GAL: A Generic Adaptation Language for describing Adaptive Hypermedia

Kees van der Sluijs[1], Jan Hidders[2],
Erwin Leonardi[2], Geert-Jan Houben[1,2]

[1] Eindhoven University of Technology
PO Box 513, NL-5600 MB Eindhoven, The Netherlands
k.a.m.sluijs@tue.nl
[2] Delft University of Technology
PO Box 5031, NL-2600 GA Delft, The Netherlands
{a.j.h.hidders, e.leonardi, g.j.p.m.houben}@tudelft.nl

Abstract. Nowadays, Web applications have become more sophisticated and interactive, and many of them also offer personalization and adaptation as their features. The mechanisms for these features are typically built on top of specific adaptive engine that has its own engine-specific adaptation language that can be very different from other engines. However, our observation leads to the fact that the main objective of these adaptive engines is the same, that is, to create adaptive navigation structure. In this paper, we present GAL (*Generic Adaptation Language*) that specifies the basic adaptive navigation structure in an engine independent way. Since GAL is *generic*, it can be compiled to difference specific adaptive engine and be used in combination with different authoring environments.

Keywords: GAL, generic adaptation language, navigation specification

1. Introduction

With the evolution of the Web, Web applications have grown more complex, interactive and more and more they try to adapt to the user. Personalization and adaptation are important features on the Web, as the Web allows a large and diverse public with diverse goals and demands to use the same Web applications. Personalization and adaptation are needed to help tailoring suboptimal one-size-fits-all solutions to an experience that fits a certain user in a certain context.

Adaptivity and personalization are complex mechanisms to build from scratch. Therefore, past research resulted in a number of adaptive engines and frameworks that simplify the creation of adaptive applications, like [1,2,3,4]. Also several authoring environments have been built that target the simplification of the creation of adaptive applications. If in the educational domain, for instance, a teacher should be able to create an adaptive course in such an adaptive engine, the authoring environment should allow such a lay user (in terms of adaptive systems) to easily create an adaptive application. Authoring environments also allow having different perspectives on adaptive applications, e.g. consider workflow or goal-oriented views.

A complicating factor however is that authoring environments are typically built on top of a specific adaptive engine, i.e. using a specific authoring environment restricts the specific adaptive engine that you may use and choosing a specific adaptive engine restricts the authoring environment that you can use. Moreover, we found that the different engines (even different versions of the same engines) typically have very different and sometimes complex engine specific languages that you need to use for specifying an adaptive application. However, even though the different adaptive engines have different functionalities and different strengths and weaknesses, in the end the core of these engines achieve the same thing: they create an adaptive navigation structure.

In this paper we present the Generic Adaptation Language (GAL). GAL is a language that allows expressing adaptive navigable Web applications over a given dataset. It is engine independent as it captures the *basic* adaptive navigation structure that is used by all (more specific) adaptive engines. This structure refers to the underlying dataset via queries. GAL is based on our previous experience with building Web applications in Hera [4]. However instead of specifying the whole applications design (domain, navigation, presentation), in GAL we focus on the adaptive navigation.

. The primitives of the GAL language should are chosen such that they support the basic types of adaptation as specified in the field of adaptive hypermedia [5]. To show that GAL is generic we will build compilers that allow compiling GAL to several specific adaptive engines. This allows a new authoring environment to author applications for all those engines by just providing support for the GAL language as an output format. Similarly compilers will be built from several existing authoring environments to GAL, which allows a new adaptive engine to use existing authoring environments by adding support for the GAL language.

Our claim is that by using this GAL language we are able to use any authoring environment in combination with any adaptive engine (for which we write a compiler). GAL adds the additional benefit of having the code in one place in an orderly way with a simple easy to understand syntax in semantics. We will give well defined semantics, which adds additional benefits like adding model checking functionality at the GAL level, so that a GAL engine can abstract away some of the tasks of the adaptive engine.

The structure of the rest of the paper is as follows. In Section 2 we first discuss some related work. After that, we present the formal syntax and an informal semantics of GAL with a running example that presents an adaptive application about the Milky Way. In Section 3 we describe the basic concepts of GAL and how it models the navigational structure of a web site. In Section 4 we describe the global structure and semantics of a GAL program. In Section 5 we explain how in GAL the components of pages are defined. In Section 6 we describe the navigation semantics of GAL programs, i.e., what operations are executed in which order when a user navigates through the defined web application, and what are the pages presented to the user. Finally in Section 7 we present our conclusions and discuss future work.

2 Related Work

Since more than a decade ago, the Web Engineering community has proposed numerous web engineering methods (e.g. Hera [4], OOHDM [6], WebML [7], and UWE [8]) for covering complete web application development cycle. These methods try to separate Web application into three main design concerns: *data design*, *application design*, and *presentation design*. Some of these methods offer techniques for adaptation. In [4], an adaptive Web information system called Hera is presented. Hera focuses on adaptation in the navigational design. RDF is used for expressing the domain and context data, and the adaptive navigation. In addition, Hera also provides facility to use RDF query expressions in the definition of application model. This allows a more fine-grained specification of adaptation and context-dependency. OOHDM [6] supports adaptation by specifying conditions on navigation concepts and relationships in the navigation class model. These conditions determine the visibility of content and links. In WebML [7], navigational adaptation conditions are specified in a *profile* as queries. The Event-Condition-Action (ECA) rules are also applied in WebML in order to attain context-awareness. UWE [8] is an UML-based web engineering approach that uses OCL constraints on the conceptual model to specify adaptation conditions. The aspect-orientation is also applied to UWE in order to specify several types of adaptive navigation, namely, *adaptive link hiding*, *adaptive link annotation*, and *adaptive link generation*.

In the field of *adaptive hypermedia* adaptive engines are built that serve applications which are personalized for their users (which is a characteristic that they share with for example Hera). Some well known adaptive hypermedia systems include AHA! [2], InterBook [3], and APeLs [1]). AHA! is an Open Source adaptive hypermedia system mainly used in education domain. It supports adaptation in a number of different ways, namely: adaptive guiding, link annotation, link hiding, and adaptive presentation support, by using conditional fragments and objects. Interbook provides an environment for authoring and serving adaptive online textbooks. It supports adaptive navigation that guides the users in their hyperspace exploration. The guidance is visualized by using annotations (e.g. icons, fonts, and colors). Adaptive Personalized eLearning Service (APeLS) is a multi-model, metadata drive adaptive hypermedia system that separates the narrative, content, and learner into different models. The adaptive engine in APeLS is a rule-based engine that produces a model for personalized courses based on a narrative and the learner model.

The researchers in adaptive hypermedia also propose several formal models for adaptive hypermedia applications, e.g. AHAM [9], The Munich Reference Model [10], and XAHM [11]. AHAM and The Munich Reference Model extend the DEXTER model [12] by separating the storage layer further. In AHAM, this layer is separated into *domain model*, *user model*, and *adaptation model*. The Munich Reference Model divides this layer into *domain meta-model*, *user meta-model*, and *adaptation meta-model*. While AHAM follows a more database point of view, the Munich Reference Model uses an object-oriented specification written in the Unified Modeling Language (UML). The adaptation is performed using a set of adaptation rules. These models support the content-adaptation and the link-adaptation (or navigation adaptation). Cannataro et al. proposed an XML-based adaptive hypermedia model called XAHM [9]. The adaptation process in XAHM is based on three

adaptivity dimensions that form *adaptation space*, namely, user's behavior (e.g. preferences and browsing activity), technology (e.g. network bandwidth and user's terminal), and external environment (e.g. location, time, and language). The adaptation process is to find the position of the user in the adaptation space. The user's behavior and external environment dimensions determine the adaptation of page content and links. The technology dimension is used in adapting the presentation layout.

The above state-of-the-arts approaches are different from GAL in the following ways. Unlike these approaches, GAL is not intended to be another system, method, or model for designing adaptive applications. Instead, GAL focuses on abstracting from a specific adaptive engine configuration by *generically* specifying basic navigation adaptation in an engine independent way. That is, GAL can be compiled into specific adaptation engines and used by these engines. Note that GAL does not focus on typical presentation aspects like text color, font face, and how pages should be rendered to be presentable on a specific device. These aspects are typically very engine specific, and therefore hardly describable in a generic way. Moreover, GAL on purpose separates these concerns and thus limits the number of language constructs in order to get a clean and simple core language.

3 The Basic Concepts of GAL

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 *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 that are to be executed when the user performs certain actions such as requesting the page or leaving the page. All data from which these units are generated is assumed to be accessible via a single RDF query endpoint, and the updates are also applied via this endpoint. In the examples in this paper we will use the SPARQL[1] query language for referring to and updating data.

The set U of units contains *unordered units* and *ordered units.* These differ in that the first represents a unit with unordered content, i.e., a bag or multi-set, and the second has ordered content, i.e., a list. Unordered units are denoted formally as **uu**(C, h) where $C \in \mathbf{B}(E)$ describes the content as a bag of content elements (in the set E, to be defined later) and the relation $h \subseteq ET \times UQ$ defines the event handlers by associating event types in the set ET with update queries in the set UQ. Here we will assume that ET = {**gal:onAccess**, **gal:onExit**} and UQ describes some set of update queries over RDF stores. Ordered units are denoted as **ou**(C, h) where $C \in \mathbf{L}(E)$ describes the content as a list of content elements and the relation h is as for unordered units.

The set E of content elements contains (1) *attributes* representing presentable content such as text, (2) *unit container* representing a single nested unit, (3) *unit sets*

[1] http://www.w3.org/TR/rdf-sparql-query/

representing a nested set of units of the same type, (4) *unit lists* representing a nested list of units and (5) *links* representing navigation links. We denote attributes as **at**(*n*, *l*, *v*) with a name $n \in EN \cup \{\perp\}$ representing the name of the attribute, a label $l \in EL \cup \{\perp\}$ representing a printable label for the attribute and a value $v \in V$ representing the value contained in the attribute. Here *EN* denotes the set of content element names here assumed to be all URIs minus the special constants **gal:_top**, **gal:_self** and **gal:_parent**, $\perp$ a special constant not in $EN \cup EL$ denoting the absence of a name, *EL* is the set of content element labels here assumed to be the set of RDF literals and *V* the set of basic values, i.e., URIs and RDF literals. Unit containers are denoted as **uc**(*n*, *l*, *u*) with a name $n \in EN \cup \{\perp\}$, a label $l \in EL \cup \{\perp\}$ and a unit $u \in U$. The unit label *l* represents a presentable label and we assume *UL* is the set of RDF literals. The name *n* is used to refer in other parts of the unit to the position of this unit container. We denote unit sets and unit lists as **us**(*l*, *UB*) and **ul**(*l*, *UL*) with a label $l \in EL \cup \{\perp\}$ and as content a bag of units $UB \in \mathbf{B}(U)$ or a list of units $UL \in \mathbf{L}(U)$. Finally, links are denoted as **ln**(*ut*, *b*, *q*, *be*, *t*) with a unit type $ut \in UT$ that indicates the type of unit to which the link points, a binding $b : X \rightharpoonup V$ that maps variable names to a value and defines the static parameters of the link, a query $q \in LQ \cup \{\perp\}$ that computes dynamic parameters of the link, a binding extension function $be : X \rightharpoonup VE$ that maps variable names to value expressions to describe additional dynamic parameters and a target $t \in EN \cup$ {"**gal:_top**", "**gal:_self**", "**gal:_parent**"} that indicates which unit in the current page will be replaced with the retrieved page. Here *LQ* denotes the set of queries over RDF stores that return a list of bindings, *UT* denotes the set of unit type which we assume here to be URIs, *X* is the set of variable names which we assume here all start with "$" and includes the distinguished variable **$user** and *VE* denotes the set of value expressions which compute a single value given an RDF store and whose syntax will be defined in more detail later on. The set of all bindings, i.e., partial functions $b : X \rightharpoonup E$, is denoted *B*.

Note that the above definitions of the set of proper units *U* and the set of content elements *E* are mutually recursive, which can be resolved by stating that we define them as the smallest sets that satisfy the given definitions. Note that therefore a unit is essentially a finite tree of content elements.

4 The Global Structure of GAL Programs

We proceed with giving the syntax of GAL along with an informal description of its semantics. The purpose of a GAL program is to describe how, given a page request in the form of a unit type, a user identifier and a binding that describes the parameters, the resulting unit is computed from the current RDF store. The top level of the grammar looks as follows:

<gal-expr> ::= <unit-decl>* .
<unit-decl> ::= <unit-type> **"["** <unit-def> **"]"** .

We assume that <unit-type> refers to the set of unit type names *UT*, and <unit-def> describes the computation of a unit. For each page request the first <unit-decl> with

the requested unit type determines the result. The request is passed on to this <unit-def> while removing the unit type and user identifier and extending the binding such that **$user** is bound to the user identifier. The syntax of <unit-def>:

<unit-def> ::= **"a"** (**"gal:unit"** | **"gal:orderedUnit"**) **";"**
(<input-var>)* (<attr-def> | <subunit-expr> | <set-unit-expr> |
<list-unit-expr> | <link-expr>)* (<on-event-expr>)* **.**

First the type of the resulting unit is specified, i.e., whether it is ordered or unordered, then the list of required parameters (excluding the implicit parameter **$user**) is given, followed by a list of content element expressions, and finally a list of event handlers is given.

To illustrate the syntax we look at a use case that uses GAL to specify an adaptive application. We look at a learning application about the Milky Way. More specifically, we look at the page that specifies information about a specific planet. We then declare something like:

```
:Planet_Unit [ a gal:unit; .. {content specification} .. ]
```

The syntax of <input-var> and <on-event-expr> is defined as follows:

<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> **"]" ";"**.

Here we assume that <var-name> describes the set of variables X, and <domain-concept> describes the set of domain concepts DC and <update-query> describes the set of update queries UQ^X parameterized with variables from X. The semantics of the <unit-def> is that it first checks if the current binding indeed defines the required variables with values that belong to the specified domain concept. If this is not the case, then the result is undefined. Otherwise the binding is restricted to the specified variables plus **$user** and passed to the content element expressions. Depending on the specified unit type, we compute **uu**(C, h) or **ou**(C, h). Here C is either the concatenation of the lists of content elements computed by the specified content element expressions or the corresponding bag of content elements. For the content element expressions we assume that they produce either an empty or a singleton list. Finally, h associates event types with update queries that are obtained by taking the specified parameterized update query and replacing the variables with their value in the binding.

For example, in the running example an input parameter of the unit will indicate which planet we want to show to user. For this we assume that an incoming link to the planet unit carries this information in the form of a variable. Therefore the following will be specified in the content specification:

```
gal:hasInputVariable [
        gal:varName "Planet";
        gal:varType Planet;
];
```

5 Content Element Expressions

We proceed with the syntax of the different content element expressions, starting with attributes:

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

Here <elem-name> describes the set *EN* of content element names defined here as the set of URIs not containing the special constants **gal:_self**, **gal:_parent** and **gal:top**. The semantics of the <attr-def> is the singleton attribute list [**at**(*n*, *l*, *v*)] with *n* the specified name (or ⊥ if it is not specified), *l* is the resulting literal of the <val-expr> (or ⊥ if it not specified or not a literal) and *v* the result of <val-expr>. If the result of <val-expr> is undefined, then the result is the empty list [].

The syntax and semantics of <val-expr> are as follows:

<val-expr> ::= <value> | “**(**” <val-expr> “**.**” <val-expr> “**)**” | “**[**” “**gal:query**” <val-query> “**]**” |
“**[**” “**gal:if**” <list-query> “**;**” (“**gal:then**” <val-expr> “**;**”)? (“**gal:else**” <val-expr> “**;**”)? “**]**” .

Here <value> describes the set of values *V* which we assume here to be the set of URIs and RDF literals, and <val-query> describes VQ^X, the set of parameterized queries over an RDF store that return a single value. Based on <val-expr> we define the earlier mentioned set of value expressions *VE* as the <val-expr> expressions without free variables. If the <val-expr> is <value> then this <value> is the result. If it is “**(**” <val-expr> “**.**” <val-expr> “**)**” then the result is the concatenation of the two literals that are the result of the two <val-expr>s, and if these do not both result in literals then the result is undefined. If it starts with **gal:query** then the result is that of <val-query> with the free variables substituted with their value in the current binding. If it starts with **gal:if** then the result is defined as follows. If the <list-query> with free variables replaced by their value in the current binding returns a non-empty list then the result of the **gal:then** <val-expr> is returned, otherwise the result of the **gal:else** <val-expr> is returned. If that particular <val-expr> is not present then the result is undefined.

To illustrate this we now provide some attributes in the running example to show information about this planet on the screen. For example, we want to print the name of the planet, and also provide an image of that planet:

```
gal:hasAttribute [
        gal:name planetName;
        gal:value [$Planet.name]

];
gal:hasAttribute [
        gal:name planetImage;
        gal:label ( "Image of: ".[$Planet.name] );
        gal:value [ gal:query // url of an appropriate picture
                 “SELECT ?image_url
                  WHERE { $Planet :hasAssociatedResource ?image
                                  :type Image;
                                  :url ?image_url.}”
```

```
            ]
      ];
```

Note the variable references for the variable named “Planet” by using the $-sign. The expression `[$Planet.name]` is a syntactic short-hand for the SPARQL query expression `[gal:query “SELECT ?name WHERE { $Planet name ?name }”]` which retrieves the name of the planet indicated by $Planet.

Now suppose we want to add an adaptive attribute that gives ‘beginner’ or ‘advanced’ information about the planet, but only if the user has advanced knowledge about the concept:

```
gal:hasAttribute [
   gal:name planetInformation;
   gal:label "Information: ";
   gal:value [
         gal:if [ gal:query
                “SELECT  ?conceptPlanetInstance
                 WHERE { $User :hasConceptInstance ?conceptPlanetInstance
                                          :name $Planet.name;
                                          :visited ?Visited;
                                          :advancedUserValue ?Advanced.
                                    FILTER (?Visited >= ?Advanced)
                        }”];
         gal:then [$Planet.AdvancedInfo];
         gal:else [$Planet.BeginnerInfo];
   ]
]
```

The query checks if the user number of visits equals or is greater than the amount of visits to the concepts that are needed to make a user an advanced user. If this is true (and the query in the ‘if’ clause returns a result) then the advanced information is showed, otherwise the beginners information is showed.

The next type of content element we consider is the unit container:

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

The semantics of <subunit-expr> is the singleton unit container list [**uc**(*n*, *l*, *u*)] where *l* is the unit label specified in <label-spec>, *n* is the unit name indicating the location specified in <name-spec> and *u* is the first unit in the unit list constructed by <unit-constr>. If the result of <unit-constr> is undefined then the result is the empty list []. The syntax and semantics <unit-constr> will be defined later on.

The following types of content element we describe are the list units and set units:

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

The semantics of <list-unit-expr> is the list-unit list [**ul**(*l*, *LU*)] where *l* is the unit label specified in <label-spec> and *LU* is the unit list constructed by <unit-constr>. The semantics of <set-unit-expr> is the same except that *LU* is replaced with *BU*, the corresponding bag of units.

To illustrate the use of unit sets in the running example consider that we want to show a list of the moons of the planet. We can use the hasSetUnit construct for this.

```
      gal:hasSetUnit[
               gal:label "The following Moon(s) rotate around “.[$Planet.name];
```

```
            gal:refersTo Moon_Unit_Short;
            gal:hasQuery "SELECT ?Moon
                        WHERE $Planet :hasMoon ?Moon";
    ];
```

The hasSetUnit construct executes a query and for every result of that query it will spawn a unit, in this case a Moon_Unit_Short. For every of those units the query result will be bound to the 'Moon' variable.

The final type of content element that can be specified is the link:

<link-expr> ::= **"gal:hasLink" "["** <link-constr> <target-spec>? **"]" ";"** .
<link-constr> ::= **"gal:refersTo"** <unit-type> **";"** <bind-list> .
<bind-list> ::= <query-bind>? <var-bind>* .
<query-bind> ::= **"gal:hasQuery" "["** <list-query> **"]" ";"**
<var-bind> ::= **"gal:assignVariable" "[" "gal:varName"** <var-name> **";"**
"gal:value" <val-expr> **"]" ";"** .
<target-spec> := **"gal:targetUnit"** (<unit-name> | **"gal:_top"** | **"gal:_self"** | **"gal:_parent"**) **";"** .

Here <list-query> describes the set LQ^X of parameterized queries over the RDF store that return a list of bindings and are parameterized with variables from X. The result of <link-expr> is the link list [**ln**(*ut*, *b*, *q*, *be*, *t*)] where *ut* is the unit type specified in <link-constr>, *b* is the current binding, *q* is the <list-query> but with the free variables replaced with their value in the current binding, *be* is the partial function that maps each <var-name> in the <var-bind> list to the first associated <val-expr> but with the free variables in this <val-expr> replaced as specified by the current binding, and finally *t* is the target in <target-spec> if this is specified and **gal:_top** if it is not.

For example, the Moon_Unit_Short that was mentioned in the preceding example could contain a link to the elaborate description as follows:

```
:Moon_Unit_Short [
    a gal:unit;
    gal:hasInputVariable [
            gal:varName "Moon";
            gal:varType Moon
    ];
    gal:hasAttribute [
            gal:name moonName;
            gal:label "Moon: ";
            gal:value [$Moon.name]
    ];
    gal:hasLink [
            gal:refersTo :Moon_Unit_Elaborate;
    ]
]
```

Note the hasLink construct that binds to the Moon_Unit_Short, meaning that every attribute in the unit becomes a link that points to the Moon_Unit_Elaborate unit. By default the current input variables, in this case $Moon, are passed as parameters of the link, so it indeed will lead to a page describing the same moon more elaborately.

The final non-terminal we consider is <unit-constr> which computes a list of units:

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

The result is defined as follows. First, the query in <bind-list> is evaluated, while replacing its free variables with their value in the current binding, and the resulting

list of bindings is extended with the binding specified by the list of <var-bind> expressions. Finally, for each of the resulting bindings we use them to extend the current binding and for this binding evaluate the <unit-def>, and finally collect all the resulting units in a list.

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

We conclude the presentation of the syntax an semantics of GAL with a possible output of the running example. Given the definitions for the Planet Unit it would yield for a specific instance, for example Jupiter, something along the lines as shown in the screenshot in Figure 1. This screenshot is based upon a GAL specification that is compiled to and executed by the AHA!3 engine [2].

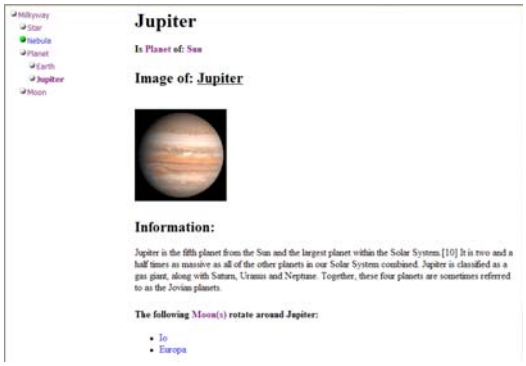

Figure 1: Screenshot of an instance of the Planet Unit

6 The Navigation Semantics of GAL

In this section we 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* and this in turn is combined with the binding that is obtained when evaluating *be*. 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.

7 Conclusions and Future Work

Many Web application frameworks nowadays offer personalization and adaptation as their features. However, until now the adaptation mechanisms are typically engine specific and the adaptation languages used by the adaptive engines are different from one another. In this paper, we have presented Generic Adaptation Language (GAL) that aims to capture basic adaptive navigation functionality in an engine independent way. GAL allows us to use every authoring environment in combination with any adaptive engines. We gave a formal description of GAL, and demonstrated that GAL can be used to specify an adaptive application. Currently we work on working out examples that show that GAL allows every type of adaptivity as specified in [5]. In further future work we look at building compilers from GAL to several adaptive engines (e.g. for AHA! version 3, Hera and the GALE engine that is being built in the GRAPPLE project[2]). Similarly we will build compilers from several authoring environments to GAL (e.g. for the Graph Author and the authoring tools within the GRAPPLE project). We expect that our formal work as presented in this paper will help us to relatively simply build these compilers. In this way we aim to show that GAL is both applicable to most adaptive engines as well as that it is generic. This might pave the path in the future for general acceptance and standardization of a engine-independent generic adaptation language.

Acknowledgements: This work was supported by the 7th Framework Program European project GRAPPLE ('Generic Responsive Adaptive Personalized Learning Environment').

[2] Cf. http://www.grapple-project.org

References

[1] Owen Conlan, Vincent P. Wade, Catherine Bruen, and Mark Gargan. **Multi-model, Metadata Driven Approach to Adaptive Hypermedia Services for Personalized eLearning**. In Proceedings of the Second Adaptive Hypermedia and Adaptive Web-Based Systems Conference (AH 2002), Malaga, Spain, May, 2002.

[2] Paul De Bra, David Smits, and Natalia Stash. **The Design of AHA!**. In Proceedings of the 17th ACM Conference on Hypertext and Hypermedia (Hypertext 2006), Odense, Denmark, August, 2006.

[3] Peter Brusilovsky, John Eklund, and Elmar W. Schwarz. **Web-based Education for All: A Tool for Development Adaptive Courseware**. In Computer Networks and ISDN Systems, 30(1-7), pp. 291-300, 1998.

[4] Geert-Jan Houben, Kees van der Sluijs, Peter Barna, Jeen Broekstra, Sven Casteleyn, Zoltán Fiala, and Flavius Frasincar. **Hera**. Book chapter: Web Engineering: Modelling and Implementing Web Applications, G. Rossi, O. Pastor, D. Schwabe, L. Olsina (Eds), Chapter 10, pp. 263-301, 2008, Human-Computer Interaction Series, Springer.

[5] Peter Brusilovsky. **Adaptive Hypermedia**. In User Modeling and User-Adapted Interaction, 11 (1-2), pp. 87-110, 2001.

[6] Gustavo Rossi and Daniel Schwabe. **Modeling and Implementing Web Applications with OOHDM**. Book chapter: Web Engineering: Modelling and Implementing Web Applications, G. Rossi, O. Pastor, D. Schwabe, L. Olsina (Eds), Chapter 6, pp. 109-155, 2008, Human-Computer Interaction Series, Springer.

[7] Marco Brambilla, Sara Comai, Piero Fraternali, and Maristella Matera. **Designing Web Applications with WebML and WebRatio**. Book chapter: Web Engineering: Modelling and Implementing Web Applications, G. Rossi, O. Pastor, D. Schwabe, L. Olsina (Eds), Chapter 9, pp. 221-261, 2008, Human-Computer Interaction Series, Springer.

[8] Nora Koch , Alexander Knapp, Gefei Zhang, and Hubert Baumeister. **Uml-Based Web Engineering: An Approach Based on Standard**. Book chapter: Web Engineering: Modelling and Implementing Web Applications, Gustavo Rossi, Oscar Pastor, Daniel Schwabe, and Luis Olsina (Eds), Chapter 7, pp. 157-191, 2008, Human-Computer Interaction Series, Springer.

[9] Paul De Bra, Geert-Jan Houben, and Hongjing Wu. **AHAM: A Dexter-based Reference Model for Adaptive Hypermedia**. In Proceedings of the 10th ACM Conference on Hypertext and Hypermedia (Hypertext'99), Darmstadt, Germany, February, 1999.

[10] Nora Koch and Martin Wirsing. **The Munich Reference Model for Adaptive Hypermedia Applications**. In Proceedings of the Second Adaptive Hypermedia and Adaptive Web-Based Systems Conference (AH 2002), Malaga, Spain, May, 2002.

[11] Mario Cannataro and Andrea Pugliese. **XAHM: An Adaptive Hypermedia Model Based on XML**. In Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE), Ischia, Italy, July, 2002.

[12] Frank G. Halasz and Mayer D. Schwartz. **The Dexter Hypertext Reference Model**. In Communications of the ACM (CACM), 37(2), pp. 30-39, Febuary, 1994.