is the set of concepts with associated concept relationships that is used as the source of an application. It will often consist of one or more (subsets of) Pure DMs, some application specific concepts, like concept that aggregate information of the application, and possibly also some concepts of Pseudo DMs (see next bullet point).
- **Pseudo Domain Models.** Pseudo DMs are neither Pure DMs nor Application DMs but rather sets of concepts that have some or another general function. So far the Layout Domain Model is the only example of this DM type that we have found, and therefore we will rather refer to 'the Layout DM'. The Layout DM contains layout concepts that can be used to have application DM concepts inherit their page layout from.

The Application DM serves as the blueprint DM for the adaptive application; only Application DMs are (supposed to be) used in the CAM to specify adaptation behaviour for. It is, of course, possible to have a Pure DM coinciding with an Application DM.

The GRAPPLE DM tool does not make a clear distinction between the three DM types. There is no particular reason for this approach, other than that it has been an implicit design decision. It has traditionally been the case that Pure DMs coincide with Application DMs, and therefore no need for this distinction has been perceived during the definition of the DM tool. Layout issues have traditionally not been handled through DMs, but the DM tool turned out to have functionalities that allowed for a quite simple method for layout definition. Given the status quo, all references to Domain Models in this thesis are about Application Domain Models, unless indicated otherwise.

4.1.2 Decoupling Application Dependent and Application Independent information

The DM tool allows authors to use any type of DM as the base of their application. Authors could thus in theory edit Pure DMs and the Layout DM for personal use. Indeed, this is undesirable since this poses the risk that DMs of these types come to contain application dependent information. The DM tool, however, has three functionalities that together can prevent the mixture of application dependent and application independent information: authoring rights, the notion of shareability and the details of the working principle of the tool's DM import mechanism. The author's awareness for the need of this separation and his/her cooperation are required though, to indeed ensure the enactment of it.

The first aid in the decoupling of application dependent and application independent information is the DM tool's notion of authoring rights. The creator of a DM can thus indicate who is allowed to edit his DM by assigning authoring rights. This ensures that the creator of the DM remains in control over the DM's content or at least over who can edit the DM as well. Especially for Pure DMs it is desirable to have only a restricted group of experts allowed to change anything in the DM.

The second functionality of the DM tool that can help to avoid application dependent information to be leaked away to other applications, is the notion of sharing. An author can indicate whether a DM that is created by him/her is available to others or not. If all authors would only share DMs that contain no application dependent information, that is, only Pure DMs and Pseudo DMs, then one could safely use any public DM. It is the responsibility of the authors themselves, though, to take care of this as well as of the assignment of authoring rights, and perhaps a more explicit notion of the different DM types would be of aid to make them more aware of the need to indeed act upon this.

Finally the last, and most important, functionality of the DM tool is that if the author imports (a part of) a DM, the source DM is *copied* into the author's new (Application) DM. This results in the creation of *new* concepts which happen to exactly resemble the concepts of the source DM, rather than that the concepts of the source DM indeed become part of the author's new DM. The difference is that in the former case the concepts have another URI than the concepts of the source DM, whereas in the latter case the concepts are in fact the same and thus share the same URI as well. Note, that this mechanism enables authors that do not have the authoring rights for a (shared) DM to use it as a part of their own Application DM after all, while it is still ensured that the right to alter the source DM is not violated.

4.1.3 The Layout Domain Model

In the DM tool there will be one special DM: the Layout Domain Model. This is a pseudo-DM that consists entirely of layout concepts. These concepts are not used to describe some sort of domain, but merely specify layouts.

A layout concept can specify a page layout in one of two ways, depending on the authoring approach that is used: it can have a *template resource*, which is a file written by the author that specifies how a page should look. These concepts are called 'template layout concepts'. Another option is that the concept has an associated (ordered) *set of page elements*; these concepts are called 'predefined layout concepts'. The template layout concepts are used in the 'template based authoring approach', and the predefined layout concepts are used in the 'automated authoring approach'; these are two of the three page structure authoring approaches that are described in Section 4.4.

The template layout concepts must be created by the author with the DM tool. The construction of a template layout concept is similar to the construction of a normal DM concept, the only difference is that it can have only one property, namely the template. As the name already suggests, the predefined layout concepts can not be authored.

Note that the Layout DM in its original form will only consist of different predefined layout concepts and one empty template concept. Furthermore, it will be shared and no end-user authors will have authoring rights for it. If an author thus wants to use layout concepts for his/her own (Application) DM, (s)he has to import the desired concepts from the Layout DM. Note that it may be useful for authors to create their own, private Layout DM in which they can store template layout concepts that they intend to reuse.

4.1.4 DM Format

The output format of the DM tool is IMS VDEX, a grammar for controlled vocabularies. Code Fragment 12 illustrates the code of a typical concept from the Milky Way example application. Each concept corresponds to a term, whose identifier is the concept's URI (which is simultaneously the concept's name). The concept's title is specified in a `<langstring>`-element within the `<caption>`-element; this enables the existence of multiple language-specific titles for the same concept. The resources that are associated with the concept are specified by `<mediaDescriptor>`-elements. Each `mediaDescriptor` has one `<mediaLocator>`-element which contains the resource's URI, and one language independent `<interpretationNote>`-element which contains a metadata identifier. The metadata identifier refers to a `<metadata-item>` that contains more

information about the resource, which is at least a name that can be used by the compiler to identify the nature of the resource (e.g. text or an image) and possibly a label.

Note, that the syntax that will be used for the metadata is not decided upon yet; probably it will either be Learning Object Metadata (LOM) or Dublin Core (DC).

```
<term>
  <termIdentifier>
    gale://grapple-project.org/Milkyway/EtaCarinae
  </termIdentifier>
  <caption>
    <langstring language="en"> EtaCarinae </langstring>
  </caption>
  <mediaDescriptor>
    <mediaLocator> gale://Milkyway/resource/etacarinae_text.xhtml
  </mediaLocator>
  <interpretationNote>
    <langstring language="x-none"> metadata-itemID-1 </langstring>
  </interpretationNote>
  <mediaDescriptor>
    <mediaLocator>gale://Milkyway/img/img_etacarinae.jpg </mediaLocator>
    <interpretationNote>
      <langstring language="x-none"> metadata-itemID-2 </langstring>
    </interpretationNote>
  </mediaDescriptor>

  <metadata>
    <metadata-item ID="metadata-itemID-1">
      <name> text </name>
      <label>
        <langstring language="en"> Information </langstring>
      </label>
    </metadata-item>
    <metadata-item ID="metadata-itemID-2">
      <name> image </name>
      <label>
        <langstring language="en"> Image of: </langstring>
      </label>
    </metadata-item>
  </metadata>
</term>
```

Code Fragment 12: Concept representation in Domain Model specification

Besides terms, the specification of the DM also contains `<relationship>`-elements which denote the concept relationships of the DM. These elements contain a source term, a target term and a relationship type which denotes the *name* of the concept relationship. Note that one should not confuse this element with a possible Concept Relationship Type that could be associated with a DM relationship. No associations with CRTs whatsoever are denoted in the DM.

```
<relationship>
  <relationshipType> rotatesAround </relationshipType>
  <sourceTerm>gale://grapple-project.org/Milkyway/Phobos</sourceTerm>
  <targetTerm>gale://grapple-project.org/Milkyway/Mars</targetTerm>
</relation>
```

Code Fragment 13: Relationship

4.2 Concept Relationship Types in GRAPPLE

Since the adaptive behaviour of an application is often driven by learning objectives, CRTs are typically of a pedagogical nature. They are in theory independent from the semantic relationships that occur in the Domain Model, although it often happens to be the case that semantic and pedagogical relationships coincide somehow. Since there is only a limited set of pedagogical structures, typically, a relatively small set of CRTs that can fulfil the needs of most authors. These CRTs are predefined in the GAT and therefore most authors will never need to define their own ones. As such, they will not need to use the CRT tool, but rather use the CAM tool to apply CRTs to various concepts.

There will, however, also be authors that need CRTs that do not exist yet; they will have to define those CRTs themselves. Since the CRTs are written in GAL, understanding of both the language and the underlying framework are necessary. An important issue to take into account for them is that there are several classes of CRTs whose differences relate to the nature of the adaptations they define; therefore authors that define their own CRTs must be aware of this classification.

4.2.1 Classes of CRTs

Although all CRTs seem the same on the outside to the authors, under the hood there are four major different classes of CRTs, one of which is a special CRT. Each CRT class has different characteristics and has its own consequences for the UM and GAL specifications. In this section we will first describe the four classes, and then we will elaborate on the consequences of each class on the Engine Independent Application Specification.

Figure 15 depicts Peter Brusilovsky's "classic loop of user modelling and adaptation" as he describes it in [Bru1996]. The figure basically illustrates how the AHS receives information about the user (by the user's actions), this information is then mapped onto a UM and based on this, the AHS produces an adaptation which results in an adaptation effect. Three of the four CRT classes are classifiable by their (technological) function, based on a refinement of this figure for GRAPPLE, which can be found in Figure 16.

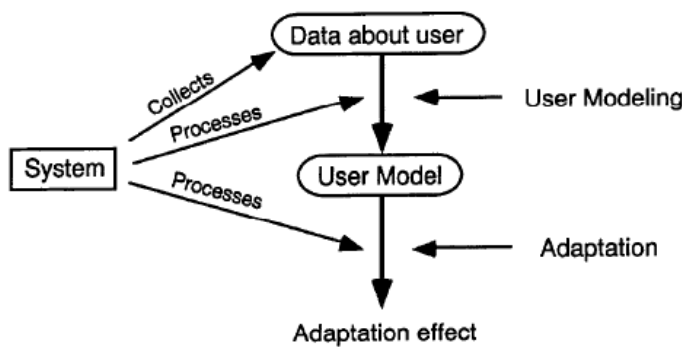


Figure 15: "Classic user modelling – adaptation loop in adaptive systems" by Peter Brusilovsky [Bru1996]

In adaptive applications in GRAPPLE, at least within the scope of this thesis, the only user action that is registered is concept access; therefore Figure 16 states that the "System Registers Concept Access", rather than that the "System Collects Data about User". Furthermore, we state that "Concept Access triggers an Explicit Update to the UM" and that (the state change of) the UM can then (again) trigger an "Implicit Update to the UM". This is more specific than stating that by user modelling the data about the user is processed into a UM, as is done in Figure 15. Finally, a dotted line that indicates that the Adaptation Effect –which is based on the UM- is delayed until all UM updates are finished is added.

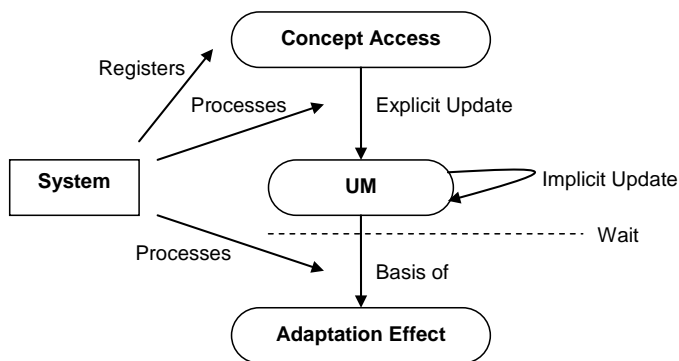


Figure 16: Refined version of Brusilovsky's figure for GRAPPLE

Starting from Figure 16, the first CRT class is the *Explicit Update CRT*, or 'Update CRTs' for short. Update CRTs specify behaviour that is explicitly triggered, that is, behaviour that is related to concept access. These CRTs merely update UM attributes, and do not have any directly visible consequences for the concept's page.

Since concept access is relevant for navigation, the attribute update query that is expressed by this CRT class is represented in the GAL specification. Furthermore, if the UM attributes involved in CRTs of this class do not exist yet at the time of compiling the CRT, the required attributes are automatically generated in the UM for the concepts that are involved with them. The details of these attributes are specified by the CRT.

The second CRT class is the *Implicit Update CRT*. These CRTs specify behaviour that is triggered implicitly, that is, they express UM attribute values in terms of other UM attributes. In fact, this CRT class has two subclasses: Invariants and Recursive CRTs. Invariant CRTs can only express UM attribute values in terms of other UM attributes; only updates to these attributes result in updates to the dependent attribute. Recursive CRTs express the UM attribute value in terms of the UM attribute itself and possibly other UM attributes; explicit updates to the attribute itself as well as updates to the other attributes that it depends on change the value of the attribute. Note, that the Recursive CRT is a sort of mixture between an Explicit Update CRT and an Invariant that apply to the same UM attribute.

Implicit Update CRTs are not reflected in an application's GAL specification. The reason for this is first of all that this class of CRT does not express a directly visible aspect of the application, like Adaptation CRTs do. Second of all, Implicit Update CRTs are not directly related to concept access, which could be the only other reason for a CRT to be represented in a GAL specification. Rather than in the GAL specification, these CRTs are expressed in the UM specification by a UM Specification Rule Expression. Again, the UM attributes involved in CRTs of this class are automatically generated in the UM for the concepts that are involved with them if they do not yet exist at compile time. The default details of these attributes are again specified by the CRT, but CRTs of this class subsequently modify the value of the (automatically generated) dependent UM attribute.

The third CRT class that can be related to Figure 16 is the *Adaptation CRT*. CRTs of this class define how the concept's page will look based on the state of the UM. CRTs do thus not update the UM, but rather only use UM attributes. Often, CRTs of this class use one or more UM attributes for some condition, and based on that condition the adaptation is determined.

Since these CRTs obviously influence the application's navigation, the adaptation constructions specified are represented in the GAL specification. Again, the UM attributes involved in CRTs of this class are automatically generated in the UM for the concepts that are involved with them if they do not yet exist at compile time. Note, however, that if no other CRT ever updates the UM attributes that are involved in an Adaptation CRT, the CRT will have no effect whatsoever.

The fourth CRT class is the *Auxiliary CRT*. CRTs of this class do not specify any behaviour, but rather define an abstract structure that is used by other CRTs. In fact it they express a 'role' that concepts that participate in the CRT fulfil with respect to each other, such that other CRTs can simply refer to that role without having to know whether a, and if so which, concept relationship connects the concepts. As an example consider the situation in which the concept relationship 'X' functions as a parent relationship for half of the concepts of a DM and concept relationship 'Y' functions as a parent relationship for the other half. By specifying this in the Auxiliary CRT 'Parent-relationship', the placeholder '\$hasParent' –which belongs to this CRT- is replaced by the compiler by the correct concept relationship ('X' or 'Y') on all occurrences.

Initially, the UM specification was used by Auxiliary CRTs to add properties that expressed these structures to each concept. Concepts would thus for example have a 'parent' property that would express what the parent for that concept is, and in the GAL specification the value of this property was used for references to the parent. A disadvantage of this, however, is that the UM specification is for *user* related information only, and this approach violated that.

Finally, it should be noted that it is possible to define complex CRTs that are in fact a mixture of CRTs of different classes. These CRTs, for example, both update some UM attributes and adapt the navigation based on the updated attributes. An example of a complex CRT is the 'Guided Tour' CRT, which will be described in depth in the next section.

4.2.1.1 Classification based on Pedagogical Aspects of the Application

The classification above is based on the CRT's function in the adaptation process. There is, however, another way to classify CRTs as well, namely by their function with respect to the pedagogical strategies of the application. For authors, which will often be teachers, a classification based on this will be more natural than a classification based on technological aspects of the CRTs. Therefore we briefly consider how CRTs would be classified if we merely consider their *role in the application*.

The two most important CRT classes from a pedagogical point of view would be the *Pedagogic Structure CRTs* and the *Pedagogic Presentation CRTs*. CRTs of the former class define the pedagogic structure of the application, for example by expressing the concept hierarchy (using the 'Parent' CRT that was described above) and by stating which concepts must be studied before others. Pedagogic Structure CRTs describe abstract aspects of the application, and we could metaphorically say that if the application's DM is a forest, then these CRTs define the roads along which the end-user must find his or her way.

Continuing the metaphor, Pedagogic Presentation CRTs could be said to describe the signage of the forest (not just the road signs). CRTs of this class both make the pedagogic structure visible for the end-user and

they may also deal with other (presentational) issues like the presentation of UM dependent content, e.g. a concept that shows a welcome page on the first visit.

Finally, there are the *User Modelling CRTs* which are CRTs that describe process that relate to the end-user. Some good examples of this CRT class are CRTs that model how the user gains (and possibly also loses) knowledge about concepts. Other CRTs might relate to the user's interest or identify problems in the user's understanding of the learning material. The attributes that are updated by User Modelling CRTs are often used by Pedagogic Structure CRTs and by Pedagogic Presentation CRTs; the simple reason for this is that CRTs of those classes are based on the UM.

Note, that the User Modelling CRTs coincide with the Update CRTs from the technological classification. The reason for this is that all Update CRTs express updates to the UM that are triggered by the user himself and thus by definition model the user. Note also, however, that not all CRTs that express a UM attribute update are User Modelling CRTs, since not all UM attributes describe the user himself. The Invariant CRTs do thus not automatically belong to the User Modelling CRTs; in fact the only example Invariant CRT that was implemented in the Milky Way example application is not a User Modelling CRT in this classification (but rather a Pedagogical Structure CRT).

4.2.2 Predefined CRTs

Most authors of an adaptive course will need only a relatively small number of CRTs, mainly because there is a limited number of pedagogical strategies that can be embedded in a course. Courses with adaptive learning material taught on the Eindhoven University of Technology, like the course on Hypermedia and the course on Adaptive Hypermedia, have already provided us with an indication of the CRTs that might be useful. These CRTs have then been applied to the Milky Way example application to explore their requirements within the GAT.

In this section, we will first describe two Explicit Update CRTs, which are generally used as a means to model the user (see Section 4.2.1.1). Then we describe two Implicit Update CRTs, followed by an Adaptation CRT and finally the three most important Auxiliary CRTs and a complex CRT.

Since it holds for all CRTs that if the CRT uses a UM attribute that does not exist yet, this attribute is automatically generated, this is not explicitly mentioned in the CRTs described below. Note again, that the details that are needed for the generation of these attributes are present in the CRTs themselves.

4.2.2.1 Explicit Update CRTs

The mostly used Explicit Update CRTs are the *Visited CRT*, which counts the user's visits to a concept, and the *Knowledge Update CRT*, which updates the user's knowledge on accessing a concept. The UM attributes that are updated by these CRTs are 'visited' and 'knowledge'.

The *Visited CRT* has a set of concepts which is traversed by a for-loop as its input. For each of its elements an update query that increases the UM attribute 'visited' by one, is added to the 'gal:onAccess'-construction of the associated GAL Unit (see Code Fragment 14).

```
gal:onAccess [
  gal:updateQuery [ Update ?V := ?V + 1
                    Where ?c um:attribute ?A
                          um:name "visited";
                          um:value ?V
                    ];
];
```

Code Fragment 14: GAL code of the Visited CRT

The *Knowledge Update CRT* updates the UM attributes 'knowledge' and 'changed', based on whether the concept is 'suitable' for the user or not. Code Fragment 15 is the code that is added to the GAL Units of the concepts that are in the input set of this CRT. The CRT thus assumes the presence of a Boolean attribute called "suitability" as well as the attributes that it updates, "knowledge" and "changed".

Before we turn to the description of this CRT it is important to know that the knowledge of a concept consists of the knowledge of the concept itself and the knowledge of the concept's descendants. All descendants contribute a share to the concept's knowledge, which is proportional to their distance to the concept and their own number of descendants (see Figure 17). If a concept has N children, accessing the concept if it is suitable, increases the knowledge by $100/(N+1)$; if it is not suitable the increase is only $35/(N+1)$. Together with the knowledge of the concept's children the knowledge can add up to 100.

The proportions 35 and 100 are chosen for historical reasons, and there may very well be pedagogical reasons to choose other proportions, as long as the knowledge that is assigned to the concept when it is visited when it is not suitable (in this case 35) is smaller than the threshold value that is used for the Prerequisite

CRT (in this case 50). The reason for this is that the transitivity of the Prerequisite CRT is then preserved; otherwise visiting a concept that is prerequisite for the next concept while it is not suitable would result in making the next concept suitable (while the prerequisite has still not been studied enough). Within the scope of this thesis, however, it merely matters that the knowledge of a concept depends from the knowledge of its children (according to the concept hierarchy) and of the suitability of the concept for the user.

The CRT thus increases the knowledge of the concept either to $35/(N+1)$, or to $100/(N+1)$, 'N' being the number of children of the concept. This increase, however, is preceded by an update of the concept's 'changed' attribute. The reason for this is that the value of the 'knowledge' attribute before its update is needed to calculate the change. It is also to ensure that this order is maintained that the GAL code is extended with 'gal:order'-constructions.

```
gal:onAccess [
  gal:hasCondition [
    gal:if [ Select ?S
      Where ?c um:attribute ?A
        um:name "suitability";
        um:value ?S
    ];
    gal:then [
      gal:updateQuery [ Update ?changed := 100/(?Nr+1) - ?K
        Where ?c um:attribute ?A
          um:name "knowledge";
          um:value ?K.
          ?c um:attribute ?A'
          um:name "changed";
          um:value ?changed.
        ];
      gal:order 1;
      gal:updateQuery [ Update ?K := 100/(?Nr+1)
        Where ?c um:attribute ?A
          um:name "knowledge";
          um:value ?K
        ];
      gal:order 2;
    ];
    gal:else [
      gal:updateQuery [ Update ?changed := 35/(?Nr+1) - ?K
        Where ?c um:attribute ?A
          um:name "knowledge";
          um:value ?K.
          ?c um:attribute ?A'
          um:name "changed";
          um:value ?changed.
        ];
      gal:order 1;
      gal:updateQuery [ Update ?K := 35/(?Nr+1)
        Where ?c um:attribute ?A
          um:name "knowledge";
          um:value ?K
        ];
      gal:order 2;
    ];
  ];
];
```

Code Fragment 15: GAL code of the Knowledge Update CRT

4.2.2.2 Implicit Update CRTs

The *Prerequisite CRT*, which specifies which concepts are to be studied in advance of other concepts, and the *Knowledge Propagation CRT*, which specifies how knowledge of one concept propagates to other concepts, is an example of an Invariant CRT and a Recursive CRT respectively.

The *Prerequisite CRT* expresses the suitability of concepts in terms of the required knowledge of the prerequisite concepts. This is done directly in the UM specification using the Generic Update Language, of which the definitive form is not known yet. For now we use the lazy (shorthand) notation, since it allows the attribute value to be denoted at the attribute itself, which improves the readability of the UM specification. The

exact expression of behaviour that the Prerequisite CRT comprises is:

Prerequisite (PrerequisiteConcepts, DependentConcepts):

$(\forall i \in \text{DependentConcepts} : i.\text{suitability} = (\forall j \in \text{PrerequisiteConcepts} : j.\text{knowledge} \geq 50 / (\text{Children}(j) + 1)), \text{Children}(j))$
 being the number of children of element 'j'.

The number of children of the prerequisite concepts is taken into account because the knowledge update of concepts depends on the number of children as well. If a concept is a prerequisite for another concept, it only matters if that prerequisite concept was visited and whether it was suitable at the time or not; and because a concept with N children can only maximally account for 1/(N+1)th of its knowledge, the knowledge threshold that is used for the Prerequisite CRT is divided by (N+1).

The code that belongs to the Prerequisite CRT can be found in Code Fragment 16. The outer for-loop traverses through the set of concepts that have the same set of prerequisite concepts; the value of the 'suitability' attribute of the current concept is bound to '?S'. The inner for-loop combines the old value of the 'suitability' attribute with the terms that relate to the prerequisite concepts. First the value of '?S', which is at compile time a string that represents a Boolean value, is assigned to a temporary variable called '?todo'. Then for each prerequisite concept '?in' the number of children of '?in' is counted, and the value of '?todo' is conjuncted with the statement that '?in.knowledge' is greater then or equal to 50 divided by the number of children of '?in'. This is repeated for all prerequisite concepts, and finally '?S' is replaced by '?todo'.

```
FOR ALL ( Select ?S
  Where ?out IN $DEPENDENT_CONCEPTS.
        ?Concept.name == ?out.
        ?out um:attribute ?A
              um:name "suitability"
              um:value ?S.
        )
{
String ?todo := ?S;
FOR ALL ( Select ?Concept'
  Where ?in IN $PREREQUISITE_CONCEPTS.
        ?Concept'.name ==?in
        )
{
  int ?Nr = [ Select Count(?child)
            Where ?child $hasParent ?in. ];
  ?todo = "?in.knowledge >= 50\(?Nr+1)" && ?todo;
}
<REPLACE>
  ?todo
</REPLACE>
}
```

Code Fragment 16: Code that produces the value of the 'suitability' attribute of concepts that have prerequisite concepts

The *Knowledge Propagation CRT* is a supplementary CRT for the *Knowledge Update CRT*: it specifies the knowledge propagation from children to their parent, thus increasing the latter's knowledge (see Figure 17).

Recall that the propagation of knowledge is needed to complete knowledge of the parent concept.

Note, that the maximal share that a child concept can add to its parent, depends on the number of children that the child concept has (again, see Figure 17). So, if, concept A has 3 children of which one child has a child of its own, then two children add 100/4 to A's knowledge, A itself contributes 100/4 to its own knowledge, the third child adds 100/4/2 to A's knowledge, and A's only grandchild also adds 100/4/2 to A's knowledge.

The structure of the code of the Knowledge Propagation CRT is printed in Code Fragment 17. The for-loop selects a concept from the set of concepts that participate in the CRT, and counts the number of children that its parent has. Then a UM Specification Rule is added to the list of Rules that belongs to the UM Specification (marked by the tag <ADD_RULE>). Since there has not been decided on the details of the UM Specification Rule language yet, it is still unclear to which part of the specification and in which form the Rules are exactly added. The most important property of the Rule, should be that it is triggered by any update to the attribute that is mentioned in the 'IF_UPDATED' clause, in this case the 'changed' attribute. The Rule then specifies how the 'changed' and 'knowledge' attributes of the concept's parent is updated. Note, that the update to the 'changed' attribute of the concept's parent triggers the Knowledge Propagation Rule of the parent, thus ensuring the knowledge propagation along the entire path to the root of the application. Finally, the concept's 'changed' attribute, which was set by the Knowledge Update CRT, is resetted to zero.

```

<UM>
FOR ALL ( ?c IN $SET ) {
  int ?Nr = [ Select Count(?child)
             Where ?c $hasParent ?p.
             ?child $hasParent ?p. ];
  <ADD_RULE>
  IF_UPDATED {Select ?K
              Where ?c um:attribute ?A
                  um:name "knowledge"
                  um:value ?K
              }
  THEN ( Update ?parent_changed := ?changed/(?Nr+1)
        Where ?p um:attribute ?A
            um:name "changed"
            um:value ?parent_changed.
        ?c um:attribute ?A
            um:name "changed"
            um:value ?changed.

        AND
        Update ?K' := ?K' + ?changed/(?Nr+1)
        Where ?p um:attribute ?A
            um:name "knowledge"
            um:value ?K'.
        ?c um:attribute ?A
            um:name "changed"
            um:value ?changed.

        AND
        Update ?changed := 0.
        Where ?c um:attribute ?A
            um:name "changed"
            um:value ?changed.
    )
  </ADD_RULE>
}
</UM>

```

Code Fragment 17: GAL code of the Knowledge Propagation CRT

An alternative to this approach in which knowledge propagation is triggered by updates to the 'knowledge' attribute, would be to calculate all knowledge updates that follow from access to a given concept in advance, and add all updates to that concept's onAccess GAL code. This would 'flatten' the chain of knowledge updates that this CRT can generate and it would turn this CRT into an Explicit Update CRT. It would be unnatural, though, to do this, since it would remove the natural transitivity of the CRT and would instead emulate that the access to the concept directly triggered these updates. Furthermore, it would require to explicitly define a partial order on the update rules, since for the mechanism to work properly, first the knowledge update that really results from the concept access must be performed and only then the knowledge updates which result from that.

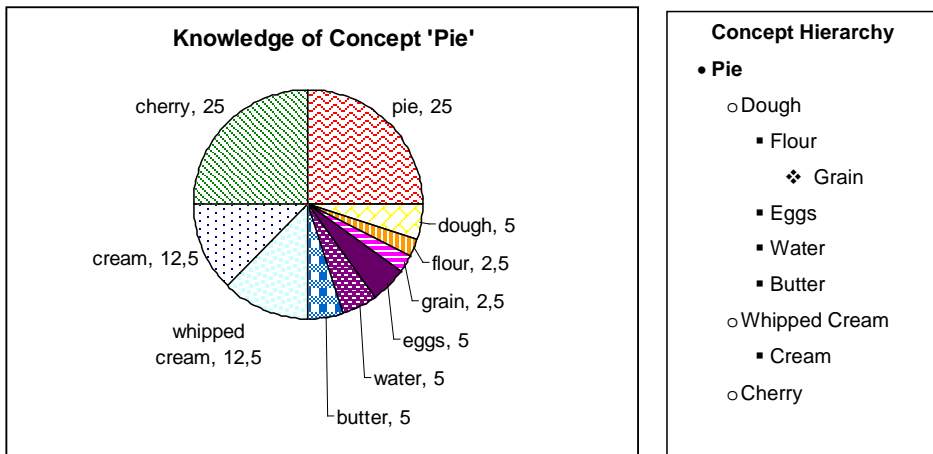


Figure 17: Share of the descendants of concept 'Pie' in its knowledge

4.2.2.3 Adaptation CRTs

The most important Adaptation CRT for adaptive applications is the CRT that specifies concept recommendation. Since GAL distinguishes three degrees of emphasis, the *Concept Recommendation CRT* boils down to expressing which condition results in which degree of emphasis. The GAL code that is added to all attributes with a link that has a concept from the input set of this CRT as its target can be found in Code Fragment 18.

```
gal:hasCondition [
  gal:if [Select?S
    Where ?q um:attribute ?A
          um:name "suitability";
          um:value ?S
  ];
  gal:then [
    gal:if [Select?V
      Where ?q um:attribute ?A
            um:name "visited";
            um:value ?V
      Filter ?V > 1
    ];
    gal:then [ gal:hasEmphasis "medium" ];
    gal:else [ gal:hasEmphasis "high" ];
  ];
  gal:else [ gal:hasEmphasis "low" ];
];
```

Code Fragment 18: GAL code that is added to all attributes with a link that has the concept under consideration as its target.

4.2.2.4 Auxiliary CRTs

Auxiliary CRTs do not specify behaviour, and therefore have no associated with CRT code. Rather, the Auxiliary CRTs specify abstract structures on the DM, sometimes by assigning 'roles' to existing DM relationships and sometimes by defining the structure without relating it to the DM. In the former case, the CRTs specify which DM relationships must replace the placeholders that are used by other CRTs. In the latter case, probably some auxiliary concept relationship must be created which can then indeed replace the placeholders, as if it were a genuine DM relationship.

The most important Auxiliary CRT is the *Parent CRT*. This CRT is used to define the application's concept hierarchy, and knowledge propagates along the structure that this CRT defines.

The Parent CRT may coincide with one or more DM relationships; the 'is-a' relationship is an example of a concept relationship that would typically be assigned the 'parent' role. Because this CRT assigns a role to DM relationships, rather than that it expresses the influence of (attributes of) concepts on other concepts, this CRT is not instantiated by dragging concepts into sockets. Rather in the CAM the author can indicate which DM

relationship(s) implement the Parent CRT, but how this will exactly be done is still unclear at the time of this writing.

The *Extends CRT* is used to specify in the CAM from which layout concepts the other concepts inherit their layout. Since there are no semantic relationships between these different types of concepts, this CRT can not be instantiated by a DM relationship. Therefore this CRT must indeed be instantiated by dragging a layout concept to one socket, and the set of concepts that inherit its layout to another socket. Again, how the CRT will effect the Engine Independent Output specifications is still unknown at the time of this writing.

The *Semantic Parent CRT* is similar to the the Parent CRT in that it coincides with one or more DM relationships. As a result this CRT is also to be implemented with one or more DM relationships. The Semantic Parent CRT indicates an alternative parent relationship that is based on a more applied level than the Parent CRT. Whereas the Parent CRT is purely hierarchical, the Semantic Parent CRT is more practice based; a good example of DM relationships that could be used to instantiate the Semantic Parent CRT are the 'rotatesAround', 'isElementOf', 'isPlanetOf', 'isMoonOf' and 'belongsTo' relationships in the Milky Way DM, or the 'eats' relationship in the (hypothetical) Food Chain domain.

4.2.2.5 Complex CRTs

Complex CRTs combine characteristics of several CRT classes into one CRT. A good example of this CRT class is the *Guided Tour CRT*, which combines the Explicit Update CRT class with the Adaptation CRT class. Features of both CRT classes are visible in the code of the Guided Tour CRT, however, it can not be stated that these two CRT classes entirely make up the characteristics of this CRT. The pseudo-code for this CRT can be found in Code Fragment 19; since there are still too many things undecided, there is no point in attempting to give explicit code. The pseudo-code describes how first a 'guidedtour' entity, which is filled with the concepts that together form the guided tour, is added to the DM specification. Then it describes that to each GAL Unit that represents the page of a concept from the DM, a reference to either the 'Guided_Tour_Included_Unit' or the 'Guided_Tour_Excluded_Unit' is added, depending on whether the concept occurs in the list with tour concepts ('\$TOUR-LIST'). Finally, the Guided_Tour_Included_Unit and the Guided_Tour_Excluded_Unit (see Section 3.5.1) are added to the GAL specification.

```
<ADD to=DM specification>
  gale://grapple-project.org/milkyway/guidedtour [
  ];
</ADD>

FOR ALL (?Concept IN $TOUR-LIST) {
  <ADD to=<DM specification>/gale://grapple-project.org/milkyway/guidedtour>
    tour:concept [
      tour:concept_name ?Concept;
      tour:position ?Concept.index();
    ];
  </ADD>
}

FOR ALL (?Concept IN DOMAIN_MODEL) {
  IF[ ?Concept' IN $TOUR-LIST ];
  THEN [
    <ADD to=<GAL specification>/?Concept'_Unit>
      gal:hasSubUnit [
        gal:name "Guided_Tour_Included_Unit";
      ];
    </ADD>
  ];
  ELSE [
    <ADD to=<GAL specification>/?Concept'_Unit>
      gal:hasSubUnit [
        gal:name "Guided_Tour_Excluded_Unit";
      ];
    </ADD>
  ];
}

<ADD to=<GAL specification>>
  <!-- Guided_Tour_Included_Unit -->
  <!-- Guided_Tour_Excluded_Unit -->
```

</ADD>

Code Fragment 19: Pseudo-code for the Guided Tour CRT

The clear influence of the Explicit Update CRT class can be found in the presence of the 'gal:onAccess'-construction in the GAL code of the guided tour (see Appendix B), which updates the UM attributes 'current_concept', 'next_concept', 'paused' and 'active'. This 'gal:onAccess'-construction is, however associated with a *specific hyperlink* to a concept, rather than with an entire Unit (which is the custom for Explicit Update CRTs). Then, the rest of the GAL code for this CRT, see Section 3.5.1., can be accounted for by the influences of the Adaptation CRT class.

There are, however, some parts of the Guided Tour CRT that can not be explained by the either of the aforementioned classes. The most important issue is the addition of the 'guidedtour' entity and its tour-concept properties (see Code Fragment 8) to the DM specification. There are no CRT classes that deal with the addition of information to the DM specification, since adaptation behaviour normally leaves the application's DM untouched. The lack of a model that describes the application structure, however, necessitates us to add the tour entity to the DM specification anyway, because the UM specification is even less suitable for it (only structures for information that differs among users should be added to the UM specification).

Furthermore, it should be noted that for the Explicit Update part of the CRT, it is not only expected that the required attributes are automatically generated, but that also for the UM specification a guided tour entity that holds these attributes must be created. Since the guided tour entity is hard-coded in the CRT code, it is probably possible to indeed implement this.

4.2.3 CRT Format

The format of the CRTs has come a long way since the start of this thesis. The initial format of a CRT as specified in the first deliverable of the CRT tool [D3.2] can be found in Code Fragment 20. The only additional information provided at the time was that entities could either be concepts, services or resources and that the cardinality could be 'binary', 'n-ary', 'group', 'binary-group' or 'n-ary-group'.

During the process of defining the GAL code that was to be inserted in the <specification name="GAL">-element, it quickly became clear that the original format was not sufficient at all. The main problems of the format were first of all that it lacked any reference to UM attributes and that the entities involved in the CRTs are not only concepts (and services and resources, but they are not relevant in the Milky Way example application), but also sets and lists of concepts.

```
<CRT UUID="02934809238409384">
  <NAME>prerequisite</NAME>
  <DESCRIPTION>some description about this CRT</DESCRIPTION>
  <SHAPE>line</SHAPE>
  <COLOUR>blue</COLOUR>
  <ENTITY TYPE="ANCHOR">CONCEPT</ENTITY>
  <ENTITY TYPE="TARGET">RESOURCE</ENTITY>
  <CARDINALITY>N-ARY</CARDINALITY>
  <specification name="GAL">
    ... pieces of GAL code ...
  </specification>
</CRT>
```

Code Fragment 20: The specification of the CRT format as provided in [D3.2]

Based on the deficiencies of the original format, a new format was proposed, of which the XML Schema can be found in Code Fragment 21. The fragments that are most important within the scope of this thesis, are explained below.

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema elementFormDefault="qualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="crt">
  <xs:complexType>
    <xs:all>
      <xs:element name="uuid" type="String"/>
      <xs:element name="author" type="String"/>
      <xs:element name="name" type="String"/>
      <xs:element name="comment" type="String"/>
      <xs:element name="pedagogicalDescription" type="String"/>
      <xs:element name="majorVersion" type="Integer"/>
      <xs:element name="minorVersion" type="Integer"/>
      <xs:element name="dateTimeCreated" type="Date"/>
      <xs:element name="dateTimeEdited" type="Date"/>
    </xs:all>
  </xs:complexType>
</xs:element>
```

```

        <xs:element name = "representation" type="representationType"/>
        <xs:element name = "constraints" type="constraintsType"/>
        <xs:element name = "adaptationBehaviour" type="adaptationType"/>
        <xs:element name = "crtSockets" type="crtSocketType"/>
    </xs:all>
</xs:complexType>
</xs:element>

<xs:complexType name="representationType">
    <xs:element name="shape" type="String"/>
    <xs:element name="colour" type="String"/>
</xs:complexType>

<xs:complexType name="constraintsType">
    <xs:element name="allowedInLoop" type="Boolean"/>
    <xs:element name="attributeConstraints" type="attr_constrType"/>
    <xs:element name="umVariableConstraints" type="umv_constrType"/>
</xs:complexType>

<xs:complexType name="adaptationType">
    <xs:element name="userModel" type="UMType"/>
    <xs:element name="code" type="String"/>
</xs:complexType>

<xs:complexType name="crtSocketType">
    <xs:element name="socket" type="socketType"/>
</xs:complexType>

<xs:complexType name="attr_constrType">
    <xs:element name="socketID" type="String"/>
    <xs:element name="attributeName" type="String"/>
    <xs:element name="requiredValue" type="String"/>
</xs:complexType>

<xs:complexType name="umv_constrType">
    <xs:element name="umVariableName" type="String"/>
</xs:complexType>

<xs:complexType name="UMType">
    <xs:element name="dmID" type="String"/>
    <xs:element name="socketID" type="String"/>
    <xs:element name="umVarName" type="String"/>
    <xs:element name="public" type="Boolean"/>
    <xs:element name="readOnly" type="Boolean"/>
    <xs:element name="location" type="locationType"/>
    <xs:element name="type" type="String"/>
    <xs:element name="range" type="rangeType"/>
    <xs:element name="default" type="type"/>
</xs:complexType>

<xs:complexType name="socketType">
    <xs:attribute name="type" type="String" use="required"/>
    <xs:element name="uuid" type="String"/>
    <xs:element name="colour" type="String"/>
    <xs:element name="minCardinality" type="Integer"/>
    <xs:element name="maxCardinality" type="Integer"/>
    <xs:element name="orderRelevant" type="Boolean" minOccurs="0"/>
    <xs:element name="name" type="nameType"/>
</xs:complexType>

<xs:complexType name="locationType">
    <xs:element name="web" type="URL"/>
    <xs:element name="remoteCourse" type="remoteCourseType"/>
</xs:complexType>

<xs:complexType name="rangeType">
    <xs:element name="from" type="Real"/>
    <xs:element name="to" type="Real"/>
    <xs:element name="elem" type="String"/>
</xs:complexType>

<xs:complexType name="name">
    <xs:attribute name="language" type="String" use="optional"/>
</xs:complexType>

```



```

<xs:complexType name="remoteCourseType">
  <xs:element name="remoteCourseName" type="String"/>
  <xs:element name="resource" type="resourceType"/>
</xs:complexType>

<xs:complexType name="resourceType">
  <xs:element name="resourceUniqueIs" type="String"/>
  <xs:element name="resourceName" type="String"/>
</xs:complexType>

</xs:schema>

```

Code Fragment 21: XML Schema of the CRT format

The 'constraints' element of the CRT specifies the requirements that the use of the CRT imposes on its context. These requirements relate to required attribute values of the concepts involved and the absence of UM attributes of concepts in the neighbourhood of the concepts that are involved in the CRT. In the Milky Way example application, no examples of such constraints have been found. Finally this element contains an element that indicates whether or not the CRT can be allowed in a loop with itself.

The 'adaptationBehaviour' element of the CRT specifies the actual behaviour that the CRT expresses. It contains two main elements: a 'userModel' element in which the required 'UMvariables' (red. UM attributes) are described, and a 'code' element in which the code that is used by the Engine Independent Compiler is described. The former element contains the aforementioned details of the UM attributes that the Engine Independent Compiler needs to create the required attributes in the UM specification. The code that is specified in the latter element, is the code that the Engine Independent Compiler will execute after the code's instantiation with concepts based on the CAM. The code is a combination of pseudo-code that tells the compiler what to do, and GAL code or code that creates UM Specification Rule Expressions. The content of the 'code' element thus describes which is the code is literally added to the DM, UM and GAL specifications. It must be stressed that the language that is used for this code is really merely a pseudo language and that its definitive format is yet to be determined. Only the fragments of GAL code are in the correct grammar (i.e. the grammar for the UM Specification Rule Expressions is not definitive either).

Finally, the 'crtSockets' element contains 'socket' elements, which specify the details of the sockets that are referred to in the code of the CRT. The most important aspect of these 'socket' elements is their type. So far, there are 3 socket types to be distinguished: 'source', 'target' and the rest. Between sockets of the type 'source' and 'target', a notion of direction is assumed, i.e. connectors between source and target concepts are arrows in the CAM tool. Any other socket types can be used as well, but these types will not have this notion of direction. The new format includes no information regarding the details of 'other socket types'.

The 'minCardinality' and 'maxCardinality' elements indicate the minimum and maximum number of concepts that should be bound to this socket respectively.

The optional 'orderRelevant' element indicates whether the concepts that are dragged into the socket in the CAM tool are added to a set or a list. The Guided Tour CRT is an example CRT for which the order in which concepts are added to the socket is indeed relevant.

Lastly, the name of the socket is the name that is used for reference to the socket in the code of the CRT and the 'uuid' element represents the identifier with which the CAM will refer to the CRT.

4.3 Conceptual Adaptation Model

The Conceptual Adaptation Model (CAM) specifies the adaptation of the application. This is done by the instantiation of CRTs with concepts and concept relationships from the DM. Only the Auxilliary CRTs 'Parent' and 'Semantic Parent' can be instantiated with concept relationships, all other CRTs –at least in the Milky Way example application- are instantiated by dragging concepts into sockets. Since it is unknown so far how the former will be accomplished, in this section we only describe how CRTs are instantiated with concepts.

First of all, recall that the CAM output file in fact *includes* the DM, UM and CRTs, when we write 'CAM', however, we only refer to the CAM-specific part. The CAM consists of 'relationship' elements only, of which the XML Schema is described in Code Fragment 22.

```

<?xml version="1.0" encoding="utf-8" ?>
<xs:schema elementFormDefault="qualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="relationship">
  <xs:complexType>
    <xs:all>
      <xs:element name="uuid" type="String" minOccurs=1 maxOccurs=1/>
      <xs:element name="assign" minOccurs=1>
        <xs:complexType>
          <xs:element name="entityId" type="String" minOccurs=1/>
          <xs:element name="dmID" type="String" minOccurs=1/>

```

```

        </xs:complexType>
      </xs:element>
    </xs:all>
  </xs:complexType>
</xs:element>
</xs:schema>

```

Code Fragment 22: XML Schema of the (only) CAM element 'relationship'

The 'uuid' element refers to the identifier of one of the CRTs; it indicates which CRT is instantiated by this element. The 'assign' elements specify which sockets of the CRT are instantiated with which DM concepts. A 'relationship' element must thus contain exactly as many 'assign' elements as the CRT has sockets (which is at least one). The 'entityId' specifies the identity of the socket that is to be binded; the 'dmID' elements are the concept identities with which the socket is bound. The number of 'dmID' elements should be within the cardinality constraints that are specified by the socket.

4.4 Three Approaches to Defining the Page Structure

Besides modelling the domain and the behaviour of the application, another important task of the author is to define the layout or structure of the application's pages. Examples of components that can typically be found on a page are a title, textual content, images and a header and footer. Usually, these components will be ordered on a page, and it is this exact ordering of components that we refer to with 'page structure'.

So far two approaches have been used for the determination of the page structure in adaptive systems [Bra2009]: manual authoring (which was done in AHA!), which we will call the 'Manual Approach', and automatic page structure generation (as was done in Hera), which we will call the 'Automated Approach'. In GRAPPLE these two approaches and a third one are supported, and each of these will be described in this section.

4.4.1 Manual Approach

With the Manual Approach, the author writes all pages individually (see Figure 18). The page of a concept is completely specified in one file, and each concept has therefore only one associated resource which is used as the basis of a page. Adaptation constructions and other resources (such as pictures) are included by hard-coding them into the page. It is possible for the author to use a template if (s)he writes one, but on each page the details must be customised to the concept(s) that the page is used for.

The advantage of the traditional approach provides extreme flexibility, since the author can specify each page into detail. And it is sometimes even possible –at least in AHA!- to select a different resource that specifies the concept's page based on a condition.

The main disadvantage of this approach, however, is that after the pages have been written –usually by copy-pasting a basic file of which the details are changed for each concept- it is very labour intensive to change anything for all pages.

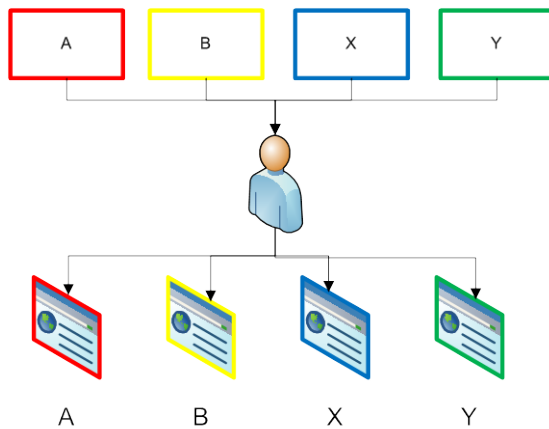


Figure 18: Manual Approach, the author writes a separate page for each concept

4.4.2 Automated Approach

With the Automated Approach the author does not specify the page layout for any concept in particular. Rather, the layout of the concept's page is automatically generated by choosing a predefined layout concept. A concept is thus only associated with resources that describe its content, and ideally these resource do not contain any application dependent information. The layout concepts contain references to generic attributes, such that the references can apply to any concept. An element would thus refer to the 'image' attribute of the concept, rather than hardcoding its value as would be done in the manual approach.

Note, that this approach requires that the attributes that are referred to in the layout concept, indeed need to be present for each concept that uses that layout concept. Furthermore, the resources from the DM should be bound to the correct resource attributes. It should be either the responsibility of the CAM tool to ensure this, or there should be a sort of 'graceful degradation' notion that simply leaves a blank space or removes a page element from the layout if the required attributes are not present. In the former case, the CAM tool could automatically generate the UM attributes that are used in the layout concept, and it could require the author to assign resources or properties from the DM to the other attributes that are used in the layout concept (not being UM attributes). It would be desirable then, that the attributes are replaced by their equivalents if there are CRTs that generate these attributes as well¹¹.

An early version of this approach was implemented in the Milky Way example application with AHA! 4.0, and served as a major breakthrough in our ideas about this authoring approach (see Figure 19). It worked as follows: in a browser the area of the window can be divided into several (rectangular) areas in a table-like way. Each area had a (hardcoded) 'view' assigned to it, and these views typically contained fragments of the page. Examples of views that were defined for the Milky Way example application are the title view, the image view, the description view and the parent tagline view. The views were associated with the different parts of the window in ConceptConfig files; these are separately stored configuration files that specify presentational details like the colour schema for hyperlinks, which processors (that were needed by the adaptation engine) could be found where and also how the page is structured and which view is presented where. The ConceptConfig file that we kept in AHA! 4.0 for each application (although this file can be used by other applications as well) specified several different page structures by defining lists of views. Each page structure belonged to a separate 'concept equivalence class' and each concept belonged to one of these classes. The disadvantages of this approach are first of all that it mixes code that should be configurable for authors (the structure of pages) with code that they should not change (processor locations). Secondly, the files were quite difficult to understand for normal users, which made it difficult for the author to indeed specify page layout.

¹¹ e.g. if a 'visited' attribute is generated for a concept because it appears in a used page element, and then a CRT that counts the visits to a concept wants to create a more elaborate 'visited' attribute for that concept, the attribute that was created for the page element (which is likely less advanced) should be replaced by the attribute created by the CRT.

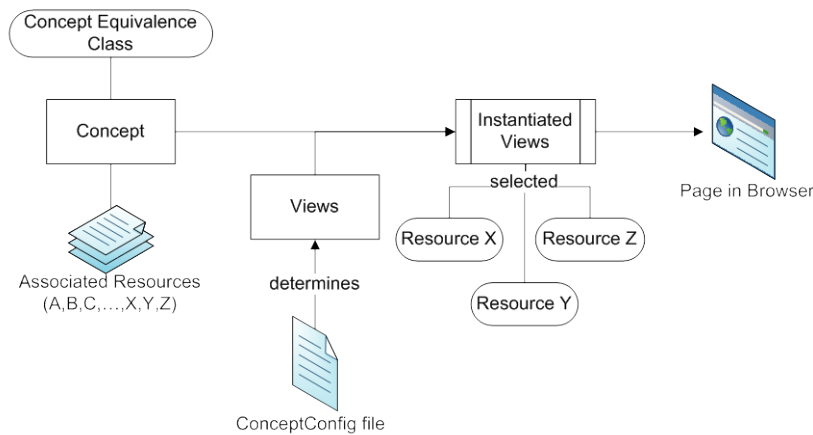


Figure 19: Automated approach in AHA! 4.0: the concept's equivalence class and the application's ConceptConfig file determine which views will be displayed on the page in what order (thus determining the page's structure).

In GRAPPLE we still use the idea of AHA! 4.0 albeit in a somewhat updated form (see). The idea of ConceptConfig files as they existed for AHA! has been banned from the GRAPPLE project, since these files combined technical information that the author should not change as well as the layout definitions. In GRAPPLE we use the layout concepts that specify a page structure, and 'normal' concepts inherit their layout from them. The layout concepts that are used for the automated approach, currently still consist from page elements which resemble the former 'views', that is, currently the page elements are really pieces of code. This is a temporary workaround, though, and in the future these page elements will merely *represent* a part of the page structure without specifying it. A processor that is still to be developed, will then automatically generate the page content, i.e. the code that currently resides within the page elements. We will refer to this processor with the name 'automatic layout processor'.

The automated approach solves the manual approach's problem of having to write many nearly identical pages, because the author merely has to associate concepts with a predefined layout. Note, that this approach is not identical to that of Hera, in which there is no intervention of the author concerning the presentation of content whatsoever.

The main disadvantage of the automated approach is that it is less flexible because it only allows the use of predefined layouts, or in the best case only the page elements are predefined and the order is freely adjustable. This causes difficulty for the presentation of exceptional, concept-specific information which can not easily be incorporated in the resources of the concept.

We illustrate the problem with the following example: pages about the concepts 'star', 'planet' and 'moon' should all have the same structure, except for the fact that we want the page about the concept 'star' to display the message *"tonight the observatory is open and students have free entrance!"* on certain dates (assuming that the system is somehow aware of today's date). Furthermore we assume that the 'star' concept has different resources for beginner students and for advanced students.

To solve this problem, and in general the issue of deviations from the normal structure, there are the following options:

- The exceptional content must be added to the resource(s), or
- A conditional fragment inclusion holding the content must be included in the code of one of the page elements in the automatic layout processor (assumes that the author has write-access to the automatic layout processor), or
- A separate page element which is conditionally included must be defined for the content (again, assumes that the author has write-access to the automatic layout processor)

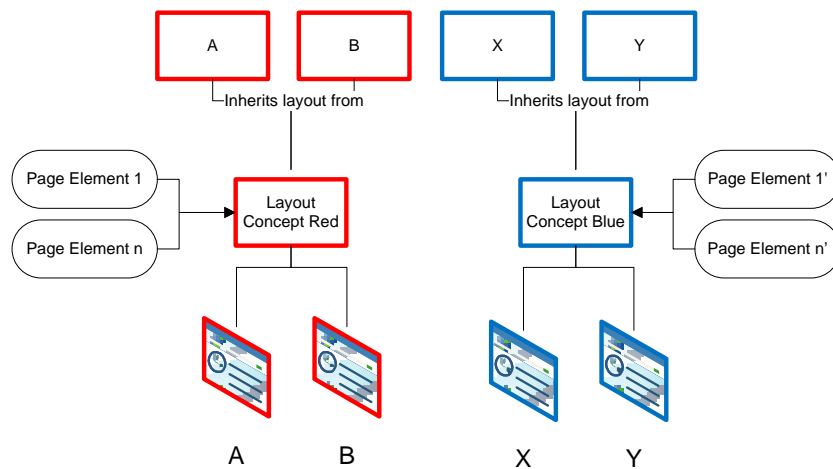


Figure 20: Automated approach in GRAPPLE: concepts inherit their layout from layout concepts that define a page structure using predefined page elements

The first option imposes problems on the authoring process if there are multiple resources that are candidates for a certain role. Next to that, if the content exception is not related to the content of the resource, it is not logical to include it in the resource and it might cause problems if the author forgets where (s)he put it. Our example suffers from both issues: it would require to conditionally include the message in both the beginner and the advanced version of the resource, and the message is not related to the teaching material about stars.

The second option first of all requires that the author has write-access to the automatic layout processor. Assuming that the author has indeed the rights to do so, which will probably not be the case, a hardcoded conditional fragment inclusion in the page element code adds extra –application specific- code and increases the complexity, which is undesirable. Next to that, the addition of content at illogical places can lead to problems later on in the authoring process. Consider for example to which page element we would have to add the message about the observatory. Probably it would be in the ‘title’ element, since that comes closest to an announcement of any kind. So assuming that we chose the ‘title’ element, we would have to include a conditional statement that is only needed for a concept with one particular URI (which might be one of several dozens); imagine how the ‘title’ element would look like if we would like to include messages from ten other concepts. Needless to say, this approach is very detrimental for the readability of the element’s code and the performance of the automatic layout processor.

In the last option, for which the author would also need write-access to the automatic layout processor, the author defines a separate page element (again in the code of the automatic layout processor) which is conditionally included in the page layout specification. Note that this new element should also be added to a layout concept somehow. This solution only moves the added complexity that is created in the previous alternative, and furthermore it would require a more difficult programming job from the author.

As these ‘solutions’ show, there is no elegant way to solve the problem of exceptional content. Furthermore, it should be noted that although GRAPPLE is completely open-source, the GRAPPLE consortium has no intention of providing the author with any support for writing their own version of the (future) automatic layout processor. This practically eliminates the latter two options from any further consideration. The template based approach, which covered in the next section, is therefore much better suited for dealing with layouts deviate from the general page structure.

Another problem of the automated approach is that it is impossible to include something depending on the presence of a page element in the page. Consider the situation in which the page element for the concept’s image is only included depending on the device that the application is run on, and that the inclusion of content in another page element, say the text element, depends on the presence of the image. The text element has no way of knowing whether the image element is present other than by testing on the same condition as the condition for showing the image element, which would thus make the text element dependent from the device rather than from the presence of the image element.

4.4.2.1 Page Elements of the Milky Way Example Application

In this section, we briefly describe the page elements that were designed for the implementation of the automated approach for the Milky Way example application. The Milky Way example currently works based on a workaround that needs page elements with code, but the future automatic layout processor should

automatically generate a page structure from merely selecting a layout concept that only describes the order of code-less page elements (they are then merely placeholders).

The contents of the elements are written in the same format as templates (see 4.4.3.1), although the constructions that are used are required to be application independent if the page elements of the workaround are to be used for multiple applications. The exact code of the elements can be found in Appendix C.

- **Header element:** includes a file named "header.xhtml", which is supposed to reside in the folder that is one level above the folder in which the header element is stored.
- **Title element:** prints the concept's title attribute in <h1>-format. Concepts that have this element included in their page layout, must possess a "title" attribute.
- **Parent element:** tests whether the current concept has a parent, and if so prints a parent tagline with the following structure: "Is", followed by a link to the parent of the concept, followed by "of: ", followed by a link to the semantic parent of the concept. An example instantiation of this element is the parent tagline of the moons of Jupiter: "*Is [Moon of: Jupiter](#)*". These concepts have 'Moon' as their parent, since they are moons, and 'Jupiter' as their semantic parent, since they rotate around Jupiter. Concepts that have this element included in their page layout, must possess both a "parent" and a "semantic parent" property.
- **Image element:** prints the label of the concept's "image" attribute, followed by the inclusion of the concept's image. Concepts that have this element included in their page layout, must possess an "image" attribute.
- **Description element:** prints the label of the concept's "info" attribute, followed by the inclusion of the concept's description (which URI resides in the "info" attribute). Concepts that have this element included in their page layout, must possess an "info" attribute.
- **Related Concepts element:** first checks whether the concept has any semantic children, and if this is so prints the following tagline: "The following ", followed by a link to the parent of the semantic children, followed by "(s) are related to ", followed by the title of the current concept, followed by ":". Below this tagline, an unordered list with links to the semantic children of the concept is printed; this is done by a for-loop. An example instantiation of this page element is the list of moons related to Mars:
"The following [Moon\(s\)](#) are related to [Mars](#):
*[Phobos](#)
*[Deimos](#)"
Concepts that have this element included in their page layout, must have children that both possess a "parent" and a "semantic parent" property.
- **Visited element:** this element prints "Visits: " followed by the value of the concept's "visited" attribute. Concepts that have this element included in their page layout, must possess a "visited" attribute.
- **Guided Tour element:** The Guided Tour element comprises the navigation of a guided tour of which the set of concepts that should be visited and their order have been specified by the author in the authoring process. The navigation consists of a collection of hyperlinks that aim to guide the user along the tour's concepts in the correct order. The most important components of this page element are the "Next >>"-link and the "Back to Tour >>"-link of which the former is displayed on pages of concepts that are part of the tour and the latter is displayed on pages of concepts that are not part of the tour. Additionally, the page element provides links that allow the user to start, pause or exit the tour.

4.4.3 Template Based Approach

The Template Based Approach combines the efficiency of the automated approach with the flexibility of manual page writing (of course, both can only be obtained to a certain extent). It works as follows: each concept inherits its layout from a layout concept, as in the automated approach. But rather than a predefined page structure definition, the layout concept contains a template file that is written by the author, like the concept's main resource in the manual approach. This template file abstractly describes the page layout and contains no concept specific content (unless the author wants to).

In the same way as in the automated approach the template refers to attribute values rather than hardcoded values, which allows for reuse of the same template for many pages. There is a great advantage over the automated approach, though: it is possible to add content that deviates from the normal structure, because it can simply be written in the template. Since the author has to write the templates, it is also up to him/her to decide what is included. For exceptional content the author can thus either include the content by conditional fragment inclusion, or (s)he can create a separate template for the concept(s) involved.

Note that also here it must be ensured that the attributes that are used in the template indeed exist and have the correct type in the DM and UM. Since it would be a major burden on the authoring tool to require it to enforce the correctness of the models with this much flexibility on the author's side, it is considered to be the Authoring of Adaptation in the GRAPPLE Project

author's own responsibility to ensure that the concepts have the correct attributes. Since the template based approach is not propagated for its simplicity of use, we do not consider this responsibility as problematic for the author. This would in fact be a problem for the automated approach, because that approach is propagated for being easy to use.

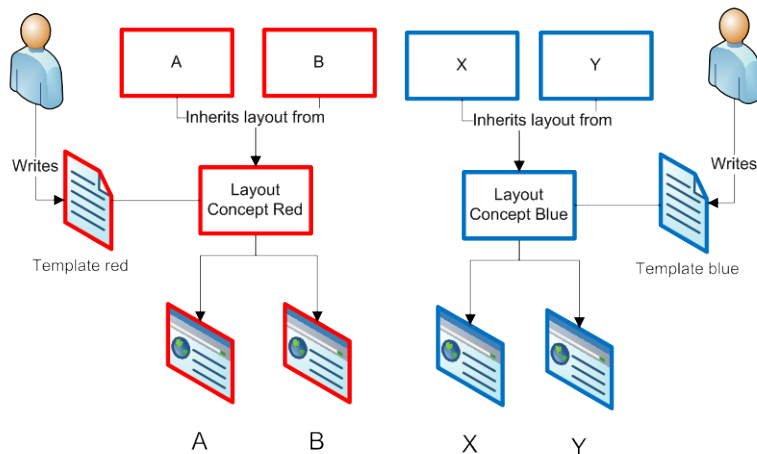


Figure 21: Template Based approach: the author writes templates on which the layout of the concept's pages is based

4.4.3.1 Writing a Template

In this section we will briefly explain the syntax of the template as it is used as direct input for the GALE Adaptation Engine. The definitive syntax that is used for the GAT will be heavily based on this syntax, but some details will differ. In this section we will discuss the presentational aspects, reference to attributes and properties, the use of concept relationships, the use of links and finally some constructions that are needed for inclusion of content.

When we refer to the 'relative name' of a concept in this section, what we mean in fact is the part of the URI that distinguishes the concept from the other concepts from the application DM. So if the URI of the application would be "gale://grapple-project.org/Milkyway/", then the name of the concept would be the part that suffixes the application's URI; when we refer to the relative name of the concept "gale://grapple-project.org/Milkyway/EtaCarinae", we mean "EtaCarinae".

First of all the template must always start with the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
  <html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:gale="http://gale.tue.nl/adaptation">
```

The <html>-tag must be closed again at the end of the document with the </html>-tag.

The basic presentation aspects, which exclude anchor tags, image tags and object tags, work the same as in normal html. The mostly used elements are:

- Headers: the largest being <h1></h1>, the smallest being <h6></h6>
- Font styles: for bold, <i></i> for italics and <u></u> for underlined text
- White lines:

- Bulleted lists: to mark the start and end of the list, and for each item

References to attributes and properties of concepts have a special, but very straightforward syntax. If the value of a property or attribute is to be denoted, we simply write '\${' with the name of the property or attribute between the curly brackets. This construction will return the content of the 'value' attribute for a property and the result of the evaluation of the content of the 'default' element for an attribute (more on this follows in Section 6.3). Furthermore, attributes are prefixed with a hash ('#') and properties are prefixed with a question mark ('?'). Finally, if an attribute or property of the currently visited concept is addressed, the concept does not

have to be specified, otherwise the name of the target concept must be specified right after the opening curly bracket. This leads to the following syntax:

- Attribute reference ("attribute"), current concept: `#{attribute}`
- Property reference ("property"), current concept: `#{?property}`
- Property reference ("property") of attribute ("attribute"), current concept: `#{attribute?property}`
- Attribute reference ("attribute"), foreign concept ("concept"): `#{concept#attribute}`

In case one does not want to refer to a particular concept, but rather to a concept that is somehow related to another concept, one can use the following construction to refer to the input and output concepts of domain relationships:

- Reference to concepts that are output concepts of the concept relationship "relationship", having the concept "concept" as a parent according to this relationship: `#{concept->(relationship)}`
- Reference to concepts that are input concepts of the concept relationship "relationship", having the concept "concept" as a child according to this relationship: `#{concept<-(relationship)}`

The key of this syntax is that the arrow ('->' or '<-') is bound to the specified concept on the left, and the right side of the arrow then indicates the concept that is wanted; depending on the orientation of the arrow the input or the output concepts are denoted. In case the unbound part of the construction can result in more than one concept, the first one of the concept hierarchy is selected. If another concept should be selected, the author can specify this writing the number of that concept between square brackets right behind the construction (e.g. `#{->(parent)[2]}`), starting by 0 (which is the first concept). Note, that this is a potentially very dangerous construction, since not all concepts that make use of the template may have enough concepts related to them as specified in the construction. The author is therefore urged to include a test that checks whether the target concept indeed exists before, this construction is used.

Similarly as with the references to attributes and properties, it holds that if it is the currently visited concept that binds the construction, the concept does not have to be specified, otherwise the relative name of the concept must be specified.

Also, note that one will often want to refer to an attribute or property of the concept that is referred to with this construction, thus often the construction will be extended with a '#' or '?' part.

The following example would denote the title of concept lo's parent planet: `#{gale://grapple-project.org/milkyway/Io<-(isMoonOf)#title}`

To inform the adaptation engine about the fact that a reference to an attribute or property of some concept (which again may have been referred to by an expression that refers to the concept's relationship with another concept) is something that the engine must act upon, the aforementioned constructions are always assigned to the *expression* attribute of an XML element: 'expr'. If the result of the expression must simply be printed, the expression is put in a `<gale:variable>`-construction, which has the following syntax: `<gale:variable expr="expression"/>`. Other XML elements that have an expression attribute are covered below.

The 'expr' attribute indeed solves the problem of XML *elements* that use an expression, but it is also possible that one would like to assign an expression as a value to the *attribute* of an HTML or XML element. Since the GALE Adaptation Engine only processes values of the attribute 'expr', it is thus impossible to assign an expression to another attribute. To solve this problem, the `<gale:attr-variable>` element has been introduced. This element has two attributes: 'name' and 'expr', of which the former is used to indicate which attribute of the parent element we would like to assign the value of 'expr' to. So, if we have an element 'X' with attribute 'A' to which we want to assign expression 'E', the construction becomes: `<X><gale:attr-variable name="A" expr="E"/></X>`.

The creation of hyperlinks to another concept in an adaptive application is somewhat different from creating normal hyperlinks in html: it exactly suffers from the problem that was just described. A normal hyperlink would have the following structure ` anchor_text `. In the template syntax, first of all the href-attribute is removed from the opening `<a>`-tag. The reason for this is that the adaptation engine must somehow be instructed that the link is not a normal link, but a link to a concept, and by removing the href-attribute from the `<a>`-tag we have the chance to indeed intercept the engine from interpreting the tag as 'normal html'. To create a reference to the target concept, we use a `<gale:attr-variable>`-construction in which the 'href' attribute is assigned the expression that specifies the target concept. The original html hyperlink-construction, is thus changed into: `<a><gale:attr-variable name="href" expr="link_target"/> anchor_text `.

Both the link target, which should be the (relative) name of the target concept, and the anchor text can be attributes and properties as expressed in the above. Note, that for the anchor text a `<gale:variable>`-construction must surround the expression.

Finally, if the adaptation engine must apply link adaptation to the link, the `<a>`-tags should be changed into

<gale:a>-tags. In the exceptional case that on the user's following the hyperlink some attribute (not property) should be updated, the <gale:a>-tag can be elaborated with the "exec" attribute (standing for 'execute'). The value of this attribute should be an expression of the following form: `# { target_attribute , new value } ;` (note the semicolon at the end). The instantiation of the target attribute works in the same way as for attribute reference as described above. The new value of the attribute can be any expression that would be valid in Java; note that attribute values are referred to with the aforementioned `{ }`-construction. Of particular interest might be the conditional expression which has the form: `(condition ? true-expression : false-expression)`. This tag should be used with extreme caution, since the author directly accesses the UM via this tag. Therefore only advanced authors should use it.

There are several alternatives for the inclusion of content:

- Object and image inclusion, in which a file is included; the latter being an image.
- Conditional (fragment) inclusion, in which any content can be included conditionally.
- For loop, this automatically iterates over some set of concepts, and repeats the body of the loop for each instance.

The syntax for inclusion of objects is relatively easy. Where the normal syntax for the inclusion of an object in html is `<object data="object_URI"></object>`, the template syntax is: `<gale:object data="object_URI" /></gale:object>` if the URI of the object is directly typed. If the URI of the object is stored in some attribute, the template syntax is:

```
<gale:object>
  <gale:attr-variable name="data" expr="attribute_reference" />
</gale:object>
```

, *attribute_reference* being an attribute reference as described before.

The syntax for inclusion of images is exactly the same as in html if the URI of the image is directly typed, e.g. `` (note that the URI is relative; an absolute URI is allowed as well). If the URI of the object is stored in some attribute the template syntax is:

```
<img>
  <gale:attr-variable name="src" expr="attribute_reference1" />
  <gale:attr-variable name="width" expr="attribute_reference2" />
</img>
```

, *attribute_reference1* and *attribute_reference2* being attribute references as described before; note that the former attribute must store a URI and the latter (optional) attribute must store an integer.

An important content inclusion construct is conditional (fragment) inclusion, which basically is an if-then-else construction for printing content rather than executing code. The syntax is very straightforward:

```
<gale:if expr="expression">
  <gale:block> true-content </gale:block>
  <gale:block> false-content </gale:block>
</gale:if>
```

The *expression* can be any valid Java expression, in which attributes can be involved using the aforementioned attribute reference. The *true-content* and *false-content* can be normal code as it if was written in the rest of the template, and finally, the block with *false-content* is not required.

The most complex construction that can be used in a template is the for-loop; it has the following syntax:

```
<gale:for var="variable" expr="expression">
  body-content
</gale:for>
```

In this construction *variable* is the for loop's variable, which acts as a placeholder for element that is used for the current iteration, and *expression* is an expression that refers to the (unordered) set over which the for loop iterates. A very usable instantiation is a concept reference through the use of a concept relationship, such that for example the loop can iterate over all input or output concepts of a certain DM relationship.

Although the syntax does not seem very complicated so far, there are some issues that one must especially consider. First of all, for any use the variable must be prefixed with the percentage mark ('%'). Furthermore, when the variable is not used in for an attribute or property reference or for a concept reference using a relationship, the (prefixed) variable must be surrounded with the string """. This is for example the case in hyperlink references. An example of a for-loop is:

```
<ul>
  <gale:for var="concept" expr="&#{&lt;- (isMoonOf)}">
    <li><gale:a>
      <gale:attr-variable name="href" expr="&quot;%concept&quot;" />
      <gale:variable expr="&#{%concept?title}" />
    </gale:a></li>
  </gale:for>
</ul>
```

Although some of the constructions –like object and image inclusion, adaptive hyperlinks (without executable code) and conditional content inclusion- and the syntax for presentational aspects already existed in projects before GRAPPLE, the syntax in its current form is mainly the result of the authoring process of the Milky Way example application. In earlier forms the syntax was quite complex, but during the process we have come to a standard that seems to be fairly usable and capable of covering the requirements that authoring can impose. Since the syntax was based on the creation of one application only, however, there might still be needs that have remained undiscovered so far.

Finally, it is very important that the author ensures that the attributes and properties that are used in the template, are also present in the concepts that (s)he intends to use the template for. It is the author's own responsibility to check for this.

5 Translation of Authoring Models into Engine Independent Specifications

In this chapter we explain how the authoring models and CRTs can be translated into the DM, UM and GAL specifications. First we consider how the DM and the layout concepts that describe the concepts' page structures are expressed in the DM, UM and GAL specifications, and then we elaborate on how the CRTs further detail the specifications thus obtained. Note, that we suggest an order in the compilation process here, that is in fact not required. The CRTs may very well not be processed only after the authoring models (especially the DM).

Finally note that when we refer to 'adding something to the specification', we in fact mean 'adding something to the part of the specification that represents the current object of consideration'.

5.1 Bases of the DM, UM and GAL specifications

The DM serves as a basis for each of the three parts of the engine independent application specification. Each DM concept is used for the generation of an overlay concept in the DM and UM specifications and for the generation of two GAL Units in the GAL specification: one Unit that represents the page of the concept and one that represents the concept in the treeview. The partial specifications that are produced based on the DM form the skeleton of the application's engine independent specifications; the CRTs will fill out the details.

5.1.1 DM specification

The skeleton of the DM specification is very similar to the original DM: only the structure is different, and therefore the translation is quite straightforward. First of all, the concept's 'termIdentifier' from the DM, see Code Fragment 12, is used to denote the start of a new concept construction in the DM specification, see Code Fragment 7. Secondly, the content of the 'caption' element's 'langstring' element (Code Fragment 12) is used as the value of the DM specification's 'dm:title' (Code Fragment 7). If there is a 'description' element present in the original DM, which is not the case in our Milky Way example application, the content of its 'langstring' element should be the object by a 'dm:description' predicate. Note, that whenever a 'langstring' element can be used in the original DM there may be multiple candidate values (one per language) for the according DM specification's elements. In this thesis, we assume that if there are multiple candidates, always preferably the English (UK) or the English (US) value is selected, and if there is no English candidate the first one is selected.

The 'mediaDescriptor' elements are less straightforwardly translated. MediaDescriptors are represented in the DM specification by a 'dm:resource'-block which contains at least a 'dm:value' element whose value is the content of the mediaDescriptor's 'mediaLocator' element. The meta-data that is associated with the mediaDescriptor (see Code Fragment 12) accounts for the remainder of the content of the 'dm:resource' element. The meta-data's 'name' element, which is a required element, is indeed the name of the resource, and is therefore the value of the 'dm:resource' element's 'dm:name' element. All other meta-data elements are optional, and are translated for the DM specification by simply prefixing the meta-data element's name with 'dm:' and use the element's value as the object for the predicate that was constructed from its name.

Any information that would in the DM specification be represented by a 'dm:fact' element, see Section 3.3, would most probably be derived from meta-data in the original DM as well. So far, however, it has been unspecified how information other than a title, a description and resources are to be represented in the DM. The IMS VDEX grammar that is used for the DM, though, is most suitable to indeed handle extra information by the addition of metadata to the term. In this case, the translation of that data for the DM specification would be similar to the approach that was taken for resources.

Finally, the original DM's concept relationships are translated by prefixing the value of the 'relationshipType' element with 'mw:' (see Section 3.3) and adding it as a predicate to the element that in the DM specification represents the DM concept specified in the 'sourceTerm'. The object of this predicate then is the value of the 'relationship' element's 'targetTerm' element. The relationship that is represented in Code Fragment 13 would thus result in "mw:rotatesAround gale://grapple-project.org/milkyway/mars" being added to the concept 'gale://grapple-project.org/milkyway/phobos'.

5.1.2 UM specification

The skeleton of the UM specification is, similarly to the DM specification, based on the original DM (in fact on the original UM, but this is an overlay model of the DM and was omitted from the discussion in this thesis). Of each DM concept the 'termIdentifier' element is again used to denote the start of a new concept in the UM specification. These concepts, however remain initially empty, since for this thesis it was assumed that there is

no special information in the UM and thus the only source of information for the UM specification are the CRTs.

5.1.3 GAL specification

In the GAL specification, each DM concept results in a Unit that represents the page of the concept (referred to as 'page Units') and a Unit that represents the concept in the treeview (referred to as 'treeview Units'). The content of the page Units is derived from the page structure that is specified in the layout concept of which the page Unit's concept inherited its layout details (see Section 3.5). In this section we will consider the GAL representation of the constructions that are used in the templates that were used for the Milky Way example application.

Before we turn to the translation of the layout constructions into GAL, however, we briefly point out how for each DM concept the corresponding GAL Units are generated. Similar to the DM and UM specifications, the value of the 'termIdentifier' element of the concept (Code Fragment 12) is used as a prefix for the name of the Unit. The complete Unit names adhere to the pattern "*termIdentifier_Unit*" and "*termIdentifier_TreeView_Unit*". Then it is declared that the construction is a GAL Unit by adding the line "a gal:Unit;". The 'empty' Units as derived from a concept in the original DM are printed in Code Fragment 23. Note, that all treeview Units by construction only express a hyperlink to the page Unit of the concept that they represent.

```
gale://grapple-project.org/milkyway/etacarinae_Unit [
  a gal:Unit;
];

gale://grapple-project.org/milkyway/etacarinae_TreeView_Unit [
  a gal:Unit;
  :hasAttribute [
    gal:value [
      gal:hasQuery gale://grapple-project.org/milkyway/etacarinae.dm:title;
    ]
    gal:hasLink [
      gal:refersTo gale://grapple-project.org/milkyway/etacarinae_Unit;
    ]
  ];
];
```

Code Fragment 23: Empty GAL Units for concept 'Eta Carinae' as they would be derived from the original DM

Appendix D contains the resource of a template layout: a template file of the Milky Way example application. The template file was originally written for pages of concepts with an 'isa' relationship with the Milky Way 'star' concept.

The template starts with a header section in which an object inclusion is done. Since the object for inclusion only is referred to by a relative path to a file, this section is problematic for the GAL specification. The reason for this is that the GAL specification must be application independent, and a relative path to a file violates this requirement. This issue can be handled in two ways: the header section can be neglected by the Engine Independent Compiler, or an attribute can be automatically generated. In the latter case, a 'dm:resource' element with a generated name and the relative path as a value should be created in the DM specification, and in the GAL specification an attribute that has the value of the generated 'dm:resource' element as its value should be created. In this thesis we opted for the former alternative which leaves the section out.

The title section that follows the header in the Milky Way application's page structure consists of a <gale:variable>-construction between <h1> tags. The heading tags are (always) neglected by the Engine Independent Compiler. The <gale:variable>-construction refers to the 'title' attribute of the concept. Since the 'title' element (and optionally the 'description' element) in the DM specification are the only concept attributes with an exceptional structure compared to the structures for the notation of factual information and resources (see Section 3.3), we assume that 'title' is recognised as a keyword by the Engine Independent Compiler such that it automatically adds a 'gal:hasAttribute'-construction with the value "*concept_name.dm:title*" to the page Unit. This assumption is necessary here, because later on we want to be able to state that the same approach is always followed when a hash ('#') is encountered in the template; and this approach would not work for the 'title' attribute because the title is an exceptional element in the original DM.

```
<h1><gale:variable expr="#{title}" /></h1>
```

Code Fragment 24: Construction denoting the title of a concept; this construction is an exception for the 'title' attribute.

The section that defines the 'parent tagline' of the concept, is one of the more complex constructions for translation into GAL. It is useful to first consider the structure of the tagline: the tagline starts with a string, followed by a hyperlink with anchor text, followed by another string and hyperlink with anchor text (see Code Fragment 25). First of all, both strings get their own GAL attribute, which is in this case simply an attribute with the string as its value. Then both hyperlinks get their own 'gal:hasAttribute'-construction, for which the anchor text is the value of the 'gal:value', which may either be a string or a query. In this case both anchor texts are queries for the title of a concept that is denoted by a reference based on a concept relationship ("*relational reference*"). Note the important difference between generic (with the same constraints on the existence of attributes as for application dependent templates) and application dependent templates here: application dependent templates are allowed to be application specific (by definition), and may thus mention the concept relationship explicitly as long as all concepts that use the template have that concept relationship. In Appendix D we may thus directly write "->(isa)". For the generic template, however, we must refer to the *role* of the relationship that must be used; the code that was used for the 'parent tagline' page element for the Milky Way example application in Appendix C demonstrates that we must write "->(~parent)", ~parent being a placeholder rather than the name of a DM relationship. The Engine Independent Compiler must fill out the correct concept relationship (in this case "isa") at compile time for the placeholder. The anchor text query of the hyperlink is in this case –because the title attribute of the queried concept is asked- translated as follows (see Code Fragment 26): the Select-clause contains a local variable, in this case '?T', and the Where-clause binds this variable depending on the form of the relational reference:

- ->(relationship)#title results in:
`<concept name of the concept for which the Unit is> dm:relationship ?P
dm:title ?T`
- <- (relationship) #title results in:
`?C dm:relationship <concept name of the concept for which the Unit is>;
dm:title ?T`
- If instead of a relationship '*relationship*' a placeholder, like '~parent' in Code Fragment 25, is used the placeholder is substituted with the correct relationship at compile time.
- ->(relationship) <-(relationship1) #title results in:
`<concept_reference name> dm:relationship ?Concept1.
?Concept2 dm:relationship1 ?Concept1;
dm:title ?T`
- ->(relationship) ->(relationship1) #title results in:
`<concept_reference name> dm:relationship ?Concept1
dm:relationship1 ?Concept2
dm:title ?T`
- <-(relationship) <-(relationship1) #title results in:
`?Concept1 dm:relationship <concept_reference name>;
dm:relationship1 ?Concept2;
dm:title ?T`

Finally the target of the hyperlink, which is denoted in the `<gale:attr-variable name="href">`-element, is the value of the 'gal:refersTo' element of the 'gal:hasLink' construction that immediately follows the query of the 'gal:value' element. The query that specifies the hyperlink's target is resolved in the same way as was done for the anchor text, in fact the anchor text and the hyperlink's target are often the same.

```
<h4>Is <gale:a><gale:variable expr="{->(~parent)#title}" /><gale:attr-variable  

name="href" expr="{->(~parent)#title}" /></gale:a> of:  

<gale:a><gale:variable expr="{->(~semantic_parent)#title}" /><gale:attr-variable  

name="href" expr="{->(~semantic_parent)#title}" /></gale:a></h4>
```

Code Fragment 25: Fragment of the predefined page element 'Parent Tagline'

```
gal:hasQuery [Select ?T  

Where <concept name of the concept for which the Unit is>  

$hasParent ?P  

dm:title ?T  

];
```

Code Fragment 26: Translation in GAL of the anchor text query of the first hyperlink in Code Fragment 25

Recall that in the code in Code Fragment 25 is based on the code that can directly be used by the GALE Adaptation Engine, and which therefore already takes care of the link adaptation by the usage of `<gale:a>` tags instead of normal `<a>` tags. This should not be the case in the future definitive format, since the Link

Recommendation CRT, which is still to be applied by the Engine Independent Compiler, should take care of this.

The next construction in Appendix D is the construction for image inclusion, which works for the translation to GAL similar as object inclusion (which is done for the description). For both constructions we see a reference to the label of an attribute (`{#image?label}`) and a reference to the attribute itself (`{#image}`); for the image inclusion after the label a reference to the concept's title attribute follows (see Code Fragment 27). The latter can not be elegantly translated into GAL, since there is no way for the Engine Independent Compiler to find out that the reference to the title should be a part of the attribute's label in the GAL code; therefore we neglect the reference to the title here.

```
<gale:variable expr="{#image?label}"/> <gale:variable expr="{#title}"/> <br/>
<img><gale:attr-variable name="src" expr="{#image}"/></img>
```

Code Fragment 27: Reference to a (resource) attribute and its label in a template

If the Engine Independent Compiler encounters a reference to an attribute in a construction that is intended for presentation, that is all constructions referred to in Section 4.4.3.1 except for the 'for-loop' construction, everything between 'exec' tags and the if-clause of the conditional fragment inclusion construction, it automatically generates a 'gal:hasAttribute' construction in the GAL specification. The reason that this can be done so easily is that it is sure that everything that appears in these parts of the code of a template or a predefined element is by definition intended for navigation and should therefore be included in the GAL specification.

To determine the value of the newly created GAL attribute, the Engine Independent Compiler must first find out whether the attribute under consideration is a UM attribute or a resource attribute. This can be done by checking the code of the CRTs for references to an UM attribute (referred to as 'UMvariable' in the CRT format). If a reference is found, the value of the GAL attribute is a query for the value of the UM attribute (see Code Fragment 36, more on this follows in this section). If no reference is found, the value of the GAL attribute must be a query for a resource attribute. Recall that factual information, which is stored in properties, is denoted by a question mark ('?') (see Section 4.4.3.1). The translation to GAL of references to the value of a property can be found in Code Fragment 28.

```
gal:hasAttribute [
  gal:name <property_name>;
  gal:value [
    gal:hasQuery [Select ?V
      Where <concept name of the concept for which the Unit is>
dm:fact ?F
      dm:name <property_name>;
      dm:value ?V.
    ];
  ];
];
```

Code Fragment 28: GAL code for the presentation of a property (factual information)

If the value, like in this case, turns out to be a resource query (see Code Fragment 29), it must somehow be determined which resource should be selected. Or to put it differently: the attribute must be associated with a resource. How this is exactly to be determined, however, is still unclear but there are two main options: one that requires the author to associate the attributes that are used in the templates to the resources, and one that abandons the use of attributes for resources.

The latter alternative, the author inserts the entire query for the correct resource directly in the template. For the automated approach this would require the author to use certain predefined names for their resources. For the template based approach this would allow the author to define the resource names on his own, and it would leave the responsibility of correctly querying the correct resource to the author.

Note, that this alternative lets the author takes care of resource selection for the template based authoring approach, and that for the automated approach it would require a special Resource Selection CRT that changes the initial value of a resource attribute into a resource selection query. Note, furthermore, that in the template based approach the author can specify *any* resource selection query, whereas in the automated approach (s)he is bound to the resource selection queries that the variety of Resource Selection CRTs define (unless the author defines his own Resource Selection CRT).

The other alternative would require the author during the authoring process for the automated approach to indicate for each of the resources to which of the (generic) attributes of the predefined layout concept they belong.

If the author uses the template based approach, there are several options: a) the GAT could provide a

mechanism that automatically requires the author to bind the attributes that are used for a concept (because it is associated to a template) to a resource. This would be similar to the method that was just described for the automated approach, except for that it is capable of binding any attribute to a resource instead of only the predefined (generic) attributes. Note, that this option requires the associations between resources and attributes to be stored somewhere, and currently this is not possible with the formats of the authoring models. Option b) would be to have the Engine Independent Compiler match resource names with attribute names, and automatically choose the resource with the name that is most similar to the attribute name as a value for the GAL attribute. Resource selection should for the latter option be taken care of by a CRT that updates the initial value of the GAL attribute into a query that returns the correct resource. For the former alternative, resource selection could either be taken care of by a CRT as well, or the mechanism that lets the author bind resources to the attribute could allow him/her to directly specify the resource selection.

For now we assume that we use the alternative that uses attributes in the template. The reason for this is that the future format that is to be used for this approach is very similar to the GALE based format that we use now, and in GALE we indeed use the approach with attributes. We also assume that the author somehow indicates which resources belong to which attribute, and that in case of resource selection they provide the query that selects the correct resource.

```
gal:hasAttribute [
  gal:name image;
  gal:value <user-specified resource-query>;
];
```

Code Fragment 29: GAL attribute for the 'image' attribute without the label; generated from a template or predefined page element.

When the Engine Independent Compiler finds a label of an attribute, it first checks the GAL specification for the presence of such attribute. If it is not present yet, the compiler creates a GAL attribute in the same way as described in the above, and fetches the label from the selected resource (it simply assumes that there is one; if there is none it generates an empty string as the value for the 'gal:label' element). If there already is an attribute with the correct name, the label is simply added. The GAL code for the attribute now looks like in Code Fragment 30.

```
gal:hasAttribute [
  gal:name image;
  gal:label <resource-query replacing predicate "dm:value" with "dm:label">;
  gal:value <user-specified resource-query>;
];
```

Code Fragment 30: GAL attribute for the 'image' attribute with label; generated from a template or predefined page element.

Note that if an attribute with a label is involved in resource selection, the construction as specified in Code Fragment 31 should be used. The reason for this is that this retains the coupling of the label to its resource. The alternative¹², which can be found in Code Fragment 32, suggests that the selection criterion for the label may differ from the selection criterion (which is in practice, with the GAT, impossible), and could thus theoretically result in a decoupling of the labels from their resources.

```
gal:hasAttribute [
  gal:name attribute name;
  gal:hasCondition [
    gal:if [user-specified resource-query selection criterion ];
    gal:then [
      gal:label user-specified resource-query case 'true' replacing predicate
        "dm:value" with "dm:label";
      gal:value user-specified resource-query case 'true';
    ];
    gal:else [
      gal:label user-specified resource-query case 'false' replacing predicate
        "dm:value" with "dm:label";
      gal:value user-specified resource-query case 'false';
    ];
  ];
];
```

¹² Which happens to have been the initial approach in writing the GAL specification for the Milky Way example application.

Code Fragment 31: Correct notation for resource selection

```
gal:hasAttribute [
  gal:name attribute name;
  gal:label [
    gal:hasCondition [
      gal:if [user-specified resource-query selection criterion ];
      gal:then [user-specified resource-query case 'true' replacing predicate
        "dm:value" with "dm:label";
      ];
      gal:else [user-specified resource-query case 'false' replacing predicate
        "dm:value" with "dm:label";
      ];
    ];
  ];
];

gal:value [
  gal:hasCondition [
    gal:if [user-specified resource-query selection criterion ];
    gal:then [user-specified resource-query case 'true';
    ];
    gal:else [user-specified resource-query case 'false';
    ];
  ];
];
];
```

Code Fragment 32: Alternative notation for resource selection

The construction for the displaying of a list of related concepts consists of a construction that we have already seen, namely a tagline, and a for-loop mechanism which lists links to the related concepts. Since the hyperlink construction was also already covered in the above, we only consider the translation of a for-loop (see Code Fragment 33) into GAL here. The for-loop consists of a query that selects the set of concepts over which the loop must iterate and a body that must be executed each iteration. The result of the query is in the code of the template or predefined page element iterated by a local variable; in GAL, however, everywhere where that local variable would occur, the complete query is used, since the 'gal:hasSetUnit'-construction takes care of the iteration. When the Engine Independent Compiler encounters a <gale:for>-tag it adds a 'gal:hasSetUnit' construction to the GAL specification, with the for-loop's query (resolved as described above for the parent tagline) as a value for the 'gal:hasQuery' element. Furthermore a 'gal:subUnit' is created *within* the 'gal:hasSetUnit', which we call 'body_Unit' here, and the phrase "gal:refersTo body_Unit;" is added to the 'gal:hasSetUnit' construction. The content of the for-loop's body is then resolved as if it were normal code, except for that in the GAL code all occurrences of the local variable from the for-loop are replaced by the entire query and all GAL code that is generated is added to the subUnit rather than the main Unit. The code then looks as in Code Fragment 34.

```
<gale:for var="concept" expr="$${<- (~semantic_parent)}">
  <li><gale:a><gale:variable expr="$${%concept#title}"/>
    <gale:attr-variable name="href" expr="&quot;%concept&quot;"/>
  </gale:a></li>
</gale:for>
```

Code Fragment 33: Code for a for-loop in a template. Again, the details of the GALE-specific format should be neglected here; the definitive format for example should not include the '"';s such that the references to the local variable 'concept' are consistent.

```
gal:hasSetUnit [
  gal:refersTo body_Unit;
  gal:hasQuery [ query from the for-loop ];

  body_Unit [
    a gal:subUnit;
    ...content...
  ];
];
```

Code Fragment 34: GAL code generated from a for-loop in the code of a template

The references to the page element 'Guided Tour' in Appendix D should be neglected. The tour mechanism will in the GAT be entirely created by the Guided Tour CRT. Again, it is because the presented code in the appendix is in fact for direct use by the GALE Adaptation Engine that this code is present.

The only reason for a reference to the guided tour mechanism in the layout concepts could be to indicate where the navigational aids of the guided tour are to be displayed. It is, however, also very reasonable to simply position them at the top or bottom of the page by default.

The last construction that we find in Appendix D is the reference to a UM attribute (see Code Fragment 35). Indeed since the attribute is intended for display, a 'gal:hasAttribute' construction is added to the GAL specification. Because, again, the Engine Independent Compiler can not now whether attribute 'visited' is a UM attribute or a resource attribute, it checks the CRTs that are applied to the concept that is currently under consideration for a reference to a UM attribute. In this case, it will indeed find a reference, namely (at least) in the Visited CRT. The value of the GAL attribute, now simply becomes the query for a UM attribute with the same name as the attribute that was found in the template or page element, see Code Fragment 36.

```
<gale:variable expr="{#visited}"/>
```

Code Fragment 35: Reference to UM attribute 'visited' in a template

```
gal:hasAttribute [
  gal:name <attribute_name>;
  gal:value [
    gal:hasQuery [Select ?V
                  Where <concept name of the concept for which the Unit is>
um:attribute ?A
                    dm:name <attribute_name>;
                    dm:value ?V.
                ];
  ];
];
```

Code Fragment 36: GAL code for the presentation of a UM attribute

Finally, there are two constructions that have not been considered so far: the construction for conditional fragment inclusion and that for associating UM attribute updates to hyperlinks.

The construction for conditional fragment inclusion, see Code Fragment 37, can be translated into GAL very straightforward. A 'gal:hasCondition' construction is added to the GAL specification, of which the 'gal:if'-clause contains the condition, the 'gal:then'-clause contains the content of first block of the construction and (optionally) the 'gal:else'-clause contains the content of the second block.

Note, that the true-clause and false-clause both have to be translated into GAL as well.

```
<gale:if expr="condition">
<gale:block>
  true-clause
</gale:block>
<gale:block>
  false-clause
</gale:block>
</gale:if>
```

Code Fragment 37: Code for conditional fragment inclusion in a template or predefined page element

The construction for associating UM attribute updates to hyperlinks (see Code Fragment 38) is also quite straightforward to translate. First of all, a 'gal:onAccess'-attribute is added to the GAL specification directly after the 'gal:hasLink'-construction that is generated by the hyperlink. Then the expression that updates the UM attribute is added in a 'gal:updateQuery'-construction, see Code Fragment 39. Note that the new value that is assigned to the attribute, may again be an expression. This expression is then resolved in the Where-clause as well, according to the rules of the GAL Query Language (see Section 3.1.1).

```
<gale:a exec="#{Concept_name#UM_attribute_name, new_value};"><gale:attr-variable
name="href" expr="query"/> anchor text </gale:a>
```

Code Fragment 38: Code for associating UM attribute updates to a hyperlink in a template or predefined page element

```
gal:onAccess [
  gal:updateQuery [
    Update ?V := new_value;
    Where Concept_name um:attribute ?A
```

```

        um:name UM_attribute_name;
        um:value ?V.
    ];
];

```

Code Fragment 39: GAL Code produced by the 'exec' attribute in Code Fragment 38

5.2 Translating CRTs into GAL and User Model specifications

CRTs fill out the details of the specifications. Using the CAM, the Engine Independent Compiler can instantiate the code of the CRTs with concepts. By applying the instantiated code of the CRTs, the application's specifications are then finalised by addition of the application's adaptive behaviour and navigational aspects.

In this section we will not explain how each of the predefined CRTs is exactly translated into fragments of the specifications, since much of this was already done in Section 4.2.2. Rather we will explain the syntax¹³ and semantics of the current pseudo language that is used to define the CRTs' 'code' elements and we will consider the requirements of the definitive language for this.

The common structure that can be observed for most CRTs is that a for-loop traverses the set of concepts with which the CRT is instantiated, and that for each concept fragments of GAL or UM Specification Rule Expressions are added to the UM and GAL specifications. In exceptional cases, existing fragments of code in the specifications are *replaced* as the result of a CRT, and for the Guided Tour CRT code is added to the *DM specification* rather than the UM specification or the GAL specification.

The general structure of the 'code' element is currently as follows (note again that this is not definitive): first of all one of the tags <specification name="GAL">, <specification name="UM"> and <specification name="DM"> is denoted to indicate to which specification the following code will apply. This then also defines what the code's possible *objects of manipulation*, i.e. the parts of the specification that the code can manipulate, can be. For the GAL specification we distinguish the following objects of manipulation:

- GAL specification: new GAL Units can be added to the specification.
- GAL Units: GAL components can be added to the GAL Units that belong to a concept.
- GAL attributes: the values of GAL attributes can be changed; in theory also 'gal:hasCondition'-constructions can be added to GAL attributes, but so far no example of this has been encountered.
- GAL links: 'gal:hasCondition' and 'gal:onAccess' constructions can be added to hyperlink constructions.

For the UM specification we distinguish the following objects of manipulation:

- UM specification: new entities can be added to the specification.
- UM concept elements: new attributes can be added to concept elements.
- UM concept element attributes: the values of attributes that have 'false' as the value of their 'persistent' property can be changed.

For the DM specification the only object of manipulation is the DM specification itself, since only new entities can be added to the specification. The rest of the DM specification is static by definition.

The precise object of manipulation is then selected in the scope declaration of the for-loop that follows the specification tag. In this part also the binding between the elements from the socket and the object of manipulation is done. The scope of the for-loop is defined with a simple Select-Where query; the Where-clause may consist of multiple parts: it always contains at least an expression to bind the elements of the socket to a variable. If the CRT adds or changes elements of the specification rather than that it adds elements to the specification itself (i.e. at the highest level), the clause also contains an expression for the selection of specification's the object of manipulation. Finally if the object of manipulation is not a concept, the clause should also an expression for the binding of the concept to which object of manipulation belongs, the *concept of manipulation*, to the socket element variable. In the scope declaration query the name of the socket is always prefixed with a dollar sign ('\$').

An example of a scope declaration for a UM specification can be found in Code Fragment 40. It clearly distinguishes the binding of the socket's elements (the concepts) to variable "?out", the binding of the socket element variable to the concept of manipulation and finally the selection of the object of manipulation.

```

FOR ALL (
    Select ?S

```

¹³ The current pseudo-language can in fact not really be said to have a 'syntax', since it is a rather informal language

```

Where ?out IN $OUT-SET.
    ?Concept.name == ?out.
    ?Concept um:attribute ?A
        um:name "suitability"
        um:value ?S. )
{

```

Code Fragment 40: Scope declaration of a for-loop that has the value of the attribute 'suitability' of each of the sockets' elements as its object of manipulation.

Another example of a scope declaration for a GAL specification can be found in Code Fragment 41. It is the most complicated scope declaration that we encountered in authoring the CRTs for the Milky Way example application, and it works as follows: first, we bind the elements from the socket to variable "?c". Then all links that refer to the Unit which belongs to "?c", that is the Unit which has the name of concept "?c" postfixed with "_Unit", are selected and bound to "?L". The selection of the object of manipulation is intermingled in this case with the binding of the concept of manipulation to the socket element variable, because the fact that a naming convention is used for Units is immediately used here in the selection of the object of manipulation. The query that belongs to the link that is currently under consideration is always bound to variable "?q", such that in the body of the for-loop one can refer to that query.

Note, that the loop iterates for *each element* of the socket (i.e. concept) over *all* hyperlinks that have the Unit of the current concept as their target.

```

FOR ALL (
    Select ?L
    Where ?c IN $DM.
        ?Unit gal:hasLink ?L
            gal:refersTo ?c."_Unit"
            gal:hasQuery ?q. )
{

```

Code Fragment 41: Scope declaration of a for-loop that has all links that have a given concept as their target as its object of manipulation.

After the for-loop's scope declaration in some cases temporary variables may be declared. Temporary variables may be needed if the CRT for example has two sockets for which for each element of the one socket some aggregation must be made over some property of all elements of the other socket. After the declaration of temporary variables another for-loop may follow, which is again often the case if there are two sockets involved in the CRT. As far as we experienced in practice the nesting of for-loops goes as deep as the number of sockets involved.

At some point in the 'code' element, there is indeed a code fragment that must be added to the specification. In most cases the code fragment must indeed be added, and the tag that indicates this is <ADD>. In some cases, however, the Engine Independent Compiler may have generated some default code in a previous stage already, and then the code fragment must *replace* the existing code. In this case a <REPLACE> tag is used. An example of a situation in which this may be necessary is when one CRT has generated a 'suitability' attribute with the default value 'true' for a concept, and later on the Prerequisite CRT needs to specify an expression that expresses the concept's 'suitability' attribute as an invariant.

The <ADD_RULE> tag which can be found in the code of the Knowledge Propagation CRT is a temporary tag, because it is not clear yet to which exact part of the UM Specification the Rules must be added. When this is decided upon, however, the <ADD_RULE> tag can be changed in a normal <ADD> tag, and the CRT's for-loop should be adjusted to return the correct object of manipulation.

Finally, it may be the case that a CRT modifies more than one specification. Consider for example Explicit Update CRTs: CRTs of this class always add GAL code to the GAL specification and they may require the addition of a UM attribute to the UM specification if the UM attribute that they update is not present for the concepts of consideration yet. The 'code' element contains in this case multiple parts, each of which is tagged with the appropriate 'specification' tag.

5.2.1 Auxiliary CRTs in the Engine Independent Application Specification

Finally, the Auxiliary CRTs should be represented in the Engine Independent Application Specification, it is not entirely clear yet, however, how this should be done. The reason that this is necessary, is that for example directly forwarded information (see shortcut on the left in Figure 9) may indeed use this information, or some

Adaptation Engines (like for example the GALE AE) may have some hardcoded references to one of the CRTs¹⁴.

One option is to extend the concept elements in the DM specification with predicates in the same way as is done for concept relationships. The disadvantages of this, however, are first of all that the concept relationships from the DM can then no longer be distinguished from the predicates that were generated because of an Auxiliary CRT, and secondly that this approach automatically forbids the author to use names for concept relationships that are already in use by Auxiliary CRTs.

A solution to the issues that are raised by adding the predicates that are generated from the Auxiliary CRTs to the DM specification, is to add them to the UM specification. Indeed this raises no practical issues. The only objection to this approach, however, is that the UM specification ought to define structures for the storage of user-dependent information only, and this is violated by adding these predicates.

Since at the time of writing there is not enough clarity with respect to the expression of the Auxiliary CRTs in the Engine Independent Application Specification yet, for now we do not make any assumptions about this. Rather, we consider the issue to be noted now, and do not go into any further detail.

¹⁴ Of course, in fact this should not be the case, but the GALE AE is already an example of an engine that does it anyway. The hard-coded names of auxiliary structures that it uses are 'parent' and 'extends'.

6 GALE

In this chapter we describe the GRAPPLE Adaptive Learning Environment (GALE) and one of the the input formats of the GALE Adaptation Engine: GDOM. Although GALE can handle any format for which it has a plugin,

6.1 Architecture of GALE

In this section we briefly describe the architecture of the GRAPPLE Adaptive Learning Environment focussing on the main components involved. The explanation is thus simplified and possibly less precise than in reality, but a more detailed explanation can be found in [D1.3].

Figure 22 shows the simplified architecture of the GALE: an Adaptation Engine, a Domain Model Service (DMS) and a User Model Service (UMS) centred around an Eventbus. As the names already indicate, indeed the UMS and DMS *serve* the adaptation engine, as a result of which they can be replaced by any other service with the correct interface¹⁵. Note, that the GALE may be in use by several users for several applications at the same time and that there may thus be many browsers communicating with the AE rather than only one.

The process works as follows: when a logged on user follows a hyperlink on a page, (s)he is in fact requesting access to a concept. How this concept will be represented in the browser must be decided by the Adaptation Engine based on the UM and the DM. The Adaptation Engine knows what to do based on adaptation rules; it checks conditions and decides how to proceed. When information that is needed for this process is not available, it sends a request for that information to the eventbus and waits for a reply. All content-related information about concepts (e.g. associated resources) is stored in the DM, and provided by the DMS. All user-related information about concepts (e.g. knowledge of prerequisites) is stored in the UM, and is provided by the UMS.

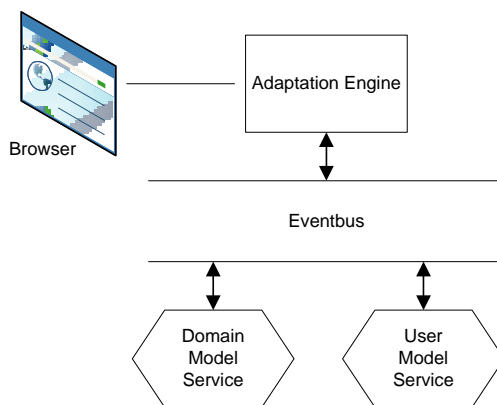


Figure 22: GALE architecture

6.1.1 Adaptation Engine

The Adaptation Engine is a much more complicated component than it seems to be in the first place. Consider for example the presentation: besides deciding to display an element, the browser must also be told *how* to do that. Since the application is adaptive, this is not a trivial issue.

To handle the complexity, the AE contains several managers and processors that each have their own specialisation (see Figure 23; the colours indicate the sequence in which the various managers and processors are invoked). A servlet called the GALEServlet is the AE's communication port to the user-side of the outside world, i.e. the browser. It receives requests (from the browser), decides how they must be handled, keeps track of all kinds of contextual information and finally sends the responses. We briefly describe this process here.

When the GALEServlet receives a request for a concept, it needs to know two things: which *concept* is requested by which *user*. In case the request is the first request since the start of the browser a new session is

¹⁵ meaning that the service should listen for and send the same messages as the service that it replaces

initiated, and the Login Manager is called to handle the authentication of the user [D1.3]. During the activities of the Login Manager, the UMS is asked for the UM which is then cached in the AE's UM cache.

The contextual information being available now, the GALEServlet continues handling the concept request. It now calls the Concept Manager, which will identify the concept that has been requested based on the http-request. Then, the Load Processor is asked to take care of selecting and fetching the resource that is appropriate for the concept based on the state of the UM; the exact process is detailed in [D1.2]. The selection of the appropriate resource is done by evaluation of the Java expression that resides in the 'resource' attribute of the concept in the DM. If the expression is a URI, the resource with this URI is simply requested to the UM cache. Otherwise, the expression is evaluated and the URI resulting from that is passed to the UM cache. Finally, the information related to the presentational aspects and processing of the concept is collected and passed to the GALEServlet [D1.3].

If the definition of the page structure is specified by a 'layout' attribute rather than in the 'resource' attribute, a similar process is delayed until the request is handled by the LayoutProcessor.

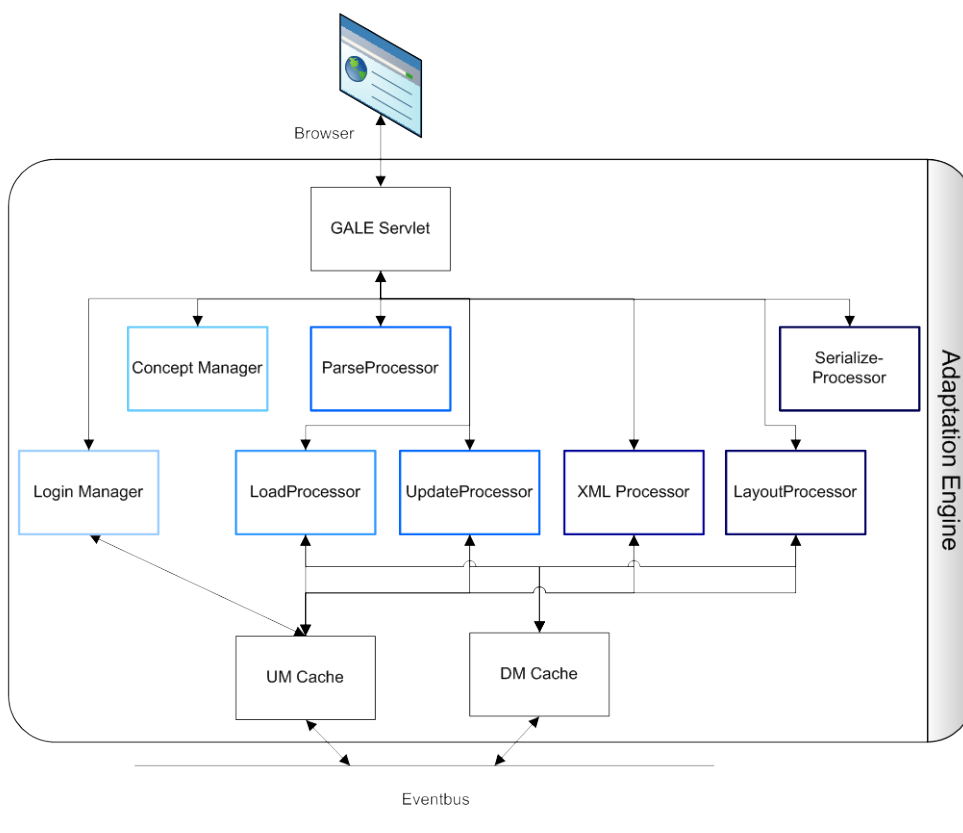


Figure 23: Architecture of the GALE Adaptation Engine; the colours indicate the sequence in which the various managers and processors are invoked

Then the GALEServlet invokes the ParseProcessor that unravels the resource that was received from the Concept Manager in the previous step. In fact the resource is cut into pieces, and each piece is then separately processed in the next step. At this point in the process, the GALEServlet also invokes the UpdateProcessor, which takes care of updating the UM by processing the 'event' code of –in this case- the GDOM file [D1.3] (see Figure 16, above the 'wait' line).

In this step of the process, the XML Processor processes all pieces of the resource that were generated in the previous step: it takes care of the *adaptation* (see Figure 16, under the 'wait' line). It thereby uses the newly updated version of the UM [D1.3].

If the adaptation is done, and the requested concept has a 'layout' attribute, the GALEServlet invokes the LayoutProcessor. This processor creates an in-memory specification of how the page that the browser has to

present will look [D1.3]. If the Layout Processor finds resources in the 'layout' attribute, like for example a template file, these resources are processed in the same way as has just been described.

When the Layout Processor is done, the GALEServlet finally invokes the SerializeProcessor. This processor puts all the pieces that the ParseProcessor created back together into one file. The GALEServlet finally sends this file to the browser [D1.3].

6.1.2 Eventbus

The Eventbus is a very important component in the GALE architecture; it could metaphorically be called the system's central nervous system. One of the strengths of the GALE framework is that components are relatively independent from each other, and this is enabled by the use of an eventbus.

The components in the GALE simply expect the outside world to understand their requests and replies, and therefore it does not matter how they are handled. A consequence of this is that other services can easily be added or used instead of the services that are standard provided.

The eventbus works as follows: the components that need or expect information from the eventbus have one or more EventListeners registered on it. When an event is sent into the eventbus, all EventListeners that listen for that event, take appropriate action. The sender of an event does not care who needs its information; it puts the information on the eventbus relying on the fact that whoever needs the information will listen for it [D7.1].

6.1.3 DM Service

The Domain Model Service facilitates the communication between the Domain Models and the "outside world" (usually the AE and the UMS). The existence of this component ensures that the other components do not need to bother where DM information comes from.

The DMS listens for 'getdm' and 'setdm' events; it sends the requested DM to the eventbus in case of the first event, and it modifies the DM in response on the second event. The 'setdm' event is by default followed by an 'updatedm' event from the DMS in order to notify all listeners that a certain DM has been changed [D7.1].

6.1.4 UM Service

The User Model Service facilitates the communication between the User Models and the "outside world". Similar to the DMS, the UMS ensures that components do not have to bother with where to get information.

User models can be split into two parts: the part that contains independent user information like his/her password and username, and a part that contains user information that is related to the concepts of a Domain Model.

The independent user information is referred to as 'entity information', and has associated events 'getentity' and 'setentity'. The UMS listens for both of these events. The 'setentity' event is, unlike the 'setdm' of the DMS and 'setum' of the UMS, not followed by an update event; the reason for this is that entity information is always requested to the UMS when needed while a cache is kept by the AE for DM information and UM dependent user information.

The dependent user information, which we will simply refer to as 'user information' and 'UM' in the remainder of this chapter, has five associated events. The 'getum', 'setum' and 'updateum' are symmetric to the events of the DMS. Furthermore the UMS listens for 'updatedm' events, such that it is informed when the expression that defines the value of non-persistent (thus calculated) UM attributes is updated, and it listens for 'queryum' events. Note that the latter event is never sent by an AE, since AEs simply use their UM cache or sent a 'getum' call if they need information about the UM. Rather a possible sender of 'queryum' would be a LMS.

An interesting aspect of the GALE is that concepts can be public, that is, they can be shared with other Domain Models. This can lead to the situation in which the value of a certain concept, say concept C, is determined by public concepts whose attribute values are not under control of the owner of C. To this end GRAPPLE has a component named the User Model Framework, or UMF for short. The UMF exists for the management of User Models and their interoperability [D6.1]. Since the details of this component are beyond the scope of this report, we refer to GRAPPLE deliverable 6.1a [D6.1] for details.

Whenever an attribute value of a concept C is dependent on another DM, there exists a mapping between the 'source concepts' that are needed for the calculation of C's attribute and C itself. This mapping is stored by the UMF, and all UM Caches know that in case such an attribute value is requested the information can be asked to the UMF (a more elaborate explanation on this mechanism follows in the next section).

6.2 Fetching an Attribute Value in GALE

If the Adaptation Engine needs an attribute value for a user (see Figure 24), it requests the User Model Service (UMS) to send it the value of the specified variable. Regardless of the content of the specified

variable, which may either be a value or an expression that results in a value, the UMS handles the whole process. When the UMS receives a request, it checks the UM for the presence of the requested attribute. If it is present, the value is read and interpreted as Java-code. Atomic values are immediately returned to the Adaptation Engine; if the variable value is a (legal) Java expression, this expression is processed. If the attribute is not found, this means that either the attribute has not been initialised yet or that the attribute value is not marked as persistent. To obtain a value after all, the UMS asks the DMS for the default value of the requested attribute, and sends this to the Adaptation Engine.

After the UMS has sent its response to the Adaptation Engine, it checks whether the attribute is persistent or not. If the 'persistent'-property is 'true' it saves the value to the. If, however, the 'persistent'-property is 'false', the attribute is deleted from the UM, and the next time it is requested the default value will have to be fetched from the DM again. Note, that although this construction may seem a waste of resources, it enables the author to let the attribute value depend on other attributes (possibly of other concepts).

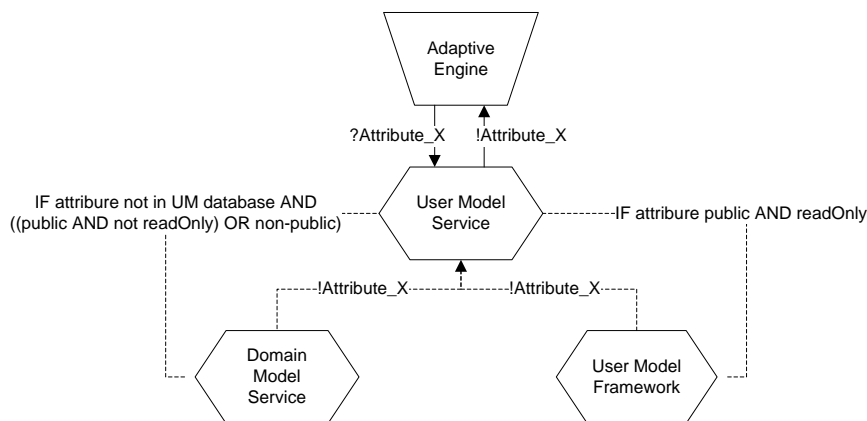


Figure 24: Retrieval of an attribute value

6.3 GDOM File

The file that serves as the application model –which is in this case the GDOM-file that is produced by the Engine Specific Compiler, but which may also be produced by e.g. an authoring tool that is tailored to GALE-, is the backbone of each adaptive application in GALE. It defines the concepts, the relationships between the concepts, it contains the information for the determination of which resources to select for a page, and it specifies UM updates. This section specifies how this information is expressed in the file and illustrates its use with an example

Until very recently, constructs that were characteristic for AHA! 4.0 could only be executed by directly inserting them into the database. Since this method is not feasible for larger-scale usage, the GDOM format was defined by David Smits. Below we define its XML Schema [W3XML][W3XML]:

```

<?xml version="1.0" encoding="utf-8" ?>
<xs:schema elementFormDefault="qualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="GDOM">
    <xs:complexType>
      <xs:element name="concept" type="conceptType" minOccurs="1"/>
      <xs:element name="relation" type="relationType"/>
    </xs:complexType>
  </xs:element>

  <xs:complexType name="conceptType">
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:element name="attribute" type="attributeType" minOccurs="1"/>
    <xs:element name="event" type="xs:string" maxOccurs="1"/>
    <xs:element name="property" type="propertyType" minOccurs="1"/>
  </xs:complexType>

  <xs:complexType name="attributeType">
    <xs:attribute name="name" type="xs:string" use="required"/>
  
```



```

        <xs:attribute name="type" type="java.lang.*" use="required"/>
        <xs:element name="default" type="xs:string" minOccurs="1"
maxOccurs="1"/>
        <xs:element name="property" type="propertyType"/>
        <xs:element name="event" type="xs:string" maxOccurs="1"/>
    </xs:complexType>

    <xs:complexType name="propertyType">
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="value" type="xs:string" use="required"/>
    </xs:complexType>

    <xs:complexType name="relationType">
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:element name="inconcept" type="xs:string" minOccurs="1"
maxOccurs="1"/>
        <xs:element name="outconcept" type="xs:string" minOccurs="1"
maxOccurs="1"/>
        <xs:element name="property" type="propertyType"/>
    </xs:complexType>

</xs:schema>

```

Code Fragment 42: XML Schema of the GDOM format

A GDOM-file consists of <concept>-elements and <relation>-elements only, which are both of the complex type. A <concept>-element, or briefly a concept, always has a name that uniquely identifies the concept. Furthermore it can have one to many <attribute>-children, zero or one <event>-child and zero to many <property>-children.

An <attribute>-element, or briefly an attribute, should always have a name and a type. The name of an attribute should be unique within the scope of its parent-element, that is, the concept. The type of the attribute can be any java.lang object. Furthermore, it contains one <default>-element which specifies the default value of the attribute; this can be a literal value (of the correct type) or a URI (if the attribute is of type 'string') or a Java expression that has this as a result. Finally attributes can also have <property>-elements and one <event>-element.

The <event>-element specifies all events that the access to its parent element should generate. The event code of a concept is thus executed when the concept is accessed (i.e. when the browser requests the concept to be displayed), and the event code of an attribute is executed when the attribute is accessed (i.e. when the AE requests the value of the attribute). Note, that attribute access does not imply concept access, since the former can take place while the concept is not requested for display.

The event code statements should be specified in Java code and be separated by semicolons (;).

A <property>-element, or briefly a property, can be child of a concept as well as of an attribute. Properties are author or system defined and can not be changed by end-user interaction with the application. The value of a property is therefore of the type 'string' by default. Properties should be uniquely identifiable by their names within the scope of their parent-element, that is, either the attribute or the concept to which they belong (it is thus perfectly legal to have a 'persistent' property for each attribute). All concepts that have to be displayed in the application have the (required) property 'type'. The type of a concept is typically 'page' or 'abstract'. The type of a concept is, among others, used to determine whether it is included for the count of the pages that have been read and for the construction of the treeview.

<relation>-Elements require the presence of a <inconcept>-child and a <outconcept>-child, whose names both refer to an existing concept. <relation>-elements are required to have a name, but the same name is allowed to be used for multiple elements. This allows the creation of graphs.

All <relation>-elements have arity one-to-one, thus requiring N-to-M concept relationships (N and M being positive integers) to result in N*M <relation>-elements.

The <relation>-element with name 'parent' is used by default for construction of the treeview, and the (optional) 'order' attribute that concepts can have is used to determine the (partial) order among siblings.

Finally, a <relation>-element can have properties in the same way as concepts can have (direct) properties, that is, <property>-elements would be direct children of the <relation>-element.

6.3.1 Layout Definition in GDOM

The layout of the entire browser window can be specified in several alternative ways; in this section we briefly describe some of the alternatives.

First of all, when a concept is to be displayed, it is always checked whether or not it has a 'layout' attribute associated with it. This attribute may either be a direct attribute of the concept, or it can be the attribute of

another concept with which the concept has an 'extends' relationship. We first consider the case in which no 'layout' attributes is found.

If there is no 'layout' attribute associated with the concept, the file that is specified in the concept's 'resource' attribute used to fill up the entire browser window. In this file the browser window may still be divided in views (recall Section 2.3.3), and files can be associated with those views. The advantage, however, is that since the concept of which the file describes the page structure is already known, it is no longer a problem if that file is tailored to that concept.

If there is neither a 'layout' attribute nor a 'resource' attribute for a concept, then the AE can not provide the browser with any information to display. For this case, there is an 'empty concept' file, in which the user is told that the requested concept is empty, that is then returned to the browser

If there is indeed a 'layout' attribute (see Code Fragment 43), there are still two options: the layout attribute may or may not include a <content/>-element. In a 'layout' attribute the browser window can be divided into rectangular areas that each can have their own <view/>-element, or the <content/>-element. A frequently used predefined view is the treeview element, which (calculates and) specifies the part of the concept hierarchy that should be visible for the concept that is visited. If the 'layout' attribute indeed contains a <content/>-element, the browser window is filled as specified by the attribute, and the area for the <content/>-element is filled according to the file of the concept's 'resource' attribute (or with the 'empty concept' file if the concept does not have a 'resource' attribute). If the attribute contains no <content/>-element, the browser window is filled only according to what the files that are associated with the views specify, i.e. the 'resource' attribute of the concept is never used at all in that case.

```
<attribute name="layout" type="java.lang.String">
  <default><![CDATA[
"<struct cols="\%20%;*\\">"+
"  <view name="\static-tree-view\"/>"+
"  <content/>"+
"/>struct">
  ]]>
  </default>
  <property name="persistent" value="false"/>
</attribute>
```

Code Fragment 43: 'layout' Attribute in GDOM; concepts can either have this attribute themselves or may inherit it from another concept

Note, that for the automated approach the latter alternative should be used, and for the template based approach one of the two alternatives that uses the 'resource' attribute of the concept should be used.

6.4 GDOM file of the Milky Way Example Application

In this section we will describe the GDOM-file of our example application. As with the Engine Independent Application Specification, the GDOM-file contains a lot of repetition and therefore we only present an example of each construction. The Guided Tour construction, however, is not described in GDOM, because there are many alternative approaches and the one that was implemented can not be easily derived from the approach that is taken in the Engine Independent Application Specification.

First, we go through the code of the concept 'Europa' (see Code Fragment 46): a moon of the planet Jupiter. The concept starts with a <concept/>-tag containing the name of the concept, which also functions as its identifier. The name is of the form *namespace/application_name/concept_name*, that is, 'gale://grapple-project.org/Milkyway/Europa'.

The first attribute of the concept is the 'resource' attribute; note, that we have selected the approach in which we define the content of the page's main view by associating the 'resource' attribute with a template file, without the use of a 'layout' attribute. The URI of, or the URL or path to the template file is specified in its <default/>-element. Note, that only content of the resource specified in the 'resource' attribute is processed by the engine and turned into a page; resources in other attributes will be neglected unless they are included by a resource that is specified in a 'resource' attribute, which may also be the 'resource' attribute of another concept. The value of the 'resource' attribute of Europa is "gale://Milkyway/xml/moon_instance_template.xhtml", in which "gale:" denotes the GALE home directory, "/Milkyway/xml/" is the path relative to the GALE home directory that leads to the directory that stores the templates for the Milky Way example application, and "moon_instance_template.xhtml" is the template that defines the page structure of this concept. Recall, that multiple concepts may use the same template. Finally, note that the attribute has a 'persistent' property which is set to 'false', which means that the attribute value is not stored in the UM, and that the default value is always retrieved from the DM. This ensures that if the author

of the application wants to change the template that is used for this concept, the AE will indeed use this new template on the next occasion that the user requests the concept.

The next attribute is 'visited', an attribute that originates from the UM specification. In this attribute, the number of times that the user has visited the concept is stored. The <event>-element, which is covered below, updates this attribute on each access. Since the value of this attribute must be preserved even across different sessions, the 'persistent' property is set to 'true'. The 'public' property is set to true and the 'readOnly' property is set to 'false', which means that indeed the Milky Way application is in charge of the attribute's value, but that other applications can use it as well.

The 'suitability' attribute of a concept, which is also originates from the UM specification, indicates whether the concept is suitable for the user. Since the value of this attribute was stored as an invariant in the UM specification, the AE must evaluate the content of the <default>-element, which is an expression, when it needs the attribute's value. The value of Europa's 'suitability' attribute is: " $(\{\text{Jupiter}\#\text{knowledge}\} \ \> \ = \ (50/1)) \ \&\& \ (\{\text{Moon}\#\text{knowledge}\} \ \> \ = \ (50/15))$ ", which is a translation of the invariant that was expressed by a UM Specification Rule Expression in the UM specification. Note, that the threshold value for the 'knowledge' of the concepts that are specified in this expression, 50, is divided by '1' for the knowledge of 'Jupiter' and by '15' for the knowledge of 'Moon'. The reason for this is that the Prerequisite CRT, which is the source of this expression, divides 50 by the number of children that the prerequisite concept has plus one (see Code Fragment 16); indeed Figure 10 shows that 'Jupiter' has no children and that 'Moon' has 14 children.

The 'persistent' property is indeed set to 'false' such that the AE calculates value of the attribute rather than storing it in the UM. The 'public' and 'readOnly' properties are set to 'false' for 'suitability'; the former because the value of the attribute is an expression, and other applications might not have access to the attributes used in that expression (thus causing problems), and the latter is set to 'false' since the AE is allowed to update the expression.

The last attribute that originates from the UM specification is the 'knowledge' attribute. Since Europa's parent, 'Moon', has the Knowledge Propagation CRT associated with its 'knowledge' attribute, the 'knowledge' attribute of Europa contains event code. This event code adds "changed.diff/15" to the knowledge of Moon: the amount of knowledge with which Europa is updated divided by the number of children of its parent plus one. The attribute is persistent, public and not read only for the AE; the former property is 'true' because the knowledge that the user possesses about the concept has to be maintained across sessions.

The three attributes that then follow, 'title', 'image' and 'description' have all three been derived from the GAL specification. For these attributes, it simply holds that if the according GAL attribute had a 'hasCondition'-construction that expressed resource selection, or in case of the 'title' attribute if it expressed a choice between the 'title' property of the concept and another string, as its value, then the translated equivalent of that query is the value of the attribute. If the GAL attribute specified a (query to a) direct value, as is the case in Appendix A, then the attribute value is simply a reference to the concept's property that contains the resource's URI, or the string value in case of the 'title' attribute. The GAL code in Code Fragment 44 would thus result in the GDOM fragment of Code Fragment 45.

The 'persistent' property of the attributes is 'false' because the AE must interpret the value of the attributes; the 'public' and 'readOnly' properties are omitted because the attributes are not UM attributes.

Finally the 'image' and 'description' attributes both have a 'label' property, which enables the author of the template to refer to 'the label of the <...> attribute'. There is one disadvantage however, since the labels are properties of the resource *attribute* in GDOM, rather than labels of the resource itself, it is not possible to conditionally return the correct label in the same way as the resource attribute conditionally returns the correct resource. The only way to solve this issue with the current means, is to store labels in their own attributes, however this would eliminate the possibility to refer to 'the label of a resource', and would thus require the author to instead refer to a separate label-attribute (which is thus no longer related to the resource attribute). Only if GDOM would allow attributes to contain <attribute>-elements themselves, this issue can elegantly be solved.

```
gal:hasAttribute [
  gal:name description;
  gal:hasCondition [
    gal:if [ Select ?A
      Where $Concept um:attribute ?A
        um:name "visited";
        um:value ?V
      Filter ?V > 1 ];
    gal:then [
      gal:label $Concept.dm:text_label;
      gal:value $Concept.dm:text;
    ];
  ];
  gal:else [
```



```

<attribute name="description" type="java.lang.String">
  <default>${?text}</default>
  <property name="persistent" value="false"/>
  <property name="label" value="Information: "/>
</attribute>
<event>
  #{#visited,#{#visited}+1};
  if (${#suitability}) {#{#knowledge,(100/1)};};
  else {#{#knowledge,(35/1)};};
</event>
<property name="type" value="page"/>
<property name="title" value="Europa"/>
<property name="text" value="gale://milkyway/resource/europa_text.xhtml"/>
<property name="image" value="gale://milkyway/img/img_europa.jpg"/>
</concept>

```

Code Fragment 46: Concept 'Europa' in GDOM

After the definitions of the concepts, the concept relationships are specified (see Code Fragment 47). This is done with the <relation>-elements that were described in the above. The most important relationship for an application is the 'parent' relationship, which is used to specify the concept hierarchy. This relationship must be derived from the Engine Independent Application Specification, however, at the time of writing it is not exactly clear yet how the 'parent' relationship is to be expressed (in the Engine Independent Application Specification). The semantic relationships are derived from the DM specification: the concept that contains the name of the relationship (the subject) is the value of the <inconcept>-element, the relationship name itself (the predicate) is the value of the <relation>-element's 'name' attribute, and the object of the triple is the value of the <outconcept>-element.

```

<relation name="isMoonOf">
  <inconcept>gale://grapple-project.org/Milkyway/Europa</inconcept>
  <outconcept>gale://grapple-project.org/Milkyway/Jupiter</outconcept>
</relation>
...
<relation name="parent">
  <inconcept>gale://grapple-project.org/Milkyway/Europa</inconcept>
  <outconcept>gale://grapple-project.org/Milkyway/Moon</outconcept>
</relation>

```

Code Fragment 47: Concept relationship definitions in GDOM

- Formatted: Dutch
- Field Code Changed
- Formatted: Dutch (Netherlands)

7 Compiling GAL into GALE Engine Format

The GALE engine can handle several input formats by the virtue of plug-ins. The input format that was originally developed for GALE is GDOM. In this chapter we explain how the Engine Independent Application Specification can be translated into GDOM. First we describe how a GDOM file is created and then we briefly reflect on the resources that specify the page structure of the concepts' pages.

Since the GAT and GALE are both based on AHA 4.0, the situation after the engine specific compilation (see Figure 9) is quite similar to the situation before the engine independent compilation. In GALE we have a DM and a UM for adaptive applications, which means that the DM specification and UM specification only have to be translated for usage. The files that are used for the page structure specification by the layout concepts are almost literally used as well, but these were first incorporated into the GAL specification. This means that the GAL specification is the only one of the three parts of the Engine Independent Application Specification that has to be unravelled.

7.1 GDOM File

The GDOM file of an application is mostly based on the DM and UM specifications; only the resource attributes and Unit event code is derived from the GAL specification. In this section, we first describe how the DM specification is represented in the GDOM file, then the UM specification and finally the GAL specification.

7.1.1 DM Specification

The skeleton of the GDOM file is created by adding a <concept>-element to the file for each concept in the DM specification. These elements in the DM specification are recognised by the "rdf:type dm:concept" statement that they contain. The name of the new GDOM concept is the same as the name of the DM specification concept. Then, the 'title' element of the concept, which is marked by the predicate "dm:title", is stored in a property named 'title'. Furthermore, for all 'resource' elements of the concept a property is created: the name and value of this property are equal to the name and value of the 'resource' element. The 'label' of the 'resource' element is used when the GDOM resource attribute is created (based on the GAL attribute to which the resource belongs). Similarly as for the DM specification's 'resource' elements, GDOM properties are also created for the 'fact' elements, which are marked by the predicate "dm:fact". Finally, a property with name 'type' and value 'page' is added to the GDOM concept.

The elements that then remain in the DM specification, are the predicates that represent concept relationships; for each concept relationship predicate a 'relationship' element is created in the GDOM file. The name of the relationship is equal to the predicate from the DM specification, except for the "mw:" prefix, which is stripped. The 'inconcept' of the newly created element is the concept that is the predicate resides in, and the 'outconcept' is the object that belongs to the predicate.

Code Fragment 48 and Code Fragment 49 illustrate how the GDOM file is derived from the DM specification.

```
gale://grapple-project.org/milkyway/etacarinae [
  rdf:type dm:concept;
  dm:title "Etacarinae";
  dm:resource [
    dm:name "image";
    dm:value gale://grapple-project.org/milkyway/img/img_etacarinae.jpg;
    dm:label "Image of: ";
  ];
  dm:resource [
    dm:name "info";
    dm:value gale://grapple-project.org/milkyway/resource/etacarinae_text.jpg;
    dm:label "Information: ";
  ];
  mw:isa gale://grapple-project.org/milkyway/star;
  mw:belongsTo gale://grapple-project.org/milkyway/milkyway;
];
```

Code Fragment 48: Concept of the DM specification

```
<concept name="gale://grapple-project.org/milkyway/etacarinae">
  <property name="title" value="Eta Carinae"/>
  <property name="text" value="gale://grapple-
```

```

project.org/milkyway/resource/etacarinae_text.xhtml"/>
  <property name="image" value="gale://grapple-
project.org/milkyway/img/img_etacarinae.jpg"/>
  <property name="type" value="page"/>
</concept>

<relation name="isa">
  <inconcept>gale://grapple-project.org/milkyway/etacarinae </inconcept>
  <outconcept>gale://grapple-project.org/milkyway/star</outconcept>
</relation>

<relation name="belongsTo">
  <inconcept>gale://grapple-project.org/milkyway/etacarinae</inconcept>
  <outconcept>gale://grapple-project.org/milkyway/milkyway</outconcept>
</relation>

```

Code Fragment 49: GDOM code that is generated based on the code in Code Fragment 48

7.1.2 UM Specification

The UM attributes of each concept, indicated by the 'um:attribute'-constructions (see Code Fragment 50), are added as 'attribute' elements to the GDOM concepts. The attributes from the UM specification contain three elements that require a 'special treatment'; the other elements can simply be translated into properties in the same way as was done for the properties that were derived from the DM specification. The first exceptional element is the 'um:name': the value of this element is used as the value of the GDOM attribute's 'name' attribute. Similarly, the 'dt:type' is used for the 'type' attribute of the GDOM attribute. Finally the 'um:value' is the content of the attribute's required 'default' element.

Note, that all UM Specification Rule Expressions must be parsed before they can be inserted in the GDOM file:

- All attribute types must be changed into "java.lang" classes,
- '>' and '<' must be changed into '>' and '<' respectively,
- `concept_name#attribute_name` must be changed into `#{concept_name#attribute_name}`,
- `concept_name?label_name` must be changed into `#{concept_name?label_name}`, and
- '&' must be changed into "&"

Code Fragment 50 and Code Fragment 51 illustrate how the GDOM file is derived from the UM specification. Note, however that the grayed out text in Code Fragment 50 exemplifies how the Auxiliary CRTs *might* be stored in the UM specification, but this code is further neglected here. If we would like to translate it into GDOM anyway, however, we could remove the "um:" prefix and create GDOM 'relationship' elements from them in the same manner as was done for the concept relationship predicates of the concepts in the DM specification. Furthermore, the grayed out event code in Code Fragment 51 is code that would have been added by the translation of the value of the UM Specification Rule of the concept. Note, that currently Invariant CRTs are indiscriminable from Recursive CRTs, apart from the fact that for the former the shorthand notation is used. This is problematic, since Recursive Update CRTs should result in event code for the attribute that causes the implicit update, and for Invariant CRTs this does not have to be done. Furthermore, note that the 'changed' attribute of the UM Specification does not have to be translated into GDOM, since the GALE engine has its own, hardcoded variable 'changed.diff' which fulfils the same role as the 'changed' attribute¹⁶. We can thus simply add the Knowledge Propagation's UM Specification Rule as event code to the 'knowledge' attribute, and we can neglect the updates to the 'changed' attribute. Note, however, that the Rule specifies propagation based on updates to the 'changed' attribute, and that the implementation in GALE propagates based on the updates to the 'knowledge' attribute. Finally, note that this implementation is facilitated by the GALE Adaptation Engine in particular; and there is thus no guaranty at all that any other Adaptation Engine can deal with knowledge propagation in this way. The 'changed' attribute is thus necessary.

```

gale://grapple-project.org/milkyway/etacarinae [
  um:attribute [
    um:name "visited";

```

¹⁶ The GALE Adaptation Engine has a variable named 'change.diff' for each numerical attribute. The variable contains the value with which the attribute is updated when it is requested.

```

    dt:type Integer;
    um:value "0";
    um:persistent "true";
    um:public "true";
    um:readOnly "false";
  ];
um:attribute [
  um:name "suitability";
  dt:type Boolean;
  um:value "gale://grapple-project.org/milkyway/star#knowledge >= 50/(2+1)";
  um:persistent "false";
  um:public "false";
  um:readOnly "false";
];
um:attribute [
  um:name "knowledge";
  dt:type Real;
  um:value "0";
  um:persistent "true";
  um:public "true";
  um:readOnly "false";
];
um:attribute [
  um:name "changed";
  dt:type Real;
  um:value "0";
  um:persistent "true";
  um:public "false";
  um:readOnly "false";
];
um:hasParent gale://grapple-project.org/milkyway/star;
um:hasSemanticParent gale://grapple-project.org/milkyway/milkyway;
];

```

Code Fragment 50: Concept of the UM specification

```

<attribute name="visited" type="java.lang.Integer">
  <default>0</default>
  <property name="persistent" value="true"/>
  <property name="public" value="true"/>
  <property name="readOnly" value="false"/>
</attribute>
<attribute name="suitability" type="java.lang.Boolean">
  <default>
    ${gale://grapple-project.org/milkyway/star#knowledge} &gt;= (50/3)
  </default>
  <property name="persistent" value="false"/>
  <property name="public" value="false"/>
  <property name="readOnly" value="false"/>
</attribute>
<attribute name="knowledge" type="java.lang.Real">
  <default>0</default>
  <property name="persistent" value="true"/>
  <property name="public" value="true"/>
  <property name="readOnly" value="false"/>
  <event>#{gale://grapple-project.org/milkyway/star#knowledge, ${gale://grapple-
project.org/milkyway/star#knowledge}+(changed.diff/3)};
  </event>
</attribute>

```

Code Fragment 51: GDOM code that is generated based on the code in Code Fragment 50

7.1.3 GAL Specification

To the GDOM file that has so far been built by the translation of the DM specification and UM specification, the event-code and the resource attributes that have to be added are based on the GAL specification. In this section we consider the GAL constructions which are translated into elements of the GDOM file.

First of all, the 'gal:onAccess'-construction can be translated into the GDOM 'event' element. Note, that because GAL can only express 'gal:onAccess'-constructions that are connected to a GAL Unit (and to 'gal:hasLink'-constructions, but these are not represented in the GDOM file), this construction can only be translated to 'event' elements at the concept level. Since the GDOM file allows only one 'event' element per concept, while GAL allows multiple 'gal:onAccess'-constructions per Unit, only the first 'gal:onAccess'-construction may create an 'event' element; the other ones may only add their content to this element.

The only (valid) content of a 'gal:onAccess'-construction is a 'gal:updateQuery', possibly wrapped with a 'gal:hasCondition'-construction. The 'gal:updateQuery'-construction can also be used in the files that are associated with the layout concepts, but we cover it only once.

The gal:updateQuery- construction can consistently be translated into GDOM with the construction: `#{target, updateQuery};` (note the semicolon after the closing curly bracket).

GAL expressions, which usually follow the structure: `$ConceptName.Attribute := f($ConceptName.Attribute)`, function *f* being some arithmetic expression that possibly includes (other) attributes. This is translated into GDOM as follows: `#{ConceptName#Attribute, f($ConceptName#Attribute)};`.

So the following example: `gal:updateQuery[Jupiter.Visited := Jupiter.Visited + 1];` is translated into GDOM as follows: `#{Jupiter#Visited,#{Jupiter#Visited}+1};`.

If the expression includes attributes of the concept in which's scope it occurs, the concept name may be omitted, changing the example's translation into: `#{#visited,#{#visited}+1};` (for the Jupiter concept).

When a 'gal:updateQuery'-construction occurs within a 'gal:hasCondition'-construction, this construction is almost literally translated, since 'gal:hasCondition [gal:if []; gal:then []; gal:else [];];' is simply changed into 'if() {} else{};'.
The GAL attributes are not so straightforwardly translated as the other two GAL constructs that we have covered so far. The reason for this is that only *resource* attributes are to be represented in the GDOM file. The Engine Specific Compiler can thus not simply translate all 'gal:hasAttribute'-constructions into GDOM attributes. With what we have so far learnt from the GAL specification and GDOM files that we have written for the Milky Way example application, the presence of the phrase "dm:resource" seems to be enough to distinguish a resource attribute from the other GAL attributes. The title of the concept, which is the only other GAL attribute that should be represented in the GDOM file, can be recognised by its name 'title'.

The GAL resource attributes are translated into GDOM depending the structure of the construction: resource attributes that involve resource selection have that expression (translated) as the value of the GDOM 'default' element, resource attributes that directly refer to a resource are translated somewhat differently. In both cases, however, the name of the GAL attribute is used as the name of the GDOM attribute, and the GDOM attribute type is always "java.lang.String". Furthermore, always a 'persistent' property with value 'false' is added, to indicate that the value of the attribute is not a string that is to be literally printed by the AE, but that the attribute value is to be resolved. The value of the 'default' element and the 'label' property indeed differ depending on whether or not resource selection is involved.

If there is no resource selection involved, the structure of the GAL attribute as presented in Code Fragment 52, is translated as presented in Code Fragment 53. The query for the resource is translated into a statement that returns the value of the property that contains the resource's URI or URL. The value of the label is retrieved from the DM specification, because it is not possible to interpret the value of a property. The AE can thus only literally print the content of the property.

If the GAL attribute indeed specifies a query for resource selection, the structure of the attribute should resemble the code as presented in Code Fragment 54. To obtain the GDOM equivalent of this code, see Code Fragment 55, several steps must be taken: first of all, the condition of the resource selection query must be translated into its GDOM equivalent. Since the GAL query language is not final yet, we do not the translation of the GAL Query Language in this thesis. The translated condition is then used as the condition of a conditional expression, of which the 'true' and 'false' clauses contain a query to the value of the property with the name of the resource that is referred to in the 'gal:hasCondition'-construction's 'then'-clause and 'else'-clause respectively. The issue with the label that was raised in Section 6.4, is not elegantly solved: the label of the resource in the 'then'-clause is simply used as the label of the GDOM resource attribute.

```
gal:hasAttribute [
  gal:name "attributeName";
  gal:label <query for label "L1">;
  gal:value <query for resource with name "R1">;
];
```

Code Fragment 52: Pseudo GAL code of a resource attribute with no resource selection

```

<attribute name= <attributeName> type="java.lang.String">
  <default> ${?R1} </default>
  <property name="label" value="<label L1>"/>
  <property name="persistent" value="false"/>
</attribute>

```

Code Fragment 53: Representation of Code Fragment 52 in GDOM

```

gal:hasAttribute [
  gal:name "attributeName";
  gal:hasCondition [
    gal:if [ <condition> ];
    gal:then [
      gal:label <query for label "L1">;
      gal:value <query for resource with name "R1">;
    ];
    gal:else [
      gal:label <query for label "L2">;
      gal:value <query for resource with name "R2">;
    ];
  ];
];

```

Code Fragment 54: Pseudo GAL code of a resource attribute with resource selection

```

<attribute name= <attributeName> type="java.lang.String">
  <default> ( <condition in GDOM> ? ${?R1} : ${?R2} ) </default>
  <property name="label" value="<label L1>"/>
  <property name="persistent" value="false"/>
</attribute>

```

Code Fragment 55: Representation of Code Fragment 54 in GDOM

7.2 Page Structure Specifications in GDOM

In this section we briefly reflect on the resources that are used by the 'layout' attributes and/or 'resource' attributes in the GDOM file. Since the files that are generated from the GAL specification are nearly identical to the resources that were originally used to generate the GAL code from (the template files), we will not cover in detail how the reverse translation from GAL to GDOM should be done.

Since a GAL Unit represents the navigation of the page of the concept that it represents, basically all GAL constructions, except for the 'gal:onAccess'-construction that is connected to the GAL Unit, are represented in the files that are used to define the page structure. If for each GAL Unit a separate file is created, though, we completely neglect the capacity of GALE to handle template files and predefined page structures. This is, however, the logical result of translating the GAL specification into GALE.

Another option for the GALE is to neglect the GAL specification, if the files that are needed for the page structures are forwarded directly to the GALE Adaptation Engine. This is what the shortcut on the left in Figure 9 would represent. The GAL specification can then be used by other engines, like Hera, that can not handle the resources of the layout concepts.

Note, that we have observed that the future version of the format that would be used for the templates, is not completely compatible with the format that GALE currently requires (this is the format as it was described in Section 4.4.3.1).

8 Future Work and Conclusions

The goals of this thesis were to clarify and document the authoring process in GRAPPLE and the models and formats involved in that process, and to do the preparations for the implementation of the Engine Independent Compiler and the Engine Specific Compiler. Based on what we have learnt from the specification and implementation of the Milky Way example application, we have described the status quo of the GRAPPLE project with respect to the authoring process and have described how the different models and formats can be translated into each other. Several issues have been identified and solutions have been suggested, some of which have been adopted, but there still remain some open ends.

At the start of my graduation project, some of the authoring models were rather marginally documented, the GALE Adaptation Engine was still under construction and the Generic Adaptation Language was not entirely defined yet. During the specification of the Milky Way example application in each of the stages of the authoring process, many issues were therefore raised and needed to be resolved.

The implementation of two versions of the Milky Way example application, first in AHA! 4.0 and later in GALE¹⁷, first of all gave insight in the page structure constructions and types of behaviour that could be of use for the average author. Furthermore, it has led to the development of two new authoring approaches for page structures: the automated approach, which is currently still based on coded page elements, and the template based approach, which allows the author to write his/her own reusable templates. For these new authoring approaches, also a format that enables a quite straightforward way of querying the Domain Model (DM) was developed. Future work on this part of the project, would be the implementation of the automatic layout processor for the automated approach.

The expression of the Milky Way example application in the output of the authoring models revealed some deficiencies in the original format of the Concept RelationshipTypes (CRTs), which were indeed solved in the new format. Although a limited set of CRTs has been described in this new format, the Knowledge Propagation CRT is not final yet, a method still needs to be found to express Auxiliary CRTs and a CRT or specification mechanism for Resource Selection is still to be developed. Furthermore, the description of the authoring process forced us to consider the intermingling of domain dependent and domain independent information in the DM, and led to the identification of three types of DMs. An issue that has not been considered in this thesis, however, is how concepts from a Pseudo DM –which have a special role in the authoring process- should be distinguished from the other concepts in the Application DM. Finally, the (GALE AE based) format that we developed for the specification of templates has turned out to be not completely suitable for authoring at the level of the GRAPPLE Authoring Tool, and could thus still be improved.

The expression of the example application in the Generic Adaptation Language (GAL) has taught us about the many different ways to encode an application and the concessions that sometimes have to be made for the sake of translatability. In some respects GAL turned out to be a very powerful language, like for the description of elements that have to be represented on concepts' pages. In others, however, it proved to be in need for additional expressivity; User Model specification rules to express certain updates to the UM, and the possibility to associate update queries to the access to a specific hyperlink on a page, are examples of what has been added. Finally, the development of the GAL code for the CRTs and the implementation of their instantiated versions led to the insight that there exist four main different classes of CRTs, which all have their own reflection on the Engine Independent Application specification.

Summarising, the various implementations of the Milky Way example application has provided us with various new insights regarding the authoring of adaptation in the GRAPPLE project, and it has forced us to clarify the specifications of the authoring models, application specifications and the authoring process. The Milky Way example application has thus proven itself worth the effort of implementing it in so many different representations, but it has also raised many issues that are still to be solved.

¹⁷ So in total there were four versions: two of each type

References

- [Nie1993] Jakob Nielsen, 1993. *Usability Engineering*, Mountain View: Morgan Kaufmann. Chapter 5 Section 2 Speak the Users' Language.
- [W3XML] http://www.w3schools.com/schema/schema_complex_any.asp, accessed 16-03-2009
- [Bra2ID55] Paul De Bra, Adaptive Systems (2ID55), lecture slides week 13, fall 2008
- [Bra2ID65] Paul De Bra, Hypermedia Course (2ID65), <http://dyn188.win.tue.nl:8080/portal>
- [D1.1] Eva Ploum (0568594), *CAM to Adaptation Rule Translator (Specification)*. February 2009. GRAPPLE Deliverable 1.1a, Version 1.0
- [D1.2] Paul De Bra, David Smits, Kees van der Sluijs, Evgeny Knutov, *Resource Selector Specification*. February 2009. GRAPPLE Deliverable 1.2a Version 1.0
- [D1.3] Paul De Bra, David Smits, Evgeny Knutov, *"Stand-alone" Adaptive Learning Environment Specification*. July 2009. GRAPPLE Deliverable 1.3b Version 0.1
- [D3.2] Christina Steiner, Alexander Nussbaumer et al., 2009. *Definition of a concept relationship tool*. December 2008. GRAPPLE Deliverable 3.2a Version 1.0.
- [D3.3] Maurice Hendrix, Alexandra Cristea, Martin Harrigan, Vincent Wade, Frederic Kleinermann and Olga De Troyer, 2009. *Design of CAM*, January 2009. GRAPPLE Deliverable 3.3a Version 1.0.
- [D6.1] Fabian Abel et al., *Distributed user model platform and retrieval service*. February 2009. GRAPPLE Deliverable 6.1a Version 1.0.
- [D7.1] Lucia Oneto et al., *Initial specification of the operational infrastructure*. January 2009. GRAPPLE Deliverable 7.1a Version 1.0.
- [Bra2009] Paul De Bra et al., *Unifying Adaptive Learning Environments: authoring styles in the GRAPPLE project*. 2009, Eindhoven
- [Bru1996] Peter Brusilovsky. 1996. Methods and Techniques of Adaptive Hypermedia. *User Modeling and User-Adapted Interaction*, July 1996, Vol. 6, no. 2-3, pp. 87-129.
- [Höök1995] Kristina Höök et al. 1995. A Glass Box Approach to Adaptive Hypermedia In: P. Brusilovsky, A. Kobsa and J. Vassileva, eds. *Adaptive Hypertext and Hypermedia*. 1998. Dordrecht: Kluwer Academic Publishers, pp. 143-170.
- [Kap1993] Craig Kaplan et al. 1993. Adaptive Hypertext Navigation Based On User Goals and Context In: P. Brusilovsky, A. Kobsa and J. Vassileva, eds. *Adaptive Hypertext and Hypermedia*. 1998. Dordrecht: Kluwer Academic Publishers, pp. 45-69.
- [Vas1996] Julita Vassileva. 1996. A Task-Centered Approach for User Modeling in a Hypermedia Office Documentation System In: P. Brusilovsky, A. Kobsa and J. Vassileva, eds. *Adaptive Hypertext and Hypermedia*. 1998. Dordrecht: Kluwer Academic Publishers, pp. 209-247.
- [Bra1999] Paul De Bra, Geert-Jan Houben and Hongjing Wu, 1999. AHAM: A Dexter-based Reference Model for Adaptive Hypermedia, Proceedings of the ACM Conference on Hypertext and Hypermedia, pp. 147-156.
- [Ste2009] Christina Steiner, Kai Michael Höver and Martin Harrigan, 2009. *Adaptive learning environments: A requirements analysis*. March 2009. [Presentation] Dublin: First annual review of the GRAPPLE project
- [Wu2002] Hongjing Wu, 2002. *A Reference Architecture for Adaptive Hypermedia Applications*. November 2002. PhD Eindhoven University of Technology.
- [Sta2006] Stash, N., Cristea, A. and De Bra, P., 2006. Adaptation to Learning Styles in E-Learning: Approach Evaluation. In: *Proceedings of E-Learn 2006 Conference*, Honolulu, Hawaii, October 2006.
- [Vos2007] Gottfried Vossen and Stephan Hagemann, 2007. *Unleashing Web 2.0*. Morgan Kaufman, Chapter 6: The Semantic Web And Web 2.0
- [Ree2003] Reinout van Rees, 2003. *Clarity in the usage of the terms Ontology, Taxonomy and Classification*. 20th CIB W78 Conference on Information Technology in Construction, Waiheke, Auckland, New Zealand, April 2003, url: http://vanrees.org/research/papers/2003_cib.pdf
- [Bra2002] Paul De Bra, Ad Aerts and Brendan Rousseau, 2002. Concept Relationship Types for AHA!2.0. *Proceedings of the AACE ELearn2002 conference*. October 2002, pp. 1386-1389.

- [McD2009] Julie McDonald, and Jari Timonen, 2009. How could pedagogy make a choice between personalized learning systems? Workshop paper: *A³H*, 7th International Workshop on Authoring of Adaptive and Adaptable Hypermedia, 2009, url: http://www.dcs.warwick.ac.uk/~acristea/A3H/UM/9_Paper.pdf
- [Bru1998] Peter Brusilovsky, John Eklund and Elmar Schwarz, 1998. Web-based education for all: A tool for developing adaptive courseware. *Computer Networks and ISDN Systems* (Proceedings of Seventh International World Wide Web Conference, 14-18 April 1998), Vol. 30 (1-7), pp. 291-300.
- [Web1996] Peter Brusilovsky, Elmar Schwarz and Gehrard Weber, 1996. ELM-ART: An intelligent tutoring system on world wide web. *Intelligent Tutoring Systems* (Proceedings of the Third International Conference ITS, 12-14 April 1996), Vol. 1086/1996, pp. 261-269.
- [Mit1998] Antonija Mitrović, 1998. Experiences in Implementing Constraint-Based Modeling in SQL-Tutor. *Intelligent Tutoring Systems* (Proceedings of the Fourth International Conference ITS, 16-19 August 1998), Vol. 1452/1998, pp. 414-423.
- [Bra1998] Paul De Bra and Licia Calvi, 1998. AHA! An open Adaptive Hypermedia Architecture. *The New Review of Hypermedia and Multimedia*, Vol. 4, pp. 115-139, Taylor Graham Publishers, 1998.

Appendix A

```
gale://grapple-project.org/milkyway/sun_Unit[
  a gal:Unit;

  gal:hasAttribute [
    gal:name title;
    gal:value gale://grapple-project.org/milkyway/sun.dm:title;
  ];

  gal:hasAttribute [
    gal:value "Is ";
  ];

  gal:hasAttribute [
    gal:value [
      gal:hasQuery [Select ?T
                    Where gale://grapple-project.org/milkyway/sun $hasParent ?P
                        dm:title ?T
                    ];
      ];
    ];

  gal:hasLink [
    gal:refersTo [
      gal:hasQuery [Select ?P."_Unit"
                    Where gale://grapple-project.org/milkyway/sun $hasParent ?P
                    ];
      ];
    ];

  gal:hasQuery [Select ?P
                Where gale://grapple-project.org/milkyway/sun $hasParent ?P
                ];
  ];

  gal:hasCondition [
    gal:if [ Select ?S
            Where gale://grapple-project.org/milkyway/sun $hasParent ?P
                um:attribute ?A
                um:name "suitability";
                um:value ?S
            ];
    gal:then [
      gal:if [Select?V
              Where gale://grapple-project.org/milkyway/sun $hasParent ?P
                  um:attribute ?A
                  um:name "visited";
                  um:value ?V
              Filter ?V > 1
            ];
      gal:then [ gal:hasEmphasis "Medium" ];
      gal:else [ gal:hasEmphasis "High" ];
    ];
    gal:else [ gal:hasEmphasis "Low" ];
  ];
];

gal:hasAttribute [
  gal:value " of: ";
];

gal:hasAttribute [
  gal:value [ gal:hasQuery [
                Select ?T
                Where (gale://grapple-project.org/milkyway/sun
$hasSemanticParent ?P)
                    dm:title ?T
                ];
  ];
];
```

```

gal:hasLink [
  gal:refersTo [
    gal:hasQuery [
      Select ?P."_Unit"
      Where gale://grapple-project.org/milkyway/sun $hasSemanticParent
?P
    ];
    gal:hasQuery [
      Select ?P
      Where gale://grapple-project.org/milkyway/sun $hasSemanticParent
?P
    ];
  ];
];
gal:hasCondition [
  gal:if [
    Select ?S
    Where gale://grapple-project.org/milkyway/sun $hasSemanticParent ?P
      um:attribute ?A
      um:name "suitability";
      um:value ?S
  ];
  gal:then [
    gal:if [ Select ?V
      Where (gale://grapple-project.org/milkyway/sun
$hasSemanticParent ?P)
      um:attribute ?A
      um:name "visited";
      um:value ?V
      Filter ?V > 1
    ];
    gal:then [ gal:hasEmphasis "Medium" ];
    gal:else [ gal:hasEmphasis "High" ];
  ];
  gal:else [ gal:hasEmphasis "Low" ];
];
];
gal:hasAttribute [
  gal:name image;
  gal:label [
    gal:hasQuery [Select ?L
      Where gale://grapple-project.org/milkyway/sun dm:resource ?R
      dm:name "image";
      dm:label ?L.
    ];
  ];
  gal:value [
    gal:hasQuery [Select ?V
      Where gale://grapple-project.org/milkyway/sun dm:resource ?R
      dm:name "image";
      dm:value ?V.
    ];
  ];
];
gal:hasAttribute [
  gal:name description;
  gal:label [
    gal:hasQuery [Select ?L
      Where gale://grapple-project.org/milkyway/sun dm:resource ?R
      dm:name "text";
      dm:label ?L.
    ];
  ];
  gal:value [

```

```

gal:hasQuery [Select ?V
              Where gale://grapple-project.org/milkyway/sun dm:resource ?R
                    dm:name "text";
                    dm:label ?V.
              ];
];
];

gal:hasAttribute [
  gal:value "The following ";
];

gal:hasAttribute [
  gal:value [
    gal:hasQuery [
      Select ?T
      Where gale://grapple-project.org/milkyway/sun $hasParent ?P.
            ?Cousin $hasSemanticParent ?P;
            dm:title ?T.
    ];
  ];
];

gal:hasLink [
  gal:refersTo [
    gal:hasQuery [
      Select ?Cousin."_Unit"
      Where gale://grapple-project.org/milkyway/sun $hasParent ?P.
            ?Cousin $hasSemanticParent ?P.
    ];
  ];
];

gal:hasQuery [
  Select ?Cousin
  Where gale://grapple-project.org/milkyway/sun $hasParent ?P.
        ?Cousin $hasSemanticParent ?P.
];

gal:hasCondition [
  gal:if [
    Select ?S
    Where gale://grapple-project.org/milkyway/sun $hasParent ?P.
          ?Cousin $hasSemanticParent ?P;
          um:attribute ?A
          um:name "suitability";
          um:value ?S.
  ];
  gal:then [
    gal:if [
      Select ?V
      Where gale://grapple-project.org/milkyway/sun $hasParent ?P.
            ?Cousin $hasSemanticParent ?P;
            um:attribute ?A
            um:name "visited";
            um:value ?V.
      Filter ?V > 1
    ];
    gal:then [ gal:hasEmphasis "Medium" ];
    gal:else [ gal:hasEmphasis "High" ];
  ];
  gal:else [ gal:hasEmphasis "Low" ];
];

gal:hasAttribute [
  gal:value "(s) rotate around: ";
];
];

```



```

gal:hasAttribute [
  gal:value [
    gal:hasQuery [ gale://grapple-project.org/milkyway/sun.dm:title ];
  ];
];

gal:hasSetUnit [
  gal:refersTo body_Unit;
  gal:hasQuery [
    Select ?Child
    Where ?Child $hasSemanticParent gale://grapple-project.org/milkyway/sun
  ];

  body_Unit [
    a gal:subUnit
    gal:hasAttribute [
      gal:value [
        gal:hasQuery [
          Select ?Child.dm:title
          Where ?Child $hasSemanticParent gale://grapple-
project.org/milkyway/sun
        ];
      ]
      gal:hasLink [
        gal:refersTo [
          gal:hasQuery [
            Select ?Child."_Unit"
            Where ?Child $hasSemanticParent gale://grapple-
project.org/milkyway/sun
          ];
        ];
        gal:hasQuery [
          Select ?Child
          Where ?Child $hasSemanticParent gale://grapple-
project.org/milkyway/sun
        ];
      ];
      gal:hasCondition [
        gal:if [ Select ?S
          Where (?Child $hasSemanticParent gale://grapple-
project.org/milkyway/sun).
          ?Child um:attribute ?A
            um:name "suitability";
            um:value ?S.
        ];
        gal:then [
          gal:if [ Select ?V
            Where ?Child um:attribute ?A;
              um:name "visited";
              um:value ?V;
            Filter ?V > 1
          ];
          gal:then [ gal:hasEmphasis "Medium" ];
          gal:else [ gal:hasEmphasis "High" ];
        ];
        gal:else [ gal:hasEmphasis "Low" ];
      ];
    ];
  ];
];

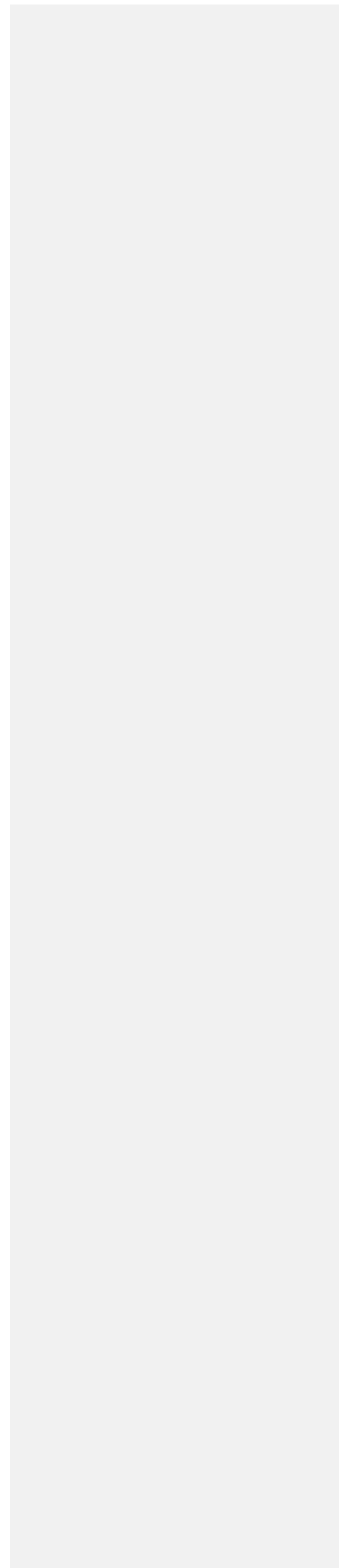
gal:hasSubUnit [
  gal:name Guided_Tour_Included_Unit;
];

gal:hasAttribute [

```



```
        um:name "visited";
        um:value ?V
    Filter ?V > 1
];
gal:then [ gal:hasEmphasis "Medium" ];
gal:else [ gal:hasEmphasis "High" ];
];
gal:else [ gal:hasEmphasis "Low" ];
];
];
];
```



Appendix B

```
Guided_Tour_Included_Unit [

  gal:hasCondition [
    gal:if [Select?A
      Where mw:guidedtour tour:attribute ?A
            tour:attribute_name "active";
            tour:value "true"
    ];
    gal:then [
      gal:if [ !($Input.dm:title == gale://grapple-
project.org/milkyway/triton.dm:title) ];
      gal:then [
        gal:hasAttribute [
          gal:name Next;
          gal:label "This link will guide you to the next concept of the Guided
Tour";
          gal:value [
            gal:hasValue "Next >>";
            gal:hasLink [
              gal:refersTo [
                gal:hasQuery [
                  Select ?V."_Unit";
                  Where mw:guidedtour tour:concept ?C
                        tour:attribute_name "next_concept";
                        tour:value ?V
                ];
              ];
            gal:hasQuery [
              Select ?C
              Where mw:guidedtour tour:attribute ?A
                    tour:attribute_name "next_concept";
                    tour:value ?C
            ];
          gal:onAccess [
            gal:updateQuery [
              Update ?V := ?Name;
              Where mw:guidedtour tour:concept ?C
                    tour:concept_name $Input
                    tour:position ?P.

              mw:guidedtour tour:concept ?C'
                tour:concept_name ?Name
                tour:position (?P)+1.

              mw:guidedtour tour:attribute ?A
                tour:attribute_name "next_concept";
                tour:value ?V.
            ];
          gal:updateQuery [
            Update ?V := ?Name;
            Where mw:guidedtour tour:concept ?C
                  tour:concept_name $Input
                  tour:position ?P.

            mw:guidedtour tour:concept ?C'
              tour:concept_name ?Name
              tour:position (?P)+1.

            mw:guidedtour tour:attribute ?A
              tour:attribute_name "current_concept";
              tour:value ?V.
          ];
        ];
      ];
    ];
  ];
];
```

```

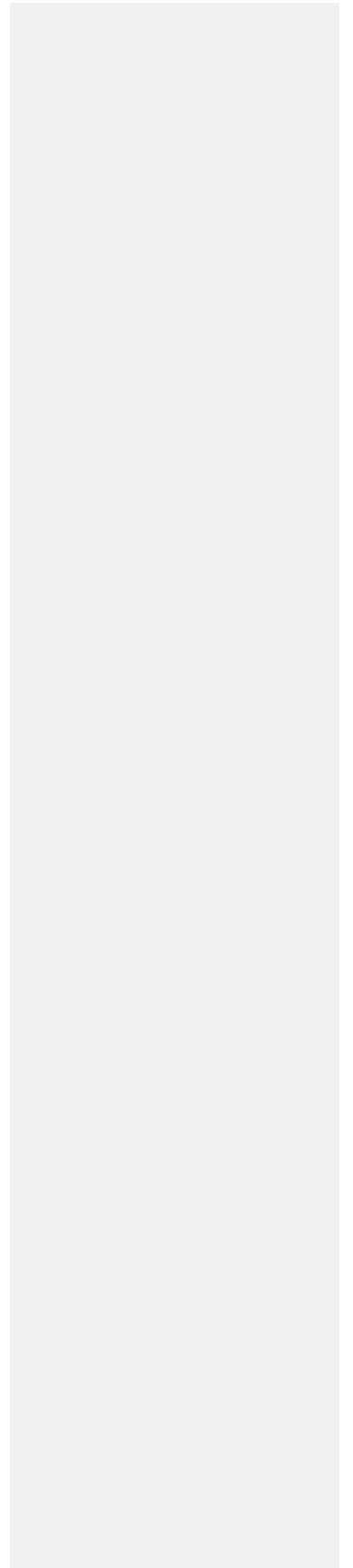
];
];
];
];
gal:else [
gal:hasAttribute [
gal:name RestartTour;
gal:label "This link will restart the Guided Tour";
gal:value [
gal:hasValue "Restart the Tour>>";
gal:hasLink [
gal:refersTo [
gal:hasQuery [
Select ?N."_Unit";
Where mw:guidedtour tour:concept ?C
tour:position "1";
tour:concept_name ?N
];
];
gal:hasQuery [
Select ?N
Where mw:guidedtour tour:concept ?C
tour:position "1";
tour:concept_name ?N
];
gal:onAccess [
gal:updateQuery [
Update ?V := ?N;
Where mw:guidedtour tour:concept ?C
tour:concept_name ?N
tour:position "2".

mw:guidedtour tour:attribute ?A
tour:attribute_name "next_concept";
tour:value ?V.
];
gal:updateQuery [
Update ?V := ?N;
Where mw:guidedtour tour:concept ?C
tour:concept_name ?N
tour:position "1".

mw:guidedtour tour:attribute ?A
tour:attribute_name "current_concept";
tour:value ?V.
];
];
];
];
];
];
gal:hasAttribute [
gal:name PauseTour;
gal:label "This link will pause the Guided Tour";
gal:value [
gal:hasValue "Pause Tour>>";
gal:hasLink [
gal:refersTo [
gal:hasQuery [$Input."_Unit"; ];
];
gal:hasQuery [ $Input ];
gal:onAccess [
gal:updateQuery [
Update ?Active := false;
Where mw:guidedtour tour:attribute ?A
tour:attribute_name "active";

```


17



Appendix C

Header:

```
<span xmlns:gale="http://gale.tue.nl/adaptation">
<gale:object data="../header.xhtml"/>
</span>
```

Title:

```
<span xmlns:gale="http://gale.tue.nl/adaptation">
<h1><gale:variable expr="{?title}"/></h1>
</span>
```

Parent tagline*:

```
<span xmlns:gale="http://gale.tue.nl/adaptation">
<h4>Is <gale:a><gale:variable expr="{-(~parent)#title}"/><gale:attr-variable
name="href" expr="{-(~parent)#title}"/></gale:a> of
<gale:a><gale:variable expr="{-(~semantic_parent)#title}"/><gale:attr-variable
name="href" expr="{-(~semantic_parent)#title}"/></gale:a></h4>
</span>
```

Image:

```
<span xmlns:gale="http://gale.tue.nl/adaptation">
<h2><gale:variable expr="{#image?label}"/> <u><gale:variable expr="{#title}"/>
</u></h2>
<br /><img><gale:attr-variable name="src" expr="{#image}"/></img>
</span>
```

Description:

```
<span xmlns:gale="http://gale.tue.nl/adaptation">
<br /><h2><gale:variable expr="{#info?label}"/></h2>
<gale:object><gale:attr-variable name="data" expr="{#info}" /></gale:object>
</span>
```

Related Concepts*:

```
<span xmlns:gale="http://gale.tue.nl/adaptation">
<gale:if expr="{<-(~semantic_parent)}.length > 0"><gale:block>
<br /><br /><b>The following < gale:a><gale:attr-variable name="href" expr="{-(~semantic_parent)<-(~parent)#title}"/><gale:variable
expr="{-(~semantic_parent)<-(~parent)#title}"/></gale:a> are related to
<gale:variable expr="{#title}"/></b>
<ul>
<gale:for var="concept" expr="{<-(~semantic_parent)}">
<li><gale:a><gale:variable expr="{%concept#title}"/>
<gale:attr-variable name="href" expr="&quot;%concept&quot;"/>
</gale:a></li>
</gale:for>
</ul>
</gale:block></gale:if>
</span>
```

Visited:

```
<span xmlns:gale="http://gale.tue.nl/adaptation">
<br /><b>Visits:</b> <gale:variable expr="{#visited}"/>
</span>
```

* 'parent' and 'semantic_parent' refer to the DM relationships which have the 'parent' and 'semantic parent' roles in the provided context

Guided Tour:

```
<span xmlns:gale="http://gale.tue.nl/adaptation">
<hr /><gale:if expr="{GuidedTour#active}">
```

```

<gale:block>
  <gale:if expr="{GuidedTour#i} &lt; {GuidedTour#last_concept}+1">
    <gale:block>
      <gale:if expr="{#GT}"> <gale:block>
        <gale:a exec="{GuidedTour#i, ({GuidedTour#i}+1)};">
          <gale:attr-variable name="href" expr="{GuidedTour#next}"/>
          <u><h4>Next >></h4></u>
        </gale:a>
      </gale:block>
      <gale:block>
        <gale:a>
          <gale:attr-variable name="href" expr="{GuidedTour#current}"/>
          <h4><u> Back to Tour >> </u></h4>
        </gale:a>
      </gale:block>
    </gale:if>
  </gale:block>
  <gale:block>
    <gale:a exec="{GuidedTour#i, 0};">
      <gale:attr-variable name="href" expr="{GuidedTour#next}"/>
      <h4><u> Restart Guided Tour >> </u></h4>
    </gale:a>
  </gale:block>
  </gale:if>

  <gale:a exec="{GuidedTour#active, false};#{GuidedTour#paused, true};">
    <gale:attr-variable name="href" expr="gale.concept().getTitle()"/>
    <h4><u> Pause Tour >></u></h4>
  </gale:a>
  <gale:a exec="{GuidedTour#active, false};#{GuidedTour#paused,
false};#{GuidedTour#i, 1};"> <gale:attr-variable name="href"
expr="gale.concept().getTitle()"/><h4><u> Exit Guided Tour >> </u></h4></gale:a>
</gale:block>
<gale:block>
  <gale:if expr="{GuidedTour#paused}">
    <gale:block>
      <gale:a exec="{GuidedTour#active, true}; #{GuidedTour#paused, false};">
        <gale:attr-variable name="href" expr="{GuidedTour#current}"/>
        <h4><u> Resume Tour >> </u></h4></gale:a>
      </gale:block>
      <gale:block>
        <gale:a exec="{GuidedTour#active, true};">
          <gale:attr-variable name="href" expr="{GuidedTour#current}"/>
          <h4><u> Start Guided Tour >> </u></h4></gale:a>
        </gale:block>
      </gale:if>
    <gale:if expr="!{GuidedTour#paused}">
      <gale:block>
        The Guided Tour will first teach you about stars, planets and moons and then
shows you the members of our solar sytem. It will be an exciting travel starting
from the Sun all the way to Neptune and his moon Triton. <br/>
      </gale:block>
    </gale:if>
  </gale:block>
</span>

```

Appendix D

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:gale="http://gale.tue.nl/adaptation">
<gale:object data="../header.xhtml"></gale:object>
<h1><gale:variable expr="#{#title}"/></h1>
<h4>Is <gale:a><gale:variable expr="{->(isa)#title}"/><gale:attr-variable name
="href" expr="{->(isa)#title}"/></gale:a> of:
<gale:a><gale:variable expr="{->(belongsTo)#title}"/><gale:attr-variable name
="href" expr="{->(belongsTo)#title}"/></gale:a></h4>
<h2><gale:variable expr="#{#image?label}"/> <u><gale:variable
expr="#{#title}"/></u></h2><br /><img><gale:attr-variable name="src"
expr="#{#image}"/></img>
<br /><h2><gale:variable
expr="#{#description?label}"/></h2><gale:object><gale:attr-variable name="data"
expr="#{#description}"/></gale:object><br />
<gale:if expr="{<-(isPlanetOf)}.length > 0"><gale:block>
<br/><b>The following <gale:a><gale:attr-variable name="href" expr="{<-(
isa)&lt;- (rotatesAround)#title}"/></gale:a> rotate around
<gale:variable expr="#{#title}"/></b>
<ul>
<gale:for var="concept" expr="{<-(isPlanetOf)}">
  <li><gale:a><gale:variable expr="{%concept#title}"/>
    <gale:attr-variable name="href" expr="&quot;%concept&quot;"/>
  </gale:a></li>
</gale:for>
</ul>
</gale:block>
<gale:block><br/></gale:block>
</gale:if>
<gale:object data="guided_tour.xhtml"></gale:object>
<br/>
<b>Visits:</b> <gale:variable expr="#{#visited}"/>
</html>
```