# GRAPPLE

D1.1b Version: 1.1

## CAM to Adaptation Rule Translator (Implementation)

| **Document Type** | Deliverable |
|---|---|
| **Editor(s):** | Kees van der Sluijs (TUE) |
| **Author(s):** | Jan Hidders (TUD), Kees van der Sluijs (TUE), Paul De Bra (TUE), David Smits (TUE), Erwin Leonardi (TUD), Geert-Jan Houben (TUD) |
| **Internal Reviewer(s):** | Avi Naim (UCAM), Vincent Wade (TCD) |
| **Work Package:** | 1 |
| **Due Date:** | 1-11-2009 |
| **Version:** | 1.1 |
| **Version Date:** | 8-2-2010 |
| **Total number of pages:** | 20 |

**Abstract:** This deliverable describes the translation from a Conceptual Adaptation Model to low-level adaptation rules for the GRAPPLE Adaptive Learning Environment (GALE). The compiler performing this translation is incorporated into GALE.

To make the GRAPPLE authoring environment usable with other adaptive delivery engines we also define an 'intermediate 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:** GALE, GAL, GRAPPLE Adaptation Language, Navigation Specification, CAM, rule translation

# Summary

This deliverable looks at the implementation of 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). This deliverable builds upon D1.1a, but also diverges from one of the main concepts put forward in that deliverable. We no longer use GAL as the intermediate language between the CAM model and the GALE engine. Instead we now use GAL as an interchange language between the GRAPPLE framework and other adaptive frameworks. Within GRAPPLE we use a direct translation from CAM to rules GALE can process. The reasons for this decision are mainly of a practical nature.

In this deliverable we describe the updated architecture. We explain in detail how we achieved translation between the CAM model and the GALE adaptation engine. We also go into more detail about the GAL language. We provide a formal definition of the language. The actual implementation of the translation between GRAPPLE's components and GAL remains a goal for the third year of the project.

# Authors


| Person | Email | Partner code |
|---|---|---|
| **Jan Hidders** | a.j.h.hidders@tudelft.nl | TUD |
| **Kees van der Sluijs** | k.a.m.sluijs@tue.nl | TUE |
| **Paul De Bra** | debra@win.tue.nl | TUE |
| **David Smits** | d.smits@tue.nl | TUE |
| **Erwin Leonardi** | E.Leonardi@tudelft.nl | TUD |
| **Geert-Jan Houben** | g.j.p.m.houben@tudelft.nl | TUD |


# Table of Contents

**SUMMARY** ..... 2

**AUTHORS** ..... 2

**TABLE OF CONTENTS** ..... 2

**TABLES AND FIGURES** ..... 3

**LIST OF ACRONYMS AND ABBREVIATIONS** ..... 3

**1 ARCHITECTURE** ..... 4

**2 FROM AUTHORING MODELS TO GALE** ..... 5

2.1.1 Domain Models ..... 6
2.1.2 User Model ..... 7
2.1.3 Concept Relationship Types ..... 7
2.1.4 Conceptual Adaptation Model ..... 8

**2.2 From DM, CRT and CAM to GALE** ..... 9

2.2.1 DM to GALE Translation ..... 9
2.2.2 Adaptation rules in CRT ..... 9
2.2.3 CAM to GALE Translation ..... 11

**3 GAL ..... 12**

**3.1 Basic Constructs ..... 13**
3.1.1 Preliminary definitions and notation ..... 14
3.1.2 The definition of units and content elements ..... 14

**3.2 Syntax of the Language ..... 15**

**3.3 Semantics of the Language ..... 16**
3.3.1 The resulting unit of a page request ..... 16
3.3.2 The side effects and result of following a navigation link ..... 19

**3.4 User Modelling in GAL ..... 19**

# Tables and Figures

### List of Figures

Figure 1: Revised architecture of Grapple framework (the part that relates to the authoring and the adaptive engine) ..... 5
Figure 2: Illustrative example of a CDM. ..... 6

# List of Acronyms and Abbreviations

| | |
|---|---|
| ALE | Adaptive Learning Environment |
| CAM | Conceptual Adaptation Model |
| DM | Domain Model |
| DoW | Description of Work |
| GAL | GRAPPLE Adaptation Language |
| GALE | GRAPPLE Adaptive Learning Environment |
| GEB | GRAPPLE Event Bus |
| GRAPPLE | Generic Responsive Adaptive Personalized Learning Environment |
| LMS | Learning Management System |
| UM | User Model |

# 1 Architecture

In deliverable D1.1a we described GAL as the intermediate language between the authoring tools and the GALE adaptation engine. However, we decided to change this. GAL is no longer an intermediate language between the CAM and GALE as originally planned, but instead is a Generic Adaptation Language that can be translated from and to the GRAPPLE framework. This alteration of the first year's planning has some practical reasons. GAL is not part of the DoW. Moreover, given that the coupling of CRT's with GAL code and translation of that code to pure GAL and then to GALE turned out to be quite complex, we decided to remove GAL from the critical path of the project. Instead we focused on the core capabilities of the GRAPPLE system by using a direct translation system, and designed the GAL language as an output language of the GRAPPLE framework for exchanging with other adaptive frameworks. Implementation of a compiler from and to GAL, and the proof of concept that GAL can also run on another adaptive engine is work currently planned for the third year of the project. As GAL is not mentioned at all in the Description of Work, this means that an implementation is not scheduled for a third year deliverable. When this work gets done, this will only result in an unofficial update of this deliverable instead of a new deliverable.

This change of the role of GAL also leads to a change in the communication architecture between the authoring models and the GALE adaptive engine, and their relationship with GAL. Figure 1 represents an updated view of this architecture.

One notable difference is the usage of the GRAPPLE Event Bus (GEB) as the communication channel within the GRAPPLE framework. The GEB is used as a uniform communication platform between all GRAPPLE components, and is also used as the communication platform of the GRAPPLE platform with the rest of the world. GEB generically exposes all the available functions to those components that subscribe to it. This allows inside and outside components to be oblivious of the specific components that exist within the framework and it removes the need of developing specific interfaces between every pair of components that might want to communicate. More about GEB can be found in deliverable D7.1b, section 3.2.

The CAM model is sent to the Engine Specific Compiler via the GEB, where it gets directly compiled to Engine-Specific Rules that can be interpreted by the GALE adaptation engine. Note that the GEB is only used in the GRAPPLE framework, for the stand-alone version of the adaptive engine (described in D1.3b) we especially made a direct link from the authoring models to the Engine-Specific Compiler, but that is the only exception of communication not being handled by the GEB in the GRAPPLE framework. Even though the Engine-Specific Compiler is depicted as a separate component in the framework, it is actually implemented as an import module of the GALE engine.

The next section will describe how the authoring models are compiled to executable GALE code, while section 3 contains the formal description of the GAL language.

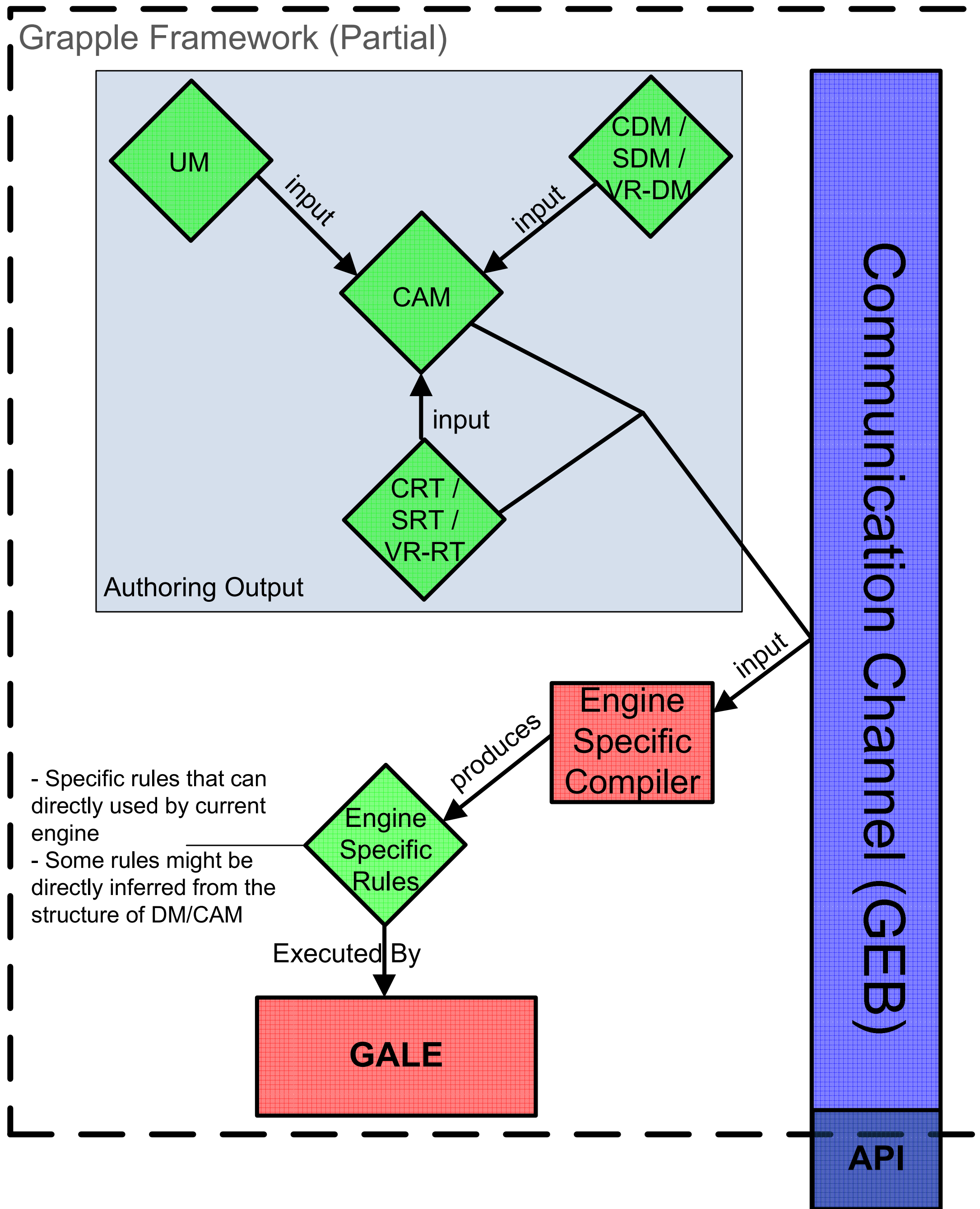


Figure 1: Revised architecture of the GRAPPLE framework (the part that relates to the authoring and the adaptive engine).

## 2 From Authoring Models to GALE

In this section we describe the process of translating a Conceptual Adaptation Model (CAM) definition (as specified in WP3, D3.3a/b) of an application to adaptation 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/b). 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.

Therefore, we will first look at the models that are the output of the authoring process. This is an extension of an earlier description of Deliverable D1.1a.

The authoring environment produces several models:

- Domain models
- User model[1]
- Concept Relationships Types
- Conceptual Adaptation Model

### 2.1.1 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.

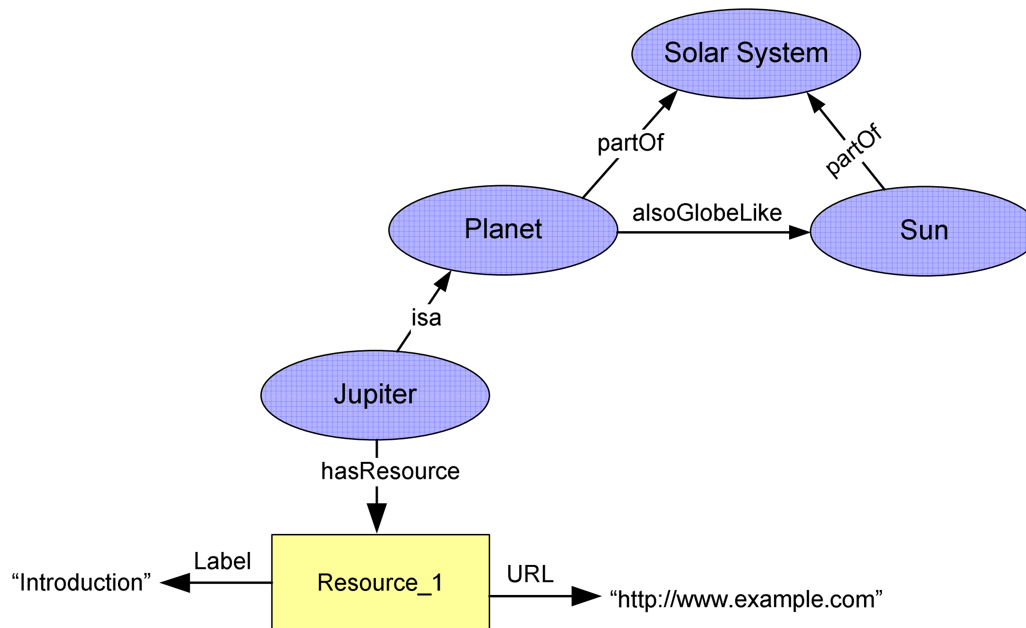


Figure 2: Illustrative example of a CDM.

Figure 2 is an illustrative example of a CDM. Concepts like "Solar System", "Planet", "Sun" and "Jupiter" are connected by relationships like "isa", "partOf" 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

[1] The User Model is not actually produced by the authoring tools, but they specify attributes that must be present in UM, and for which concepts.

a pedagogical attribute that can be referred to during course design. For instance, if a teacher wants every concept to show introductory resources, he can refer to these resources via a label (or a configured attribute) with the value “Introduction”.

The CDM is explained in more detail in D3.1a/b. The SDM is explained in D3.5a/b and VRDM models are explained in D3.4a/b.

### 2.1.2 User Model

The User Model (UM) will typically be 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 that 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).

GRAPPLE’s User Modeling Framework (GUMF) 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 GUMF needs to be contacted. Deliverable D3.2a/b/c describes how references to user model data in GUMF are defined through the CRT authoring tool).

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.

### 2.1.3 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 semantic defining rules. An instance of a 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 to 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 than 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 GALE constructs. The CRT authoring tool offers a "placeholder" for a template piece of adaptation code, which can be written in any language. We will describe the language used to define GALE templates using a number of examples. The following figure is taken from deliverable D3.2b:

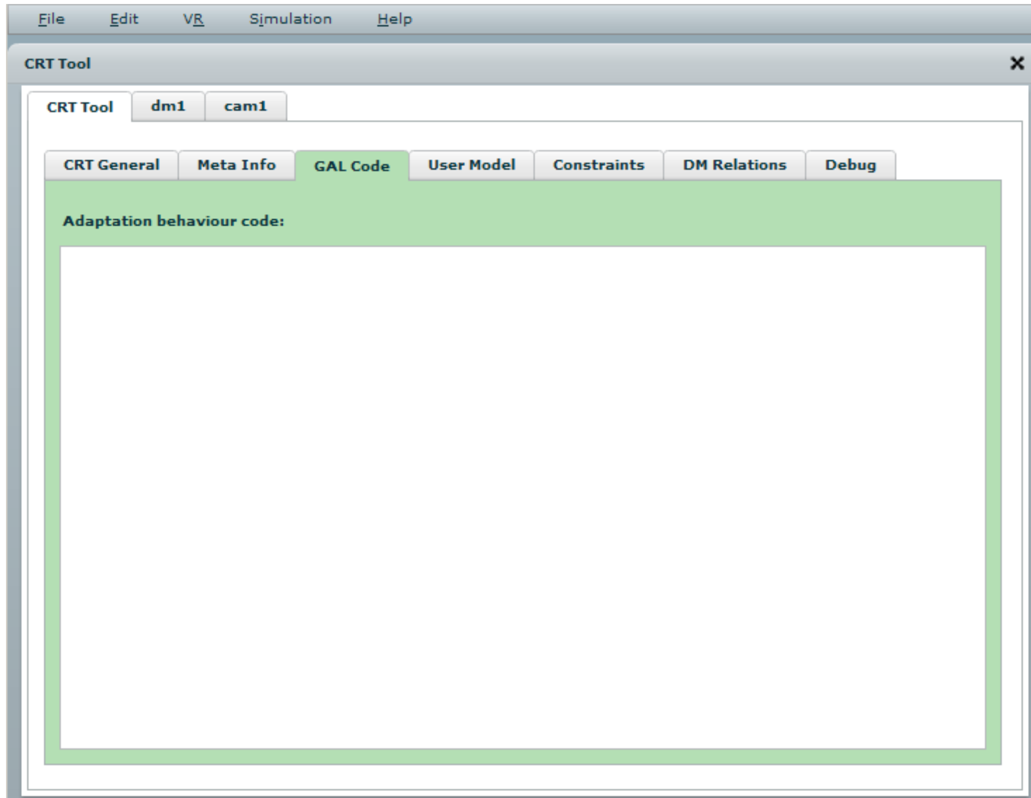


Figure 2: Entering adaptation code in the CRT authoring tool

A concept relationship can be *unary* (between a concept or a set of concepts and itself), binary (between a concept or set of concepts and another concept or set of concepts) or n-ary (between any number of sets of concepts). In the CRT tool only binary relationships can be *directed*. All other relationships are *undirected*. The concept relationship type (CRT) has placeholders for the concepts or sets of concepts. In the CRT tool these are called *sockets*. In the template code we will refer to these placeholders by name (like %source% or %target%). The engine-specific (GALE) compiler translates the template code (with placeholders) into real code by replacing the placeholders by actual concepts, and thereby replicating the code not only for the instances of the CRT but also for the instances of concepts inside the sockets (see below).

### 2.1.4 Conceptual Adaptation Model

The Conceptual Adaptation Model (CAM) specifies the semantic or pedagogical structure of a course or application. It contains multiple sub-models that together specify the navigation and the presentation of the course. Below is 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 for 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/b/c.

## 2.2 From DM, CRT and CAM to GALE

The DM, CRT and CAM to GALE translation requires some basic understanding of the data structures used by GALE. We refer to deliverable D1.3b for details, and only recall the essence here.

A GALE application consists of (named) *concepts* with domain model (DM) *properties* and user model (UM) *attributes*, which can also have *properties*. There can be relationships between individual concepts. Typically a “parent” relationship is used to define a concept hierarchy. The “extends” relationship is used to make concepts inherit properties from other concepts. Concept relationships created with the DM authoring tool automatically become relationships in GALE as well.

### 2.2.1 DM to GALE Translation

The DM authoring tool (deliverables D3.1a/b) is used to define the concepts and concept relationships that together make up the conceptual structure of an application or course. The structure of DM is as follows:

- There are arbitrarily many *concepts*. Each *concept* has an internal *identity* that ensures the concept is globally unique (across applications), but this identity is not used by authors. Each concept has a:
    - *name*: it is up to the author to ensure the name is unique as the name is used in other parts of the authoring tools to refer to concepts. In GALE this becomes the last part of the concept’s *name* (which is a URI containing also the server name and the application name).
    - *description*: this becomes a *description* property in the GALE DM. It is typically not used anywhere in the application (but it can be).
    - *properties*, each having a *name*, *value* and *description*. These become GALE DM properties with a *name* and a *value*. (The *description* is discarded in the translation.)
    - *resources*, each having a *URL* and a list of *properties* with a *name* and *value*. Deliverable D1.2b (Sections 2.2.1 and 2.2.2) describe the different ways in which the *resources* can be translated into GALE code. The translation results in a *resource* attribute in the GALE UM. (It is in the UM because it has conditions to select a resource from the list.)
- Concepts are linked through binary *relationships*. Each relationship has a *type*, a *source* concept and a *destination* concept. Each relationship becomes a GALE DM *relation* with a *name* (corresponding to what the DM tool calls the *type*), with an *inconcept* (the *source*) and an *outconcept* (the *destination*). GALE has an adaptation language construct to find all concepts having a specific relation *to* a given concept or all concepts to which a given concept has a specific relation.
  `Titan->(parent)` gives the list of concepts to which “Titan” has a “parent” relation.
  `Jupiter<-(parent)` gives the list of concepts which have a “parent” relation to “Jupiter”.

### 2.2.2 Adaptation rules in CRT

The definition of a concept relationship type contains some “hints” about the meaning of a CRT and the used DM and UM properties, as well as a bit of template code entered into the form shown in Figure 2 above.

- A CRT is a *unary*, *binary* or *n-ary* connection between sets of concepts. A *unary* or *n-ary undirected* CRT has an arbitrary number of “sockets”, and a *binary directed* CRT has two “sockets”, commonly referred to as “source” and “target” (but they can have arbitrary names). The sockets are referred to in the adaptation rule code as shown below.
- A CRT also has a “User Model” part. It is used to indicate that concepts used in a socket expect or require some UM attribute(s). The CRT tool lets you specify that attributes have the type “integer”, “float”, “string” or “boolean”. The compiler will ensure that the concepts in GALE have the corresponding UM attributes of the appropriate type. However, it is possible to override these definitions when writing the adaptation rules as shown below.

- A CRT can be automatically associated with some DM relationship types. This implies that instances of the CRT will be added to the CAM automatically. As a result, adaptation rules for the CRT will automatically be instantiated for the instances of the relationships in the DM.

In the "GALE code" tab template adaptation rules are written that are instantiated into "real" adaptation rules when the CAM is "deployed" to GALE. We illustrate the use of template adaptation rules with a few examples, to show the differences between GALE adaptation rules and adaptation rule templates.

In GALE there are three places where adaptation rule code is used: in *event* code, in *default* code and in pages. We will ignore the code that occurs in pages as it is not part of the translation of the conceptual structure of an application. The code is actually Java code, with some "shorthand" notation to refer to concepts, attributes and properties. The details are in deliverable D1.3b. We mainly illustrate the extensions here that are used to refer to sockets of the CRT.

- *default code*. For every UM attribute there is a *default* value. This can be defined as a constant, but it can also be a GALE expression, which is actually a Java expression. An example of such an expression could be:

  ```
  ${Earth#knowledge}>80 && ${Moon#knowledge}>80
  ```

  This example is a Boolean expression that evaluates to true when the knowledge for both the concept Earth and the concept Moon is greater than 80.
  In the CRT we associate the expression to the *default* value of some attribute of one of the sockets, and we can also use a socket in the expression. A prerequisite relationship could have the following GALE template code:

  ```
  %target% { #suitability & !`${%source%#knowledge}>80` }
  ```

  When this CRT is instantiated (in a CAM) this rule will first of all be replicated for each concept in the `%target%` socket. In the instantiation the `${%source%#knowledge}>80` expression will be replicated for each concept in the `%source%` socket, and these expressions are combined with the logical "and" operator (because the template says `&`). The combined expressions are considered to be executable code and not just a string value because of the use of '!' in the template code.
  So if "Earth" and "Moon" are both prerequisites for a concept X then the *default* code for the `suitability` attribute of X will be the expression shown above. Should there be multiple CRT instantiations defining a default expression for an attribute the expressions must be Boolean and will always be combined with "and".

- *event code*. For every concept and for every UM attribute there is *event code*. The event code for a concept is executed when the end-user accesses (requests) that concept. The event code for an attribute is executed when the value of that attribute changes. The *event code* is a Java statement, or a sequence of statements. It is specified in the CRT tool as a string. Instantiation is done by concatenating the strings. The strings are delimited by back quote characters (`) because the Java statements themselves may contain regular single (') or double (") quotes. Here is an example of template event code:

  ```
  %self% {

     event +
        `if (${#suitability})
            #{#own_knowledge, 100};
        else if (${#own_knowledge} < 35)
            #{#own_knowledge, 35};`

     #own_knowledge {
        event +
        `#{#knowledge, #knowledge + changed.diff/(${<-(parent)}.length+1)};`
     }

  }
  ```

  To understand this piece of code we need to know that in GALE it is customary to have the attributes `knowledge` and `own_knowledge`. This is done to distinguish between the knowledge contribution from reading a page and the knowledge contribution from reading pages of child-concepts in the hierarchy. For each attribute (in this case the attribute `own_knowledge`) GALE maintains a property `changed` with three fields: `old`, `new` and `diff` which is used to keep track of the latest change to the value of the attribute. (`diff` only exists for numerical attributes, not for Booleans and strings.)

The code in this example is replicated for each concept in the %self% socket. When the user accesses a concept the `own_knowledge` value is updated, depending on the suitability of the concept. The change to the `own_knowledge` attribute triggers the second bit of event code: the `knowledge` attribute is updated to take into account the knowledge gain from studying the concept.

Note: instead of `#own_knowledge { … }` in the example we could have written `#own_knowledge:Integer { … }` to override the data type that was specified in the User Model tab in the CRT tool. The new type must be the name of a Java class (Integer in the example).

For a slightly more complex example we consider the “knowledge propagation” (CRT) which is used to propagate knowledge up the concept hierarchy. This is typically associated with the “parent” relationship. We assume that the sockets for the knowledge propagation CRT are named “child” and “parent” (rather than “source” and “target” which is the default in the CRT tool).

```
%child% {

   #knowledge {
      event + `#{%parent%#knowledge, %parent%#knowledge +
           changed.diff / (${->(parent)-<(parent)}.length+1)};`
   }

}
```

Normally when using the knowledge propagation CRT we expect concepts to have a single parent. However, the template adaptation rule does not make that assumption. For each child of a concept the `knowledge` attribute gets this event code which is instantiated once for every parent of the concept. Note that the `parent` relationship used in `${->(parent)-<(parent)}` is unrelated to the `%parent%` socket name.

There is one additional issue in the translation of CRTs to GALE: the definition of UM attributes. There are three ways (each one overriding the previous) to determine which attribute concepts must have:

- When an attribute is used in template code and is not otherwise defined the compiler will try to determine the data type for the attribute from its use. This is not the recommended approach!
- Attributes used in a socket should be declared in the CRT tool’s User Model part. The types “integer”, “float”, “string” and “boolean” are understood by the CRT tool and the compiler (which translates them to Java classes).
- Attributes may be typed in the template code by adding a : and class behind their name. This class takes precedence over any type declared in the CRT User Model part. Any Java class may be used (so it is not limited to Integer, Float, String and Boolean). The “default” expression for a prerequisite can be rewritten as:

  ```
  %target% { #suitability:Boolean & !`${%source%#knowledge:Float}>80` }
  ```

  and then it implies that the `suitability` attribute of the target concepts must be `Boolean` and the `knowledge` attribute of the source must be of the class `Float`.

### 2.2.3 CAM to GALE Translation

In the conceptual adaptation model (CAM) of an application the concepts from one or more domain models are connected by relationships of zero or more CRTs. The CAM defines which instances of the CRTs exist in the application. The CAM authoring tool produces a file that contains not only the CAM but also the DMs and CRTs that are used in the CAM. As a result a single file is the input for the CAM to GALE compiler.

Considering the description in Sections 2.4.1 and 2.4.2 of how DM parts are translated to GALE code and how CRTs contain pieces of code that are instantiated for the concepts that occur in the sockets in a CAM the whole translation becomes a fairly straightforward process:

- For each concept in DM a GALE concept is defined. Section 2.4.1 describes how properties are translated. (Deliverable D1.2b adds an explanation of how resources are translated.)
- For each concept relationship in DM a GALE relation is created. These relations are typically used to “traverse” them in order to find for instance children and parent concepts.
- For each CRT used in the CAM the template adaptation rules are translated to “real” adaptation rules and inserted in the appropriate place in the structure of the concepts that appear in the sockets.

Note that a CAM may refer to more than one DM. A typical use of this is to make concepts *inherit* parts from generic concepts in another DM. GALE is delivered together with a standard “admin” application. In that “admin” application there is a concept “_layout”, with an attribute “layout”. The default value for this attribute is:

```
"<struct cols=\"20%;*\"><view name=\"static-tree-view\"/><content/></struct>"
```

The layout attribute is used by GALE to decide on a presentation structure for a concept. Concepts that “extend” the “_layout” concept inherit this default layout. This default layout shows a navigation menu on the left and the content of the concept (page) on the right. In order for a concept “Milkyway” to “extend” the “_layout” the following relation is needed:

```
<relation name="extends">
   <inconcept>gale://grapple-project.org/Milkyway/Milkyway</inconcept>
   <outconcept>gale://gale.tue.nl/admin/_layout</outconcept>
</relation>
```

Since the “_layout” concept has only one attribute (layout) and no properties, the only effect of this “extends” relation is that the concept (Milkyway) gets a “layout” attribute with the same default value as the “_layout” concept from the “admin” application.

# 3 GAL

As explained in section 1 we decided to move GAL away from the critical path in GRAPPLE. However, that doesn’t mean that we don’t value the original goals of GAL anymore. In GRAPPLE we aim for interoperability of user models, as well as interoperability between an adaptive engines and LMSs and other possible components. GAL extends that vision by making the adaptivity itself interoperable between authoring environments and adaptive engines. Given that GAL is not part of the Description of Work, we now treat it as a extra in the GRAPPLE framework. We don’t promise the implementation of a compiler from and to GAL for the third GRAPPLE year, nor the implementation of compilers for other engines. However, we have done enough work to present the formal basis of the complete GAL language. We certainly do plan to construct these compilers, either during the third GRAPPLE year or after the project. As we think this future GRAPPLE addition can be quite valuable we still include our progressing work on this subject in this deliverable, even though other work has our main focus for now.

Given the status change of GAL within the GRAPPLE project, this also resulted in a change of the GAL position in the architecture. Figure 3 represents the architecture of the GRAPPLE framework that includes GAL (note that it is an extension of Figure 1). Engine Specific Adaptation Rules can be translated into engine-independent GAL code, via the Engine Independent Compiler. This compiler is also envisioned to be part of the GALE adaptation engine, as an export module. This GAL code can be executed by adaptation engines other then GALE. Vice versa, these adaptation engines can provide GAL code that in turn can be compiled back to engine specific rules so that these can be used by the GALE engine. In this way, we changed the architecture without actually throwing away the versatility we had with using the GAL as a middle language between the authoring tools and the adaptive engines. It also means that an authored GALE application in the GRAPPLE Authoring Tools (GAT, see deliverables D3.1b, D3.2b and D3.3b) can be executed by different adaptive engines by transforming the authoring models into GALE specific code and then transforming this generated code into GAL. The other way around, by allowing decompiling GALE code into GAT models we envision facilitating representation and editing of adaptive programs authored for external adaptive engines.

In this section we give a formal description of the GAL language. For an informal description of GAL, with examples of GAL code, refer to deliverable D1.1a. The following formal description is completely revised and matured compared with the (semi-) formal description found in deliverable D1.1a.

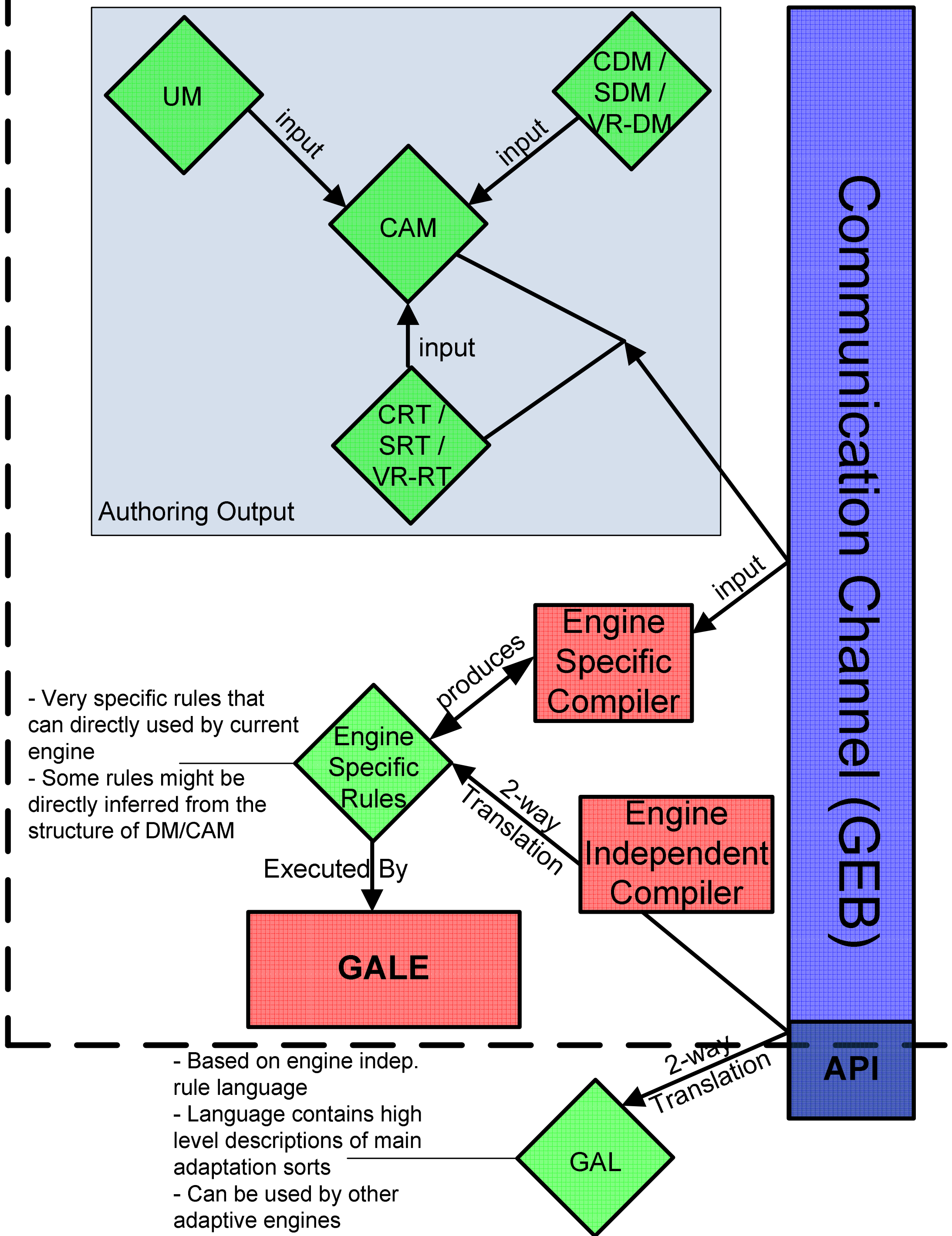


Figure 3: Revised architecture of the GRAPPLE framework including GAL

## 3.1 Basic Constructs

The purpose of GAL is to describe the navigational structure of a web application and how this adapts itself to the actions of the users. The central concept is that of a unit which is an abstract representation of a page, as it is shown to a certain user after a certain request. Basically it is a hierarchical structure that contains the content and links that are to be shown, and also updates what is to be executed when the user performs certain actions such as requesting the page or leaving the page. All data from which these units generate is

assumed to be accessible via a single RDF query endpoint, and the updates are also applied via this endpoint.

### 3.1.1 Preliminary definitions and notation

For the purpose of formalizing the syntax and semantics of GAL we postulate the following sets and constants: *ET* (event types), *DB* (database instances, which are queried and updated through the RDF query endpoint), *VQ* (value queries over the RDF query endpoint that retrieve a single value), *LQ* (list queries that retrieve lists of bindings from the RDF query endpoint), *UQ* (update queries that update the RDF query endpoint), *A* (attribute names), *V* (values, which includes RDF literals and URIs), *EL* (element labels), *UT* (unit types, which we here assume to be a URI consisting of a namespace and a local identifier), *EN* (element names), *X* (variable names containing a distinguished variable **user**), *DC* (domain concepts). We use the special constant $\perp$ which is not in any of the postulated sets, and the special constants **"gal:_top"**, **"gal:_self"** and **"gal:_parent"** which are not in *UL*. We define the set *B* to be the set of bindings, i.e., partial functions $b : X \rightharpoonup V$ that map variables to value. For partial and total functions $b_1$ and $b_2$ we define their combination denotes as $b_1\#b_2$ such that $(b_1\#b_2)(x)=b_1(x)$ if $b_1(x)$ is defined and $b_2(x)$ is undefined, $(b_1\#b_2)(x)=b_2(x)$ if $b_2(x)$ is defined and $(b_1\#b_2)(x)$ is undefined otherwise. We generalize the # operator such that $(b_1\#\ldots\#b_n)=(b_1\#\ldots\#b_{n-1})\#b_n$ for $n>0$ and $(b_1\#\ldots\#b_n)=\emptyset$ for $n=0$.

Additionally we introduce the following notation: $\mathcal{L}(S)$ the set of finite lists with elements from *S*. Lists are denoted as [1,1,2], and list concatenation as $L_1 \cdot L_2$. We write $x \in L$ to denote that an element *x* occurs in list *L*. We use list-comprehension syntax, such as $\left[ f(x,y) \mid x \in L_1,\ y \in L_2 \right]$ to construct lists and bags with the usual semantics.

For queries we assume that they contain variables, which can be replaced with a value. Concretely we assume the queries are expressed in SPARQL where variables are used in the graph patterns and are preceded with "? " or "$". For such queries *q* we let *q*[*b*] denote the query that is obtained when the variables in *q* for which *b* is defined are replaced with their value in *b*. Strictly speaking for SPARQL queries this can result in an invalid query if variables in the SELECT clause are replaced, but we ignore this for the moment. We assume that for all queries *q* a semantics is given, viz., $[[q]]:DB \rightarrow V$ for value queries $q \in VQ$, $[[q]]:DB \rightarrow \mathcal{L}(B)$ for list queries $q \in LQ$ and $[[q]]:DB \rightarrow DB$ for update queries. We also assume that it is stored in the database which values denote an instance of a domain concept. For a database instance $db \in DB$ and domain concept $dc \in DC$ the set of instances of *dc* is given by a set $[[dc]]_{db} \subseteq V$. Concretely we will assume this set is retrieved by the SPARQL query `SELECT ?v WHERE { ?v rdf:type dc } .`

We now define the syntax of value expressions. We assume that the following non-terminals already are defined: <value> describes *V*, <val-query> describes *VQ*, <list-query> describes *LQ*.

<val-expr> ::= <value> | "$"<var-name> | "("<val-expr>**"."**<val-expr>")" |
"[" **"gal:query"** <val-query> "]" |
"[" **"gal:if"**<list-query> **";"** (**"gal:then"**<val-expr>**";"**)? (**"gal:else"**<val-expr>**";"**)? "]".

The set of all value expressions, i.e., those belonging to <val-expr>, is denoted as *VE*. A value expression binding is a partial function $be : X \rightharpoonup VE$.

### 3.1.2 The definition of units and content elements

We first define het set *NL* of navigation links as they are represented in a unit. It contains exactly all tuples of the form **ln**(*ut*,*b*,*q*,*be*,*t*) with a unit type $ut \in UT$, a value binding $be : X \rightharpoonup VE$, a query $q \in LQ \cup \{\perp\}$, a value expression binding $be : X \rightharpoonup VE$ and a link target $t \in N \cup$ { "**gal:_top**", "**gal:_self**", "**gal:_parent**" } .

The intuition of the definition of navigation links is as follows. The unit type *ut* indicates to what type of unit the link will lead, the value binding *b* specifies the fixed link parameters that are passed with the page request, the query *q* computes a binding that defines a set of additional parameters which are computed at the time the link is followed, the value expression binding *be* specifies even more additional parameters that are computed at navigation time by given value expressions for individual variables and finally the target *t*

indicates which part of the current page will be replaced by the result of the request that is generated by navigating the link.

We define the sets *U* of *proper units* and *E* of *content elements* as the smallest sets such that (a) *U* contains

1. all *units*, i.e., tuples $\mathbf{un}(C,h,nl)$ where $C \in \mathcal{L}(E)$ is a finite list of content elements, a relation $h \subseteq ET \times UQ$ and the navigation link $nl \in NL \cup \{\perp\}$,

and (b) that *E* contains

1. all *attributes*, i.e, pairs $\mathbf{at}(n,l,v)$ with name $n \in EN \cup \{\perp\}$, label $l \in EL \cup \{\perp\}$ and value $v \in V$,
2. all *unit containers*, i.e., tuples $\mathbf{uc}(n,l,u)$ with name $n \in EN \cup \{\perp\}$, label $l \in EL \cup \{\perp\}$, and unit $u \in U$,
3. all *unit lists*, i.e., tuples $\mathbf{ul}(l,UL)$ with label $l \in EL \cup \{\perp\}$, and unit list $UL \in \mathcal{L}(U)$.

Note that since *U* and *E* are defined as the smallest sets that satisfy the properties (a) and (b), the mutual recursion is finite, i.e., the trees that are defined by mutually nested units and content elements are of finite height.

The intuition behind the concepts of units and content elements is as follows. A unit consists of (1) its content *C*, (2) the relation *h* that indicates which update queries are associate with which event type, i.e., are executed when these events occur, and (3) the navigation link *nl* that will be associated with the representation of the unit.

The intuition behind the definition of content elements is as follows. Attributes represent basic content and consist of (1) the name *n*, which is mostly there to indicate the meaning of the attributes, (2) the label *l*, which will be used in the presentation on the page and (3) the value *v*, which represents the content of the attribute. Unit containers represent nested units and consist of (1) the name *n*, which is used to identify the particular position of this nested unit, (2) the label *l*, which is used in the representation of the nested unit and (3) the nested unit *u*. The unit sets and unit lists represent a nested collection of units and consist of (1) a label *l*, which is used the representation of these collections, and (2) the bag or list of units that is the collection.

## 3.2 Syntax of the Language

We assume that the following non-terminals already are defined: <unit-type-name> describes *UT*, <var-name> describes *X*, <domain-concept> describes *DC*, <literal> describes RDF literals, <val-query> describes *VQ*, <list-query> describes *LQ*, <update-query> describes *UQ*, <unit-label> describes *UL*, <unit-name> describes *UN* and <event-type> describes *ET*.

The set of GAL expressions is then defined by the following syntax:

<gal-expr> ::= ( <unit-decl> "**.**")* .

<unit-decl> ::= <unit-type-name> <unit-type-def> .

<unit-type-def> ::= "**a**" "**gal:unit**" "**;**" <input-var>* <content> <link-expr>? <on-event-expr>*

<content> ::= "**gal:contains**" "(" ( "[" ( <attr-def> | <subunit-expr> | <list-unit-expr> ) "]" )* ")" .

<input-var> ::= **"gal:hasInputVariable"** "[" **"gal:varName"** <var-name> **";"**<br>"**gal:varType**" <domain-concept> "]" ";" .

<on-event-expr> ::= "**gal:onEvent**" "[" "**gal:eventType**" ("**gal:onAccess"** | **"gal:onExit"**) ";"<br>"**gal:update**" <update-query> "]" ";".

<attr-def> ::= "**a**" "**gal:attribute**" ";" <name-spec>? <label-spec>? "**gal:value**" <val-expr> .

<name-spec> ::= "**gal:name**" <elem-name> ";" .

<label-spec> ::= "**gal:label**" <val-expr> ";" .

<subunit-expr> ::= "**a**" "**gal:subUnit**" <name-spec>? <label-spec>? <unit-constr> ";" .

<list-unit-expr> ::= "**a**" "**gal:listUnit**" <label-spec>? <unit-constr> ";" .

<link-expr> ::= "**gal:hasLink**" "[" <link-constr> <target-spec>? "]" ";" .

<link-constr> ::= "**gal:refersTo**" <unit-type-name> ";" <bind-list> .

<target-spec> ::= "**gal:targetUnit**" ( <unit-name> | "**gal:_top**" | "**gal:_self**" | "**gal:_parent**") ";" .

<bind-list> ::= <query-bind>? <var-bind>* .

<query-bind> ::= "**gal:hasQuery**" "[" <list-query> "]" ";" .

<var-bind> ::= "**gal:assignVariable**" "[" "**gal:varName**" <var-name> ";"
"**gal:value**" <val-expr> "]" ";" .

<unit-constr> ::= "**gal:refersTo**" "[" <unit-type-def> "]" ";" <bind-list> .

We let each non-terminal also denote the set of expressions associated with it by the syntax, so <gal-expr> is the set of all GAL expressions.

We extend the syntax with some syntactic sugar: inside a <unit-constr> the fragment "[" <unit-type-def> "]" can be replaced by a <unit-type-name> if in the total expression this is associated with this <unit-type-def>. In other words, if a <unit-type-name> occurs in the <unit-constr> after the **"gal:refersTo"** then it is interpreted as the associated "[" <unit-type-def> "]".

## 3.3 Semantics of the Language

The semantics of a GAL expression is described in two parts. In the first part the semantics of a GAL expression describes how a page request results in a unit. In the second part it is defined how a total navigation step is performed, i.e., what are the updates to the database and the resulting current unit in the browser if in a certain unit a certain navigation link is followed.

### 3.3.1 The resulting unit of a page request

The semantics of the syntax is defined by giving functions for each non-terminal, for example, for expressions *ge* in <gal-expr> we define a partial function that maps a user identifier (the requesting user), a unit type, a binding and a database to a unit, denoted as $[\![ge]\!] : V \times UT \times B \times DB \rightharpoonup U$. We overload this notation for all non-terminals, but not in all cases with the same signature.

Before we proceed with the semantics of GAL expressions, we first define the semantics of <val-expr> defined earlier. We recall the syntax rule:

<val-expr> ::= <value> | "$" <var-name> | "(" <val-expr> "." <val-expr> ")" |
"[" "**gal:query**" <val-query> "]" |
"[" "**gal:if**" <list-query> ";"
("**gal:then**" <val-expr> **";"**)? (**"gal:else"** <val-expr> **";"**)? "]" .

For $ve \in$ <val-expr> we define $[\![ve]\!] : B \times DB \rightharpoonup V$ as follows. If $ve = v \in V$ then $[\![ve]\!](b,db) = v$. If $ve$ = "$"$x$ with $x \in X$ then $[\![ve]\!](b,db) = b(x)$ if $b(x)$ is defined and undefined otherwise. If $ve$=**("**$ve_1$**"."**$ve_2$**")"** then, if $[\![ve]\!](b,db)$ is a string $s_1$ and $[\![ve]\!](b,db)$ is a string $s_2$, then $[\![ve]\!](b,db)$ is the string concatenation of $s_1$ and $s_2$, otherwise the result is undefined. If $ve$="[""**gal:query**"$vq$"]" then, if $q[b] \in VQ$ then $[\![ve]\!](b,db) = [\![q[b]]\!](db)$, and undefined otherwise. If $ve$=**"[""gal:if"**$q$**";""gal:then"**$ve_1$**";""gal:else"**$ve_2$**";""]"** then $[\![ve]\!](b,db) = [\![ve_1]\!](b,db)$ if $[\![q[b]]\!](db) \neq []$, and $[\![ve_2]\!](b,db)$ otherwise. If $ve_1$ or $ve_2$ are not specified in *ie* then $[\![ve_1]\!](b,db)$ or $[\![ve_2]\!](b,db)$ are presumed to be undefined.

We now proceed with defining the semantics for GAL expressions. We first recall the relevant syntax rules, and then give the corresponding semantics.

<gal-expr> ::= ( <unit-decl> **“.”**)* .

<unit-decl> ::= <unit-type-name> <unit-type-def> .

For *ge* ∈ <gal-expr> we define [[*ge*]]: $V \times UT \times B \times DB$ as follows. Assume that $ge = ud_1$**“.”**…$ud_n$**“.”**, with each $ud_i = ut_i\ udf_i$. Let *j* be the smallest number such that $ut_j = ut$ then $[\![ge]\!](v,ut,b,db) = [\![udf_j]\!](\ b\#\ \{(\mathbf{user},v)\},db)$, and $[\![ge]\!](v,ut,b,db)$ is undefined if such a *j* does not exist.

<unit-type-def> ::= “**a**” “**gal:unit**” “;” <input-var>* <content> <link-expr>? <on-event-expr>*

<content> ::= “**gal:contains**” “(” ( “[” ( <attr-def> | <subunit-expr> | <list-unit-expr> ) “]” )* “)” .

<input-var> ::= “**gal:hasInputVariable**” “[” “**gal:varName**” <var-name> “;”
“**gal:varType**” <domain-concept> “]” “;” .

<on-event-expr> ::= “**gal:onEvent**” “[” “**gal:eventType**” (“**gal:onAccess**” | “**gal:onExit**”) “;”
“**gal:update**” <update-query> “]” “;”.

For *ud* ∈ <unit-type-def> we define $[\![ud]\!]: B \times DB \rightharpoonup U$ as follows. Let us assume that
*ud*=**“a”“gal:unit”“;”**$iv_1 \ldots iv_n$ *cont le* $eve_1 \ldots eve_p$, with each
$iv_i$=**“gal:hasInputVariable”“[”“gal:varName”**$x_i$**“;”“gal:varType”**$dc_i$**“]”“;”**,
*cont*=**“gal:contains”“(”“[”**$ce_1$**“]”**…**“[”**$ce_m$**“]”“)”**,
each $ce_j \in$ <attr-def>∪<subunit-type-def> ∪<list-unit-expr>, *le* ∈ <link-expr>, and each
$eve_k$=**“gal:onEvent”“[”“gal:eventType”**$et_k$**“;”“gal:update”**$uq_k$**“]”“;”**. We say that the binding $b \in B$ is correct under database $db \in DB$ if for all $1 \le i \le n$ it holds that $b(x_i)$ is defined and $b(x_i) \in [[dc_i]]_{db}$. If *b* is not correct under *db* then $[\![ud]\!](b,db)$ is undefined, and otherwise $[\![ud]\!](b,db) = \mathbf{un}(C,h,nl)$ where
$C = [\![ce_1]\!](b',db) \bullet \ldots \bullet [\![ce_m]\!](b',db)$, $h = \{(et_1,uq_1),\ldots,(et_p,uq_p)\}$, $b' = \{(x,v) \in b \mid x \in \{\boldsymbol{user}, x1, \ldots, xn\}\}$
and $nl = [\![le]\!](b',db)$. If *le* is missing in *ud* then the result is the same except that $nl = \bot$.

Note that the specified input variables are used to restrict the bindings under which the nested content expressions evaluation. Also note that these expressions are assumed to result in a list, which will either be a singleton list containing a content element or an empty list if there no well-defined resulting content element.

<attr-def> ::= **“gal:hasAttribute”“[”** <name-spec>? <label-spec>? **“gal:value”** <val-expr> **“]”**.

<name-spec> ::= **“gal:name”** <elem-name> **“;”** .

<label-spec> ::= **“gal:label”** <val-expr> **“;”** .

For *ad* ∈ <attr-def> we define $[\![ad]\!]: B \times DB \to \mathcal{L}(E)$ as follows. If

*ad* = “**gal:hasAttribute**” “[” “**gal:name**”*an*“;” “**gal:value**”“[”*ve*“]”“]”, then $[\![ad]\!](b,db) = [\mathbf{at}(an, [\![ve]\!](b,db),])$ if $[\![ve]\!](b,db)$ is defined and [] otherwise. If *an* is not present then we assume that $an = \bot$.

<subunit-expr> ::= “**gal:hasSubUnit**” “[”<name-spec>? <label-spec>? <unit-constr> “]” “;” .

For *su* ∈ <subunit-expr> we define $\llbracket su \rrbracket : B \times DB \to \mathcal{L}(E)$ as follows. If *su* = “**gal:hasSubUnit**” “[” *ls ns uc*“]” “;” with *ls*=“**gal:label**”*ul*“**;**”, *ns*=“**gal:name**”*un*“**;**” and *ud* ∈ <unit-descr> then $\llbracket su \rrbracket$(b,bd) = [**uc**($ul,un,u_1$)] if $\llbracket uc \rrbracket(b,db)$ = [$u_1,\ldots,u_n$] and [] otherwise. If *ls* or *ns* is not present in *su* then the result is the same except that *ul* = ⊥ or *un* = ⊥.

<list-unit-expr> ::= “**gal:hasListUnit**” “[” <label-spec>? <unit-constr> “]” “;” .

For *lue*∈<list-unit-expr> we define $\llbracket lue \rrbracket : B \times DB \to \mathcal{L}(E)$ as follows. If *sue*=“**gal:hasListUnit**” “[”*ls uc*“]” “;” with *ls*=“**gal:label**”*ul* “**;**” and *uc* ∈ <unit-constr> then $\llbracket lue \rrbracket(b,db)$=[**ul**($ul,\mathcal{L}$[[*uc*]]($b,db$))]. If ls is not present then we assume that *ul* = ⊥.

<link-expr> ::= “**gal:hasLink**” “**[**” <link-constr> <target-spec>? “**]**” “**;**” .
<link-constr> ::= “**gal:refersTo**” <unit-type-name> “**;**” <bind-list> .
<target-spec> := “**gal:targetUnit**” ( <unit-name> | “**gal:_top**”| “**gal:_self**”| “**gal:_parent**”) “**;**” .

For *le* ∈ <link-expr> we define $\llbracket le \rrbracket : B \times DB \to NL$ as follows. Assume that *le* = “**gal:hasLink**” “[”*lc ts* “]” “;” with *lc*=“**gal:refersTo**”*ut*“**;**”*bl* and *bl*=*q* $vb_1 \ldots vb_n$ with *q*∈<query> and each $vb_i$=“**gal:assignVariable**”“**[**”“**gal:varName**”$x_i$“**;**”“**gal:value**”$ve_i$“**]**”“**;**” with $x_i$∈<var-name> and $ve_i$∈<val-expr>. Also assume that *ts*=“**gal:targetUnit**”*tu*“**;**” then $\llbracket le \rrbracket(b,db)$ = **ln**($ut,b,q,be,tu$) with $be = \{(x_1, ve_1),\ldots,(x_n, ve_n)\}$. If *ts* is missing in *le* then the result is the same except that *tu*=“**gal:_top"**.

Note that in a link expression the interpretation of the binding list *bl* is different from that in subunit, list-unit and set-unit expressions. Rather than computing one or more bindings it is more or less stored as is such that it can be computed when the link is followed.

<bind-list> ::= <query-bind>? <var-bind>* .

For *bl*∈<bind-list> we define [[*bl*]]:$B \times DB \to \mathcal{L}(B)$ as follows. Let *bl*=$be_1 \ldots be_n$ with $be_i$∈<query-bind>∪<var-bind>. Then $\llbracket bl \rrbracket(b, db) = [b_1 \# \ldots \# b_m \mid b_1 \in \llbracket be_1 \rrbracket, \ldots, b_n \in \llbracket be_n \rrbracket]$.

<query-bind> ::= “**gal:hasQuery**” “**[**” <list-query> “**]**” “**;**”

For *qb* ∈ <query-bind> we define $\llbracket bl \rrbracket : B \times DB \to \mathcal{L}(B)$ as follows. If

*qb*=“**gal:hasQuery**” “[” *q* “]” “;” then $\llbracket qb \rrbracket$ ($b,db$)= $\llbracket q[b] \rrbracket$ ($db$) if *q*[*b*]∈*LQ* and [] otherwise.

<var-bind> ::= “**gal:assignVariable**” “[” “**gal:varName**” <var-name> “**;**”
“**gal:value**” <val-expr> “]” “;” .

For *vb* ∈ <var-bind> we define $\llbracket be \rrbracket : B \times DB \to \mathcal{L}(B)$ as follows. If

*vb* = “**gal:assignVariable**” “[” “**gal:varName**”*x* “;” “**gal:value**”*ve* “]” “;” then $\llbracket vb \rrbracket(b,db)$=[($x$, $\llbracket ve \rrbracket(b,db)$)] if $\llbracket ve \rrbracket(b,db)$ is defined and [] otherwise.

<unit-constr> ::= **“gal:refersTo” “[”** <unit-type-def> **“]” “;”** <bind-list> .

For *uc*∈ <unit-constr> we define $[\![uc]\!] : B \times DB \rightarrow \mathcal{L}(U)$. Assume that

*uc*= “**gal:refersTo**” “[”*ud*“]” “;”*be* and that $[\![be]\!](b, db) = [b1, \ldots, bn]$ then $[\![uc]\!](b, db) = [u \mid b_i \in [\![be]\!](b, db)$, $u = [\![ud]\!](b'_i .db)]$ with $b'_i = b \# b_i \# \{(\boldsymbol{user}, b(\boldsymbol{user}))\}$.

### 3.3.2 The side effects and result of following a navigation link

In this section we (only informally) describe the navigation process that is defined by a certain GAL program. At the core of the navigation semantics is the previously described computation of the unit that is the result of the request of a user. The resulting unit then is presented to the user and becomes the current unit, i.e., the unit that is currently presented to the user. In addition all the update queries associated with the gal:onAcess events in it at any nesting depth are applied to the RDF store in some arbitrary order.

If subsequently the user navigates to an external link then all the update queries associated in it with the gal:onExit event at any nesting depth are applied to the RDF store in some arbitrary order. If the user follows an internal link **ln**(*ut*,*b*,*q*,*be*,*t*) in the current unit, then the following happens. First the target unit in the current unit is determined as follows. If the target is a unit name then this is the unit in the first unit container with that name. If it is **“gal:_self"** then it is the containing unit, i.e., the unit in which the link is directly nested. If it is **“gal:_parent"** then it is the parent unit, i.e., the unit which has as a content element a unit container, unit set or unit list that refers to the containing unit. If it is **“gal:_top"** then it is the root unit of the current unit. If in any of the preceding cases the result is not well defined then the target unit is the root unit of the current unit. When the target unit is determined then all **“gal:onExit”** events that are directly or indirectly nested in it are applied to the RDF store in some arbitrary order.

Then, the binding that describes the parameters is computed as follows: the binding *b* is combined with the first binding in the result of *q*[*b*] and this in turn is combined with the binding that is obtained when evaluating be for binding *b* and the current database instance. The resulting binding is sent together with the specified unit type as a request. Then, if this request results in a unit, the **“gal:onAccess”** update queries in this unit are executed as described before. In the final step the new current unit is the old current unit but with the target unit replaced with the resulting unit of the request.

## 3.4 User Modelling in GAL

GAL describes the navigational structure of an adaptive web application. This description has a page-based granularity. This means that it is not possible in GAL to make statements that hold generally about the complete GAL Web application. However, it might be important to specify general ‘rules’ about the user model, think for instance about knowledge propagation. Consider for example the following knowledge propagation rule in an ALE:

```
Trigger(X.Knowledge)
If ( X.knowledgeChange>0 && X.Knowledge>60 && relatedTo(X,Y) )  ->
                            { Y. knowledgeChange = ⌊0.2* X.knowledgeChange⌋;
                              Y.Knowledge = Y.Knowledge+ Y. knowledgeChange }
```

This means that when the knowledge for some concept X for some user has changed positively, and the knowledge of this concept is higher than 60, this is propagated to the knowledge of concept Y for that user if there exists a relationship called ‘relatedTo’ between concept X and Y. In that case, the knowledge of Y is updated with 20% of the change of X. Note the ⌊ and ⌋ symbol in this example that indicate that result is floored to the nearest integer. This means that the propagation of this value through the DM (sub-)graph will eventually halt because the value change will converge to zero. Note that the rule is ‘triggered’ only once every time the knowledge value of concept X changes.

Suppose that in our GAL program we update the knowledge of some concept if the user visits a page about some concept. We could then try to capture this knowledge propagation rule in one or more SPARQL update query. However, this is problematic. In order to be able to formulate this query one needs to know the path length of the update. If the knowledge update because of the page visit is fixed, we could precompute the

number of steps necessary. However, this update is not necessary fixed, which poses a problem. To illustrate this issue, consider the following (simplified) update SPARQL queries[2]:

```
MODIFY GRAPH
DELETE { $currentConcept um:Knowledge ?oldknowledge .}
INSERT ( $currentConcept um:Knowledge (?oldKnowledge+60) . }
WHERE { $currentConcept um:Knowledge ?oldknowledge . }
```

And

```
MODIFY GRAPH
DELETE { ?A um:Knowledge ?k1 .
           ?B um:Knowledge ?k2 .}
INSERT { ?A um:Knowledge (?k1+12) .
           ?B um:Knowledge (?k2+2) .}
WHERE { $currentConcept dm:relatedTo ?A .
          ?A um:Knowledge ?k1.
          ?A dm:relatedTo ?B .
          ?B um:Knowedge ?k2 . }
```

In this case we know that the initial change is 60. In the first propagation round this will reduce to $\lfloor 60/5 \rfloor$=12, in the second round this will reduce to $\lfloor 12/5 \rfloor$=2 and as $\lfloor 2/5 \rfloor$=0 there will be no third round. Therefore we can fully write the two-round propagation step. However, if the number of propagation is large the query will get large as well, while if the number of propagation steps is not known beforehand (e.g. is a fraction of the current value) we have a serious problem in this way of querying.

The GAL solution for this problem is anyway a quick-and-dirty fix. Not only might the GAL queries get very long and complex for more complex UM rules, they also have to be repeated in every Unit in which we update the variable that could trigger the rule. A better solution is to keep the UM reasoning rules in the UM definition. A complete Web application, described by GAL, therefore does not only contain a definition of the navigational structure in GAL, but also RDF versions of the DM and its instance data, and a RDF description of the UM and its reasoning rules.

For fixing the reasoning rules syntax we depend on the SWRL RDF rule language[3]. For our needs, we extended this syntax with triggers. Without a trigger a SWRL rule will execute a rule as long as an antecedent holds. A trigger refers to variable in the UM, and whenever the variable is modified, the SWRL rule is executed once. The trigger can be written at front of a SWRL rule. Its syntax is simple:

TRIGGER{ swrl variables }.

The list of SWRL variables that are associated with the trigger, and a further defined in the SWRL rule body, are monitored, and upon change trigger the one-time execution of the SWRL rule.

---

[2] For more on the SPARQL Update language refer to http://www.w3.org/Submission/SPARQL-Update/

[3] Refer to http://www.w3.org/Submission/SWRL/