Creating Adaptive Applications with AHA! Tutorial for AHA! version 3.0

Paul De Bra, Natalia Stash, David Smits[1]

[1] Eindhoven University of Technology
Department of Computer Science
PO Box 513, NL 5600 MB Eindhoven
The Netherlands
`{debra,nstach,dsmits}@win.tue.nl`

Abstract. Creating adaptive applications consists of three aspects: creating a conceptual structure of the application domain, including concept relationships that can be used for adaptation, creating content to match the conceptual structure, and setting up a software environment that performs the adaptive delivery of that content to the end-user. In this tutorial we first show how to install the AHA! system. We then show how to create a concept structure using AHA!'s authoring tools and how to develop content so that AHA! can serve and adapt it to the individual user. AHA! is an open source Java-servlet-based software environment that works with the tomcat webserver, on Linux (or Unix) as well as on Microsoft Windows. It is available from http://aha.win.tue.nl/.

1 Introduction

Adaptive hypermedia systems [1,2] have started to appear around 1990, when researchers combined the concepts of hypertext/hypermedia with user modeling and user adaptation. The first and foremost application of adaptive hypermedia was in education, where the navigational freedom of hypermedia was introduced into the area of intelligent tutoring systems. But since then applications in information systems, information retrieval and filtering, electronic shopping, recommender systems, etc. have been realized. The advent of the Web has made the use of (basic) hypermedia facilities easier, through the use of HTML. However, creating adaptive hypermedia on the Web requires server-side functionality for user modeling and for the adaptive generation of (HTML) pages. Until recently almost every adaptive hypermedia application was based on a special-purpose (server-side) system. The development of adaptive hypermedia applications and systems has had a one-to-one relationship. This has seriously hindered the development of interesting new adaptive applications by researchers with insufficient skills or financial means to develop their own adaptive hypermedia system. The AHA! system [6], or Adaptive Hypermedia Architecture, was designed and implemented at the Eindhoven University of Technology, and sponsored by the NLnet Foundation through the AHA! project, or Adaptive Hypermedia for All. AHA! is an open source general-purpose adaptive hypermedia system, through which very different adaptive applications can be created. AHA! offers low-level facilities for creating exactly the desired look-and-feel for each application and for fine-tuning the adaptation, and it offers high-level facilities for creating the conceptual structure of an application, using *concepts* and *concept relationships*. Since AHA! is essentially an adaptive client and server at the same time it can be used as a component in the content delivery pipeline and thus integrated into other server environments. (AHA! is an HTTP server, but can also request pages from other HTTP servers and filter them.)

In this tutorial we cover all the steps involved in the creation of adaptive applications using AHA! version 3.0. We put the emphasis on the creation of the *adaptation* in the application by using *concepts* and *concept relationships*. This is not only the most important part that distinguishes adaptive from non-adaptive applications, but it is also the part that is best supported by authoring tools in this version. We also cover low-level and advanced features that are available, but their use requires a more in-depth knowledge of the technology used by AHA!.

2 Overall AHA! Architecture

In order to work with AHA! (as authors of applications) we need to understand how the AHA! system works in general. For the most part AHA! works as a Web server. Users request pages by clicking on links in a browser, and AHA! delivers the pages that correspond to these links. However, in order to *generate* these pages AHA! uses three types of information:

- The *domain model* (DM) contains a conceptual description of the application's content. It consists of *concepts* and *concept relationships*. In AHA! every page that can be presented to the end-user must have a corresponding concept. It is also possible to have *conditionally included fragments* in pages. For each "place" where a decision needs to be made what to include a concept must be defined. (Such a concept can be shared between different pages on which the same information is conditionally included.) Pages are normally grouped into sections or chapters or other high-level structures. AHA! makes use of a *concept hierarchy* through which one can easily have "knowledge" propagated from pages to sections and chapters, and through which AHA! can automatically generate and present a hierarchical table of contents. Concepts can be connected to each other through *concept relationships*. In AHA! there can be arbitrarily many *types* of concept relationships. A number of types are predefined to get you going quickly as an author. A typical example of a (predefined) relationship type is *prerequisite*. When concept A is a prerequisite for concept B the end-user should be advised (or forced) to study or read about concept A before continuing with concept B. In AHA! prerequisite relationships result in changes in the presentation of hypertext link anchors. By default the link anchors will have the normal Web colors (blue or purple) when the link leads to a page concept for which all the prerequisites are met, and will have the color black when some prerequisites are not met. Creating a domain model is easiest using the *Graph Author* tool, described in Sect. 5. Fig. 1 shows an example of a domain model (taken from the on-line AHA! 2.0 tutorial) as it would appear in the Graph Author.

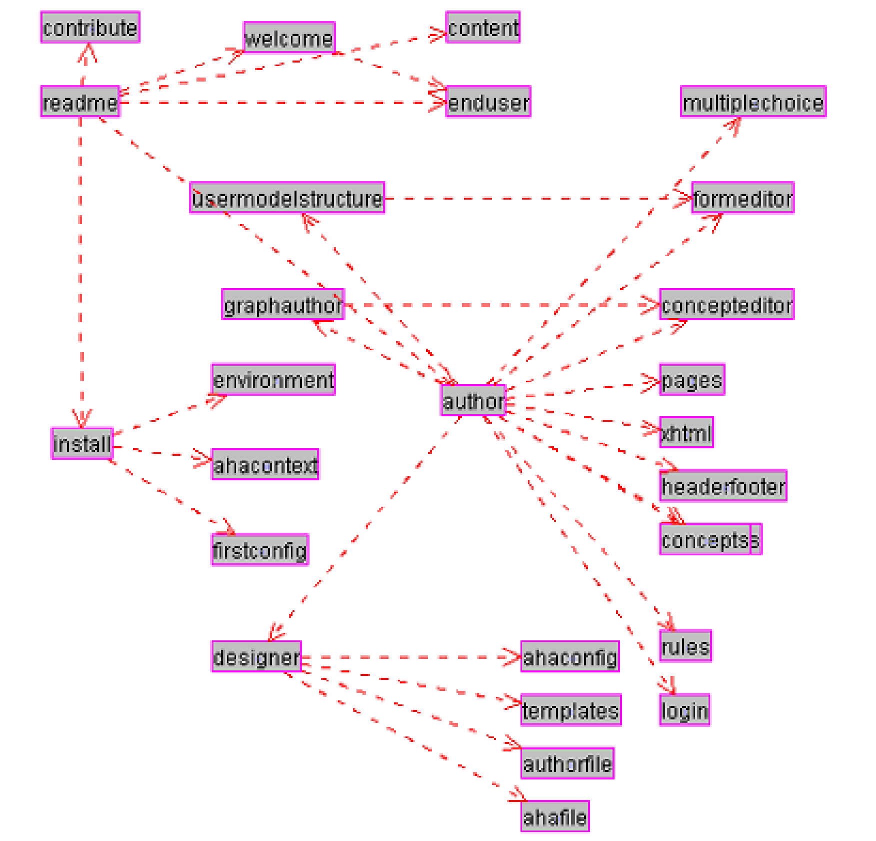


Fig.1. Example domain model with concepts and prerequisite relationships.

- The *user model* (UM) in AHA! consists of a set of *concepts* with *attributes* (and *attribute values*). UM contains an *overlay model*, which means that for every concept in DM there is a concept in UM. In addition to this, UM can contain additional concepts (that have no meaning in DM) and it always contains a special pseudo-concept named "personal". This concept has attributes to describe the user, and includes such items as login and password. When a user accesses an AHA! application the login form may contain arbitrary (usually hidden) fields that contain values for attributes of the "personal" concept. It is thus possible to initialize *preferences* through the login form. To get you going quickly as an author the AHA! authoring tools provide a number of UM concept *templates* (see Sect 5.2), resulting in concepts with predefined attributes. Typical attributes are "knowledge" and "interest", to indicate the user's knowledge of or interest in a certain concept. AHA! will automatically propagate an increase in knowledge of a concept to higher-level concepts (higher in the concept hierarchy of DM). It will also record a lower knowledge increase when studying concepts for which the prerequisites are not yet known.

- The *adaptation model* (AM) is what drives the *adaptation engine*. It defines how user actions are translated into user model updates and into the generation of an adapted presentation of a requested page. AM consists of *adaptation rules* that are actually *event-condition-action rules*. Most authors will never have to learn about AM because the rules are generated automatically by the Graph Author, but in order to really create the adaptive applications that do exactly what you want you should get to know either the *Concept Editor* presented in Sect. 6 or get to know how to define *concept relationship templates* used by the Graph Author.

Now that we have seen the "components" that make up an AHA! application we can explain what exactly happens when the end-user clicks on a link in a page served by AHA!:

1. In AHA! there are two types of links: links to a *page* and links to a *concept*. Since in DM pages are linked to concepts AHA! can find out which concept corresponds to a page and which page corresponds to a concept.
2. The adaptation engine starts by executing the *rules* associated with the attribute "access" of the requested concept (or the concept that corresponds to the requested page). "Access" is a system-defined attribute that is used specifically for the purpose of starting the rule execution.
3. Each rule may update the value of some attribute(s) of some concept(s). Each such update triggers the rules associated with these attributes of these concepts. (We explain the rules in Sect. 11.1).
4. When the rules have been executed AHA! determines which page was requested. There may be several pages associated with a concept. In Sect. 5.2 we explain how AHA! determines which page to present to the user. Processing the requested page may involve three types of actions:
   a. The page may contain *conditionally included fragments* and *conditionally included objects*. The decision whether to include a fragment or not is based on a *condition* which is a Boolean expression using attributes of concepts (and constants). A conditionally included fragment may conditionally include other fragments. The page with all its fragments is a single (X)HTML file. Conditionally included objects are defined through the <object> tag, with a parameter that refers to a concept. When the AHA! engine encounters a conditionally included object it executes adaptation rules associated with the "access" attribute of that concept (and these rules may again trigger other rules). UM is thus updated each time a conditionally included object is encountered. One of the rules will tell the engine which file (or "resource") to include. This is explained in Sect. 5.2 and Sect. 10 and follows the same procedure as for the access to a page. The selected resource is *inserted into the parse stream* and must thus be a valid (X)HTML fragment. It may contain other <object> tags that cause the conditional inclusion of more objects.
   b. The page may contain links (<a> tags) to other concepts or pages. If a link (anchor) is of the class "conditional" the AHA! engine checks the user model to decide upon the *suitability* of the link destination (concept or page). If the destination is not suitable the link anchor will be displayed in black, otherwise the anchor will be displayed in blue or purple depending on the *visited* status of the link destination (unvisited or visited). The blue/purple/black color scheme can be changed. In AHA! the colors are referred to as GOOD, NEUTRAL and BAD.
   c. Any other content of the page is passed to the browser unchanged. It must however be valid (X)HTML. It is also possible to use an XML format different from XHTML. We have already experimented with SMIL for multimedia documents.

3 Installing the AHA! software environment

AHA! is based on Java Servlet technology. In theory it should be possible to use it with any Java-enabled platform, on any Java-based webserver. We have tried (and succeeded) in installing AHA! only on Microsoft Windows and Linux, using the Apache Tomcat server (version 4 or 5), either the standalone version or the version from Sun's JWSDP package. In this section we describe the AHA! installation with Tomcat only (Apache or Sun's version makes no difference).

3.1 Installing Tomcat and logging on

When installing Tomcat (on Windows this is somewhat more automated than on Linux) you have to choose a name and password for the administrator. After starting the server you should log on as this administrator in order to install AHA!. We will assume here that the Tomcat server is installed on "localhost" on port 8080. In most cases this will be the default setting. To log in as administrator you start a web browser and browse to the location http://localhost:8080/admin which brings up the following login screen:

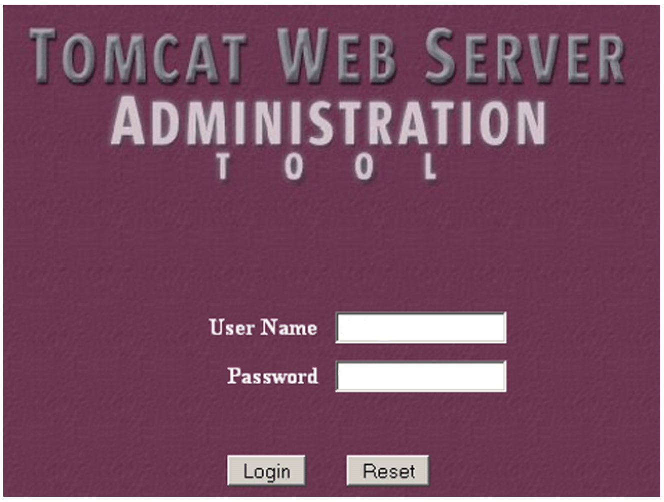


Fig. 2. Tomcat login screen.

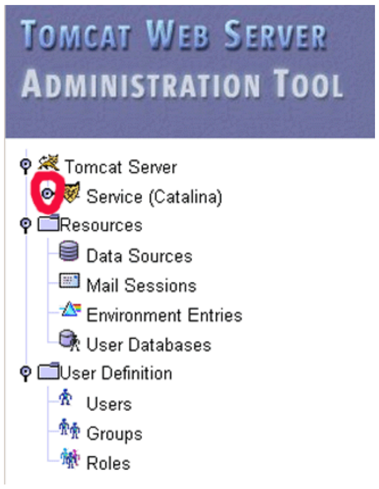
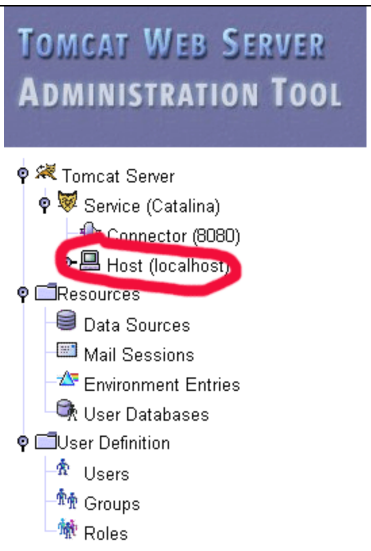
3.2 Creating the AHA! context

After logging in using the name and password you chose you can create a "context" for AHA!. Please note that Tomcat (at least up to version 5) stores the administrator name and password as cleartext in the file conf/tomcat-users.xml. Some versions even have a standard user with name "tomcat" and password "tomcat" predefined. You should make sure only the name you chose exists in this file, and that you do not use your regular password for Tomcat as a precaution.

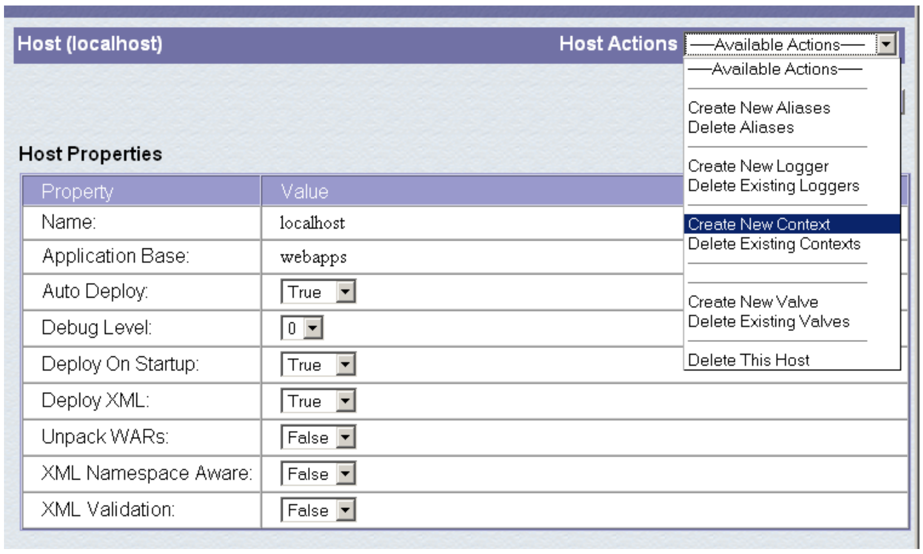
AHA! is distributed as a zip archive that you can extract anywhere on your system. The archive is the same whether you are using Windows or Linux. In the installation description and screendumps we will use the directory "d:\aha3" for this purpose but this name is completely arbitrary.

After logging on you should open up the "Service" item by clicking on the icon circled in red in Fig. 3.

After this you should click on "localhost" (not the small circle to the left of it) in the menu shown in Fig. 4.

 TOMCAT WEB SERVER ADMINISTRATION TOOL Tomcat Server Service (Catalina) Resources Data Sources Mail Sessions Environment Entries User Databases User Definition Users Groups Roles	 TOMCAT WEB SERVER ADMINISTRATION TOOL Tomcat Server Service (Catalina) Connector (8080) Host (localhost) Resources Data Sources Mail Sessions Environment Entries User Databases User Definition Users Groups Roles
Fig. 3. Initial Tomcat administration menu.	**Fig. 4.** Tomcat administration tool with service submenu opened up.

This brings up a menu of available actions of which you should select to create a new context, as shown in Fig. 5.



Host Properties

Property	Value
Name:	localhost
Application Base:	webapps
Auto Deploy:	True
Debug Level:	0
Deploy On Startup:	True
Deploy XML:	True
Unpack WARs:	False
XML Namespace Aware:	False
XML Validation:	False

Fig. 5. Menu with selection to create a new context.

Fig. 6. (on the next page) shows the form for creating the AHA! context. The items circled in red are the most important:

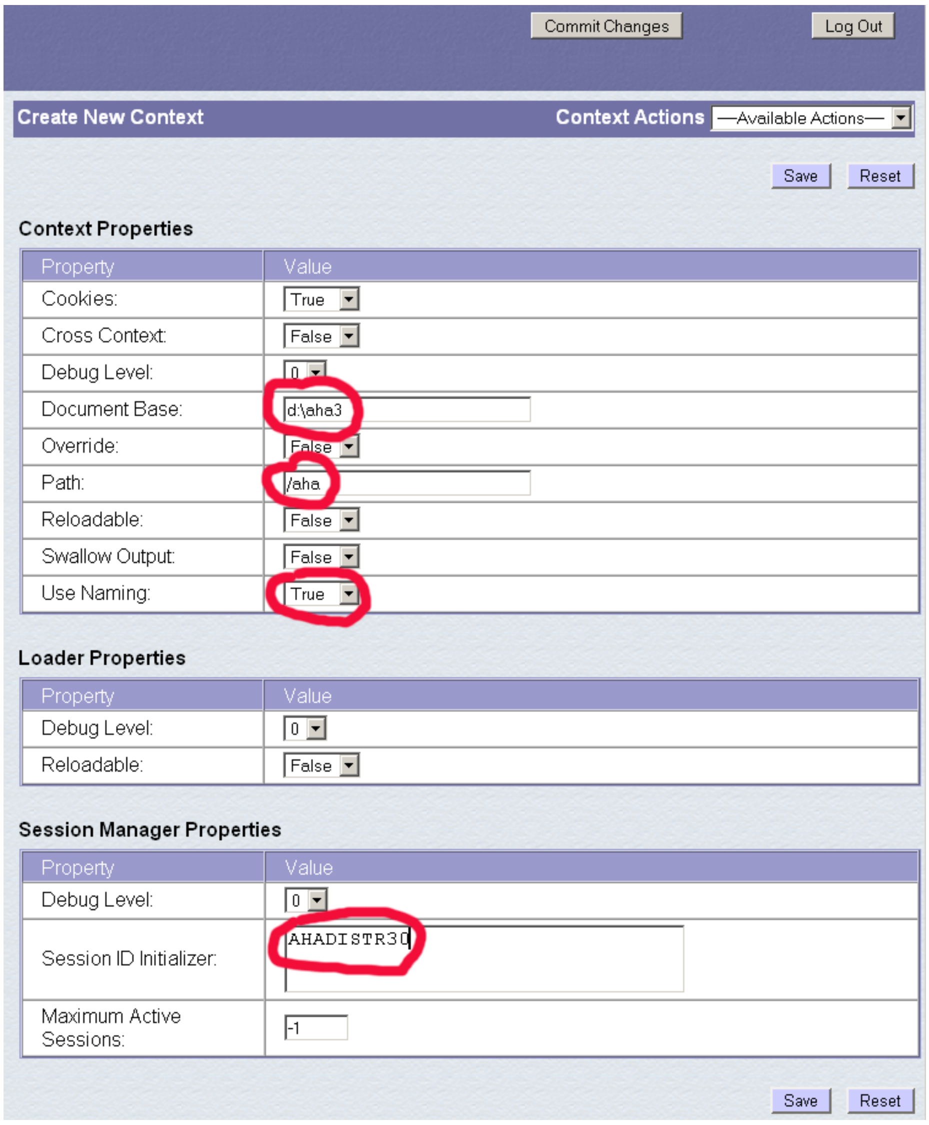
- The Document Base is the name of the directory where the AHA! archive was extracted. It need not be a subdirectory of Tomcat's installation directory. You can install AHA! anywhere you want. (We use d:\aha3 as an example.)
- The Path is the name through which all AHA!-related files will be referenced. It starts with a "/" but is always relative to the document base. When you install an application named "tutorial" on this server the tutorial's starting page will be referenced as http://localhost:8080/aha/tutorial/. (Using our example document base the application's main directory will then be d:\aha3\tutorial.)
- Use Naming has to be set to True. (It is False by default.) This parameter is important in order for AHA! to be able to retrieve its configuration information from its configuration file.
- The Session ID Initializer must be defined. It can be any string.

We suggest to leave other parameters at their default. One parameter that can be changed as desired is the Loader Property Reloadable. In case you wish to modify the AHA! software while the server is running the new code will be loaded if reloadable is set to true. However, this implies that each time a Java class is accessed the server checks to see if it was modified. This slows down the AHA! system noticeably. When you set the Context Reloadable property to true this should eliminate the need for a server restart (see Sect. 3.3) after the initial AHA! configuration, but our experience is that this feature appears not to work. You can also set the Context Property Maximum Active Sessions to limit the number of simultaneous users.

After filling out this form you must first press the "Save" button and then the "Commit Changes" button. The AHA! context is created and you can turn to configuring AHA! itself.

3.3 Initial AHA! configuration

Once Tomcat is up and running and the AHA! context created you should browse to the location http://localhost:8080/aha/Config. The effect of this action is that AHA! automatically configures itself using the parameters from its context. This means that AHA! takes into account what its Document Base is. After this initial configuration step it is imperative that you shut down the Tomcat server and restart it. The next configuration steps (see Sect. 3.4) will not work unless you restart the server.



Context Properties

Property	Value
Cookies:	True
Cross Context:	False
Debug Level:	0
Document Base:	d:\aha3
Override:	False
Path:	/aha
Reloadable:	False
Swallow Output:	False
Use Naming:	True

Loader Properties

Property	Value
Debug Level:	0
Reloadable:	False

Session Manager Properties

Property	Value
Debug Level:	0
Session ID Initializer:	AHADISTR30
Maximum Active Sessions:	-1

Fig. 6. Form for configuring the AHA! context.

3.4 The AHA! configurator

When visiting the location http://localhost:8080/aha/Config for a second time you are presented with a login form. The initial name you must use is “aha” and the password is empty. You are then shown the page in Fig. 7.

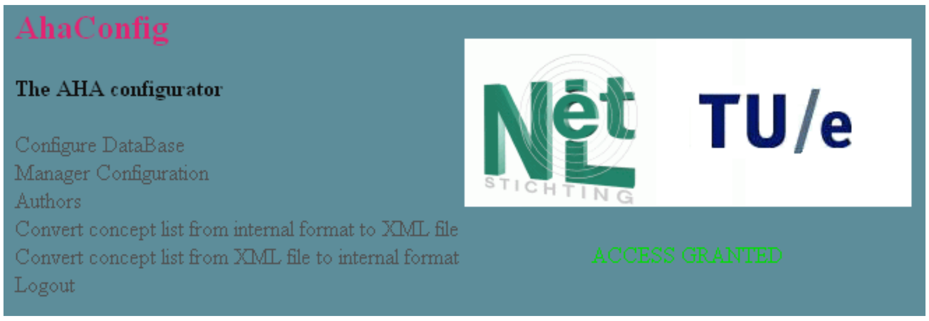
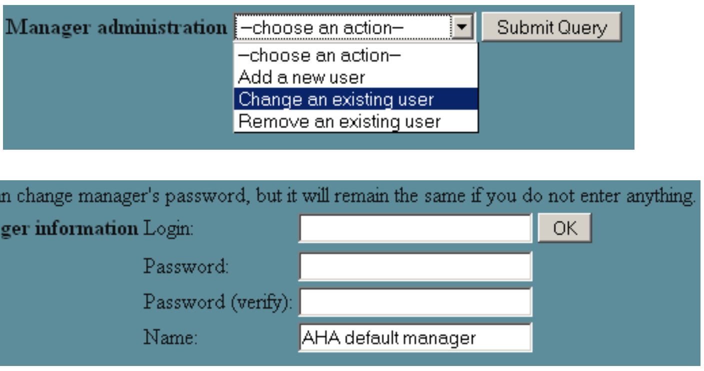


Fig.7. The AHA! configurator

The first thing you should do is to change the “Manager Configuration” to select a different name and password for the manager. You select “Change an existing user” and then fill out the “Manager information” form, as shown in Fig.8. Unlike Tomcat, AHA! stores the manager password in encrypted form. There is no serious security threat if someone reads AHA!’s configuration file containing the password information.



You can change manager's password, but it will remain the same if you do not enter anything.

Manager information Login:

Password:

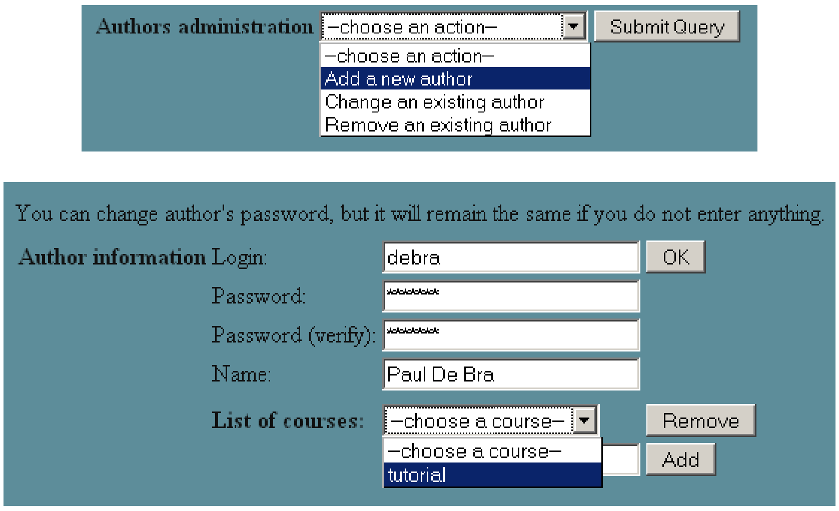
Password (verify):

Name: AHA default manager

Fig. 8. Manager administration menu to change the default manager.

The other items in the AHA! configurator are:

- Configure Database: this lets you select the internal storage format used by AHA!. The default is to use XML files to represent concepts and user models. It is possible to use a MySQL database instead. Through this menu you can copy the information from the XML representation to a MySQL database and back. In this tutorial we will assume that you leave the representation at its default setting which is to use XML files.
- Authors: this lets you create and change authors of AHA! applications installed on your server. It also lets you assign applications to authors. Fig. 9. shows how to create an author and assign an application. Note that “application” is sometimes referred to as “course” because AHA! was initially only used for on-line courses.



You can change author's password, but it will remain the same if you do not enter anything.

Author information Login: debra

Password: *******

Password (verify): *******

Name: Paul De Bra

Fig. 9. Creating an author and assigning an application to the author.

- Convert concept list from internal format to XML file: this lets you retrieve the concepts of a single application from the internal format (which is XML or MySQL) and convert it into a single XML file as used by the Concept Editor (see Sect. 6). The generated XML file is automatically placed in the author’s working directory.
- Convert concept list from XML file to internal format: this lets you convert a single application from the XML file as used by the Concept Editor (see Sect. 6) to the internal storage format (XML or MySQL). The AHA! authoring tools have a button for performing this function, so normally it is never necessary to use this menu item in the AHA! configurator. However, when importing an application from an external source (possibly created with different authoring tools) this menu item lets you “register” the application with the AHA! server.

4 AHA! for the end-user

This tutorial is really for AHA! authors, but it is important to first understand what an AHA! application does for the end-user. We will then learn how to make AHA! achieve what the user expects. Every AHA! application is accessed through a *login form*. First time users must register with the AHA! server through a *registration form*. It is possible to combine both forms. Users are identified through a unique *identity* which is chosen in the registration form. AHA! can also allow *anonymous users* for which the identity is chosen by the system. That identity is remembered by the browser through the "cookie" mechanism. A user can restart an anonymous session only from the same machine using the same browser. Non-anonymous (normal) users have a password to prevent unauthorized use of their login. An application may offer a form to let users change their password.

One AHA! server can contain several applications. Users can reuse their identity when they go from one application to another. They can also return to the first application or go back and forth. However, one should not try to use two applications simultaneously from different browsers or browser windows, because AHA! maintains only one session per user at a time. Using multiple applications simultaneously can lead to unexpected results (like the name of one application being displayed in the other application, or the wrong background image on pages).

An AHA! application performs adaptation based on a *user model* that is created and updated each time the user clicks on a link. An author can provide forms that let the user inspect and change certain attribute values for certain concepts. It is thus possible to have user-selectable preferences, for instance to choose image over text, or to select audio or video in addition to text.

The adaptation is normally based on the user model instance at the time of a page request. However, it is possible in AHA! to keep the presentation of a concept "stable", which means that once the concept has been presented it is always presented in the same way, even if the user model instance would suggest otherwise. Some users may find it disturbing when a presentation changes on them. Stable presentations alleviate this problem. The *stability* can be bound to just a session or to a Boolean expression, so it does not have to be forever.

The adaptation observed by the end-user consists of:

- The adaptive choice of link destinations. When a link is bound to a *concept* (rather than directly leading to a *page*) the adaptation engine uses a rule to determine which page to show, depending on the user model instance. A link to a concept may for instance lead to an introductory page for users who are missing some prerequisite knowledge, whereas advanced users will see a more detailed description of the concept. Such adaptation is quite drastic, and most likely very obvious to the user.
- Less drastic adaptation is the conditional inclusion of information or objects. Depending on the user model instance information items can be shown or hidden or a choice can be made (by the system) between different alternative items to be included in the page. The use of this adaptation technique serves two purposes:
  - Depending on what the user knows or doesn't know the system may decide to give an extra explanation. Reasons for doing so include compensation of missing foreknowledge and showing interesting details or related information to users who can understand it. This is what we would call true *adaptive* behavior.
  - Conditional inclusion can also be used to choose between different representations of the same information. When the same items are available as text, slides, audio and video a representation can be chosen based on user preferences. Such systematic and stable type of adaptation is called *adaptable* behavior.
- The changes in link colors (and possibly also annotation with colored icons). The basic link annotation scheme in AHA! uses three link colors (for adaptive links):
  - GOOD: the link points to a suitable page the user has not visited before. The standard color for such link anchors is blue.
  - NEUTRAL: the link points to a suitable page the user has visited before. The standard color for such link anchors is purple.
  - BAD: the link points to an unsuitable page. Whether or not the user visited this page before, the standard color for such link anchors is (almost) black.

  Depending on how the AHA! application is authored the color scheme is either determined by a style sheet (defined by the author) or is generated from preferences that can be changed by the end-user through a form.

When the user navigates through the browser's history (using the back and forward buttons) the browser may not request the pages from the AHA! server but may use cached versions. This results in pages being shown exactly as the user saw them before, and not in the "new" form that would correspond to the most recent user model.[1]

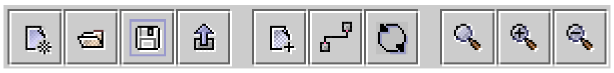
AHA! applications do not have a common look and feel. An author can either create a complete custom presentation, using HTML frames and Javascript to synchronize the content of the different frames. (This is done in the AHA! 2.0 tutorial at http://aha.win.tue.nl/ for instance.) An author can also use the *layout model* possibilities that AHA! 3.0 offers to use some automatically generated windows or frames for displaying a partial table of contents and information about concepts.

[1] We are very much interested in hearing about a general way to completely disable the caching in the browser. Until now we have been unable to find a method that works with every browser.

5 Authoring with the Graph Author tool

To create the conceptual structure of an application you can use the Graph Author tool, or the Concept Editor described in Sect. 6. The Graph Author is a higher level tool than the Concept Editor. Most authors will never need to use the Concept Editor, but for those who wish to we describe it later.

A standard AHA! installation has a link to the authoring tools on its "home" page. Since we assume that AHA! is installed with /aha as its path, the authoring tools will be available from the /aha/author/ page. The Graph Author is a Java applet with Servlets on the server side to support the I/O. After entering your (author) name and pasword it lets you create a new application or open an application that is assigned to you. (You cannot open other authors' applications.) In the Graph Author you create *concepts* and *concept relationships*. The Graph Author window is split into two parts, showing the concepts (as a hierarchy) on the left and showing the concept relationships (as a graph) on the right. Fig. 10 shows a screen shot of the Graph Author. The graph represents the structure of *prerequisite relationships* in a "tutorial" application. The concepts are structured as a hierarchy which in fact also is a structure of concept relationships (and always present in an AHA! application). Every type of relationship has a different meaning related to the adaptation an application provides (we explain this later) and is represented using different color and style of arrows. In Fig. 10 (on the next page) we only see the prerequisite relationships. Other types of relationships are filtered out to prevent clutter. We will briefly explain the operations provided by the buttons on the toolbar (which looks like the image below):



This button is used to create a new application. It is initially unnamed but you can assign it a name when you save it. The application is then automatically added to the list of applications assigned to you. Every newly created application has a "toplevel" concept. All concepts you create later will be below this concept in the *concept hierarchy*.

This button is used to open an existing application that is assigned to you (as an author). You are presented with a dialog box that lets you select one of the assigned applications.

This button is used to save the application. You get a dialog box that lets you optionally change the name of the application too. When the application is saved it does not automatically become served from your AHA! server. It is only saved in "authoring format" so that you can later open it again and continue editing.

This button is also used to save the application. In addition to saving into the authoring format this button also commits the application to the server's database (either in XML or in mySQL representation). After restarting the Tomcat server the edited application becomes available to end-users.

This button is used to create a new concept. A dialog box is shown to enter a name, description and template (type) for the concept and optionally a resource (file name). The resource uses a name relative to the server path. If an application like the "tutorial" shown in Fig. 10 is accessed as /aha/tutorial then the resource to be entered will be file:/tutorial/… (and not /aha/tutorial). The new concept will become a child of the last concept that was selected (by clicking on it).

This button is used to filter the presentation of the concept relationship graph. Fig. 10 shows a structure with only prerequisite relationships. Every type of relationship has a different visual representation. You can select which types are visible and which are hidden. (All remain present. This is only a visualization effect.)

This button activates a check for cycles in the concept relationship graph. Although cycles are allowed some kinds of cycles may cause the adaptation engine to enter an infinite loop of user model updates. This check is performed automatically as you are creating concepts and relationships but it can be useful to execute it on a freshly loaded application.

These three buttons are used to zoom to 100%, zoom in and zoom out. The Graph Author window can also be resized to enlarge the portion of the relationship graph you can see. When resizing and zooming does not help enough to get a good overview of the graph or of some details, try moving concepts around in the graph visualization, or use the filter button (described above).

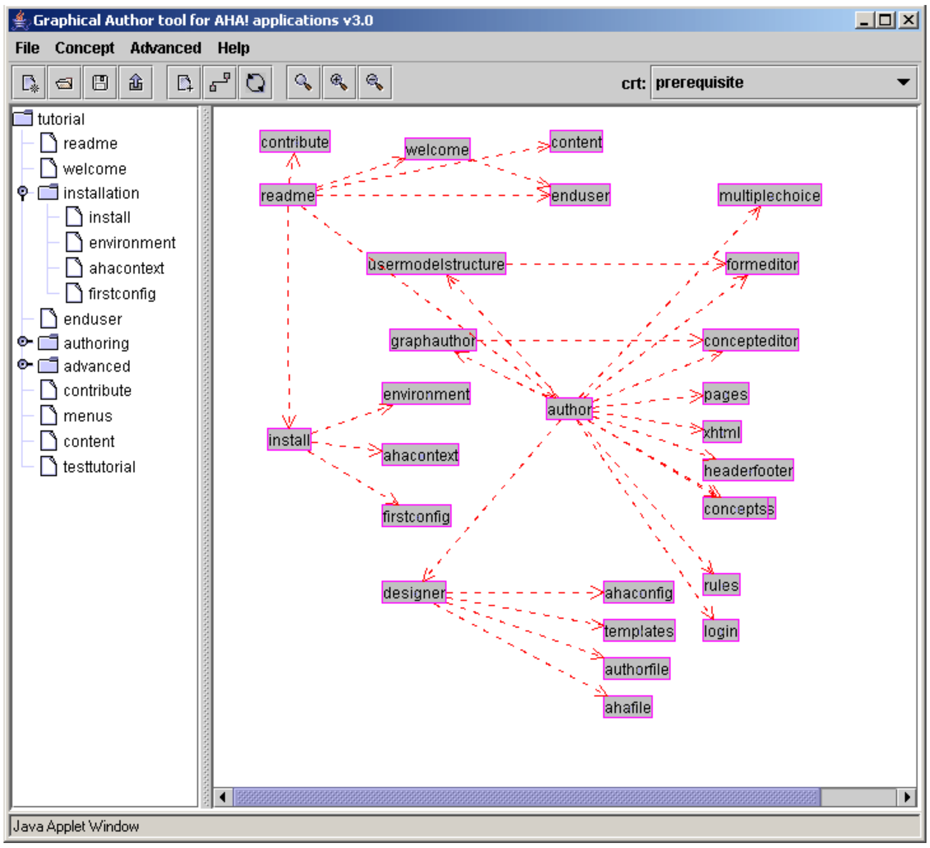


Fig. 10. Screen shot of the Graph Author.

5.1 Concept relationships

The *concept relationship graph* is created by dragging concepts from the hierarchy shown on the left (see Fig. 10) to the drawing pane, and by then drawing arrows between the concepts. You first select the appropriate concept relationship type from the drop-down list (top right in Fig. 10) and then click on the source concept and drag to the destination concept. Some concept relationships may have an optional parameter. By clicking on the arrow a textfield appears in which the parameter value can be entered. For a prerequisite for instance the amount of knowledge that must be exceeded in order for AHA! to consider the prerequisite to be fulfilled is a parameter. (Its default value is 50 in this case.)

Since prerequisite relationships are most used we will explain how to use them and also how they work. All relationship types in AHA! are translated into *adaptation rules*. The details of these rules are explained in Sect. 11.1. For now all we need to know is that a rule assigns a value to an attribute of a concept. Also, the rules are executed (conditionally) when a page (associated with the concept) is accessed. In an application like the “tutorial” one we have three types of concept relationships that play a role:

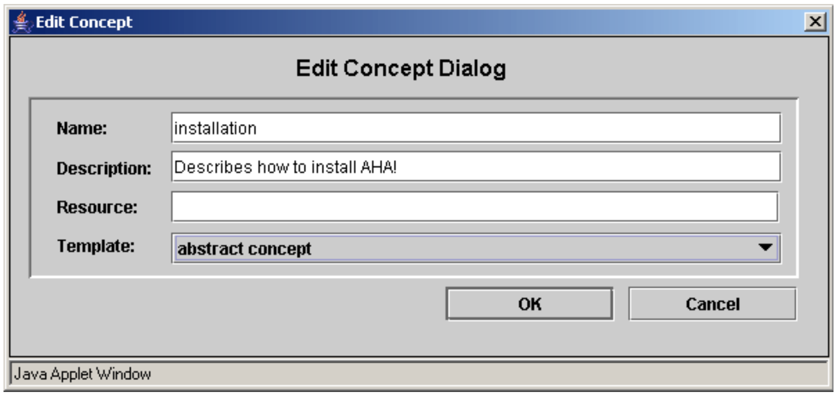
- For every page there is a unary relationship (a relationship from the page to itself), called *knowledge update*. When a page is read the action that is performed depends on the *suitability* of the page. If the *suitability* attribute is “true” then the *knowledge* of the page is set to 100. If it is false then the *knowledge* of the page is increased to 35. (By this we mean the value is set to 35 if it is lower but left at its previous value if that was already over 35.)
- The concept hierarchy shown in the Graph Author is used for *knowledge propagation*. When the *knowledge* of a concept changes that change is propagated to the concepts that are higher in the concept hierarchy. How much knowledge is propagated depends on the number of siblings the concept has. The idea is that when all siblings reach a knowledge level of 100 the parent should have 100 as well. (But due to integer arithmetic and truncation that value may end up being slightly lower.)
- The *prerequisite relationships* determine the *suitability* of a concept. If A is a prerequisite for B, expressed by drawing a prerequisite arc from A to B in the graph, the *suitability* of B depends on the *knowledge* of A. The standard rule requires the knowledge of A to be higher than 50 in order for B to be considered suitable.

As you can see from the description above there is a close interplay between *knowledge* and *suitability*. The interaction between these is such that when a page is read for which the concept is considered not suitable then that concept gets some knowledge value but not enough to make the concepts for which it is a prerequisite suitable.

Hence *prerequisite relationships* exhibit the property of *transitivity*. If A is a prerequisite for B and B for C then A is automatically a prerequisite for C. Without this property the graph of Fig. 1 would have looked a lot more complicated.

5.2 Concept templates

Fig. 11 shows a dialog box for creating/editing a concept in the Graph Author.



Name: installation
Description: Describes how to install AHA!
Resource:
Template: abstract concept

Fig. 11. Dialog box for creating/editing a concept.

Every concept has a name and optional description, and is of a certain *type*, represented by a template. The template determines which attributes the concept has, and whether it must have a resource (file) associated with it or not. An *abstract concept* does not have a resource. A *page concept* must have a resource, like file:/tutorial/xml/install.xhtml. Advanced users can create new templates. AHA! does not currently offer an authoring tool for doing so. The templates are listed in /aha/author/authorfiles/templatelist.txt and are xml files in the directory /aha/author/authorfiles/templates. Below we show part of the page template.

```
<?xml version="1.0"?>

<!DOCTYPE template SYSTEM 'template.dtd'>
<template>
<name>page concept</name>
<attributes>
   <attribute>
      <name>access</name>
      <description>triggered by page access</description>
      <default>false</default>
      <type>bool</type>
      <isPersistent>false</isPersistent>
      <isSystem>true</isSystem>
      <isChangeable>false</isChangeable>
   </attribute>
   <attribute>
      <name>knowledge</name>
      <description>knowledge about this concept</description>
      <default>0</default>
      <type>int</type>
      <isPersistent>true</isPersistent>
      <isSystem>false</isSystem>
      <isChangeable>true</isChangeable>
   </attribute>
   <attribute>
      <name>visited</name>
      <description>has this page been visited?</description>
      <default>0</default>
      <type>int</type>
      <isPersistent>true</isPersistent>
      <isSystem>true</isSystem>
      <isChangeable>false</isChangeable>
   </attribute>
   <attribute>
      <name>suitability</name>
      <description>the suitability of this page</description>
      <default>true</default>
```

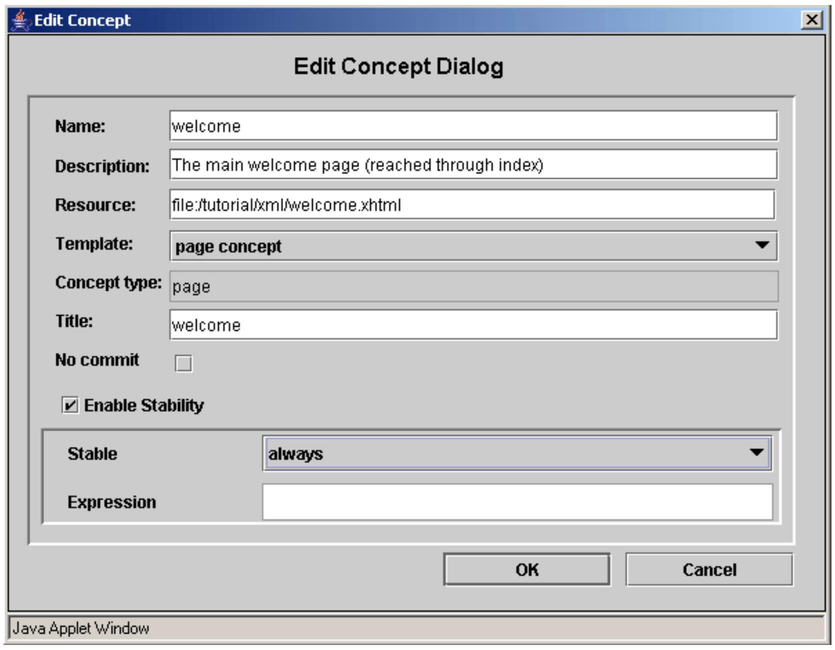
```
        <type>bool</type>
        <isPersistent>false</isPersistent>
        <isSystem>true</isSystem>
        <isChangeable>false</isChangeable>
      </attribute>
   </attributes>
   <hasresource>true</hasresource>
   <concepttype>page</concepttype>
   <conceptrelations>
             <conceptrelation>
               <name>knowledge_update</name>
               <label>35</label>
             </conceptrelation>
   </conceptrelations>
   </template>
```

We see that a page concept has 4 attributes and one unary concept relationship which is the *knowledge update* described earlier. The attributes play the following role:

- **access**: When the page is accessed the rules associated with this attribute are executed (first). The attribute only serves the purpose of starting the adaptation rule engine. `<isSystem>true</isSystem>` indicates that this attribute has a special meaning to the AHA! engine. `<isPersistent>false</isPersistent>` means that the value of this attribute is not stored in the user model.
- **knowledge**: This attribute is typical for educational applications. It stores the system's idea of the user's knowledge of the concept. The knowledge attribute has no special meaning to the AHA! system. It has `<isSystem>false</isSystem>` to indicate this. It stores an integer value in the user model (as can be seen from the `<isPersistent>true</isPersistent>` property. Initially the user's knowledge of every concept is considered to be 0 (`<default>0</default>`). By default the user is allowed to change this value through a form you can create as an author (see Sect. 7), because of `<isChangeable>true</isChangeable>`.
- **visited**: This attribute is used by the system (`<isSystem>true</isSystem>`) to remember (`<isPersistent>true</isPersistent>`) whether the user visited the page or not. This attribute is used by the system to decide whether a suitable link will be displayed using the GOOD (blue) or NEUTRAL (purple) color. The system however does not *set* the value of this attribute. The default *knowledge update* rule will set the value of this attribute (as well as the knowledge attribute). It will only register the page as visited if it was suitable.
- **suitability**: This attribute is used by the system (`<isSystem>true</isSystem>`) to decide on the presentation of links to the page. If the suitability is true the link anchors will be GOOD (blue) or NEUTRAL (purple) depending on the visited status. If the suitability is false the anchors will be BAD (black). The standard style sheet used by AHA! also does not underline links. As a result BAD links are effectively hidden.

In the *advanced* mode of the graph author more aspects of a concept can be controlled. We discuss two of them: stability and resource selection. Fig. 12 shows the edit concept dialog box in advanced mode. The stability can be chosen to be forever after the first adaptation, stability during the current session, or stability while a Boolean expression remains true. A *stable* concept is adapted to the current user model the first time it is presented, and afterwards (during the chosen stability period) it is always presented in the same way even if the user model would suggest a different presentation.

When a concept is defined using a template that has a **showability** attribute (or when a showability attribute is added) the resource can be selected based on expressions. Fig. 13 shows the resource dialog box. When a concept is accessed the resource that will be presented to the user is determined by examining expressions (from top to bottom as entered in Fig. 13). This construct can be used for deciding which page to present when following a link to a concept, but also to decide which resource to include when the <object> tag is used to conditionally include an object. AHA! comes with a "page" template for concepts that are tied to a single page (resource), and a "page concept" template for concepts that have alternative (page) presentations, chosen through the value of the showability attribute. As an author you specify the conditions for selecting a resource (page), as shown in Fig. 13. The Graph Author translates this to values for the showability attribute, and adaptation rules for ensuring that the correct resource is chosen, based on the given conditions.


Edit Concept

Edit Concept Dialog

Name: welcome

Description: The main welcome page (reached through index)

Resource: file:/tutorial/xml/welcome.xhtml

Template: page concept

Concept type: page

Title: welcome

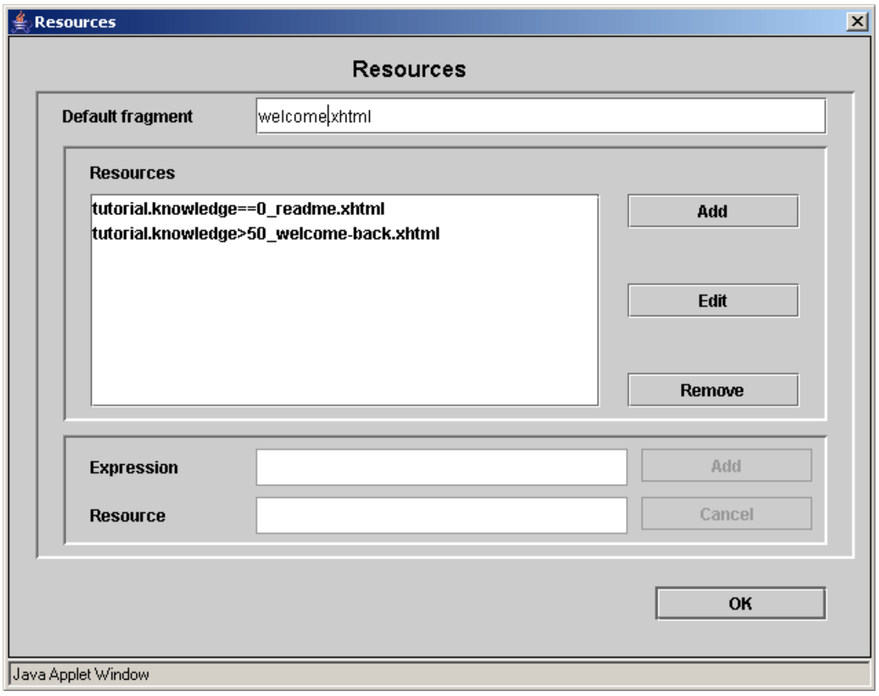
No commit ☐

☑ Enable Stability

Stable: always

Expression:

Fig. 12. The Edit Concept Dialog in advanced mode.


Resources

Resources

Default fragment: welcome.xhtml

Resources

tutorial.knowledge==0_readme.xhtml
tutorial.knowledge>50_welcome-back.xhtml

Expression:

Resource:

Fig. 13. Dialog box for performing resource selection.

5.3 Concept relationship types

Concept relationship types are defined using *templates*, just like concepts. The definition of a concept relationship type consists of two parts: a *presentation* part (used by the Graph Author to decide how to show relationships of the type) and an *implementation* part (used by the Graph Author to translate a conceptual model into the actual adaptation rules used by the AHA! engine. Concept relationship types are defined through xml files. We show the files for *prerequisite relationships* as an example.

```
<?xml version="1.0"?>

<!DOCTYPE author_relation_type SYSTEM 'author_relation_type.dtd'>
<author_relation_type>
   <name> prerequisite </name>
   <color> red </color>
   <style> dashed </style>
   <properties acyclic="true" />
</author_relation_type>
```

This file shows that prerequisites are shown as red dashed arrows (as can be seen in Fig. 10), and that cycles are not allowed (`<properties acyclic="true" />`).

```
<?xml version="1.0"?>

<!DOCTYPE aha_relation_type SYSTEM 'aha_relation_type.dtd'>
<aha_relation_type>
   <name> prerequisite </name>
   <listitems>
      <setdefault location ="destination.suitability"
        combination="AND">source.knowledge &gt; var:50
      </setdefault>
   </listitems>
</aha_relation_type>
```

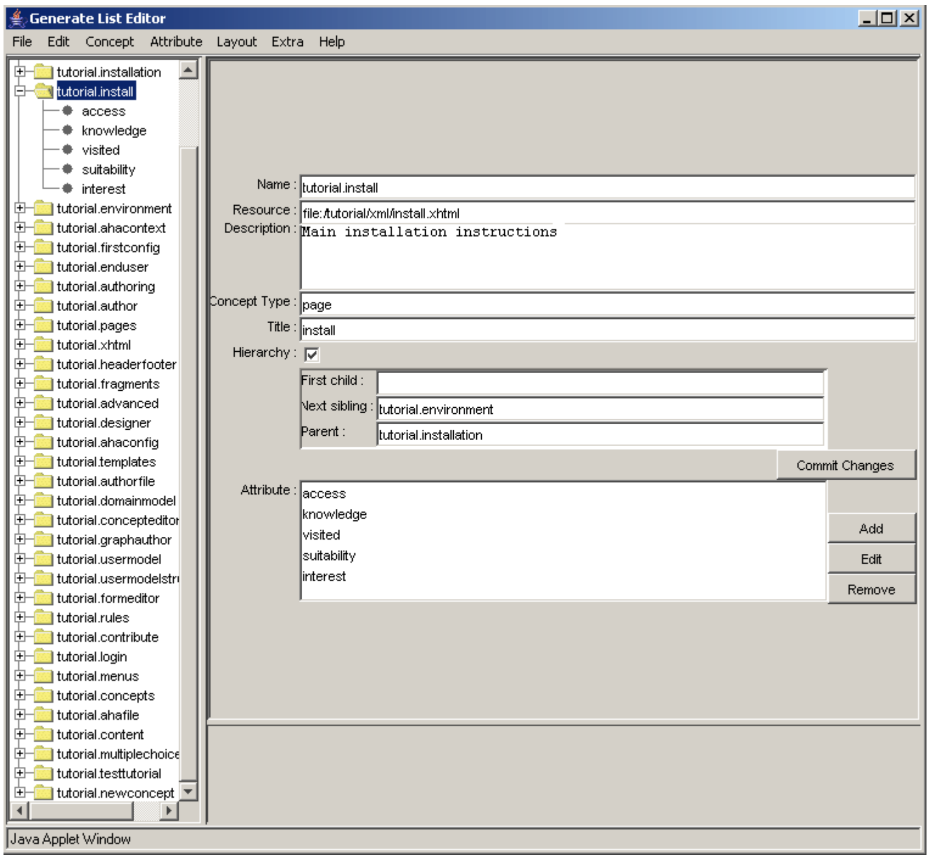
This file shows that the effect of a prerequisite is that the *knowledge* of the source concept should be greater than 50. The result of the comparison between that knowledge and 50 is stored as the *default value* for the *suitability* attribute of the destination. Since suitability is not a persistent attribute its value is not stored in the user model but is calculated each time it is needed. It is possible for a concept to have several prerequisites that *all* need to be fulfilled. This is achieved by combining the expressions with the logical "AND" operator. Note that "OR" is also allowed in a relationship type definition. The templates for other relationship types like the *knowledge update* are more complicated as they define event-condition-action rules that must be generated. We discuss such rules in the next section.

6 The concept editor

When you save an application in the Graph Author, a file in "authoring format" is created that can be edited (but also created from scratch) using the Concept Editor tool. This is a low-level tool in which every bit of functionality of AHA! (regarding concepts, attributes and adaptation rules) can be controlled. Note that whereas the Graph Author can generate files in the Concept Editor's authoring format it cannot import them. Some user interface differences between the Graph Author and the Concept Editor exist for historical reasons only. They may be eliminated in a future version, depending on available manpower.

When you start the Concept Editor it asks for a name and password. After that you can create an application or open an existing application assigned to you. (You have to type the name of the application as we have not yet copied the code that lets you select it from a list in the Graph Author.) Fig. 14 shows the Concept Editor with the same "tutorial" example used earlier. The concept hierarchy is represented inside the concepts, but the Concept Editor (unfortunately) does not show that hierarchy in its left frame. Also, the Concept Editor is sometimes referred to as "Generatelist Editor" for historical reasons. The authoring format with concepts and adaptation rules is called the "generatelist format".

The editor lists all the concepts (of a single application) on the left, and shows details of a selected concept on the right. We will briefly look at the different items that make up all the information related to a concept:

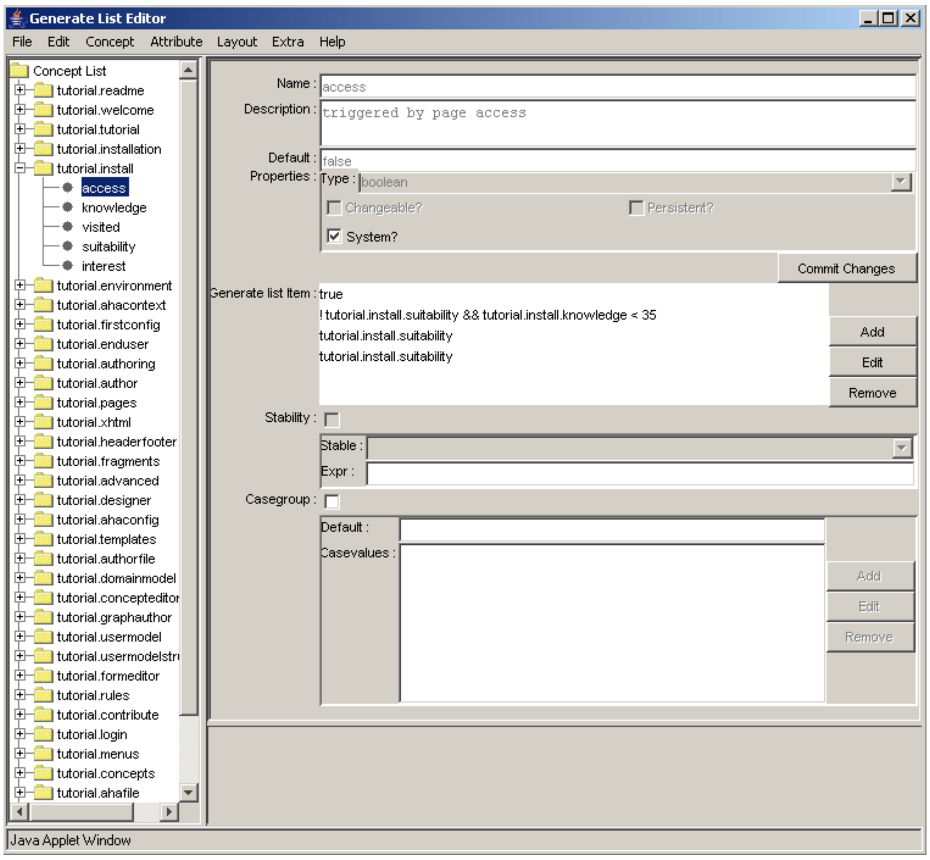


- tutorial.installation
- tutorial.install
  - access
  - knowledge
  - visited
  - suitability
  - interest
- tutorial.environment
- tutorial.ahacontext
- tutorial.firstconfig
- tutorial.enduser
- tutorial.authoring
- tutorial.author
- tutorial.pages
- tutorial.xhtml
- tutorial.headerfooter
- tutorial.fragments
- tutorial.advanced
- tutorial.designer
- tutorial.ahaconfig
- tutorial.templates
- tutorial.authorfile
- tutorial.domainmodel
- tutorial.concepteditor
- tutorial.graphauthor
- tutorial.usermodel
- tutorial.usermodelstr
- tutorial.formeditor
- tutorial.rules
- tutorial.contribute
- tutorial.login
- tutorial.menus
- tutorial.concepts
- tutorial.ahafile
- tutorial.content
- tutorial.multiplechoice
- tutorial.testtutorial
- tutorial.newconcept

Name : tutorial.install
Resource : file:/tutorial/xml/install.xhtml
Description : Main installation instructions
Concept Type : page
Title : install
Hierarchy : ☑
First child :
Next sibling : tutorial.environment
Parent : tutorial.installation
Attribute : access, knowledge, visited, suitability, interest

Fig. 14. The Concept Editor.

- **Name**: The name of a concept consists of “application name” . “concept name”. The Concept Editor mentions the application name explicitly (unlike the Graph Author) because it lets you use concepts from other applications.
- **Resource**: The name of the page, in case of a concept with an associated web page.
- **Description**: An optional description of what the concept means or is used for.
- **Concept type**: The type or template used to create the concept. This determines which attributes are automatically created and which adaptation rules.
- **Title**: AHA! can automatically create an adaptive table of contents. Each concept in that title can be shown with just its name or with an alternative name or “title”.
- **Hierarchy**: A concept can be part of the concept hierarchy (which is shown in the table of contents, used for knowledge propagation, etc.) or not. If the concept is part of the hierarchy it must have a parent unless it is the top of the hierarchy. Concepts can have children in the hierarchy and siblings. The siblings are arranged sequentially. (This order is used in the table of contents.)
- **Attributes**: A concept has a number of attributes, and with each attribute a number of adaptation rules can be associated. We look at a few adaptation rules in detail below.
  Each attribute has a name, optional description and default value. In Fig. 15 we show the *access* attribute, which is a “System” attribute. (Hence the template has determined the name, description and default and in the editor this cannot be changed without first making this attribute no longer a system attribute.) An attribute can be “Changeable”, meaning that it can be used in a form (see Sect. 7 about the Form Editor) that lets end-users change the attribute value. An attribute can be “Persistent”, meaning that the value is stored (permanently, but can be updated) in the user model. Three other aspects of a concept are the adaptation rules, stability and “casegroup”.



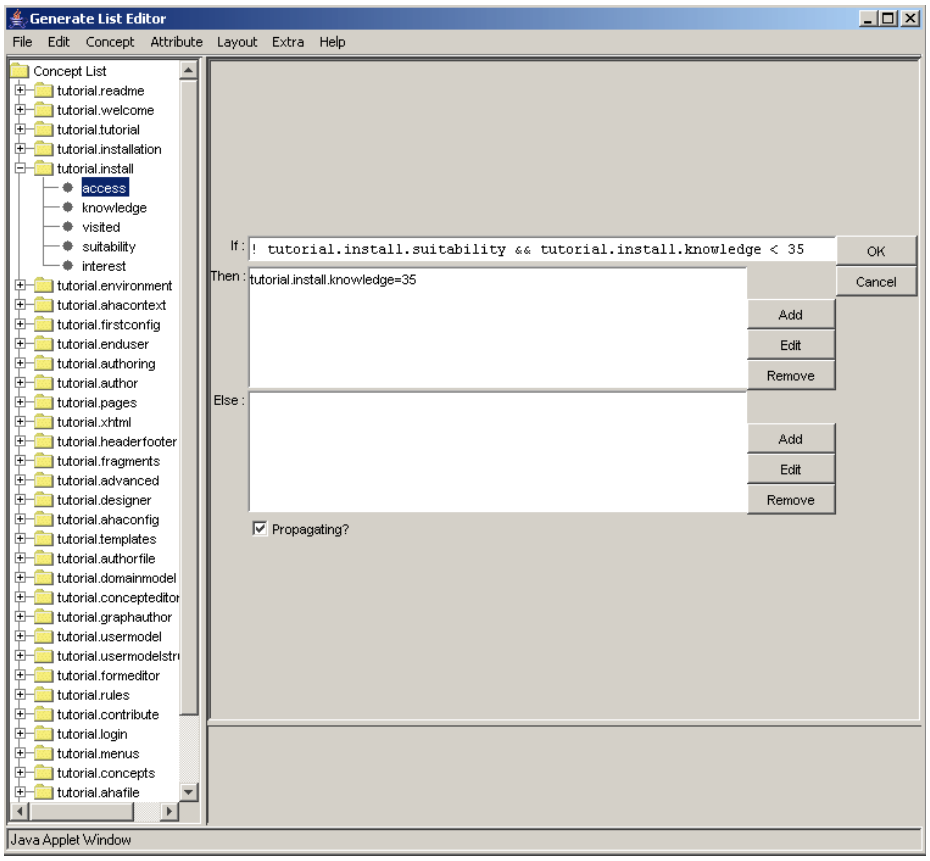
Concept List
- tutorial.readme
- tutorial.welcome
- tutorial.tutorial
- tutorial.installation
- tutorial.install
  - access
  - knowledge
  - visited
  - suitability
  - interest
- tutorial.environment
- tutorial.ahacontext
- tutorial.firstconfig
- tutorial.enduser
- tutorial.authoring
- tutorial.author
- tutorial.pages
- tutorial.xhtml
- tutorial.headerfooter
- tutorial.fragments
- tutorial.advanced
- tutorial.designer
- tutorial.ahaconfig
- tutorial.templates
- tutorial.authorfile
- tutorial.domainmodel
- tutorial.concepteditor
- tutorial.graphauthor
- tutorial.usermodel
- tutorial.usermodelstr
- tutorial.formeditor
- tutorial.rules
- tutorial.contribute
- tutorial.login
- tutorial.menus
- tutorial.concepts
- tutorial.ahafile

Name : access
Description : triggered by page access
Default : false
Properties : Type : boolean
☐ Changeable? ☐ Persistent?
☑ System?

Generate list Item :
```
true
!tutorial.install.suitability && tutorial.install.knowledge < 35
tutorial.install.suitability
tutorial.install.suitability
```
Stability : ☐
Stable :
Expr :
Casegroup : ☐
Default :
Casevalues :

Fig. 15. Information about an attribute in the Concept Editor.

- **Adaptation rules** (or “generate list items”) are *event-condition-action rules*. The *event* that triggers the execution of a rule is a change to the attribute value. For the “pseudo-attribute” *access* the event is the page access. (It temporarily changes the value of the *access* attribute from false to true. Fig. 15 shows the *condition* of each of the rules associated with the attribute (one line per rule). When we click on the condition and then “edit” we get to see the actions that are performed when the condition is true, and the optional actions for when the condition is false. Fig. 16 shows the actions for !tutorial.install.suitability && tutorial.install.knowledge < 35. The rule shows that the knowledge is set to 35 if the condition is true.
- When the “**Stability**” property is checked the presentation of the concept can be “frozen” in three ways (selected from the drop-down list). The stability “*always*” means that the presentation is adapted to the user model state the first time the concept is presented, and after that the presentation remains the same. The “*session*” stability means that the adaptation will happen the first time in every (login) session. The “*freeze*” stability means that the presentation is frozen as long as the associated expression remains true. (When false then normal adaptation is done.)
- The **“Casegroup**” checkmark indicates that multiple resources are associated with the concept and that has different resources associated with it. This feature is normally only used with a system-attribute called *showability*. It is used to conditionally select a resource (file) to include in the presentation in the location of a special <object> tag (See Sect. 10) or to select the page to present when a link to a concept is followed (instead of a link directly to a page).



If : `! tutorial.install.suitability && tutorial.install.knowledge < 35`

Then : `tutorial.install.knowledge=35`

Else :

☑ Propagating?

Fig. 16. Editing an event-condition-action rule in the Concept Editor.

7 The Form Editor

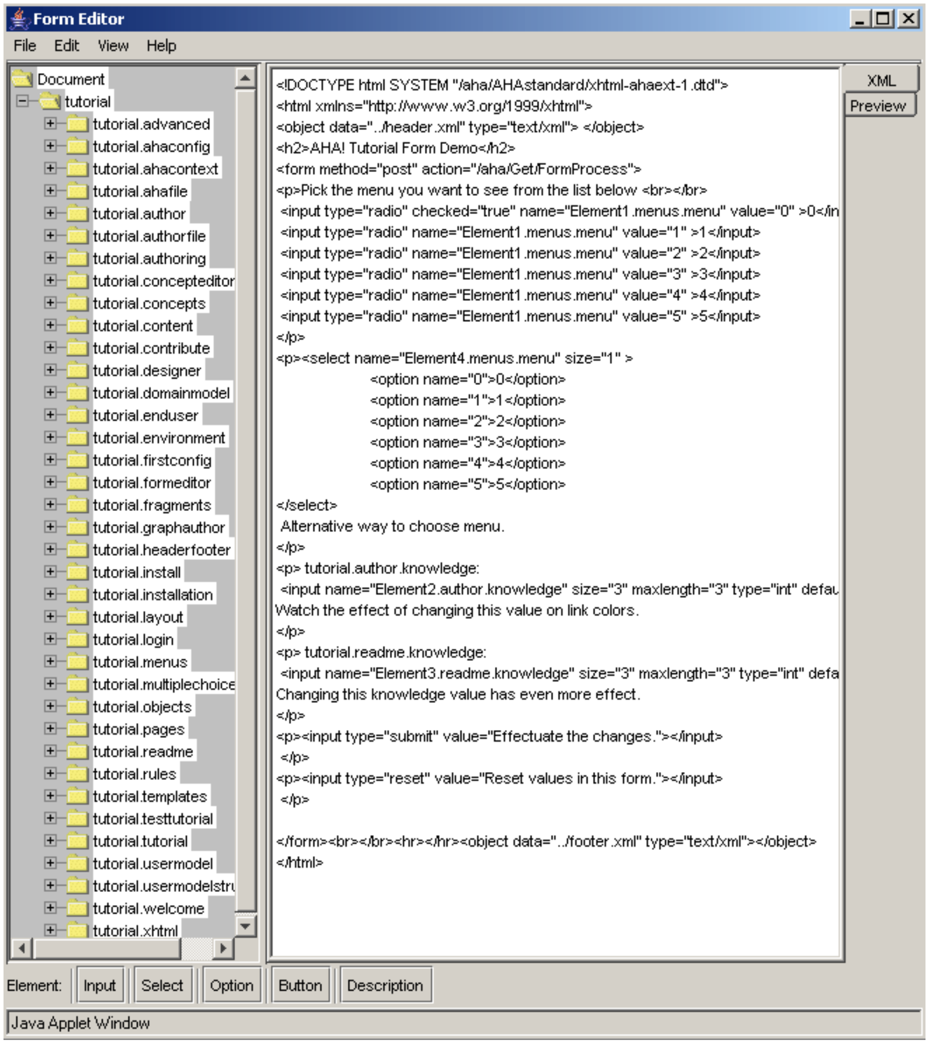
Attributes of concepts defined as “changeable” can be included in a custom-made form. The Form Editor lets you create an (X)HTML form in which form elements for the attributes of concepts are inserted automatically, and in which you create the remainder of the presentation by means of plain HTML code.

A form is bound to an application, so when creating a new form you have to “load” the conceptual structure of that application. (File, Load AHA! application). The form editor creates a skeleton representing an empty form. You can add HTML code for the presentation and use the buttons “Input”, “Select”, “Option” and “Button” to add form elements. You first select a changeable attribute of some concept and then decide on the type of form element to create:

- **Input**: This is a text field in which numbers or arbitrary text can be entered by the end-user. You define the *Size* (the number of characters there is room for on the screen), *MaxLength* (the maximum length of the input the user is allowed to provide; this is typically at least *Size*), a *Default* value to show when the field is presented to the end-user, and an optional *Description* that will preceed the form element in the presentation of the form. AHA! takes the type of the attribute into account and will check whether the end-user entered numbers only when the attribute is an integer.
- **Select**: This is a selection list. It can display a number of choices at once (*Size*) and use a scrollbar when the list is longer. The list itself (*EnumList*) must be a list of values separated by semicolons (;).
- **Option**: This is a series of checkboxes or radio buttons. Radio buttons are used when the *Allowed* number of choices is 1. Otherwise checkboxes are used. The *EnumList* of choices is again a semicolon-separated list of values.
- **Button**: This creates a *submit* or *reset* button. It can be given a *name* that is displayed in the button.

A form can be viewed as HTML source and can be *previewed*. The Form Editor uses a standard Java HTML editor class to do this. At the time of writing this tutorial this standard class is not yet fully XHTML compliant, so you will have to use a somewhat simplified HTML, which is normally enough for a simple form.

Forms created with the Form Editor are saved in your authoring directory. You have to copy them to whichever location on the server you want to use to refer to them from within the content pages that have a link to them. (We are working on a tool that lets you create forms inside the application's document tree right away.)



```
<!DOCTYPE html SYSTEM "/aha/AHAstandard/xhtml-ahaext-1.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<object data="../header.xml" type="text/xml"> </object>
<h2>AHA! Tutorial Form Demo</h2>
<form method="post" action="/aha/Get/FormProcess">
<p>Pick the menu you want to see from the list below <br></br>
 <input type="radio" checked="true" name="Element1.menus.menu" value="0" >0</in
 <input type="radio" name="Element1.menus.menu" value="1" >1</input>
 <input type="radio" name="Element1.menus.menu" value="2" >2</input>
 <input type="radio" name="Element1.menus.menu" value="3" >3</input>
 <input type="radio" name="Element1.menus.menu" value="4" >4</input>
 <input type="radio" name="Element1.menus.menu" value="5" >5</input>
</p>
<p><select name="Element4.menus.menu" size="1" >
                <option name="0">0</option>
                <option name="1">1</option>
                <option name="2">2</option>
                <option name="3">3</option>
                <option name="4">4</option>
                <option name="5">5</option>
</select>
 Alternative way to choose menu.
</p>
<p> tutorial.author.knowledge:
 <input name="Element2.author.knowledge" size="3" maxlength="3" type="int" defau
Watch the effect of changing this value on link colors.
</p>
<p> tutorial.readme.knowledge:
 <input name="Element3.readme.knowledge" size="3" maxlength="3" type="int" defa
Changing this knowledge value has even more effect.
</p>
<p><input type="submit" value="Effectuate the changes."></input>
 </p>
<p><input type="reset" value="Reset values in this form."></input>
 </p>

</form><br></br><hr></hr><object data="../footer.xml" type="text/xml"></object>
</html>
```

Fig. 17. Form Editor with an example form.

8 Multiple choice tests

AHA! 3.0 has a new module for multiple choice tests. It was developed at the University of Cordoba (by Cristobal Romero). You can create tests with an arbitrary number of questions and arbitrary numbers of answers per question, and have the system automatically select questions and answers to present. You can give elaborate feedback or just a score. We will not describe the details of the Multiple-Choice Test Editor and the use of tests in this tutorial.

9 The Layout Manager

Web-based applications (whether adaptive or not) often have a distinctive *look and feel*. They consist of several HTML frames, each used for a specific purpose, like a header, a menu or table of contents, and a main frame with text. Some existing adaptive hypermedia systems, like Interbook [3], have a very distinctive presentation style that cannot be changed. Older versions of AHA! had no presentation style, so you could create any style using frames (and style sheets) at will. But they also did not offer a way to implement a standard style and apply it to a whole presentation.

In AHA! 3.0 we use a *Layout Model* to determine the style of a presentation [4]. We will introduce this feature through a number of examples. The layout model consists of three types of entities:

- A **view** is an atomic presentation unit. A view can present an HTML file from the application, or one of a number of automatically generated content or links to other (secondary) views.
- A **viewgroup** is a grouping of views to form what is presented in a single window. A viewgroup is translated to an HTML frameset in which every view is a frame. "Compounds" are used to define the frames structure and identify which view is to be presented in which frame. When an application contains different types of presentations, e.g. information pages, a table of contents, a glossary, etc., different viewgroups can be defined. Viewgroups indicated as "secondary" will always presented in a separate window.
- A **layout** a set of viewgroups that together form a presentation. A single application may have different layouts for different types of concepts. The presentation for page concepts (with additional information) can be different from the presentation of an abstract concept for instance.

Currently the layout of an application must be named LayoutConfig.xml and reside in the main directory of the application. It consists of a `<viewlist>` which provides the names of all the views that exist in he application, and a `<layoutlist>` which defines how the different layouts are assembled from viewgroups using the views from the viewlist. The example we describe below tries to mimic (part of) the layout of Interbook applications. Note that this is just one of many possible layouts you can define for similar applications.

- The simplest example of a view is the "main" view used to contain the contents of a page, corresponding to a requested concept (which we call the "current page" or "current concept". It is defined as:

  ```
  <view name="v3" type="MainView" />
  ```

  A view can have several attributes. Name is obvious. Type indicates which information needs to be shown in the view. "MainView" is the type that represents that the view will simply present the contents of a (HTML) file. This is always the "main" part of the application. Apart from MainView AHA! currently offers a number of types of views that are inspired by Interbook [3]. More view types will be added in the future as needed.
  - EmptyView has the obvious meaning of representing an empty view, used to take up space (possibly with a background image) but presenting no content.
  - TOCView is a view presenting a complete table of contents, generated from the concept hierarchy. The items in the table of contents are indented according to their level in the hierarchy.
  - PathView is a partial table of contents, showing the path from the top of the concept hierarchy down to the current concept, and it also shows the siblings of that concept.
  - ChildrenView is a view presenting a list of children of the current concept (in the concept hierarchy).
  - TreeView is a view that combines PathView and ChildrenView: it shows the path from the top of the concept hierarchy down to the current concept, the siblings and the children of that concept.
  - ConceptbarView is a view presenting a list of concepts to which the current page contributes knowledge.
  - GlossaryView is a view presenting glossary concepts in Interbook style.
  - ToolboxView is a view with links to secondary windows.
- Common parameters that can be assigned to a view, using the "params" attribute, are *leftspace* to create a left margin, *cType* to indicate the type (template) of concepts to show in the Glossary, and different parameters to indicate certain graphical artifacts or icons. A background image is indicated using a different parameter called "background".
- A *viewgroup* represents a window. A number of parameters of the viewgroup tag define properties of the viewgroup, such as whether it is resizable and what should be the initial width and height. A viewgroup can consist of:
  - A single view to be displayed in the window, e.g.:

    ```
    <viewgroup name="TOC" secondary="true"
           wndOpt="status=1,menubar=1,resizable=1,toolbar=1,
           width=300,height=400">
        <viewref name="v1"/>
    </viewgroup>
    ```

    As shown above the view is referred to by name (v1), using a viewref tag.
  - A *compound* defines a structure with different HTML frames. It indicates which view should be presented in which frame. A compound may contain other compounds and viewrefs at the same time.

    ```
    <compound framestruct="rows=20%,*" border="0">
        <compound framestruct="cols=*,145" border="0">
             <viewref name="v7" />
             <viewref name="v5" />
        </compound>
        <compound framestruct="cols=70,*,145" >
    ```

```
            <viewref name="v6" />
            <compound framestruct="rows=*,25%" >
               <viewref name="v3" />
               <viewref name="v8" />
            </compound>
            <viewref name="v2" />
        </compound>
    </compound>
```

The adaptation that is performed in a given layout is defined in the file ConceptTypeConfig.xml (also in the application's main directory). In AHA! each concept has a *type* (as can be seen in Fig. 12). Concepts of a type are presented using the layout that is defined for that type. The way in which adaptation or annotation is done can be different for every type. One can define icons to be placed in front of a link, for instance as:

```
    <fronticon>
        <good>icons/GreenBall.gif</good>
        <bad>icons/RedBall.gif</bad>
        <neutral>icons/WhiteBall.gif</neutral>
        <unconditional></unconditional>
    </fronticon>
```

This produces Interbook-style annotation using green, white and red balls. One can also produce icons (after a link) that depend on the value of an attribute. In Interbook checkmarks can be placed behind concepts in the ConceptBar view to indicate the knowledge level of the concept. In AHA! this is done as follows:

```
    <iconanno>
        <attribute>
            <name>knowledge</name>
            <distribution>
                <boundary>0</boundary>
                <boundary>33</boundary>
                <boundary>55</boundary>
                <boundary>76</boundary>
                <boundary>101</boundary>
            </distribution>
            <results>
                <result>icons/NoCheckM.gif</result>
                <result>icons/SmallCheckM.gif</result>
                <result>icons/MedCheckM.gif</result>
                <result>icons/BigCheckM.gif</result>
             </results>
        </attribute>
    </iconanno>
```

Similarly annotations can be defined for links in a PathView or TreeView to indicate whether a concept has children or not.

The initial implementation and examples of the use of predefined layout definitions was inspired by Interbook. However, views have been defined that are not present in Interbook (e.g. the TreeView) and most of the look and feel of the application is in the choice of icons and colors, which are all configurable in the layout. The on-line version of the AHA! 3.0 tutorial (also included in the distribution) has a distinctive look of its own.

10 The application content (pages)

Every AHA! application consists of pages, in HTML or XHTML (and possibly some other formats like SMIL), and of auxiliary files like icons and images. All the files of an application are stored in the directory tree that starts with the name of the application. The example "tutorial" application will be known to the browser as /aha/tutorial/ but will be stored in the directory /tutorial on the server (below the AHA! documentroot). All the files of the application can be stored in this directory or in subdirectories, and they can refer to each other using relative or absolute addresses.

- All pages (or objects) referred to using a relative address (like "readme.xhtml" or "../images/example.gif") are processed by the AHA! engine. All references to pages (or objects) that contain conditional fragments or conditionally presented or adapted links must be of this type.
- All pages (or objects) referred to using an absolute local address (like "/aha/tutorial/images/example.gif" are served directly by the Tomcat server. No adadaptation is done to these resources.

Although AHA! contains some limited backward compatibility support for plain HTML, it works best with XHTML, with or without some AHA! extensions. In AHA! XHTML pages the most important elements are:

- Links in pages are indicated with an <a> tag like in standard HTML. AHA! considers three types of links: “conditional”, “unconditional” and “external”. Links in (X)HTML can have a class assigned to them. A conditional link will look like:
  `<a href=”link-dest.xhtml” class=”conditional”>anchor text</a>`
  and likewise for an “unconditional” link. Links without a class or with another class value are considered external links. They are shown using colors that are different from conditional and unconditional links. A link can point to a page or a concept. When a link points to a concept the adaptation rules (the “Casegroup” in particular) are used to determine which page to show. (Sect. 5.2 explains how to assign resources to concepts.)
- Conditional fragments in XHTML pages can be created using an <if> tag that exists in an AHA! extension to XHTML. (Use xhtml-ahaext-1.dtd.) The “if” contains an expression using user model concepts and attributes. An example:

```
<if expr=”tutorial.rules.knowledge&gt;50”>
<block>
  Here some text for people with knowledge about rules.
</block>
<block>
  Here some text for people without knowledge about rules.
</block>
</if>
```

- In AHA! it is also possible to conditionally include fragments without using the special additional <if> tag. The <object> tag is used instead, with a special type of “aha/text” [5]. The tag looks like:

```
<object name=”tutorial.conditionalobject” type=”aha/text”>
```

  The name refers to a concept that has a “Casegroup” definition to determine which file to include depending on an expression using user model concepts and attributes. Fig. 13 shows how to create a “Casegroup” in the Graph Author; Fig. 15. does the same for the Concept Editor.

At this time AHA! does not offer a special tool for creating content pages on the server. You need a login or ftp upload facilities for creating pages, images and other auxiliary files, and also to place forms created with the Form Editor within the application’s directory tree. (An upload/download tool is being developed.) Also, the current AHA! implementation creates a “temp” directory in the application directory tree. When you change the concept structure you have to remove that directory and let AHA! recreate it. (This will change in a future release.)

11 Advanced topics

11.1 Event-Condition-Action rules

In Sect. 6 we have explained the adaptation rules. We briefly show an example of the syntax of the rules as they appear in the “authoring format” used by the Graph Author and the Concept Editor.

```
<generateListItem isPropagating="true">
   <requirement>! tutorial.readme.suitability &amp;&amp;
              tutorial.readme.knowledge &lt; 35</requirement>
   <trueActions>
      <action>
         <conceptName>tutorial.readme</conceptName>
         <attributeName>knowledge</attributeName>
         <expression>35</expression>
      </action>
   </trueActions>
</generateListItem>
<generateListItem isPropagating="true">
   <requirement>tutorial.readme.suitability</requirement>
   <trueActions>
      <action>
         <conceptName>tutorial.readme</conceptName>
         <attributeName>knowledge</attributeName>
         <expression>100</expression>
      </action>
   </trueActions>
</generateListItem>
<generateListItem isPropagating="true">
   <requirement>tutorial.readme.suitability</requirement>
```

```
      <trueActions>
         <action>
            <conceptName>tutorial.readme</conceptName>
            <attributeName>visited</attributeName>
            <expression>100</expression>
         </action>
      </trueActions>
   </generateListItem>
```

These three event-condition-action rules are tied to an attribute (in this case “access”) of a concept (in this case “readme”) of the tutorial application. (The attribute and concept cannot be seen in the example.) When a rule is triggered its condition, called “requirement” is checked first. It is a Boolean expression. The first rule is executed when tutorial.readme.suitability is false and tutorial.readme.knowledge is lower than 35. Note the strange escape sequences &amp; and &lt; in the expression. These are needed because the XML parser will translate them to & and <. Without the escaping the XML parser would *interpret* them as & and < instead of *translating* them into & and <. Depending on the outcomeof evaluating the expression either the “trueactions” or the “falseactions” will be performed. In the example there are no “falseactions”. The action will assign the value 35 to the knowledge attribute of tutorial.readme. The second rule assigns the value 100 to the knowledge attribute of tutorial.readme if the suitability of tutorial.readme is true, and the third rule sets the visited attribute of tutorial.readme to 100 if the suitability of tutorial.readme is true. Note that the second and third rule could have been combined into a single rule. This was not the case because the rules were automatically generated by the Graph Author.

11.2 Special Forms and Reports

AHA! has a number of special tags for generating some built-in reports and forms. These tags can be used in a header or footer, included with a normal <object> tag (not of type aha/text but normal text/xml). The possible tags are defined in the standard “headerfooter.dtd”. Each tag is used either as a *handler* or a *variable*. We just mention a few as examples:

- **username**, **loginid**, **email**, **university**, **department**, **course** and **title** are variables that are replaced by their value from the user model, if that exists. Such variables exist if the registration form for the application has a field that asks for their values or that gives values in hidden fields.
- **numberdone** and **numbertodo** are variables that are replaced by the number of pages read and the number of pages still to read within the current application.
- **done** and **todo** are handlers for links to a page that lists the names of the visited resp. unvisited pages (corresponding to concepts).
- **knowledgesettings** is a handler for a link to a standard form that shows the value of the knowledge for all concepts that have this attribute and for which that attribute is marked as “changeable”.

12. Concluding remarks and future work

This tutorial describes most of the functionality of AHA! version 3.0. The current implementation is still a prerelease, so while using the tutorial you might find some features that behave differently or incorrectly or not at all. They will be corrected in the future. Also, AHA! is always distributed with an embedded tutorial that reflects the functionality of the distributed release. Please check the included tutorial for the latest details.

AHA! is continuously being further developed and improved. We welcome contributions from other groups and will do our best to include the contributions in future releases. We also welcome remarks and suggestions for improvement of the tutorial.

Acknowledgement

The development of AHA! versions 2.0 and 3.0 was largely possible thanks to a grant from the NLnet Foundation. The Minerva ADAPT project (101144-CP-1-2002-NL-MINERVA-MPP) has also contributed, as well as the PROLEARN Network of Excellence in Professional Learning.

References

1. Brusilovsky, P., Methods and techniques of adaptive hypermedia. User Modeling and User-Adapted Interaction 6 (2-3), pp. 87-129, 1996.
2. Brusilovsky, P., Adaptive hypermedia. User Modeling and User-Adapted Interaction, 11 (1-2), pp. 87-110, 2001.
3. Brusilovsky, P., Eklund, J., Schwarz, E. Web-based education for all: A tool for developing adaptive courseware. Computer Networks and ISDN Systems (Proceedings of the 7th International World Wide Web Conference, pp. 240-246, 2003.
4. Brusilovsky, P., Santic, T., De Bra, P., A Flexible Layout Model for a Web-Based Adaptive Hypermedia Architecture. Proceedings of the AH2003 Workshop, TU/e CSN 03/04, pp. 77-86, 2003.
5. De Bra, P., Aerts, A., Berden, B., De Lange, B., Escape from the Tyranny of the Textbook: Adaptive Object Inclusion in AHA!. Proceedings of the AACE ELearn Conference, pp. 65-71, 2003.
6. De Bra, P., Aerts, A., Berden, B., De Lange, B., Rousseau, B., Santic, T., Smits, D., Stash, N., AHA ! The Adaptive Hypermedia Architecture, Proceedings of the 14 th ACM Conference on Hypertext and Hypermedia, pp. 81-84, 2003.