GRAPPLE

D1.1a Version: 1.0

CAM to Adaptation Rule Translator (Specification)

Document Type	Deliverable
Editor(s):	Kees van der Sluijs (TUE)
Author(s):	Kees van der Sluijs (TUE), Jan Hidders (TUD), Milos Kravcik (OUNL), Geert-Jan Houben (TUD), Eva Ploum (TUE)
Internal Reviewer(s):	Martin Harrigan (TCD), Alexandra Cristea (WARWICK)
Work Package:	1
Due Date:	1-2-2009
Version:	1.0
Version Date:	25-2-2009
Total number of pages:	37

Abstract: This deliverable specifies a translation from a Conceptual Adaptation Model to low-level adaptation rules for the GRAPPLE Adaptive Learning Environment. We do this by specifying a 'middle language', which is an abstract and engine independent language named GAL. GAL allows for the specification of an adaptive navigation structure in which the higher level authoring constructs can be translated to and which abstracts from the engine specific rules of the adaptive engines that are in existence.

Keyword list: GAL, Grapple Adaptation Language, Navigation Specification, CAM, rule translation

Summary

The goal of this deliverable is to look at the translation of the Conceptual Adaptation Model (CAM) instance that is created by the authors of adaptive content, to rules that can be applied by the GRAPPLE Adaptive Learning Environment (GALE). Instead of choosing to build a compiler that makes a one-to-one translation between the CAM and GALE, we built something more generically useful, the GRAPPLE Adaptation Language (GAL). GAL abstracts from the actual authoring models and the specific engine implementation by forming a generic language to define adaptive navigation. In this deliverable we first put things into perspective by looking at the requirements that the learning domain puts on adaptive navigation and then to the specific relationships between models in the GRAPPLE project. Then we will look at the GAL language, which we will introduce and exemplify construct by construct. Finally we will give a formalization of the GAL language which should simplify the process of building the compilers from CAM to GAL and from GAL to GALE specific rules.

Authors

Person	Email	Partner code
Kees van der Sluijs	k.a.m.sluijs@tue.nl	TUE
Jan Hidders	a.j.h.hidders@tudelft.nl	TUD
Milos Kravcik	Milos.Kravcik@ou.nl	OUNL
Geert-Jan Houben	g.j.p.m.houben@tudelft.nl	TUD

Table of Contents

Tables and Figures

List of Figures

List of Acronyms and Abbreviations

ALE	Adaptive Learning Environment
CAM	Conceptual Adaptation Model
DM	Domain Model
GAL	GRAPPLE Adaptation Language
GRAPPLE	Generic Responsive Adaptive Personalized Learning Environment
LMS	Learning Management System
UM	User Model

1 Introduction

In this deliverable we look at the process of translating a Conceptual Adaptation Model (CAM) definition (as specified in WP3, D3.3a) of an application to engine rules that implement the application and its adaptive behaviour. In our case this would be the GRAPPLE Adaptive Learning Environment (GALE) rules (as specified in D1.3a). In other words, we will look at how we can translate the definition of the adaptive application as specified by the author into rules that can be executed by the running engine. This last definition is a bit more general than the first as we abstract from the concrete CAM and GALE. This is because there exist several adaptive engines (and versions of adaptive engines) that allow a personalized learning experienced. Also, several LMSs have begun to partially implement adaptive behaviour. When authoring tools, of which several exist as well, define a course, eventually this course needs to be translated to something that can be executed by an adaptive engine. Therefore, instead of making only a one-to-one translation between the CAM and GALE of the GRAPPLE project we aim to be more generic by specifying an intermediate language that can be used as the basis for translation between any authoring environment and any adaptive engine.

We define this generic language such that it captures the basic goal behind every adaptation engine and every authoring environment, namely defining a *navigation structure* that can be *adapted* and *personalized* for its users. We call the language GAL (GRAPPLE Adaptation Language). In this way we will get three levels of models that try to capture different things. Authoring languages (like the D3.3a CAM model, but authoring based adaptation languages like IMS LD [1] and LAG [2]) specify the application from an authoring perspective. This includes various viewpoints like the pedagogical view, but also the Virtual Reality (VR) and Simulation (refer to D3.4 and D3.5a) perspective. These models (partly) specify and restrict the adaptive navigation structure of the application. This adaptive navigation structure can then be explicitly modelled in GAL, which defines the adaptive navigation structure. As such it is closes to the engine that also eventually specifies the adaptive navigation structure, however GAL covers the adaptive navigation in an abstract readable way, so that it abstracts from engine specific constructs and presentation specific code.

We proceed with the following steps:

1. In section 2 we first give an overview of the adaptation requirements of the educational domain. We also look more specifically at the GRAPPLE project to see which models play a role in the project and what relationships exist between them. This section is useful to understand the structure and the connection of the GRAPPLE authoring environment with GALE. It is also useful to recognize where GAL fits in the picture.
2. Then in section 3 we step by step look at the constructs we need in our GAL language. Every step is exemplified through executable code that shows what the construct would practically look like. In this way we cover the basic expressivity that is needed for our GAL language.
3. The informal approach of section 3 is then formalized in section 4. In this way we aim to get a complete formal description of the GAL language. This is important to avoid ambiguity and later on, it should simplify the process of making compilers for CAM-to-GAL and GAL-to-GALE models.

2 From Authoring Models to GAL

In this section we try to put GAL into perspective. We do this in two steps. Firstly, we look generally at the educational domain and see which requirements this domain puts on the authoring models. Secondly, we look more specifically at the architecture of the GRAPPLE project. We see which models play a role and what the information flow in the system is.

2.1 The Educational Domain

GAL is a language that specifies an adaptive navigation structure. To understand the kind of models that need to be translated to GAL we analyse the assumptions, models and layers of adaptation that are of importance in the educational domain (which is our main focus in this project).

2.1.1 Assumptions

Given that we have a special target of learning applications we look at some of the implicit assumptions we make when creating an adaptive learning framework.

- Personalized adaptive learning should support the decision process narrowing the wide spectrum of choices according to e.g. the learner's preferences and background.
- The learner has the main responsibility for her learning, i.e. she can control her preferences and has to take the ultimate decision in case of ambiguities.
- To reduce the cognitive overload and to improve efficiency the learner delegates certain decisions to the tutor and to the system she has chosen.

2.1.2 Models

Several types of knowledge are needed to provide personalized adaptive learning. Ideally, their representations should be separated from each other to support reusability and flexibility as much as possible. Therefore, we consider the models to be relatively autonomous. On the other hand, their interoperability should be achieved via commonly used standards and specifications.

Domain model: domain concepts (knowledge space) interconnected with related learning resources (hyperspace); assignment of learning resources to domain concepts can be static or dynamic

Learner / User model: relevant learner characteristics – competences (knowledge, skills, attitudes, qualification), preferences (learning goal, language, interests, cognitive and learning style); preferences might be context dependent

Context model: settings in which an event occurs – physical (geographic and time coordinates and constraints, delivery device, platform) and semantic context (role, chosen learning strategy, e.g. deep study vs. quick repetition)

Pedagogical model: educational knowledge – both procedural (structure and design of learning processes) and declarative (metadata – pedagogical roles, relationships)

Adaptation model: adaptation semantics – adaptation strategies (learning activity selection, content selection, problem solving support, collaboration support), adaptation techniques (adaptive sorting, hiding, annotation)

Presentation model: presentation specifications – how to present adaptation strategies and techniques as well as how objects with a particular status should be presented to the user in the current context

2.1.3 Layers of Adaptation

The adaptation need in learning systems can in general be separated in five different layers (e.g. based on Dexter[3] and AHAM[4]):

1. *Competence:* the estimated ability (disposition) of an actor to deal effectively and efficiently with critical events, problems or tasks that can occur in a certain situation (at work, at home, etc). This estimation can be based on self assessment, informal assessment by others, and formal assessment by others (including the system). Competences are used when specifying high level learning objectives.

   Aim: Select a set (structure) of target competences according to the learner's ambitions, needs, interests, and preferences. The selected set (structure) of target competences suggests an order in which they should be learned (trained).

2. *Learning Plan:* a structured setup to support the acquisition of a specific competence. There can be alternative plans (curricula) for development of a certain competence and the learner (tutor, system) can choose one that best matches the learner's requirements and preferences. A learning plan is assigned to a competence and contains learning activities.

   Aim: Select a learning plan for a target competence according to the learner's competences, preferences, and constraints. The selected learning plan specifies a structure of learning activities to be followed by the learner.

3. *Learning Activity:* an action the learner has to perform to meet certain educational objectives. It contains a description of the activity and a list of related learning resources. A learning activity is part of a learning plan. It contains (lower level) learning activities and learning resources. Learning activities are considered generally, i.e. not only computer mediated activities.

   Aim: Based on the chosen learning plan and the learner's progress, select and order learning activities considering the learner's preferences and constraints as well as the current context.

4. *Learning Resource*: any kind of knowledge resource (not only digital) that can be used in learning. A reusable and sharable learning resource can be linked to multiple learning activities. A learning activity can reference one or more learning resources.

   Examples: learning objects (documents, multimedia), learning peers (tutors, co-learners), learning services (navigation support, learning maps, recommendation of additional resources, facilities for synchronous and asynchronous communication and collaboration – including annotation and discussion forums)

   Aim: Based on the current learning activity select and order learning resources considering the learner's preferences and constraints as well as the current context (e.g. learning strategy).

5. *Presentation*: Selected learning resources can be presented in various ways, depending on the learner's preferences and the current context (e.g. delivery device, time constraints). Each learning resource can have alternative representations in various media (e.g. text, image, audio, video) and in various versions (e.g. quality, difficulty, target group).

   Domain concepts are linked with learning resources and this means that two alternative structures exist – a conceptual network (comparable to a book index) and a network (or hierarchy) of learning resources (analogous to a table of contents). For a learner it is important that she can browse each of these networks that are seamlessly interconnected.

2.2 GRAPPLE Architecture

Section 2.1 discussed the general assumptions, models and layers of adaptation in the learning domain. They are the basis of the various authoring models as (for GRAPPLE) described in D3.2a, D3.3a, D3.4a and D3.5a. This section is intended to clarify the GALE high-level architecture in combination with the authoring environment. It gives an overview of the information flow and intends to give an intuitive feeling of the contents of the models involved and the processes that take place when translating from CAM to GALE.

Figure 1 denotes the most important GALE components and the information flow starting from the input models from the authoring environment towards GALE. The different components of this model are discussed in the sequel.

2.3 Authoring Models

The authoring environment produces several models:

- Domain models
- User model
- Concept Relationships Types
- Conceptual Adaptation Model

2.3.1 Domain Models

For the domain model we discern sub-models. If we refer to DM, we mean the union of these sub-models. The sub-models are:

- **Content Domain Model (CDM)** The CDM models the content domain of discourse.
- **Service Domain Model (SDM)** The SDM models the service and process oriented actions that apply to the CDM.
- **VR Domain Model (VRDM)** The VRDM models the VR specific items not expressible in the CDM.

A CDM consists of two parts: the concept network and references to resources that instantiate hypertext that is associated with those concepts.

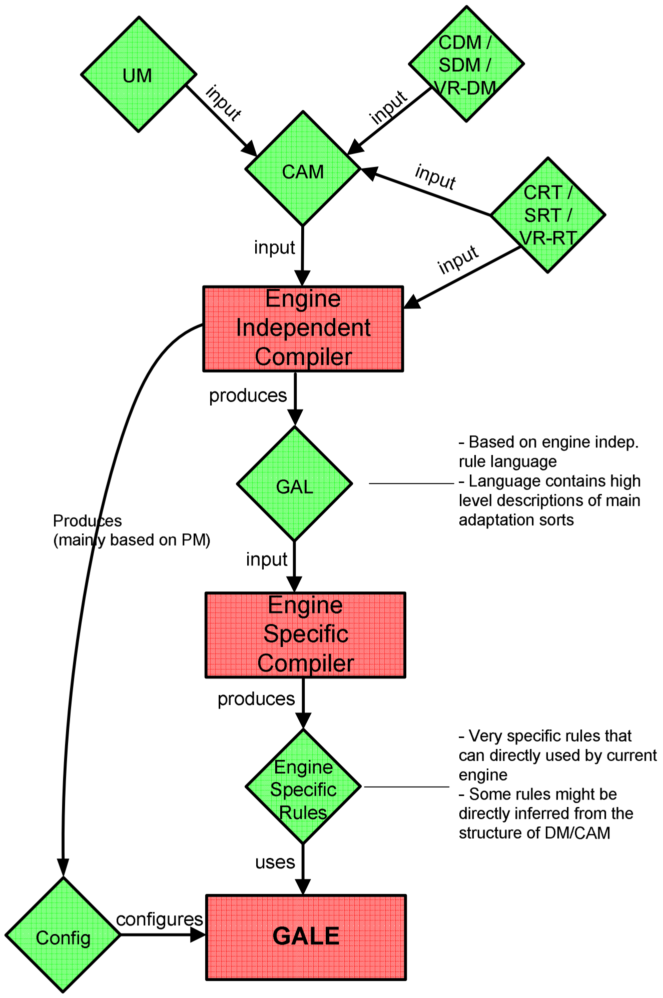

UM
CDM / SDM / VR-DM
CAM
CRT / SRT / VR-RT
input
input
input
input
input
Engine Independent Compiler
produces
GAL
- Based on engine indep. rule language
- Language contains high level descriptions of main adaptation sorts
Produces (mainly based on PM)
input
Engine Specific Compiler
produces
Engine Specific Rules
- Very specific rules that can directly used by current engine
- Some rules might be directly inferred from the structure of DM/CAM
uses
Config
configures
GALE

Figure 1: Information flow between GALE components and the input from the CAM (and related models).

Figure 2 is an illustrative example of a CDM. Concepts like “Solar System”, “Planet”, “Sun” and “Jupiter” are connected by relationships like “isa”, “part of” and “alsoGlobeLike”. Any number of relationships is possible and these relationships have no known predefined semantics. Often used relationships like “partOf” and “isa” can be specified in the CRT to assign knowledge in how to interpret these relationships and authors can use these standard CRTs when they do create a CDM.

A CDM might contain many more concepts than those needed in a course. Since the introduction of the Semantic Web many ontologies have become available and a user might want to import such ontologies into the Domain Editor and select only some of the concepts from that particular domain. This selection is documented in the CAM. The rest of the system will use this view on the domain as “the domain”.

Resources are properly modelled via a relationship like “hasResource” (the semantics of this property need to be modelled in a CRT). A resource typically refers to an URL of an actual hyperdocument. It also can have a Pedagogical attribute that can be referred to during course design. For instance if a teacher for every concept first wants to show introductory resources, he can refer to these resources via a label (or to be configured attribute) with value “Introduction”.

The CDM will be explained in more detail in D3.1a/b/c. SDM will be explained in D3.5a/b/c and VRDM models will be specified in D3.4a/b/c.

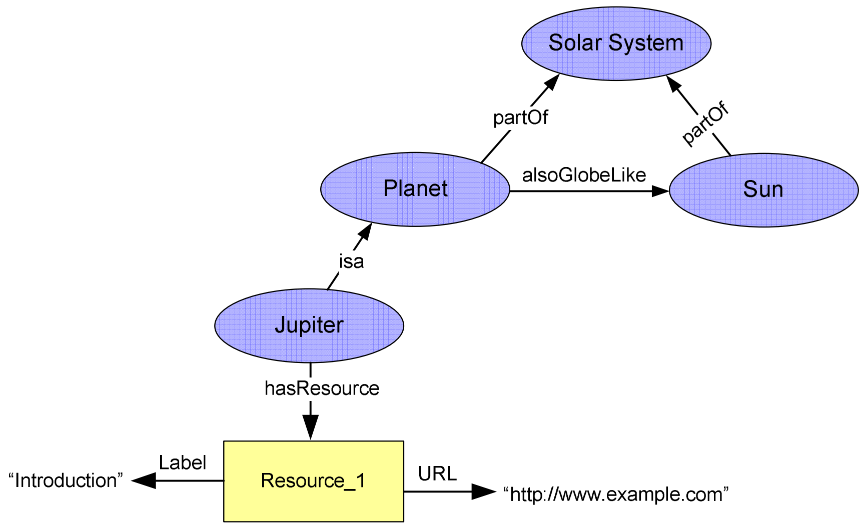


Figure 2: Illustrative example of a CDM.

2.3.2 User Model

The User Model (UM) will be typically an overlay over the DM, but also may contain some application independent data. Moreover, the UM can also refer to information in another application's UM.

The standard overlay model can for a large part be implicitly inferred from the domain. However, the author needs to be able to specify the application independent part in some way.

The author or user might also want to (re-)use information from another application's UM (which we call "foreign UM") in its own user model (which we call "native UM"). Therefore, he of course needs to know the structure of the foreign UM. He must also be able to define how to translate from the foreign UM to the native UM. In the simple case the author just points to concepts in the foreign UM which correspond with a concept in the native UM. For example, if a foreign UM records information about the concept "Sun" then the author can use the information from that UM to estimate the user's knowledge of the concept "Solar System" without teaching the student about the concept "Sun". However, sometimes the operation may be more complex than just pointing to a concept in the native UM. In that case more complex transformation steps might be necessary. One example is if the foreign UM records information about "Jupiter", "Saturn", etc, and the native UM only wants to know the user's knowledge about planets. In that case an aggregation of the "Jupiter", "Saturn", etc, concepts needs to be calculated. Another example requiring instance detail is where the foreign UM records the first and last name of a student and the native UM only supports the full name property of a student. The author should then write a translation rule that specifies how the first and last name properties can translate to the one full name property (i.e. via a concatenation operation on the literals).

The UM Framework created in WP2 and WP6 will design a rule language that can be used by authors to specify this translation between UMs. The authoring environment should allow users to express these rules. Furthermore, authors should be able to formulate "dummy" concepts in the UM that can be referred to by CAM / CRT and that get filled via a UM update. This means that the CAM/CRT are in general oblivious to these UM translation rules, but treat the "dummy" concepts as regular concepts.

The translation rules should also contain engine directives that include an identifier of which application governs this information and when and how often this information should be refreshed. This information is needed by the engine and will be used to know when the UM Framework needs to be contacted.

Note that the UM also contains information that in other work would be typically called context model, i.e. we regard everything that describes the current status of the user as part of the user model (e.g. similar as described in [5]).

2.3.3 Concept Relationship Types

We want to build a navigation structure based on the DM. One of the basic building blocks to specify this navigation is the Concept Relationship Type (CRT) model. The CRT defines the semantics of pedagogical and domain relationships between concepts or classes of concepts. "CRT" refers to the collection of these semantics defining rules. An instance of CRT is indicated by CRT-relationship. Note that a CRT is, in principle, application dependent. However, many CRT-rules can be put in a general library so that authors can reuse them in their applications.

In the CRT we define how relationships in the DM should be interpreted. We could, for instance, typically define that "isa" and "part of" relationships need be interpreted as relationships that we can use as a basis for the navigation hierarchy of the course. These kinds of rules can be a default part of a CRT for (new) applications. For a specific application domain we could define that we want to use the "alsoGlobeLike" relationship between domain concepts to denote links between resources of those concepts. We might also choose to ignore relationships, e.g. the relationships "soundsLike" might not be appropriate to derive navigation behaviour from.

We also define in the CRT constraints that restrict the navigation. For personalization these constraints refer to the user model to decide some kind of adaptation. An example of such a CRT-relationship is the prerequisite rule which restricts the accessibility of concepts based on prior knowledge of the user of other concepts.

CRT-relationships contain parameters for concepts (or resources) that need to be bound to actual concepts (or resources) in the CAM for the specific application. In this way a CRT-relationship can be used in more then one specific application. For instance the prerequisite relationship is applicable to many applications.

CRT- relationships can be used as a basis to translate the CAM (that uses CRTs) and its corresponding DM concepts to GAL constructs. An example of a CRT- relationship is:

prerequisite (Or($A,$B),$C):: if (Query($A) or Query($B)) then $C.suitable=true

In words, this means that for the prerequisites of the parameter $C to be satisfied, either concept $A must be satisfied **or** $B must be satisfied. Satisfaction is defined in this case by a query to the user model that results in a true or false value.

The CRT will be explained in more detail in D3.2a/b/c.

2.3.4 Conceptual Adaptation Model

The Conceptual Adaptation Model (CAM) specifies the course application as far it is specified by the author. It contains multiple sub-models that together specify the navigation and the presentation of the course. Here follows a very brief (and incomplete) overview of things that are specified in the CAM.

- **A DM-view**. As the CDM might contain more concepts than are relevant to the course, especially for imported domains, the selection of which concepts and resources are relevant for the course is specified in the CAM. Similar issues might apply to the SDM and VRDM.
- **CRT instantiation**. CRTs specify the navigation structure in a concept independent way, i.e. they specify the semantics of the relationships. These relationships are instantiated in the CAM. The relationships are here tied to concrete concepts from the DM. For example we can specify that Planet is a prerequisite of Jupiter.
- **Strategies**. An example of a strategy is first showing resources with the label "Introduction" and only then showing resources with the label "Advanced". This behaviour can again be expressed using CRTs.
- **Presentation.** The CAM specifies presentation specific elements of the course.

The CAM is explained in more detail in D3.3a.

2.4 GRAPPLE Adaptation Language

The GRAPPLE Adaptation Language (GAL) is an intermediate language between the authoring level and the concrete engine level. The GAL specifies navigation adaptation in an engine independent way and it allows specifying the most frequently used adaptive mechanisms and behaviours.

The purpose of GAL is to have an abstract language in which the main application experience of the student can be expressed in terms of navigation.

It is important to understand what GAL is not. The GAL is not intended to be able to specify everything that is expressible in CAM models, nor is it intended to capture all possible adaptation features that a specific engine can perform. In general GAL is not intended to become another fully-fledged programming language to 'program' adaptation. Instead, the GAL is intentionally restricted to the most common adaptive navigation primitives only. Note that even though we target the learning domain in the GRAPPLE project, GAL is more generic and is applicable in every domain that uses adaptive navigation.

The GAL offers some simple presentational constructs, but only for presentation aspects that are crucial for the navigation experience of the user. We intentionally do not go further than adding rudimentary sorting and emphasis constructs, because we regard the actual presentation specification a separate concern from navigation specification and also most adaptive systems differ greatly in what they support for presentation adaptation. Automatic behaviour in adaptive engines like automatically deriving that the knowledge of a concept equals the summation of the knowledge of its sub-concepts is also not supported in the GAL language, but is left to the engine.

3 GAL Language Constructs (Informal)

In this section we give an informal description of the GAL constructs. We step by step describe the functionality that we need, which language construct we use to fulfil this functionality and an example in an abbreviated RDF/Turtle[1] format to illustrate the use of this language construct. We use a running example of an application that presents an adaptive application about the Milky Way. Note that the use of the RDF/Turtle format is only chosen because it is simple and brief. We mainly use it because of its graph structure that we can exploit for sake of reuse of GAL constructs (which is harder with e.g. XML). However, it is no problem to use another syntax as long as it is able the constructs as we express them in this section.

3.1 Introduction

GAL specifies adaptive navigation. With navigation we mean here the whole navigation experience, which includes the structure of pages, and the links between those pages. What we will not cover are presentational issues like the colour of text, the font, how pages should be rendered to be presentable on devices with small screen sizes, etc. We have targeted here the concentrate and the constructive elements needed to describe adaptive navigation and have sought inspiration from existing languages for adaptive navigation (e.g. [6]).

In the following paragraphs we discuss the basic building blocks we need to compose adaptive pages. We do this on the basis of an example in the Planet domain, where we want to define the structure of pages of an application in this domain as well as the structure between those pages. In section 3.4 we show that the constructs that we specify together allow us to specify the main types of adaptation as specified by Brusilovsky [7].

Let us first see what we need in order to express adaptive navigation.

3.2 Main GAL Constructs

3.2.1 GAL Component

GAL describes the hypermedia structure of adaptive pages (i.e. the ones to be presented to the user), as well as the structure between these pages. Let us first look at the internal structure of such a page. The composition of a page is described with a hierarchy of GAL Components.

At the lowest level in the hierarchy we have GAL Components that are filled in with concrete resources (i.e. text, images, html-page, etc). So for this, what we need are constructions to represent the composition of GAL Components and to assign resources to GAL Components. In our hierarchy we more specifically discern two types of GAL Components: units are the non-leaf nodes in our tree, where attributes are the leaves in our tree. Units can be seen as a grouping element to group those elements that semantically will be shown together to the user. If we consider a concrete page, then the entire page can be called a Unit as well. We refer to these units as top-level units (even though they are not discernable from other units).

Let us look at our example. Suppose we talk about a unit on a Planet. We then declare something like (in our RDF/Turtle syntax)

```
:PlanetUnit a :Unit
```

[1] http://www.w3.org/2007/02/turtle/primer/

3.2.2 Attributes

Assume that we want to show the name of a Planet. The name is an example of a GAL Component that forms a leaf in our tree and we call it an Attribute. Then inside of that declaration we say something like

```
:PlanetUnit :hasAttribute [ here the details of the attribute ]
```

To give the details of an attribute we could give for instance its label. We could write this as

```
:label “Planet Name: ”
```

The label in this case can be used by the adaptive engine to show a label of what the value of the Attribute represents, in this case the name of the planet.

Arbitrary properties can be defined for attributes (besides fixed properties like label). However, if arbitrary properties are used the adaptive engine needs to interpret these properties. For instance, for GALE the class property is useful to define the presentation style.

3.2.3 Queries

Every attribute requires a value:

```
:value [Expression]
```

[Expression] is then a either a query expression retrieving a value (from the underlying database that maintains the UM/DM) or a constant. In this case the result for [Query] should be one value. A constant can also be used, i.e. if the attribute to be displayed is known beforehand. Think for instance of introductory text like "Welcome to this page".

Queries provide personalization and adaptation. The underlying database should provide access to both the domain model (for actual data) and the user model (for personalization) for this purpose, i.e. both accessible from the same query

Queries are not only used to fill in attribute values, but can be used to fill in every kind of value, e.g. also the name or label of an attribute, unit or link.

3.2.4 SubUnits

A Unit can also contain SubUnits, and if we would want a SubUnit for the planet’s parent star for example that would look like:

```
:PlanetUnit :hasUnit [SubUnit specification]
```

The SubUnit specification is named. We give it the following name:

```
name "parentStar" ;
```

The SubUnit refers to an actual Unit that represents information to be shown to the user. In this case we want to use the unit about a star, so we refer to:

```
  :refersTo :StarUnit ;
```

In order to select which star we need to present information about in the SubUnit, we refer to the actual star value via a query.

```
  :hasQuery [Query]
```

[Query] is (like before), a query expression that retrieves a value (from the underlying database), which will be forwarded to the SubUnit.

The SubUnit for star looks like:

```
  :StarUnit a :Unit
  :StarUnit :hasAttribute [
       :label “Star name”;
       :value [ :hasQuery [Query]
              ]
                             ]
```

3.2.5 Variables

The StarUnit inside the PlanetUnit depends on the containing Planet unit. Therefore, some information has to be passed from the PlanetUnit to the StarUnit. In our case we earlier passed the data about which star the StarUnit will show information about. In order to later refer to this variable we have to make it explicit and name it:

```
  :hasInput [
              :variable [
                         :varName "M";
                        :varType :Star
                        ]
              ]
```

The :varType here refers to a concept of the domain (in this case the concept “Star”).

3.2.6 Sets

A SubUnit as we specified now, can contain information about a single star. We can also imagine that we want to represent information about the set of moons of a planet. Therefore we have a hasSetUnit constructor. To represent information about every moon of the planet, we write:

```
  :PlanetUnit hasSetUnit [SubUnit specification]
```

In the SubUnit specification we specify what we also specify for regular SubUnits, i.e. label, unit reference, and query to select the right moons. The key difference is that the query for a regular SubUnit should have only one result (the star in our previous example), where the query for the setunit contains several results (every moon). The SubUnit specification would thus look like:

```
  :label "Moons" ;
  :refersTo :MoonUnit ;
  :hasQuery [Query]
```

Where the result for [Query] contains a set of results. With the number of results equal to 1, a setUnit thus is equivalent to SubUnit.

A special kind of set is the Tour. A Tour is like a set, but has the semantic difference that in a tour not all solutions of a query will be shown together, but that they will be shown one by one. For example if we would have written:

```
  :PlanetUnit hasTourUnit [SubUnit specification]
```

If this is about the moons this would mean that a user would first see information about one moon and only then (e.g. after some user action) see information about the next moon. It is for the specific adaptation engine to make a concrete implementation for the tour (e.g. by providing “next buttons”).

3.2.7 Links

Now we have (hierarchical) units that can represent content (composition). The second step is to represent links between those units.

```
  :PlanetUnit :hasLink [Link specification] .
```

Links are specified similarly as SubUnit, only the semantics differ as SubUnits are embedded in a unit, where links navigate from the current unit to the target unit as specified in the link. The Link specification for a Planet to a page that presents its in-depth technical details would look something like:

```
  :label "Planet In-Depth Technical Details" ;
  :refersTo :PlanetInDepthUnit ;
  :hasQuery [Query]
```

where the result for [Query] is one value, retrieved from the database or constant.

A link is an element of a Unit. The semantics of the link is that in the content presentation all direct attributes (i.e. not the attributes in SubUnits of the unit) are linked via the hasLink specification. The hasLink relation can also be coupled to Attributes.

3.2.8 Conditions

Queries can provide for adaptation and personalization. However, relying on this mechanism only is not very versatile. In this way it is for instance hard to specify information hiding or a choice between alternatives. Therefore we introduce conditions. For example consider the following attribute:

```
  :hasAttribute [
    :name "Extensive Description";
    :value [
            :hasCondition [ Condition ]
             ]
```

The condition consists of an if-then-else construction. The ‘if’ expression is a query. If this query has results (>0), a "then" clause is evaluated. The then clause contains an expression (which might be a query, but also another GAL Construct), e.g. in the Attribute case to compute the value of the actual attribute. If the ‘if’ query has no results the else clause is executed (which is an expression like the then-expression). The Condition thus looks like:

```
 :if [Query1]
 :then [Expression2]
 :else [Expression3]
```

Both :then and :else are optional. If one of them is omitted their evaluation has the special meaning that the GAL Component in which the Condition is used will not be initiated (i.e. hidden). For example

```
 :if [Query1]
 :then [Expression2]
```

If Query1 has no results the GAL Component will not be initiated. If it is the other way around

```
 :if [Query1]
 :else [Expression3]
```

This means that if Query1 has results the GAL Component will not be initiated. Theoretically also both :then and :else may be omitted, but this would mean that the GAL Component will never be initiated no matter what the query result is, which seems not a practically useful (as you might have left out the GAL Component in the first place).

Note that Query1 may be a composition of queries connected via Boolean operators, i.e. ¬Query1 is also a query. Also note that queries in adaptive conditions are used differently as queries elsewhere. Conditional queries can be used instead of all previous defined Query clauses (i.e. Attributes, SubUnits, SetUnits, Links).

3.2.9 Updates

For adaptation we also need to be able to update data in the UM. For this we use update queries. An update query is element of a Unit. It looks like:

```
 :updateQuery [UpdateQuery]
```

[UpdateQuery] is then a query expression changing a value in the user model database. The update query can refer to other values in the database as well as refer to variables in the current unit (or one of its superunits).

Update queries can also be used to update the DM, even though it might conceptually be a bit strange to adapt a given domain of discourse and adapt that on basis of the input of a student. But there is no practical inhibitor that forbids the updating of the domain model, i.e. if the underlying database permits both read and writes on a model this can be used in the GAL code.

3.2.10 Events

There are two points in time upon which we want to trigger an update: on access of a unit and on exit of the unit. The on access updates might be necessary for an update that always will be executed for a unit (e.g. also if the user closes the application) and for which the entire query is known beforehand. An example of such a query is maintaining the number of views for a unit for the user. Other update queries might depend on actions of the user (e.g. which link is clicked). In order to facilitate queries at one of these instances the :onAccess and :onExit properties can be used:

```
 :onAccess [
   :updateQuery [UpdateQuery]
 ];
```

or

```
 :onExit [
   :updateQuery [UpdateQuery]
 ];
```

Currently we only consider onAccess (e.g. to count the number of page views) and onExit (e.g. to measure the time that the page has been viewed) events. Other events can be useful as well, but require a scripting mechanism in the adaptive engine that triggers on other events (e.g. mouse events, key events, time events, etc). As most adaptive engines do not explicitly support this we for now do not consider these other types of events. However, as both VR and Simulation in the GRAPPLE project (refer to D4.1a and D4.2a) probably do need the extension of the type of events so we explicitly leave this option open.

3.2.11 Emphasis

GAL Components can be emphasized by giving them the property

```
 :hasEmphasis “high”
```

We discern four types of emphasis:

- “low” emphasis means under-emphasis. A component with “low” emphasis should get less emphasis than normal objects.
- default emphasis is if no hasEmphasis property has been defined, i.e. the default for all GAL Components.
- “normal” emphasis means more emphasis than normal. This level allows the adaptive engine to differentiate between more important objects (“normal” emphasis) and most important objects (“high” emphasis).
- “high” emphasis is the most emphasis a GAL Component can have.

Emphasis is a construct that directly influences the presentation. Even though we mainly target adaptive navigation, emphasizing components (especially links) on a page can strongly influence the navigation behaviour of the user and it is a regular adaptive construction [7]. Therefore, it was chosen to include this construct in our language.

It is up to the engine to decide how to present an emphasized GAL Components, e.g. by highlighting. Emphasis for a Unit means emphasis for every element in that unit.

3.2.12 Order

Order is an important aspect of adaptive navigation. Moreover, it might be important for the internal logical structure of a page.

GAL Components can have an order element that specifies an ordering. The value of an ordering element is either an integer, or a (conditional) query resulting in an integer. For example we can give an attribute "PlanetName" an order element with value 1 as follows:

```
:PlanetUnit :hasAttribute [
                              :name PlanetName;
                              :order 1
                   ]
```

The order element dictates a partial order in GAL Components within the scope of a particular Unit. This means that if two or more GAL Components have an order attribute with an integer value, the subcomponent with the lowest value is guaranteed to be displayed before the subcomponent with a higher ordering value. Subcomponent with the same ordering value are interpreted as relatively unordered, meaning that there is no guarantee which of these subcomponent is displayed before the other. The ordering of GAL components that have a non-integral or missing ordering property is undefined

In general we do not assume any particular order based on the order of GAL code. However, we do allow specifying to strictly follow the GAL code order within a GAL Component by defining a default-ordering property for that unit by simple declaring:

```
:default-ordering true
```

Note that using an explicit order attribute takes precedence over the default ordering.

For database results (e.g. for the SetUnit construct) we automatically assume the default-ordering to be true, unless specified otherwise:

```
:default-ordering false
```

3.3 Additional Basic GAL Constructs

With the constructs we showed thus far we can express the main aspects of adaptive navigation namely GAL Components, their composition, their linking and adaptation.

Additional 'basic' constructs can be defined as well (basic in the sense that they cannot be expressed in the previous constructs). We discuss these additional basic constructs shortly, but in order to start from a generic language design we focus on the core and common elements.

- **Forms**, to submit more complex information of the user than link clicks.
- **Frames**, for navigating only part of the page
- **Scripts**, to add language specific constructs which will be treated as is. This can be useful, for example, to achieve Web 2.0 functionality like asynchronous updating of the page.
- **Web Services**, to contact an external service that provides a part of the content of the page.

3.4 Compositional GAL Constructs

Our basic GAL constructs thus far expressed adaptation on a very basic level. This allows for very generic types of adaptation. However, we also consider more specific adaptation technologies as proposed in Brusilovsky's adaptation taxonomy [7], which gives us the following adaptation techniques:

- Adaptive multimedia presentation
- Adaptive text presentation
- Direct guidance
- Link sorting
- Link hiding
- Link annotation
- Map adaptation

We review these techniques and present how GAL can be used to apply these techniques and we also provide some constructs that help to apply some of the techniques and we explain how these additional constructs can be translated into basic GAL constructs.

In this way we show that GAL supports the basic types of adaptation as recognized in the field of adaptive systems. To show that you also actually express the adaptive navigation of a full Web application in GAL we provide a full example of such a GAL application in Appendix A.

3.4.1 Adaptive Multimedia Presentation

Usually in adaptive systems Adaptive Multimedia Presentations means personalized media selection, i.e. given that there is a choice of a number of multimedia clips the system chooses the one that fits best for the user.

A selection would look like this:

```
:select  [
     :case [    :casenumber “1”;
                :condition [Query1];
                :hasAttribute [ here the details of multimedia attribute 1]
           ] ;
     …
     :case [    :casenumber “N”;
                :condition [QueryN];
                :hasAttribute [ here the details of multimedia attribute N]
           ] ;
     :case [    :casedefault “true”;
                :hasAttribute [ here the details of multimedia attribute in the default case]
           ]
   ]
```

The semantics of this statement is that the first case that matches its corresponding condition will be shown and otherwise the default case will be shown.

One way of translating this in terms of GAL if-statements of one attribute with differing values is as follows:

```
 :hasAttribute [
                    (some general details of the attribute)
           :if [Query1] ;
           :then [query of multimedia attribute 1];
           :else[ … :if [QueryN];
```

```
                        :then [query of multimedia attribute N];
                        :else [query of default multimedia attribute]
                    ]
            ]
```

Direct adaptation of multimedia objects cannot be done directly in GAL. However, if a Web Service exists that allows adaptation of a multimedia object this can be supported in GAL via the Web Service construct. Personalization can than be done. GAL can send variable parameters to the Web Service call. However, the Web Service should send a result that can be handled within the GAL language, e.g. text or an URL.

3.4.2 Adaptive Text Presentation

In GAL an attribute can be a piece of text as well as any other type of media (identified by a URL). So the choice of construction in the previous paragraph also applies to text. This means that alternative page fragments can be selected. Moreover, general text adaptation can be done using the (conditional) query mechanism in GAL that can refer to both domain concepts and the UM.

We offer a specific construct to weave different versions of a document together. This allows to quickly select a group of items to be shown based on a condition (in contrast with the selection of a single GAL Component in section 3.4.1).

At the start of a Unit for which we want to define different versions together we declare something like

```
  :alternative  [
        :case [    :casenumber “1”;
                   :condition [Query1];
                   :selectid “alternative1”
              ] ;
        …
        :case [    :casenumber “N”;
                   :condition [QueryN];
                   :selectid “alternativeN”
              ] ;
        :case [    :casedefault “true”;
                   :selectid “default”
              ]
   ]
```

Based on this selection criterion we now define in the rest of the unit which GAL sub-Components belong to which selection id by adding the :selectid property with the appropriate name, for example:

```
    :has Attribute [
                   :label “Planet Name:”;
                   :value [query];
                   :selectid “alternativeN”
                 ]
```

In this way all GAL Components with selected “alternative” will be initialized, while all attributes that belong to one of the other alternatives are not initialized.

This can be done for all GAL Components that accept queries. This alternative construction can be used to prevent repeating selection queries and to get a better overview for which page-alternative a construct is used. The semantics of this construction is that all GAL Components with a specific selectid will be shown only if it's the first case in the alternative statement for which the condition is satisfied.

The translation into GAL constructs (internally) is done by repeating the conditions for every GAL component to check if the condition for the specific GAL component is the first that satisfies the condition of the case statement.

3.4.3 Direct Guidance

Direct Guidance means that the system decides what the best next link for a user is. In order to support this we use a slightly adapted version of the 'alternative' construction above, namely

```
:emphasize  [
        :case [    :casenumber "1";
                   :condition [Query1];
                   :emphasisid "emphasis1";
                   :hasEmphasis "high"


               ] ;
        …
        :case [    :casenumber "N";
                   :condition [QueryN];
                   : emphasisid "emphasisN"
               ] ;
        :case [    :casedefault "true";
                   : emphasisid "default"
               ]
          ]
```

So instead of a selectid property we use a emphasisid, where the id points to GAL Components. Typically for Direct Guidance we would only point to links, but this construct can also be used to emphasize other GAL Components other than links.

Translation in GAL terms is slightly different here, because conditions cannot be attached to hasEmphasis properties, but only to query elements. Instead we generate two versions of the same GAL Component that is referred in the emphasisid clause, an emphasized and a non-emphasized one, and based on conditions we choose one of them. For example

```
:hasAttribute [
                    :label "Planet Name:";
                    :value [query];
                    :emphasisid "emphasisN"
                  ];
```

would translate to the attribute pair

```
:hasAttribute [
                    :label "Planet Name:";
                    if [QueryN];
                    then [query];
                    :hasEmphasis "true"
```

```
                ];
    :hasAttribute [
                 :label “Planet Name:”;
                 if [QueryN];
                 else [query]
                ]
```

3.4.4 Link Sorting

Adaptive link sorting is already directly supported in GAL via the Set Unit in combination with ordering. This assumes that the query language supports ordering. By referring in the query to the UM (as well as the DM) allows to retrieve a set of links that depend on the UM.

3.4.5 Link Hiding

Link hiding (as well as hiding of other GAL Components) is also directly supported in GAL via conditions. By not specifying a then or else part in the condition means that a link (or other component) is not initiated and thus effectively hidden.

If the GAL Component should not be really hidden but just displayed differently (e.g. grayed out) the emphasis level “low” can be used.

3.4.6 Link Annotation

Link annotation means that there is some form of adaptive comment or presentational makeup, to give visual cues to the user. We do not bother ourselves with the presentational aspects of link annotation in GAL.

However, what is necessary from GAL perspective is that user information can be updated and used to adapt different aspects of a link. This is possible in GAL because the value of the properties in a link can be determined via queries in combination with adaptive conditions. For example, given GALE, we could adaptively determine the class of an attribute or a link. GALE then is responsible for the concrete different forms of annotation of links.

3.4.7 Map Adaptation

Map adaptation is used to for instance generate menu structures. One typical type of map adaptation is a hierarchical structure of links. This can be defined in a hierarchical construction as follows:

```
    :hasHierarchy [
         :item [    :itemid “ch1”;
                    :subitem :root;
                    :hasLink [Link specification ch1]
               ] ;
         :item [    :itemid “ch1.1”;
                    :subitem “ch1”;
                    :hasLink [Link specification ch1.1]
               ] ;
         :item [    :itemid “ch1.1.1”;
                    :subitem “ch1”;
                    :hasLink [Link specification ch1]
               ] ;
         ….
         :item [    :itemid “chN”;
```

```
                :subitem :root;
                :hasLink [Link specification chN]
            ]
    ]
```

This can be translated in GAL terms by using the hierarchical structures of Units in combination with the Order attribute. For example consider the following (partial) translation:

```
 :hasUnit [
          :hasLink [ :order 1;
                    {Further Link specification ch1}
                   ];
          :hasUnit[ :order 2;
                  {Reference to SubUnit that specifies ch1.1 and ch1.1.1}
                  ]
….
          :hasLink [ :order 3;
                    {Further Link specification chN}
                   ];

  ]
```

4 The Formal Description of GAL

In this section we give a more formal description of the core of GAL. Basically the language expresses a mapping from the models that contain the information on which the web site is based to the navigation model of the web site. Therefore we discuss in the next session the structure of these models. Then, in the following section, we discuss an extension of this information with derived relationships, which we will call the conceptual relationship types (refer to D3.2a for more about CRTs). After this we present an abstract syntax of the language, followed by a description of the semantics of this language.

4.1 Domain and User Models

For the purpose of our language we will assume that all the models such as the domain models, the user models, and any other model that is used to generate the navigation structure, are represented together as one big RDF graph. This is formally defined as follows:

Definition (URI, Blank node, RDF literal) We postulate the pair wise disjoint infinite sets I, B and L, of URIs, blank nodes and RDF literals.

Definition (RDF graph) An RDF graph is a finite subset of $(I \cup B) \times I \times (I \cup B \cup L)$. The set of all RDF graphs is denoted as *RG*.

Note that this RDF graph describes the models at the conceptual level, and so in a real implementation they might or might not be represented in RDF. We will call this RDF graph the *models graph*. In the following subsections we will give some hints as to what might be found in this models graph, but this is only to informally illustrate their contents. Formally we make no assumptions about the contents of the models graph. As a simplification we will assume that the DM and UM together form an RDF graph. These models are actually modelled differently (refer to D3.1a for the DM), but can be easily translated into a RDF graph (i.e. this doesn’t change the validity of formal description).

4.1.1 DM: The Domain Models

In the DMs we may for example have the following classes, i.e., instances of rdfs:Class:

- **gal:Concept** This contains all the concepts that play a role in the course material presented by the system. These may or may not have the gal:name property ending in rdfs:Literal, but if they do then it is assumed to be unique. It also can have the property gal:hasResources of type rdf:Bag (containing instances of gal:Resource).
- **gal:Relationship** This represents a relationship between concepts that is a relevant part of the domain discussed in course material that is to be presented to the users. So it is a subclass of rdf:Property and has gal:Concept as its domain and range.
- **gal:Resource** This represents a certain resource for information about a concept in the form of a URI. It has the required property gal:URI with the range rdfs:Resource, and can in addition have several other properties with a literal value.

4.1.2 UM: The User Models

In the UMs we may for example have, in addition to those in the DMs, the following classes:

- **gal:User** This contains all users that are described in the user model. These have the property gal:name which represent the unique name by which they are identified, and possibly many other properties with a literal value.
- **gal:HasKnowledge** This represents the fact that a certain user has knowledge about a certain concept. It has the required properties gal:user, gal:concept and gal:level (indicating the knowledge level) which end in gal:User, gal:Concept and rdfs:Literal.
- **gal:IsSuitable** This represents the fact that a certain concept is suitable for a certain user. It has the same required properties as gal:HasKnowledge.
- **gal:isRecommended** This represents the fact that a certain concept is recommended to a certain user. It also has the same required properties as gal:HasKnowledge.

4.2 CRT: The Conceptual Relationship Types

The CRTs extend the models graph with derived relationships between the classes in the models graph. These relationships are presumed to be relevant for the generation of the navigation structure. The structure of the relationships is defined by a CRT schema:

Definition (CRT Schema) A CRT schema is a pair (*P*, *ar*) where *P* is a finite set of predicate names and *ar* a function that gives the arity of each predicate in *P*.

An example of such a schema could be (*P*, *ar*) where *P* = { suitableAtLevel, isa } with ar(suitableAtLevel) = 3 since the relationship holds between a user, a concept and a suitability level, and *ar*(isa) = 2 since it always holds between two concepts. The meaning of these relationships is defined by a finite set of some type of Horn clauses that describe how they are derived from the models graph. These Horn clauses are defined as follows:

Definition (Variables) We postulate the infinite set *V* of variables, which is pair wise disjoint with the earlier postulated sets. We will assume they always start with a question mark like ?x and ?name.

Definition (CRT Horn clause) Given a CRT schema (*P*, *ar*) a CRT Horn clause is an expression of the form

$$atom_1 \textbf{ :- } atom_2, \ldots, atom_n.$$

with $n > 0$ and where each $atom_i$ is either
(i) a triple pattern, i.e., an expression of the form $(r_1\ r_2\ r_3)$ where each $r_i \in I \cup L \cup V$, i.e., is either a IRI, a literal or a variable.
(ii) a comparison, i.e., an expression of the form $(t_1, eqop, t_2)$ where t_i is a term consisting of literals, variables and simple arithmetic and string operators, and *eqop* one of =, < or ≤.
(iii) a predicate pattern, i.e., an expression of the form $p(r_1 \ldots r_n)$ with $p \in P$ a predicate name such that ar(p)=n and each $r_i \in I \cup L \cup V$.
We call $atom_1$ the *head* and $atom_2, \ldots, atom_n$ the tail. Moreover, a Horn clause must satisfy the

following restrictions: (1) the head must be a predicate pattern, (2) all variables occurring in the head must appear in at least one triple or predicate pattern in the tail and (3) all variables occurring in a comparison in the tail must also occur in a triple or a predicate term in the tail. Finally, note that we explicitly allow the tail to be empty.

A complete specification of the CRTs is then defined as follows:

Definition (CRT specification) A CRT specification is a tuple (*P*, *ar*, *R*) where (*P*, *ar*) is a CRT schema and R a finite set of CRT Horn clauses over (*P*, *ar*).

For example, the following rules specify that the concept C is suitable for a user at a certain level if this user already knows A and B at that level:

```
suitableAtLevel(?c ?u ?l) :-
        (?a gal:name "A"), (?c gal:name "C"), (?b gal:name "B"),
        (?k1 rdf:type gal:HasKnowledge),
                (?k1 gal:user ?u), (?k1 gal:concept ?a), (?k1 gal:level ?l),
        (?k2 rdf:type gal:HasKnowledge),
                (?k2 gal:user ?u), (?k2 gal:concept ?b), (?k2 gal:level ?l).
```

Another example would be the de-reification of a relationship:

```
isa(?c1 ?c2) :- (?k rdf:type gal:Relationship),
        (?k rdf:subject ?c1), (?k rdf:predicate gal:isa),
        (?k rdf:object ?c2).
```

Note that this needs to be done explicitly, since RDFS and OWL do not do this automatically. Or in addition computing the reflexive and transitive closure of the is-a relationship:

```
isa(?c1 ?c1) :- (?c1 rdf:type gal:Concept).
isa(?c1 ?c3) :- isa(?c1 ?c2), isa(?c2 ?c3).
```

The semantics of a set of rules is defined as usual in logic programming: each rule is interpreted as an implication from right to left, so the tail implies the head, and the variables that it contains are all universally quantified. The logical model for these logical formulas consist of a combination of the models graph and an interpretation of the predicates in the CRT schema:

Definition (Extended Models Graph) Given a CRT schema (*P*, ar) an extended models graph is a finite set of propositions where these propositions are either of the form $(r_1\ r_2\ r_3) \in (I \cup B) \times I \times (I \cup B \cup L)$ or of the form $p(r_1 \ldots r_n)$ with $p \in P$, $n = ar(p)$, and all $r_i \in (I \cup B \cup L)$. The set of all extended models graphs is denoted as *EMG*.

For a given extended model graph the truth-value of an atom with no variables is straightforwardly defined. This in turn defines the truth-value of first order formulas built up from such atoms, such as the CRT Horn clauses. We can then define the result of a CRT specification (*P*, *ar*, *R*) applied to an RDF graph *G* as the smallest extended models graph over (*P, ar*) that is a superset of *G* and for which all rules in *R* hold. It is not hard to see that this extended models graph is always uniquely defined, and that it can be computed in polynomial time in the size of *G*. We denote this result at [[*P*, *ar*, *R*]](*G*).

4.3 The Navigation Model

The result of a GAL expression is a navigation model that describes at an abstract level the navigation structure and the content of the pages. This result will of course depend upon the extended models graph that is used as the input of the GAL expression. We will describe here the basic navigation model that consists of only units and attributes and no additional GAL constructs.

The first notion we define is that of the navigation skeleton which describes the potential navigation points in terms of a unit name that identifies a type of page (like Star, Planet or Order) and a set of parameter names (such as star name, or order number) of the parameters that are in addition required to identify a certain page.

Definition (parameter name and unit name) We postulate an infinite set *PN* of parameter names and a set ***U*** of unit names.

Definition (navigation skeleton) A *navigation skeleton* is a pair (*U*, *pn*) where $U \subseteq \boldsymbol{U}$ is a finite set of root unit names and $pn : U \rightarrow 2^{PN}$ a function that maps each unit name in *U* to a finite set of parameter names.

Given a navigation skeleton we can define the set of all possible page identifiers as follows.

Definition (page identifier) Given a navigation skeleton (U, pn) a page identifier is a pair (u, pv) where $u \in U$ and $pv : pn(u) \rightarrow (I \cup L \cup B)$ the function that gives the parameter values for each of the parameters of u. The set of all page identifiers for (U, pn) is denoted as PI(U, pn).

A complete navigation structure describes how each page identifier is mapped to a page instance. Informally this represents the total state of a certain web site for a certain user at a certain moment. Before we define this formally we first define exactly the notion of page instance.

Definition (attribute name, content) We postulate an infinite set AN of attribute names and an infinite set CV of content values which denote values that can be somehow represented in the presentation of a page.

Definition (page structure) Given a navigation skeleton (U, pn) a page structure is defined as a tuple (N, E, <, an, ac, nl, src, oa, oe) where

1. N is a finite set of nodes, called GAL components, partitioned into a set of unit nodes N^U and a set of attribute nodes N^A,
2. $E \subseteq N \times N$ is a finite set of edges such that (a) (N, E) is a tree and (b) the root of this tree and all non-leaf nodes are unit nodes,
3. $< \subseteq N \times N$ is a strict partial order over N that only compares siblings,
4. $an : N^A \rightarrow AN$ is a partial function that maps some attribute nodes to their name,
5. ac : $N^A \rightarrow CV$ is a total function that maps each attribute node to its presentable content,
6. $nl : N^U \rightarrow$ PI(U, pn) is a partial function that maps some unit nodes to a page identifier which represents a navigation link,
7. $src : N^U \rightarrow N^U$ is a partial function that maps some unit nodes for which a navigation link is defined to a unit node in the tree which represents the location where the result of following the navigation link is placed,
8. $oa : N \rightarrow (EMG \rightarrow RG)$ a partial function that maps some nodes to an update over the models graph, which is executed in the case of an onAccess event, and
9. $oe : N \rightarrow (EMG \rightarrow RG)$ partial function that maps some nodes to an update of the models graph, which is executed in the case of an onExit event.

The set of all page structures for a navigation skeleton (U, pn) is denoted as PS(U, pn), and the set of page structures simply as PS.

This concludes the definition of navigation structures. Note that the language we will present has as its main purpose to define how the navigation structure looks for a certain user given a certain extended models graph. So the semantics of an expression in GAL will be a function that maps an extended models graph, a user identifier, a unit name and a binding of the unit parameters to a page structure.

4.4 The Language

In this section we describe the abstract syntax of GAL. For querying the extended models graph and updating the models graph we will assume the existence of separate languages, which are described by the non-terminals <query> and <update> respectively. This will be similar to languages like SPARQL and SeRQL but extended with extra constructs to deal the Conceptual Relationships in the CRT. For both these languages we will assume that their expression can contain free variables from the postulated set V. We also assume that if these free variables are replaced with elements from $I \cup L$ then the expression remains a valid expression of the language. The semantics of an update expression ue with no free variables is denoted as $[[ue]] : EMG \rightarrow RG$ which maps an extended models graph to the new models graph. Note that the update is based on the entire extended models graph, but only updates the RDF graph that represents the models graph. The semantics of queries is based on the notion of variable binding:

Definition (variable binding) A variable binding is a partial function $vb : V \rightarrow (B \cup I \cup L)$ that is defined for a finite set of variable names. The domain of this function, i.e., the variable names for which it is defined is denoted as **dom**(vb). The set of all variable bindings is denoted as VB.

The semantics of a query expression qe with no free variables is denoted as $[[qe]] : EMG \rightarrow 2^{VB}$ which is a total function that maps an extended models graph to a finite set of bindings.

We proceed with the abstract syntax of GAL expressions. We assume the following non-terminals as pre-defined: <variable> for elements of *V*, <unit-name> for elements of ***U***, <attr-name> for elements of *AN*, <literal> for element of *L*,<number> for the subset of *L* that represent integer numbers, <query> for expressions in the query language, possibly with free variables, and <update> for expressions in the update language, also possibly with free variables. In the meta-syntax we use "<a> | <b>" to the choice between <a> and <b>, "<a>*" to denote a sequence of zero or more expressions of type <a>, and "<a>?" to denote an optional sub-expression. The abstract syntax, i.e, the syntax without keywords and brackets, is then defined in EBNF style as follows:

```
<gal-expr> ::= <unit-def>*.
<unit-def> ::= <unit-name>? <unit-params>? <unit-content> <link-expr>?  <source-expr>?
               <order-expr>? <on-access-expr>? <on-exit-expr>? .
<unit-params> ::= <variable>* .
<unit-content> ::= ( <unit-def> |  <attr-def> | <set-unit-expr> )* .
<attr-def> ::= <attr-name>? <val-expr> <order-expr>? .
<val-expr> ::= <literal> | <query> | <cond-val> .
<cond-val> ::= <cond> <then-expr>? <else-expr>? .
<cond> ::= <query> | <and-cond> | <or-cond> | <not-cond> .
<and-cond> ::= <cond-expr> <cond-expr> .
<or-cond> ::= <cond-expr> <cond-expr> .
<not-cond> ::= <cond-expr> .
<then-expr> ::= <val-expr> .
<else-expr> ::= <val-expr> .
<order-expr> ::= <val-expr> .
<set-unit-expr> ::= <unit-def> <query> .
<link-expr> ::= <unit-name> <query> .
<source-expr> ::= <unit-name> .
<on-access-expr> ::= <update> .
<on-exit-expr> ::= <update> .
```

Note that expressions can contain free variables, in particular in <query> and <update>. We have the following three binding rules: (1) Within a <unit-def> the variable names in <unit-params> bind their occurrences in the following components within the <unit-def>. (2) Within <set-unit-expr> the variable names that are bound by the query bind their occurrences in the preceding <unit-def>. (3) The special variable ?user which identifies the user that retrieves the page is always bound. For a correct GAL expression it must hold that (1) it contains no unbound variables, (2) only the <unit-def>s at the top level have a <unit-params> component and (3) the <unit-name> of each <unit-def> must be unique within the whole <gal-expr>.

4.5 The Semantics of the Language

In this section we describe the semantics of the language. The semantics of a GAL expression *ge* is a partial function [[*ge*]] : $EMG \times (I \cup B \cup L) \times \boldsymbol{U} \times VB \rightarrow PS$. The first argument is the current extended models graph, the second argument is the value that identifies the user and the third argument is the binding that represents the parameters that are supplied to the page. The result is the page structure that will be presented to the user that requests the page.

The result of [[*ge*]](*emg*, *user*, *unit*, *vb*) is determined by selecting the unique <unit-def> in *ge* where the <unit-name> is equal to *unit*. If such a <unit-def> does not exist then the result is not defined. Also if the binding vb must be defined for all the variables in the <unit-params> of this <unit-def>, otherwise the result is also not defined. If this is all correct, then we substitute all the free variables in the unit-def with the values given in *vb* and the variable ?user is replaced with the argument *user*. Observe that for a correct GAL expression the result will then be a <unit-def> with no free variables. So what remains to be defined is the semantics of a <unit-def> with no free variables given an extended models graph *emg*.

To describe the semantics of <unit-def>s and <attr-def>s we introduce the notion of *page fragment*, which roughly corresponds to a fragment of a page structure.

Definition (page fragment) A page fragment is a tuple (*N*, *E*, *an*, *ac*, *nl*, *src*, *oa*, *oe, on, un*) where the first part (*N*, *E*, …, *oa*, *oe*) is equal to a page structure except that (a) there is no ordering <, (b) the root node is allowed to be an attribute node and (c) the function $src : N^U \rightarrow N^U$ is replaced with $src' : N^U \rightarrow \boldsymbol{N}$ which maps unit nodes to unit names, and the extra component is a partial function $on : N \rightarrow Nat$ that maps some nodes to a natural number that indicates the order, and $un : N^U \rightarrow \boldsymbol{N}$ maps unit nodes to their unit name. The set of all page fragments is denoted as PF.

We now define what we will call the pre-semantics of expressions of languages associated with several of the non-terminals:

- {[*ud*]} : EMG → PF with *ud* a <unit-def> which results in a page fragment with a unit root
- {[*uc*]} : EMG → 2^{PF} with *uc* a <unit-content> which results in a set of page fragments
- {[*ad*]} : EMG → PF with *ad* a <attr-def> which results in a page fragment with an attribute root
- {[*ve*]} : EMG → ($L \cup \{\bot\}$) with *ve* a <val-expr> which results in an RDF literal or undefined (⊥)
- {[c*v*]} : EMG → ($L \cup \{\bot\}$) with *ve* a <cond-val> which results in an RDF literal or undefined
- {[c*nd*]} : EMG → {**t**, **f**} with *cnd* a <cond> which results in a Boolean
- {[*ac*]} : EMG → {**t**, **f**} with *ac* a <and-cond> which results in a Boolean
- {[oc]} : EMG → {**t**, **f**} with *oc* a <or-cond> which results in a Boolean
- {[*nc*]} : EMG → {**t**, **f**} with *nc* a <not-cond> which results in a Boolean
- {[*sue*]} : : EMG → 2^{PF} with *sue* a <set-unit-expr> resulting in a set of page fragments with unit roots

We define all these functions as follows, with induction upon the syntactic structure of the involved expression:

{[*ud*]} : Assume that *ud* consists of the syntactic components *une*, *up*, *uc*, *le*, *se*, *oe*, *oae*, *oee* with *une* a <unit-name>, *up* a <unit-params>, *uc* a <unit-content>, *le* a <link-expr>, *se* a <source-expr>, *oe* a <order-expr>, *oae* a <on-access-expr>, and *oee* a <on-exit-expr>. We construct {[ud]}(emg) as follows. The root node is a unit node n_r. For this node the functions *an* (attribute name) and *ac* (attribute content) are undefined. The result of $nl(n_r)$ (navigation link) is defined if *le* exists and consists of the <unit-name> in *le* and the unique binding in the result of [[*qe*]](*emg*) if *qe* is the <query> in *le*. If there is no such binding, or more than one, then the result of $nl(n_r)$ is undefined. The result of $src'(n_r)$ (the navigation link source) is the unit name in the component *se* and undefined if this component is missing. The result of $oa(n_r)$ and $oe(n_r)$ (the on-access and on-exit updates) map to [[*ue*]] if *ue* is the update expression in *oae* and *oee*, respectively. The result of $on(n_r)$ is {[*oe*]}(emg) if this represents a number and undefined otherwise. The result of $un(n_r)$ is the unit name in *une*. The trees below n_r are exactly those in {[uc]}(emg) assuming that these do not share nodes or contain n_r.

{[*uc*]} : The components of uc will be either <unit-def> <attr-def> or <set-unit-expr>, which result in either single page fragments or a set of page fragments. The result of {[*uc*]}(emg) is a set of page fragments that contains a separate copy for each single page fragment in the result of each component.

{[*ad*]} : Assume that *ad* consists of the syntactic components *ane*, *ve*, *oe* with *ane* a <attr-name>, *ve* a <val-expr> and *oe* a <order-expr>. We construct {[*ad*]}(*emg*) as follows. The root node is an attribute node n_r. For this node the result of $an(n_r)$ (attribute name) is equal to *ane* if it exists and undefined otherwise. The result of $ac(n_r)$ (attribute content) is equal to {[ve]}(emg) if this is an RDF literal and undefined if this is ⊥. Like for unit definitions the result of $on(n_r)$ is {[oe]}(emg) if this is a number and undefined otherwise.

{[*ve*]} : (a) Assume that *ve* consists of *l* in <literal>, then the result of {[*ve*]}(emg) is *l*. (b) Assume that *ve* consists of *qe* in <query> then the result is the literal *l* if [[qe]](emg) returns a single binding that binds only one variable and binds it to l, and the result is ⊥ otherwise. (c) Assume that ve consists of cv in <cond-vall> then the result is equal to {[cv]}(emg).

{[c*v*]} : Assume that *cv* consists of *cnd*, *then*, *else* with *cnd* in <cond>, *then* in <then-expr> and *else* in <else-expr>. If {[*cnd*]}(*emg*) = **t** then the result of {[*cv*]}(*emg*) = {[*then*]}(*emg*) if the *then* component is present, and ⊥ otherwise. If {[*cnd*]}(emg) = **f** then the result of {[*cv*]}(*emg*) = {[*else*]}(*emg*) if the *else* component is present, and ⊥ otherwise.

{[c*nd*]} : (a) Assume that *cnd* consists of a *qe* in <query>. Then the result of {[*cnd*]}(*emg*) = **t** if [[*qe*]](*emg*) is non-empty, and **f** otherwise. (b) Assume that cnd consists of an *ac*, *oc* or *nc* in <and-cond>, <or-cond> or <not-cond>, respectively. Then the result is {[*ac*]}(*emg*), {[*oc*]}(*emg*) or {[*nc*]}(*emg*), respectively.

{[*ac*]} : Assume that ac consists of cnd_1, cnd_2 in <cond>, then {[*ac*]}(emg) = {[cnd_1]}(*emg*) ∧ {[cnd_2]}(*emg*).

{[*oc*]} : Assume that *oc* consists of cnd_1, cnd_2 in <cond>, then {[*ac*]}(emg) = {[cnd_1]}(*emg*) ∨ {[cnd_2]}(*emg*).

{[*nc*]} : Assume that *nc* consists of *cnd* in <cond>, then {[*ac*]}(emg) = ¬{[*cnd*]}(*emg*).

{[*sue*]} : Assume that *sue* consists of the syntactic components *ud* and *qe* with *ud* a <unit-def> and *qe* a <query>. The result of {[*sue*]}(*emg*) is a set of page fragments which is constructed as follows. For each binding *vb* in the result of [[*qe*]](*emg*) this set contains a distinct copy of the page fragment {[ud_{vb}]}(*emg*) where ud_{vb} is equal to *ud* except that every free occurrence of a variable in **dom**(*vb*) is replaced with its value in *vb*.

This completes the definition of all the functions and especially {[ud]}, and so we can then define the result of [[ud]](emg) by taking the result of {[ud]}(emg) and transforming it into a page structure as follows: (1) The functions *on* and *un* are removed. (2) The ordering < over *N* is added and defined such that $n_1 < n_2$ iff n_1 and n_2 are siblings and $on(n_1) < on(n_2)$. (3) The function $src' : N^U \rightarrow \boldsymbol{N}$ is replaced with $src : N^U \rightarrow N^U$ such that $src(n_1) = n_2$ iff $un(n_2) = src'(n_1)$.

5 Concluding Remarks

In this deliverable we discussed the steps that need to be taken from models in the authoring environment to specific code for the adaptive engine. We observed that several different adaptive engines exist that all use their own specific (and often inaccessible) syntax and semantics. Configuration is also often fragmented and spread over several files.

Therefore we presented the GAL language. GAL is based on the observation that in the end all the adaptive engines target to do the same thing, namely offer an adaptive Web application. GAL therefore offers the basic building blocks to express an adaptive navigation structure, which is the core part of the experience of the application. Moreover the syntax and semantics of the GAL language is kept as simple as possible (e.g. by restricting the number of constructs to a minimum), while maintaining the necessary expressive power. We showed that GAL is able to do the basic types of adaptation as recognized by the field of adaptive hypermedia. In Appendix A we expressed a small but complete adaptive application in the GAL language.

There is still much to do, though. We claim that higher level constructs as expressed in the CAM (as detailed in D3.3a) authoring language can be translated in equivalent GAL expressions. We of course need to show this by building a compiler that does exactly that. Similarly we need to show that GAL can be used as a basis by a number of adaptive engines (in the first place GALE, the GRAPPLE adaptive engine), for which we also need to build compilers. These are both tasks that will be detailed in D1.1b.

We do see a number of great benefits of our approach. If we are able to build compilers for several adaptive engines this would mean that the LMSs do not depend on a particular adaptive engine, but that they can choose the one that fits them best. Furthermore, the well defined semantics of GAL allow for additional features like model checking capabilities, e.g. for finding problems like cycles, dead ends, etc. Similarly, if we are able to build compilers for models from several authoring environments this will allow the author to choose the authoring environment of his liking, without the worry if this particular authoring is supported by the particular adaptive engine used by the LMS.

Appendix A: Application Example in GAL

In this section we show a complete example of a Milky Way application in GAL code.

Consider the following (abstract) DM model.

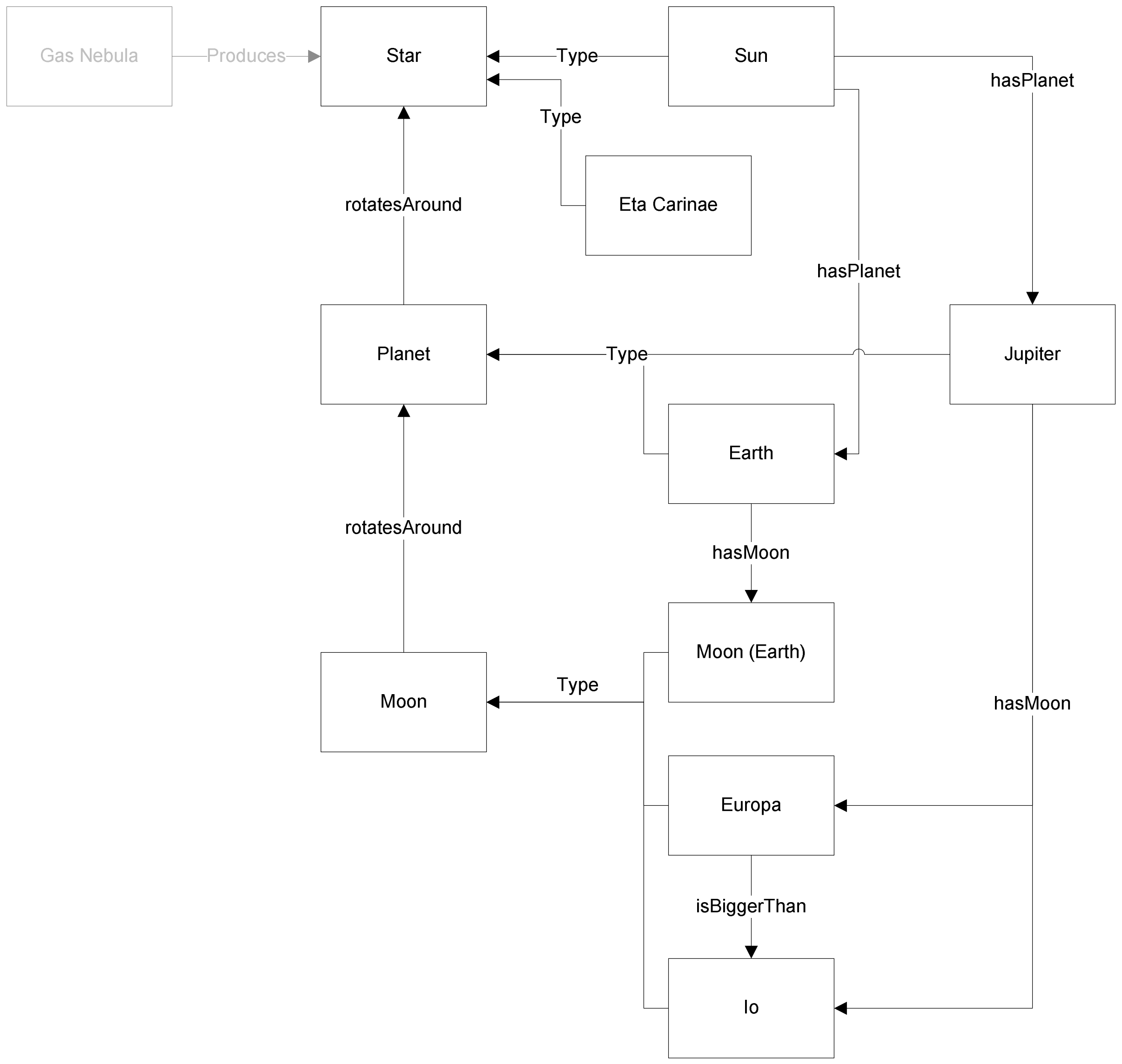
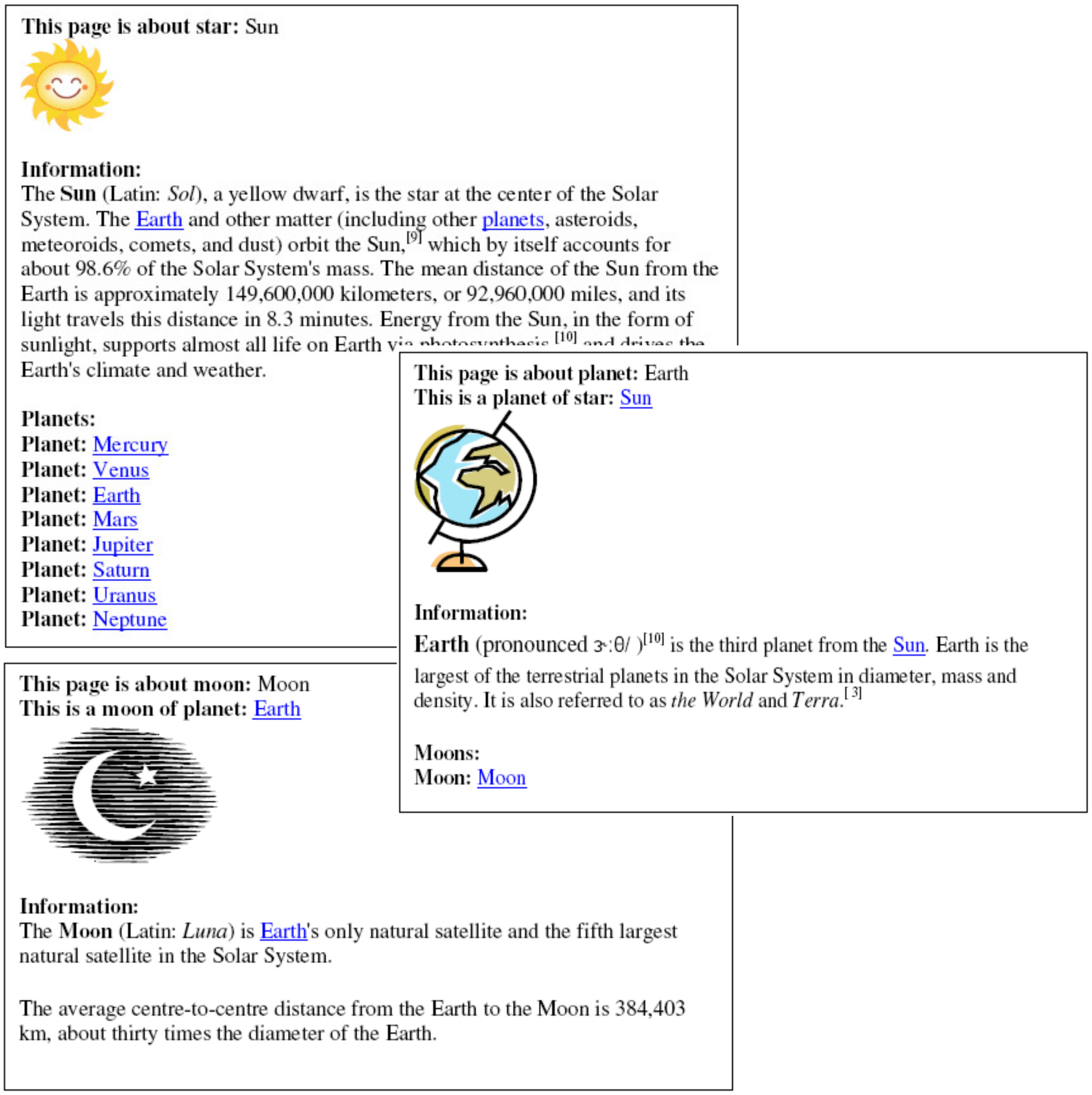


Figure 3: Milky Way example DM

Suppose we want to build an application based on that DM, which would roughly look like the following screen shots:



This page is about star: Sun

Information:
The **Sun** (Latin: *Sol*), a yellow dwarf, is the star at the center of the Solar System. The Earth and other matter (including other planets, asteroids, meteoroids, comets, and dust) orbit the Sun,[9] which by itself accounts for about 98.6% of the Solar System's mass. The mean distance of the Sun from the Earth is approximately 149,600,000 kilometers, or 92,960,000 miles, and its light travels this distance in 8.3 minutes. Energy from the Sun, in the form of sunlight, supports almost all life on Earth via photosynthesis [10] and drives the Earth's climate and weather.

Planets:
Planet: Mercury
Planet: Venus
Planet: Earth
Planet: Mars
Planet: Jupiter
Planet: Saturn
Planet: Uranus
Planet: Neptune

This page is about planet: Earth
This is a planet of star: Sun

Information:
Earth (pronounced ɜː.θ/)[10] is the third planet from the Sun. Earth is the largest of the terrestrial planets in the Solar System in diameter, mass and density. It is also referred to as *the World* and *Terra*.[3]

Moons:
Moon: Moon

This page is about moon: Moon
This is a moon of planet: Earth

Information:
The **Moon** (Latin: *Luna*) is Earth's only natural satellite and the fifth largest natural satellite in the Solar System.

The average centre-to-centre distance from the Earth to the Moon is 384,403 km, about thirty times the diameter of the Earth.

This would then expressed by the following GAL code.

```
:Star_Unit rdf:type Unit [
      :hasInput [
         :variable [
                  :varName "Star";
                  :varType Star
         ];
      ];
      :hasAttribute [
                  :name "starName";
                  :label "This page is about star: ";
                  :value [$Star.name]


      ];
```

```
:hasAttribute [
            :name "starImage";
            :value [ //url of an appropriate picture
                     SELECT   ?image_url
                     WHERE {  $Star :hasAssociatedResource ?image
                                                :type Image;
                                                :url ?image_url.
                     }
            ]
];
:hasAttribute [
            :name starInformation;
            :label "Information: ";
            :value [
                     :if [ //empty selection results in FALSE; otherwise TRUE
                              SELECT   ?conceptStarInstance
                              WHERE {  $User :hasConceptInstance ?conceptStarInstance
                                                         :name $Star.name;
                                                         :visited ?Visited;
                                                         :advancedUserValue ?Advanced.
                              FILTER (?Visited >= ?Advanced)
                              }
                     ]
                     :then [:value [
                                          SELECT   ?url
                                          WHERE {  $Star :hasAssociatedResource ?resource
                                                                     :type Text;
                                                                     :description ?keyword;
                                                                     :url ?url.
                                          FILTER (?keyword == "advanced")
                                          }
                                 ]        //high knowledge
                     ]
                     :else [:value [
                                          SELECT   ?url
                                          WHERE {  $Star :hasAssociatedResource ?resource
                                                                     :type Text;
                                                                     :description ?keyword;
                                                                     :url ?url.
                                          FILTER (?keyword == "basic")
                                          }
                                 ]        //low knowledge
                     ]
            ]
];
:hasSetUnit[
            :label "Planets: ";
            :refersTo Planet_Unit_Short;
            :hasQuery "$Star :hasPlanet ?Planet"
      ];
      :onAccess [
         :updateQuery [$Star.Visited := $Star.Visited + 1]
      ]
]

:Planet_Unit_Short [
      :hasInput [
            :variable [
                     :varName "Planet";
                     :varType Planet
            ]
      ];
      :hasAttribute [
            :name planetName;
            :label "Planet: ";
            :value [$Planet.Name]
      ];
      :hasLink [
            :refersTo :Planet_Unit_Elaborate;
            :hasQuery [$Planet]
      ]
]



:Planet_Unit_Elaborate rdf:type Unit [
      :hasInput [
            :variable [
                     :varName "Planet";
                     :varType Planet
            ];
      ];
      :hasAttribute [
            :name "planetName";
            :label "This page is about planet: ";
            :value [$Planet.name]


      ];
      :hasLink [
            :label "This is a planet of star: ";
            :refersTo Star_Unit;
            :hasQuery [
                              SELECT   ?Parent_Star_Name
                              WHERE             ?Parent_Star :hasPlanet $Planet;
                                                           :name ?Parent_Star_Name.
            ]
      ];
```

```
:hasAttribute [
            :name planetImage;
            :value [ //url of an appropriate picture
                     SELECT   ?image_url
                     WHERE {  $Planet :hasAssociatedResource ?image
                                         :type Image;
                                         :url ?image_url.
                     }
            ]
];
:hasAttribute [
            :name planetInformation;
            :label "Information: ";
            :value [
                     :if [
                              SELECT   ?conceptPlanetInstance
                              WHERE {  $User :hasConceptInstance ?conceptPlanetInstance
                                                  :name $Planet.name;
                                                  :visited ?Visited;
                                                  :advancedUserValue ?Advanced.
                              FILTER (?Visited >= ?Advanced)
                              }
                     ]
                     :then [:value [
                                         SELECT   ?url
                                         WHERE {  $Planet :hasAssociatedResource ?resource
                                                             :type Text;
                                                             :description ?keyword;
                                                             :url ?url.
                                         FILTER (?keyword == "advanced")
                                         }
                                ]        //high knowledge
                     ]
                     :else [:value [
                                         SELECT   ?url
                                         WHERE {  $Planet :hasAssociatedResource ?resource
                                                             :type Text;
                                                             :description ?keyword;
                                                             :url ?url.
                                         FILTER (?keyword == "basic")
                                         }
                                ]        //low knowledge
                     ]
            ]
];
:hasSetUnit[
            :label "Moons: ";
            :refersTo Moon_Unit_Short;
```

```
                :hasQuery "$Planet :hasMoon ?Moon"
      ];
      :onAccess [
         :updateQuery [$Planet.Visited := $Planet.Visited + 1]
      ]
]

:Moon_Unit_Short [
      :hasInput [
                :variable [
                         :varName "Moon";
                         :varType Moon
                ]
      ];
      :hasAttribute [
                :name moonName;
                :label "Moon: ";
                :value [$Moon.name]
      ];
      :hasLink [
                :refersTo :Moon_Unit_Elaborate;
                :hasQuery [$Moon]
      ]
]

:Moon_Unit_Elaborate rdf:type Unit
      :hasInput [
                :variable [
                         :varName "Moon";
                         :varType Moon
                ];
      ];
      :hasAttribute [
                :name moonName;
                :label "This page is about moon: ";
                :value [$Moon.name]
      ];
      :hasLink [
                :label "This is a moon of planet: ";
                :refersTo Planet_Unit;
                :hasQuery [
                                  SELECT   ?Parent_Planet_Name
                                  WHERE             ?Parent_Planet :hasMoon $Moon;
                                                             :name ?Parent_Planet_Name.
                ]
      ];
      :hasAttribute [
                :name moonImage;
```

```
            :value [ //url of an appropriate picture
                     SELECT   ?image_url
                     WHERE {  $Moon :hasAssociatedResource ?image
                                           :type Image;
                                           :url ?image_url.
                     }
            ]
      ];
      :hasAttribute [
            :name moonInformation;
            :label "Information: ";
            :value [
                     :if [
                              SELECT   ?conceptMoonInstance
                              WHERE {  $User :hasConceptInstance ?conceptMoonInstance
                                                    :name $Moon.name;
                                                    :visited ?Visited;
                                                    :advancedUserValue ?Advanced.
                              FILTER (?Visited >= ?Advanced)
                              }
                     ]
                     :then [:value [
                                           SELECT   ?url
                                           WHERE {  $Moon :hasAssociatedResource ?resource
                                                                 :type Text;
                                                                 :description ?keyword;
                                                                 :url ?url.
                                           FILTER (?keyword == "advanced")
                                           }
                                  ]        //high knowledge
                     ]
                     :else [:value [
                                           SELECT   ?url
                                           WHERE {  $Moon :hasAssociatedResource ?resource
                                                                 :type Text;
                                                                 :description ?keyword;
                                                                 :url ?url.
                                           FILTER (?keyword == "basic")
                                           }
                                  ]        //low knowledge
                     ]
            ]
      ];
      :onAccess [
         :updateQuery [$Moon.Visited := $Moon.Visited + 1]
      ]
]
```

References

1. Koper, R. and Tattersall, C. (Eds.). Learning Design - A Handbook on Modelling and Delivering Networked Education and Training. Heidelberg: Springer Verlag, 2005
2. Cristea, A. and Calvi, L., The Three Layers of Adaptation Granularity, in Proceedings of the ninth International Conference on User Modeling, LNCS 2702, Springer , 2003, pp. 145-156
3. Halasz, F. and Schwartz, M. The Dexter Reference Model. In Proceedings of the NIST Hypertext Standardization Workshop, 1990, pp. 95-133
4. De Bra, P., Houben, G.J., Wu, H., AHAM: A Dexter-based Reference Model for Adaptive Hypermedia, Proceedings of the ACM Conference on Hypertext and Hypermedia, 1999, pp. 147-156
5. Kobsa, A., Koenemann, J. and Pohl, W., Personalised Hypermedia Presentation Techniques for Improving Online Customer Relationships, in The Knowledge Engineering Review, 16(2), 2001, pp. 111-155
6. Houben, G.J., van der Sluijs, K., Barna, P., Broekstra, J., Casteleyn, S., Fiala, Z. and Frasincar, F., Hera, Book chapter in: Web Engineering: Modelling and Implementing Web Applications , Springer (2008), Human-Computer Interaction Series (12), eds: Gustavo Rossi, Oscar Pastor, Daniel Schwabe and Luis Olsina, isbn: 978-1-84628-922-4, pp. 263-301
7. Brusilovsky, P. Methods and Techniques of Adaptive Hypermedia, in User Modeling and User-Adapted Interaction 6, 1996, pp. 87-129