Creating Adaptive Web-Based Applications

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. The subjects of adaptation and user modeling go hand in hand. User modeling is often performed in order to adapt an application to the user, and the user's interaction with an application is the basis for the user modeling. This tutorial describes the overall architecture of adaptive Web-based (hypermedia) applications, based on the request/response paradigm used on the Web. Events (like page requests or forms that are filled out and submitted) trigger some form of "rules" that are used to update a user model. The requested information (or other response of the system) is filtered based on the current user model state. The content and presentation of information pages can be adapted, and the links that are shown inside or alongside the pages can be adapted too. After studying the general framework we describe how to create adaptive Web-based applications using the AHA! system (version 3.0). In this system the adaptation is defined using *concepts* and *concept relationships*. These can be created through a graphical (graph-based) editor, or through other tools that provide a translation to the xml-based AHA! authoring format(s). The content of an application is created using standard xhtml (and other media objects that can be embedded in xhtml pages). For education applications AHA! also offers a module to create multiple-choice tests. 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

When World Wide Web was first introduced it was intended as a means for sharing information. It uses information pages and links, and is therefore a form of *hypermedia*. Nowadays the Web is being used as a common user interface platform for all different kinds of applications, from communication (through discussion forums), shopping, banking and other global types of services down to controlling your heating, lighting and other types of devices in the home, and using interface devices ranging from mobile phones to huge computer monitors and wall-size displays. In this tutorial we are not studying the possibilities for *adaptive technology* in all these aspects of the Web, but concentrate on the Web as the hypermedia platform it is mainly used for: we look at applications in which users interact with information (pages), like in on-line courses, news-services, museum sites, (shopping) catalogs, encyclopedia, etc. We look at how these applications can adapt themselves to their individual users or user groups, through the interaction of the users with the applications.

Adaptive hypermedia systems (or AHS for short) [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. Creating adaptive hypermedia on the Web requires server-side functionality for user modeling and for the adaptive generation of (HTML) pages. In this tutorial we are going to study both these aspects. The emphasis is on adaptation of the presented *information* and of the *links* between different information items (or pages). The interaction of the user with the system consists mainly of *page requests* (by clicking on link anchors). In a Web-based environment much of the interaction is local to the browser or client-side, like moving the mouse, scrolling the page while reading, perhaps zooming in or out (in images or in text by changing the font size). These parts of the interaction cannot be used as a basis for adaptation in a Web-based environment as we study it in this tutorial. (There is research on this micro-interaction, for instance by performing eye-tracking and relaying that information to the server side. We do not consider such micro-interaction in this tutorial.)

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. We will show a classification of the main types of adaptation offered by the different systems (based on [1,2]). Even though these systems and their features are very different, we can capture all this functionality in one common reference model, the Dexter-based [8] AHAM model [7]. This model suggests that it must be possible to design and develop a single AHS that captures most of the adaptive functionality found in other systems. The AHA! system [6], or Adaptive Hypermedia Architecture, was designed and implemented at the Eindhoven University of Technology specifically with the purpose of being a *general-purpose* AHS. This project was sponsored by the NLnet Foundation through the AHA! project, or Adaptive Hypermedia for All. AHA! offers high-level facilities for creating the conceptual structure of an application, using *concepts* and *concept relationships*. This structure forms the basis for the adaptation (and it is translated into low-level adaptation rules used by the adaptive engine). A secondary aspect of being truly *general-purpose* is the ability to also recreate the *look and feel* of other AHS. AHA! offers facilities for creating exactly the desired look-and-feel for each application. AHA! is not only an adaptive server. It can also function as a client for other servers (requesting and filtering pages from other servers). It can thus be used as a component in a content delivery *pipeline* and thus integrated into other server environments.

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 Overview of Adaptive Hypermedia Methods and Techniques

In this section we first have a brief look at the reasons *why* we want to use adaptation, and then consider the possible ways to achieve such adaptation in a Web-based information presentation.

2.1 Why Adaptive Hypermedia?

In order to determine why we want to use adaptation in hypermedia and Web-based applications we investigate the potential problems that are apparent in non-adaptive applications, and that might solved through adaptation.

When creating a hypermedia application it is important to give the end-user a lot of navigational freedom. After all, hypermedia breaks out of the linear confines of the book metaphor that plagued us for centuries. However, imagine what happens when you open an average book on a random page. Chances are you won't fully understand the text. And you won't be surprised either! But now imagine that you go to a hyperdocument, and click on a link. Chances are you won't fully understand what the page says. But this time you will be surprised, disappointed maybe. Surely the author has foreseen that some users would follow that link, because the author created that link, no? Well, that may be true but it is impossible to foresee all possible navigation paths, given that there are exponentially more paths than links. So using hypermedia is difficult, first of all because you often encounter information you don't quite understand. You experience this difficulty because the author expected you to visit some other page first.

Another problem is that when the hyperspace is reasonably large you can easily get lost in that hyperspace. A graphical overview is only a partial solution because the hyperspace is usually much too large to be displayed. You need some kind of "localized" map which shows you where you have been, where you are and where you may go next.

Adaptive hypermedia can *in principle* solve these problems: because your actions are monitored the system can compensate for missing foreknowledge. It "knows" whether you have read the pages the author expects you to read before following a certain link. It can add an explanation if needed (and leave it out when not needed). Also, since the system knows that following a link may lead to a page you cannot (yet) understand it can warn you about this. Adaptive hypermedia can also help with the "lost in hyperspace" problem because it can offer an overview of just that part of the hyperspace that is of interest to you. Not only will this overview be much smaller than a complete one, it will only contain references to material you can relate to instead of a mass of topics you are not interested in and probably do not understand.

Whether you need the *content* to be adapted to your knowledge, interest and goals, or whether the *links* should be adapted to guide you towards appropriate information depends on the application area of the adaptive website.

- *Adaptive educational hypermedia systems* are derived from the area of intelligent tutoring systems. The main difference is that in an intelligent tutoring system the focus is on the *tutor* who decides what the learner should study next. These decisions are based on what the learner reads and how he or she performs on tests. The main function of an intelligent tutoring system is often described as *adaptive course* *sequencing* and is thus achieved through *link adaptation*. In an adaptive educational hypermedia application the focus is on the *learner*. The system may provide guidance, but should not enforce a single reading order upon the learner. As a result, different learners will study the material in a different order, and *content adaptation* in the form of additional explanations may be needed to avoid the problem of the learner not understanding the material.

- In *adaptive information systems*, like electronic shopping sites, tourist services (like hotel and airline reservations), but also information services like encyclopedias, the systems need to adapt to the user's goals and interests. The services can make use of permanent user information (like name, gender, age, address) as well as session-bound goals such as a topic of interest, travel destination, etc. Information systems often require users to fill out forms and to select options in an otherwise more or less fixed process. The automatic insertion of adaptive *defaults* is a very useful form of content adaptation. But also the reduction of the number of choices (by eliminating very unlikely or even impossible choices) can greatly simplify the interaction. Such operations can be considered as *link adaptation* (because links associated with the eliminated choices are eliminated as well).
- In *adaptive recommender systems*, like an (adaptive) on-line TV guide, the adaptation is based on (long term) *interests* and *preferences* of the users, and properties of the individual as well as a group. Links to the suggested information items (TV programs for instance) will be sorted according to *relevance*. Alternatively the (links to the) suggested items may be annotated with colors or icons to indicate relevance. Either way the system is again using *link adaptation*.

2.2 What do we adapt in AH?

As described above, we encounter two types of adaptation: the *information* you see on a page may be different from what someone else sees because it is adapted to your needs. Brusilovsky [1,2] calls this *adaptive presentation*, because in general the adaptation can deal with the message itself or just with the presentation form of that message. The same information can often be conveyed using a long or short text, an image, video or audio fragment. Some people find it easier to understand a long textual presentation, others prefer it short, and others prefer a video. The second type of adaptation is the adaptation to the *order* in which we access and read, view or hear the presentation. This order is influenced by manipulating the structure and presentation of links. Brusilovsky [1,2] calls it *adaptive navigation support*. Figures 1 and 2 show a detailed taxonomy of the adaptation techniques, and are adapted from [2].

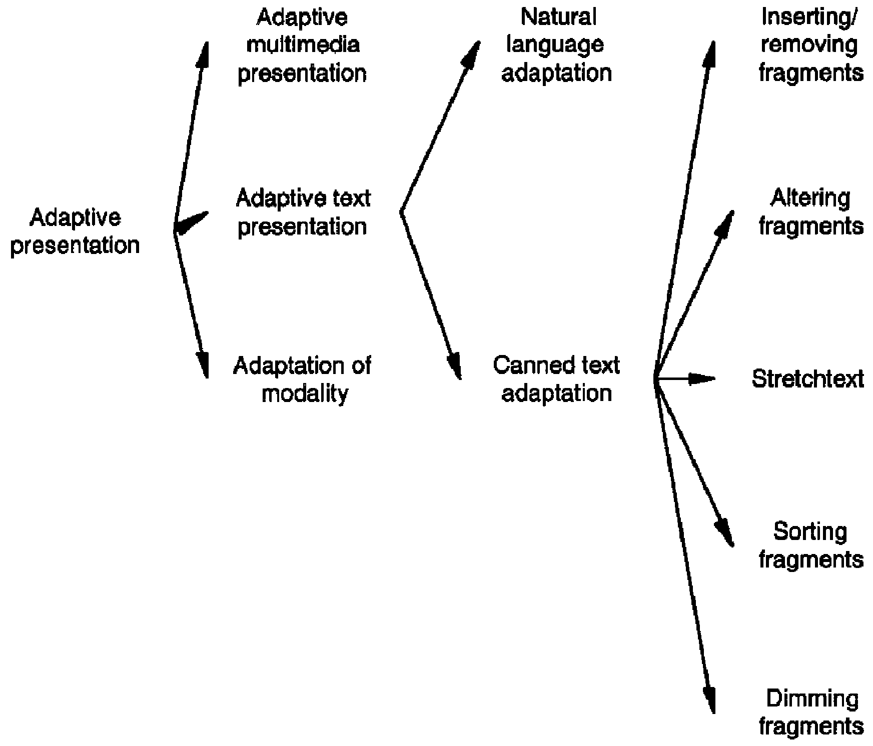


Figure 1. Adaptive presentation techniques.

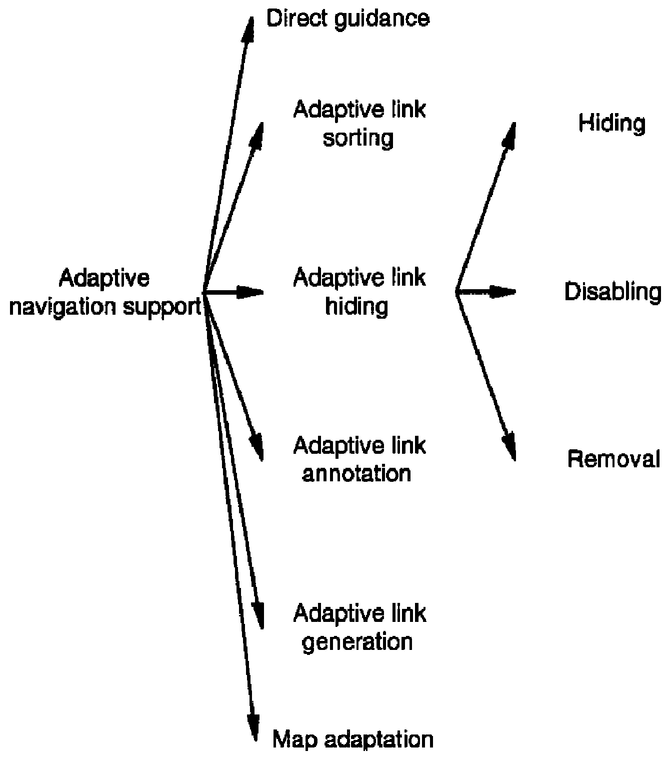


Figure 2. Adaptive navigation support techniques.

Most of the research on *adaptive presentation* deals with adaptive text presentation, and even then mostly with canned text presentation (and not natural language generation). In multimedia the selection of a presentation mode, or the presentation medium (text, image, video audio) is most feasible. Automatic adaptation of multimedia content, like in automatic summarization of video or audio, is still very much future work. In this tutorial we will look mainly into *canned text adaptation*.

Among the many ways to perform adaptation to text the technique of *inserting or removing fragments* is the most popular. This is probably due to the fact that this technique is easy to implement. With a fragment a condition can be associated, a Boolean expression on information from the user model, and this condition determines whether a fragment will be shown or not. We distinguish three areas in which this technique is often used:

- In *prerequisite explanations* an extra explanation is added for users who need it. A page that uses a technical term or a name the user has not yet seen may conditionally include a short introduction or explanation for that term or name. In our hypermedia course (2L690) for instance we sometimes refer to the system "Xanadu". For students who have not yet read the page describing Xanadu a one-line explanation of Xanadu is added to pages referring to it.

  > In Xanadu (a fully distributed hypertext system, developed by Ted Nelson at Brown University,
  > from 1965 on) there was only one protocol, so that part could be missing.

  That one-liner disappears after reading the full page about Xanadu:

  > In Xanadu there was only one protocol, so that part could be missing.

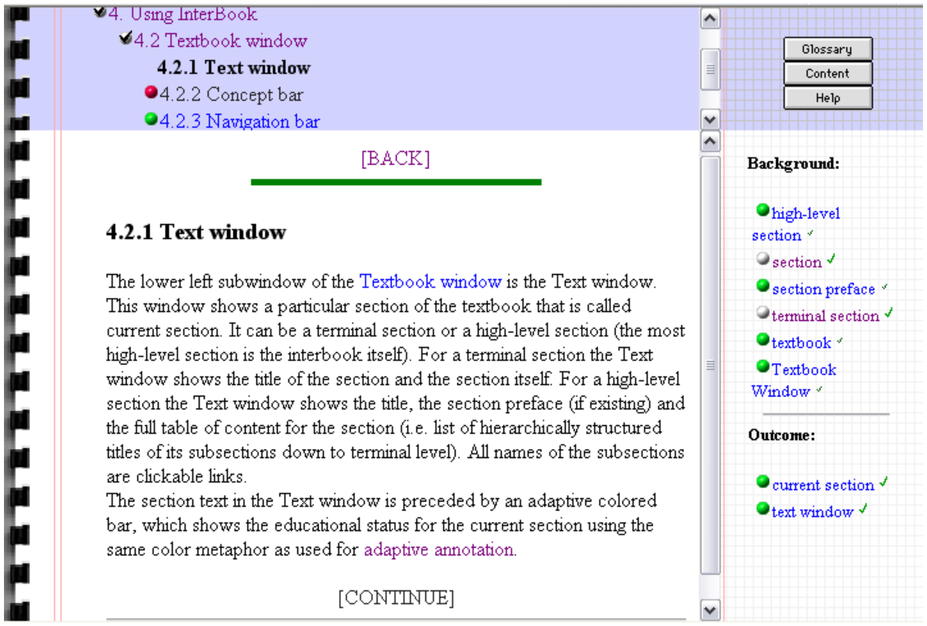
- *Additional explanations* can be given to users who are ready for them. Whereas prerequisite explanations try to compensate for missing foreknowledge additional explanations take advantage of users' knowledge to offer more in-depth information to users who can understand it.
- A special kind of additional explanations are the *comparative explanations*. This technique refers to a comparison between topics described on different pages. The comparison can only be understood by users who have read both pages. So when visiting one of these pages first, the comparison will not be made, but when visiting the other page the comparison appears.

We will not discuss most other adaptive presentation techniques (altering fragments, stretchtext, sorting) but only mention *dimming fragments*. There are many ways in which some information can be *emphasized* or *deem-*

phasized. Less important or urgent information can be presented using a smaller font, in a sidebar, as a footnote, as a pop-up activated when you move the mouse over a tooltip icon, etc.
Regarding *adaptive navigation support* figure 2 shows the following techniques:

- *Direct guidance* is a technique to offer users a possibility to be guided as in a guided tour. Typically a "next" button invites you to go to the "next" page. But unlike in a static guided tour the adaptive system determines the destination of that "next" button, so different users may go to a different page when clicking on the "next" button on the same page and when you revisit a page the "next" button on that page may take you to a different page than the previous time (but that can be confusing). Of course direct guidance can also be more subtle. Apart from buttons that clearly lead to a tour other links on a page may also have adaptively determined link destinations. You may have the impression that there is a lot more navigational freedom than is actually the case, because links may not lead to where you think they do.
- *Adaptive link generation* goes one step further and not only generates link destinations but the link anchors as well. There are many ways in which the system can decide to create new links. In *open hypermedia* all links are always generated. This is done by matching text on a page with a database of links. Adaptive link generation can also be based on the discovery of similarities between (the topics of) pages. This is certainly adaptive if it is done in pages from an open corpus of documents. The list of links that result from a search request in information retrieval or filtering systems is also adaptively generated.
- *Adaptive link annotation* is the most popular link adaptation technique. It is the least restrictive technique: all the links are accessible. Annotations are used to indicate how interesting the link is for you, at the time of reading the page containing the link. Many systems use some kind of icon in front of or behind the link anchor to indicate the relevance of the link. Since the Web has been extended with style sheets it has also become possible to use the color of the link anchor itself as an annotation. This is not without drawbacks: some users are so used to links on the Web being *blue* or *purple* that they do not recognize words in other colors as being link anchors.
- *Adaptive link hiding* means that links that are not considered relevant for you (at this time) are hidden, disabled or removed in some way. Link hiding means that the link anchor cannot be seen as being a link anchor. When the text on a page is black, a black link anchor, not underlined, looks just like plain text. If the link is still there many browsers will show a special cursor when the mouse pointer is moved over the anchor. The link can also be *disabled*, meaning that the anchor text is no longer a link anchor. On the Web this is easy to realize by removing the anchor tag. However, that performs hiding as well as disabling. It is possible to use font color and optionally underlining to make the anchor still look like a link anchor, but this is seldom done because it is frustrating for users to see link anchors that do not work as links.
- *Adaptive link removal* means that the anchor text (for undesired links) is removed, thereby automatically disabling the link as well. Link removal can easily be done in a list of links, but not in running text because removing words from the text may seriously alter its meaning and also disrupt the reading process (especially if sentences with words removed are no longer valid sentences). When asked in an informal setting a large majority of users has indicated that they preferred links in a list to be annotated or "hidden", but not removed.
- The final form of adaptive navigation support is *map adaptation*. In order to give you an idea of the whole hyperspace, and some orientation support regarding where you are in this space, many applications offer some kind of map. Websites often offer a textual sitemap, mostly because this is easy to generate. A graphical map, preferably based on conceptual relationships rather than link relationships, is a better tool for giving insight into the application's structure. However, maps are often too large to be insightful. A map can adaptively be reduced so that you can still grasp the overall picture. Nodes on the map can also be annotated to indicate relevance, to indicate where you have gone before, and perhaps even to indicate where other users have gone.

Figure 3 shows a small example of an adaptive course, using several of the techniques described above. The page is taken from the InterBook [3] manual, but presented through the AHA! system [6], after an automatic translation. The figure shows a webpage divided into 5 frames. In the *text window* (as the figure explains) you see a section from the course. It has links that are annotated with color, *blue* meaning recommended and not previously visited, and *purple* meaning recommended but already visited. This is the standard color scheme used on the Web, but that coloring of links is in fact a form of *link annotation*. The "back" and "continue" buttons are also annotated, and the "continue" button has the color *black* which means the link is not recommended. In the top frame we see a partial table of contents. The links are annotated using the same color scheme (so we indeed see that section 4.2.2, which is the next section, is not recommended. In the table of contents several link adaptation techniques are used. The link to the current section (4.2.1) is *disabled*. The links to subsections of other chapters (other than 4) are *removed*. (You cannot see that because they would require scrolling to see them.) The links are annotated also with an icon in front of the link anchor. The *green ball* indicates that the link is recommended. The *red ball* means the link is not recommended. The *white ball* indicates that the link does not lead to a page that will increase your knowledge level. Checkmarks inside the balls indicate your knowledge of the concept explained in the page (that is the link destination). There are three sizes of checkmarks. On the right we see a frame that shows the “background” concepts. In InterBook this is the term used for what most people name “prerequisite” concepts. The list of concepts actually consists of links to pages explaining these concepts. Again the blue/purple color scheme, the colored balls and checkmarks are used to indicate the status of these links and of your knowledge of these concepts. The frame on the right at the bottom shows the “outcome” concepts, the concepts of which the current page provides knowledge. This list again consists of links annotated with colors, balls and checkmarks.



4.2.1 Text window

The lower left subwindow of the Textbook window is the Text window. This window shows a particular section of the textbook that is called current section. It can be a terminal section or a high-level section (the most high-level section is the interbook itself). For a terminal section the Text window shows the title of the section and the section itself. For a high-level section the Text window shows the title, the section preface (if existing) and the full table of content for the section (i.e. list of hierarchically structured titles of its subsections down to terminal level). All names of the subsections are clickable links.
The section text in the Text window is preceded by an adaptive colored bar, which shows the educational status for the current section using the same color metaphor as used for adaptive annotation.

Figure 3. Example course page where several link adaptation techniques are used.

There is also a form of adaptive presentation in Figure 3: The text window contains a green horizontal bar, which serves as a reminder that you are reading a recommended page (with new knowledge). (When reading a non-recommended page the bar would be red.)

2.3 What can we adapt to?

Most adaptive hypermedia systems are educational systems. It is therefore not surprising that adaptation to the user’s *knowledge* is popular. A simple approach to adapting to knowledge is to distinguish a few global knowledge levels for the whole application, like “beginner”, “intermediate” and “expert”. While this may be sufficient when you first start using the application, it fails to take into account that you are becoming more knowledgeable about the part you studied than about the rest.

To solve this most applications use an *overlay model*. The whole application is described using a set or hierarchy of concepts. The system keeps track of your increasing knowledge about these concepts. Adaptation, like link annotation or the inclusion of prerequisite explanations, can be based on your knowledge level about a specific topic. Sometimes the system may wish to have a very fine grained representation of the concept structure, perhaps down to the page level or even individual fragments or paragraphs. This is for instance needed to perform adaptation that depends on having read a certain definition, or having seen a certain picture. But in many cases a more coarse grained representation is sufficient, like the knowledge of a high-level concept or a chapter of a course.

The biggest challenge in performing adaptation to your knowledge is how to determine what that knowledge about a concept actually is. The system can register which pages you visit and use an association between pages and concepts to deduce knowledge about these concepts. But of course the system does not know how much you actually understand of a visited page. You may spend little time reading it (but instead go get coffee) or may read the page without understanding it. Your performance on tests would be a much more reliable way to determine your knowledge, and therefore test modules, like multiple-choice tests, are popular adaptive educational hypermedia. However, since the purpose of the application is to present information for you to study, spending a lot of time on tests may not be very helpful to you, and as a result you may not like an on-line course with many tests.

For *adaptation to goals or tasks* it is helpful to have a representation of the hyperspace using concepts (like with educational applications), so that the goals can be described using an overlay model, just like for knowledge. Unlike with knowledge however it is much harder for the system to determine your goal by observing pages you visit. After all, you may visit pages that do not correspond to your goal, but you only discover that after accessing such pages, and you cannot tell the system "oh no, this isn't it". And unlike with knowledge your goal may change almost instantly, perhaps triggered by something you read. Adaptation to goals or tasks works best if these goals or tasks are entered explicitly into the system. This is for instance common practice in workflow systems and that makes these systems ideal for the use of adaptation to goals and tasks.

Other aspects an adaptive hypermedia system can adapt to are not related to the *information* the system has to offer. The system can adapt to your *background* for instance, which represents aspects like your education, profession, experience in using hypermedia applications, and possibly also your experience with the particular adaptive system, with the hyperspace and how to navigate through it.

Preferences are also global aspects of the user, like whether you prefer a video presentation over images or text, and it also covers *cognitive style* aspects that determine whether you need more or less guidance, whether you prefer to see examples before or after definitions, etc. There are ways to *deduce* such preferences from the browsing behavior, but these do not always work reliably and they require you to first work for a while with the system not yet adapted to your preferences. Therefore it is easier and better to initialize the preference settings through a form or questionnaire.

The *context* or *work environment* can also be adapted to, again using mostly things like media choice and media quality settings. When you are using a pda with a low bandwidth connection high-definition video will have to be replaced by low resolution images, and text pages must be reasonably small to avoid excessive scrolling. Unlike preferences these contextual aspects are relatively easy to determine automatically. They are also less stable, so they need to be verified at least at the beginning of every session, but perhaps also intermittently because properties like network bandwidth may vary over time.

3 A Reference Architecture for Adaptive Web-Based Hypermedia Systems

3.1 How Web-Based Adaptive Systems Work

The first generation adaptive hypermedia systems were not Web-based because they predated the Web. In some of these applications a very close interaction between the user interface and the back-end adaptive engine was achieved. Web-based adaptive systems are very different. The protocol linking the user interface, a Web browser, and the server is quite primitive. The HTTP or HyperText Transfer Protocol is a pure *request-response* protocol. The browser sends only one type of signal to the server: the request for a URL (or Uniform Resource Locator). The server responds with a header, possibly followed by a block of data, which is usually an HTML page but sometimes something different, like an image, or code for a Java applet.

A URL need not be the address of a webpage. It is merely an abstract "address" that can be interpreted by the server in any way that server is programmed to do. In an adaptive website the URL will actually be a reference to a server-side program, combined with some parameters for that program. Early websites used the *Common Gateway Interface* or CGI to invoke programs. Newer websites often use *Java Servlets*, a much more efficient solution as modern servers are tightly integrated with servlets, whereas CGI-scripts are completely separate programs. In a Web-based AHS the server-side program uses part of the URL to determine the page or concept the user wishes to access. The request is combined with information from the user model to "filter" the information, thereby using *adaptive presentation* and *adaptive navigation support* techniques. We will see later that the process of combining the request with the user model can be described using *adaptation rules*.

When the requested and adapted information is sent to your browser the server-side program updates your user model. The process of performing *user model updates* and *adaptation based on the user model* requires the system (designer) to make a choice which of these two "steps" to perform first.

- In an educational application that keeps track of the evolution of your knowledge "logic" dictates that the requested page needs to be adapted to the user model as it exists *before* your request. When you read the page your knowledge is going to change. Ideally the page should change according to your knowledge change while you are reading (but of course you do not really want such disturbing dynamic behavior). Suppose that you need to study page A before page B (i.e. A is a prerequisite for B) and suppose that page A contains a link to page B. The system should indicate that you are not ready to following the link to B when you just start reading page A, but when you are done the link to B should be recommended. Unfortunately, both requirements contradict each other. The page will be sent to your browser only once, with the link to page B either recommended or not recommended. The link annotation cannot change while you are reading the page. As a compromise adaptive educational systems always present links using their status *after the user model update*, meaning the user model is updated *before* the page adaptation is done.

- Also in an educational application, *prerequisite explanations* are sometimes inserted in a page. The decision whether to include these explanations should again be based on the user model as it exists *before* your request. If the user model update is performed first, as suggested by the previous bullet, reading a page would make an included prerequisite explanation on that page superfluous, and hence the explanation would never appear. However, it isn't hard to design an application so that only prerequisite explanations are included that depend on *other knowledge* than the one offered by the page including the explanation. When this measure is taken it is ok to have the user model update precede the page adaptation.
- In a recommender system, or any other application that performs adaptation on the user's *interest* or *goals*, then accessing a page provides the system with information about the user's interest or goal, so the access to the page should be taken into account in the adaptation. This again suggests that the user model update must precede the adaptation.

Because of the above reasons the *standard* way in which AHS work is to update the user model first, and then perform the adaptation to the updated model. In the reference model described in this section the choice when to perform the user model update versus the adaptation is left open, but in practice any updates that influence the adaptation must be performed before that adaptation.

3.2 The AHAM reference model

There exist numerous adaptive hypermedia systems. These systems are wildly different. Still, if we look closely at them we see that the basic structures and functionality of different adaptive hypermedia systems show strong similarities. When we try to capture these similarities into a model that model then needs to have "hooks" to include very application-specific functionality. We call our model a reference architecture because it provides a reference point for comparing different adaptive hypermedia systems. The purpose of a reference architecture is also to provide a means of formally describing different adaptive hypermedia systems using one single framework. The descriptions allow us to compare systems, and perhaps even to develop a translation from one system to another.

We describe adaptive hypermedia systems, or applications, using a modular approach so that we can describe functionally different parts separately. We base our model on a widely accepted reference model for non-adaptive hypermedia systems: the *Dexter* model [8]. The Dexter model concentrates on an abstract hypermedia layer that describes the nodes and links structure with all its properties. Dexter does not describe the details of user-interfaces or of the server-side data storage and retrieval system. It provides interface layers between the abstract hypermedia core and the user-interface on the one hand, and between the hypermedia core and the data storage on the other hand. These interface layers remain essentially unchanged in our reference architecture, because we will concentrate on the adaptive functionality in the core hypermedia layer, called the storage layer in the Dexter model.

Figure 4 shows the overall AHAM structure [7].

- At the top there is the *run-time layer* which represents the user-interface. We do not describe what the user-interface should do exactly, but we provide abstractions of what it should do by means of *presentation specifications*. Roughly speaking presentation specifications can indicate that content should be emphasized or that a link should be annotated in some way. It is up to the run-time layer to decide, possibly guided through style sheets, how to make these presentation aspects visible to the end-user.
- At the bottom the *within-component layer* describes the internal, implementation-specific data objects. The (adaptive) hypermedia systems can access these objects by means of *anchoring*.
- The core of the AHAM model is the *storage layer*, just like in the Dexter model. In AHAM this layer consists of three functionally different parts. The *domain model* contains a conceptual representation of the application domain. In the Dexter model the storage layer only contained what we call the domain model. The *user model* contains a conceptual representation of all the aspects of the user that are relevant for the adaptive hypermedia application. This includes an *overlay model* of the domain, but also the user's *background*, *experience*, *preferences* or anything else that contributes towards the adaptation. The *adaptation model* describes how an event, such as the user following a link, results in a presentation, by combining elements from the domain model and the user model.

We will now describe the details of the three models in the storage layer.

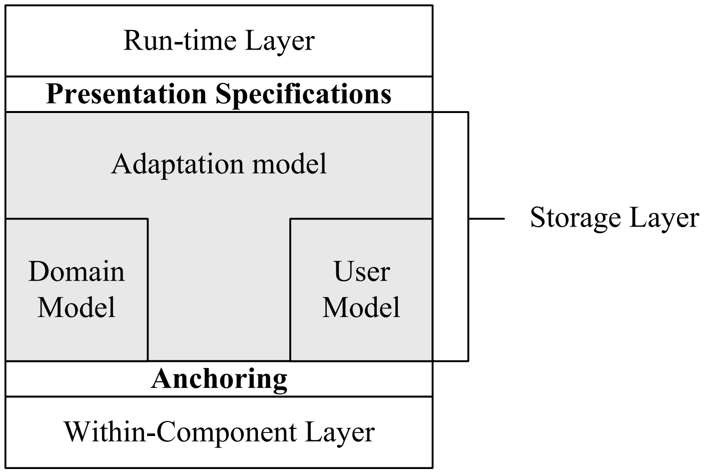


Figure 4. The AHAM reference model.

3.2.1 The Domain Model

The *domain model* is structured according to the *storage layer* of the Dexter Model [8]. In the Dexter Model it consists of "components", which are abstract entitities, and "relationships". In AHAM we have chosen to use the term "concepts" and "concept relationships" because that expresses more closely what the components and relationships typically represent. The structure of a *concept* is as follows:

- A concept has a *unique identity* (uid). You can think of this as a unique name, although names are often not as unique as one would like. The identity must be unique within the entire system, even when a system is running multiple (adaptive) applications simultaneously. You may at some point want to carry over knowledge about a concept in one course to another course, and need a unique way to refer to the concept.
- A concept has what we call "*concept information*" or c-info. The c-info has a fixed part, namely a *sequence of anchors* and a *presentation specification*, and variable part represented as a set of *attribute-value pairs*.

We have to keep in mind that a concept is an abstract entity. If a concept for instance corresponds to a fragment of text, the fragment of text will be part of the within-component layer and can be "linked" to the concept through an attribute-value pair, but the text is not something that can be manipulated within the domain model. It is possible to implement this "link" between a fragment concept and the actual contents through a URL that refers to an HTML file, but the reference model does not require any *specific* way to implement the link between concepts and content.

The *sequence of anchors* identifies substructures within a concept. These can be used as anchor point for links. Anchors can be used as the source or destination of a link.

The *presentation specification* is used to tell the run-time layer how to display the concept. In AHAM it is only a default presentation specification. Adaptation can imply that a different presentation specification will be passed on to the run-time layer.

The set of attribute-value pairs is in principle completely arbitrary. Typical examples include a set of keywords or a concept type. The Dexter model says that the attribute-value pairs can represent anything except the actual content of a component. However, we will consider the content of a fragment concept as an attribute, and we will also represent composite structures by using a *children* attribute. Fragments are *atomic concepts*. We consider two types of *composite concepts*: pages and abstract concepts. Pages are composites of which all the children are fragments, or atomic concepts. Abstract concepts are composites of which all the children are composite concepts (either abstract or pages). The structure of composition of concepts must be a hierarchy, or a directed acyclic graph. This means that no concept can thus be a descendent of itself either directly or indirectly. The *concept hierarchy* is typically used to "propagate" a knowledge increase from pages to sections and from sections to chapters.

Concept relationships are abstract relationships between concepts. Roughly speaking they are like hyperlinks, but most concept relationships cannot be used for navigation whereas all hyperlinks can. A relationship has a *sequence of specifiers*, or *endpoints*. A relationship basically "links" a number of concepts together. There are *directed* and *undirected* relationships.

- The endpoints are identified using the uid of the concept and an anchor id within that concept. Just like links in HTML pages both the source and the destination of a link can be a specific "part" or "place", not just a whole page.

- Relationships also have a direction. For most concept relationships the direction will be either FROM or TO. A typical kind of relationship is the prerequisite. Concept A is a prerequisite for concept B when you need to have knowledge of A before being recommended to visit B. The specifier for A will then be FROM and the specifier for B will be TO. Nevertheless, this relationship does not imply that there must be a hypertext link from A to B. Relationships do not have to be directed and binary like this example. The relationship can be symmetric, be it *bidirectional* or *undirected*. And a relationship may also group any number of concepts together to express some meaning. So whereas directed binary relationships are most common the reference architecture does not exclude the existence of other relationship types.
- A concept relationship has a presentation specification to help the run-time layer in deciding how to present the relationship (or its anchors).
- As concept relationships are "components" (in Dexter terminology) they have "component information" with attribute-value pairs. In AHAM we require concept relationships to have a *type* attribute. The hypertext link is good example of a relationship type. In the Dexter model this is the only relationship type that exists. In adaptive hypermedia the *prerequisite relationship* is an example of a relationship type that is not a link. When A is a prerequisite for B it is not necessary that there exists a link from a page corresponding to concept A to a page corresponding to concept B. Note also that unlike on the Web, hyperlinks are not necessarily links from one page to one other page.

3.2.2 Anchoring

As we have seen anchors form the basis for attaching links or relationships to pages or concepts. Each anchor has a unique identity, and a value that is used to locate the anchor within the concept. In the case of a text fragment the anchor value can for instance be a character index of the start of the anchor, together with the length of the anchor. The *anchor value* belongs to the within-conponent layer because it is implementation dependent. When the fragment is edited for instance, the anchor may move. Its identity however remains the same.

For composite concepts we can use anchors to identify not a location within the concept but within a sub-concept. An anchor for a page concept for instance identifies the identity of a fragment and an anchor id within that fragment. For an abstract concept the anchor id within that fragment again identifies an anchor whose value must be interpreted to travel down the concept hierarchy to end up with a "real" anchor within a fragment.

An anchor within a composite can also just be the uid of a sub-component, in case no anchors within that sub-component need to be referred to.

3.2.3 Access to Concepts and Pages

Because in the Dexter and AHAM models we have composite components we need to pay special attention to the problem of accessing and presenting pages. Concept relationships, even the ones of type link, do not only exist between pages, but between abstract concepts as well. The Dexter model defines two functions that work together to find and access pages: the resolver and accessor.

- The *resolver* function is used to find concepts based on a description. This function can be as complex as a search engine or as simple as a function that converts a filename to a complete URL. Apart from the use in search engines the resolver function is most interesting for implementing the access to abstract concepts. When an abstract concept is accessed the system needs to go down the concept hierarchy to find the uid of a page to display. When the hypermedia systems can only display one page at a time a choice has to be made which page to display, among the potentially many pages that correspond to the abstract concept. We therefore sometimes call the resolver function the *page selector*.
- The *accessor* function is used to retrieve content, given the uid of a concept. If the uid refers to an abstract concept (meaning that the resolver did not translate down to the page level) a page has to be constructed, probably with links to let the end-user decide how to further resolve the concept to a page. (The partial table of contents, list of background and outcome concepts in Figure 3 are such automatically generated pages.) If the uid already refers to a page the accessor has to retrieve the fragments and construct a page out of these fragments. We call the accessor a *page constructor* because in either case the system has to construct a page to show to the user.

3.2.4 The User Model

The first major extension of AHAM versus the Dexter model is the introduction of the user model, a must-have for all adaptive hypermedia systems.

In order to be able to model all possible adaptive hypermedia systems we construct the user model out of concepts with arbitrary sets of attribute-value pairs. When each concept of the domain model is also present in the user model we have what is called an *overlay model*. Typically the user model will contain other concepts as well, in order to represent aspects of the user that are independent of the application domain, and aspects of the context or environment as well.

Every concept may have different attributes. For page concepts for instance it makes sense to have a "visited" attribute, whereas for abstract concepts this may not make sense. The values can be of any type. When more complex structures are needed than a single layer of attribute-value pairs the value of an attribute can be a concept id so that the value can be associated with another structure of arbitrarily many concept-value pairs.

You might have the impression that in AHAM we can only represent fairly simple user models. However, looking closely we see that we in fact have objects with attributes of arbitrary types. It is possible to have "classes" of objects, all having the same attributes. We can have subclasses which inherit attributes from a superclass and have some attributes of their own. Also, the data type of an attribute can be anything, including identities of concepts. The objects can thus have attributes that refer to other attributes, which means that we can have object containment. As a consequence, the user model in AHAM can have an arbitrarily complex object-oriented structure.

3.2.5 The Adaptation Model

The *adaptation model* describes how the domain model and user model are combined to generate adaptation. Virtually all AHS have a user model that is stored permanently. Good adaptation requires that information about the user is gathered over a long period of time. Although adaptation is possible during a single session, based only on information gathered during that session, and stored in *cookies* on the client, we will consider a more permanent user model, stored and maintained on a server.

The adaptation model consists of a set of *adaptation rules*. Each rule will either define a user model update or define a desired adaptation by setting a presentation specification. For simplicity we assume that we have an overlay model and that corresponding concepts in the domain model and in the user model have the same identity. We also assume that no attributes from the domain model have names that occur in the user model, so that an expression like `concept.attribute` unambiguously refers to a domain model attribute or a user model attribute. Adaptation rules are *event-condition-action rules* because they consist of the following elements:

- Rules are triggered by an *event*. A typical event is the *access event* that represents that the user accesses a certain concept by following a link. But changes to the user model, either as a result of another rule or of another system module like a multiple-choice test evaluator, can also be used as events that trigger the execution of rules.
- When a rule is triggered the *condition* is evaluated. The condition is a Boolean expression that uses attribute values of concepts from the domain or user model, and possibly also some constant values. Conditions are used for instance to check whether a *prerequisite relationship* is satisfied.
- When the condition is true the *action* is executed. The action performs an update to an attribute value. For the sake of simplicity we do not distinguish between user model updates and the setting of presentation specifications. Any rule can perform either kind of update. We can allow more than one update in an action or not. This does not change the expressive power of the model because we can always have several rules with an identical event and condition and then with just one update in the action.
- The update of a user model attribute may be considered as an event that triggers adaptation rules. Through a *propagation field* we control whether the updates that result from the action of the rule are considered rule triggering events. The use of this field makes it possible to guarantee that the rule triggering stops without having to analyze all possible user model instances and the resulting truth-value of the conditions of all triggered rules.
- In order to control the execution of the rules more easily the rules are grouped into *execution phases*. When rules trigger each other the triggered rules may belong to different phases, which means that their execution will be postponed until that phase is considered. It would be much more difficult to enforce a similar sequencing of rules through additional attributes and conditions, but it could be done.

The structure of rules described above results in a rule executions model that resembles that used in the field of *active databases*. We do not require actual adaptive hypermedia systems to implement a system of such rules. We only use this rule system as a way to describe how user model updating and adaptation *can* work, not how it *must* work.

Using *event-condition-action rules* for user model updates may seem like a complicated way to define the behavior of an adaptive application. However, we can define *generic rules* that are applied to concepts that appear in concept relationships of a certain type. That way we only have to define the rules for relationship types and then the author of an adaptive application only has to define concept relationships, not adaptation rules.

Below we show a few examples of adaptation rules. For simplicity we only show the event, condition and action.

```
Event: access(C)
Condition: true
Action: C.visited := true
```

This is an example of a *generic* rule that says that when a concept C is accessed the “visited” attribute is set to true. In this case there is no condition (expressed by having a condition that is the constant “true”).

```
Event: visited(C)
Condition: C.recommended = true
Action: C.knowledge := “well learned”
```

This rule expresses that when concept C is visited (the visited attribute is set to true) and the concept was recommended, then the knowledge of C is set to “well learned”. This again is a generic rule.

The actual execution semantics of the *adaptation engine* is beyond the scope of this tutorial. The adaptation rules only describe *possible* behavior of the system. The *actual* behavior depends on how the adaptation engine schedules rules for execution. When several rules are associated with an event the order in which the rules are executed may determine the outcome. And since each rule may perform actions that trigger other rules, there are many ways in which these rules can be scheduled for execution. Also, the system can evaluate the conditions when rules are triggered, or when rules are actually executed (and the condition may no longer be true at the time of execution).

In the next section we will study the AHA! system, a general-purpose adaptive hypermedia system that is a close match to the AHAM reference architecture.

4 AHA! The Adaptive Hypermedia Architecture

In this tutorial we are going to study how to create adaptive applications with the AHA! system.

In order to work with AHA! (as authors of applications) we need to understand how the AHA! system works in general. We will use AHAM terminology, so that the correspondence between the system and the general reference model becomes clear. 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 Section 4.3. Figure 5 shows an example of a domain model (taken from the older on-line AHA! 2.0 tutorial) as it would appear in the Graph Author.

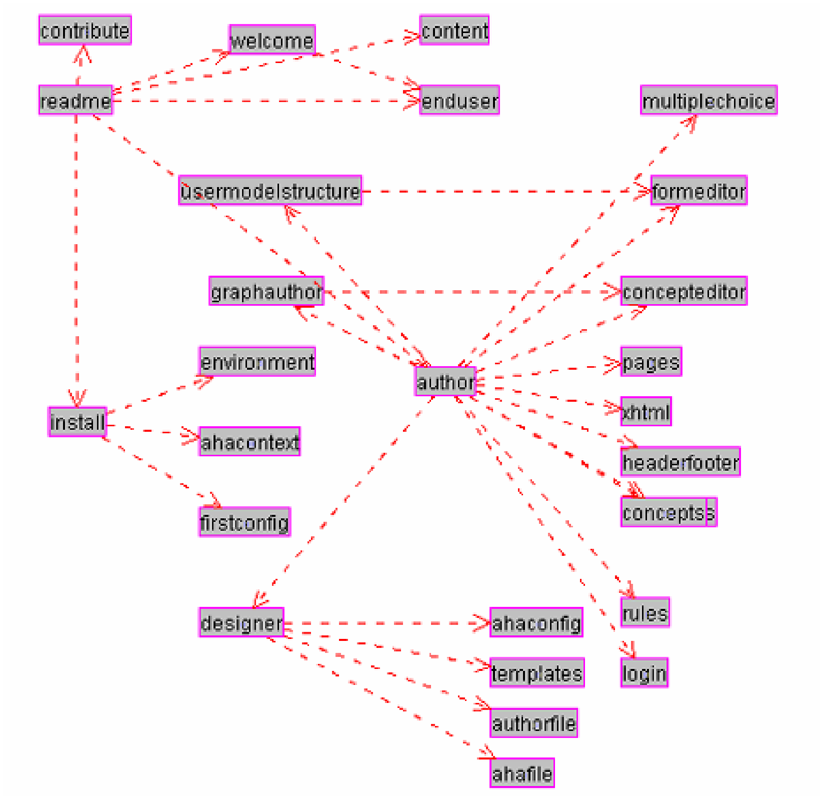


Figure 5. Example domain model with concepts and prerequisite relationships.

- 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 Section 4.3.2), 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 Section 4.4 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 Section 4.9.1).
4. When the rules have been executed AHA! determines which page was requested. There may be several pages associated with a concept. In Section 4.3.2 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 Section 4.3.2 and Section 4.8 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.

4.1 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).

4.1.1 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:

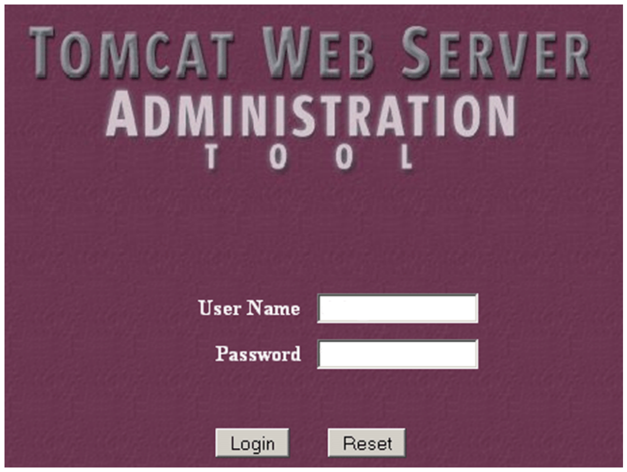


Figure 6. Tomcat login screen.

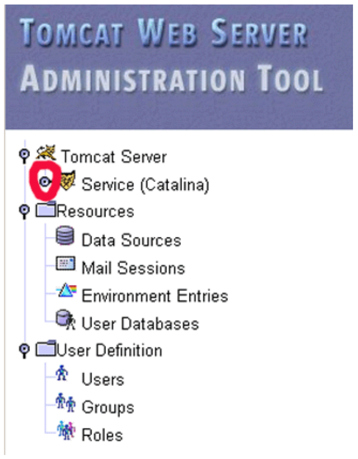
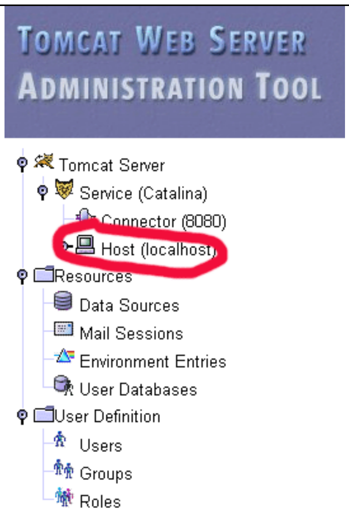
4.1.2 Creating the AHA! context

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.

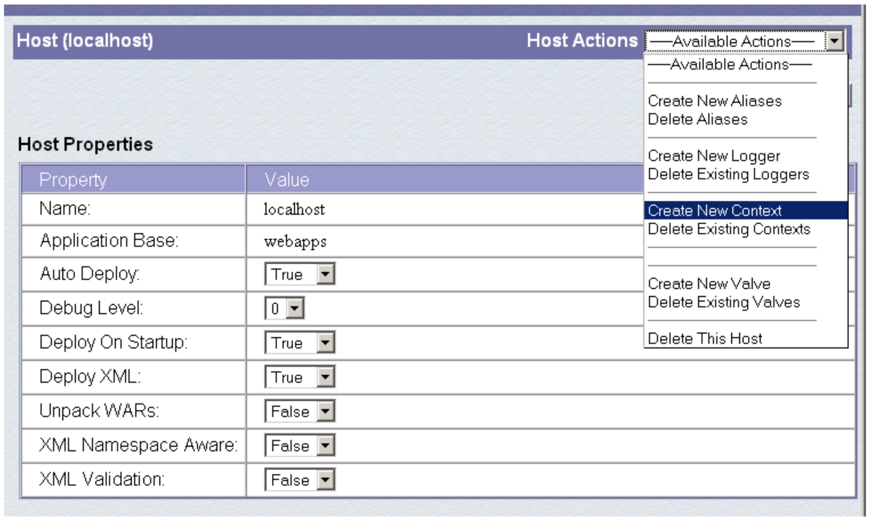
AHA! is distributed as a zip archive that you can extract anywhere on your system. The archive is the same whether you are using Windows or Linux. In the installation description and screendumps we will use the directory "d:\aha3" for this purpose but this name is completely arbitrary.

After logging on you should open up the "Service" item by clicking on the icon circled in red in Figure 7.

After this you should click on "localhost" (not the small circle to the left of it) in the menu shown in Figure 8.

 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
Figure 7. Initial Tomcat administration menu.	**Figure 8.** Tomcat administration tool with service submenu opened up.

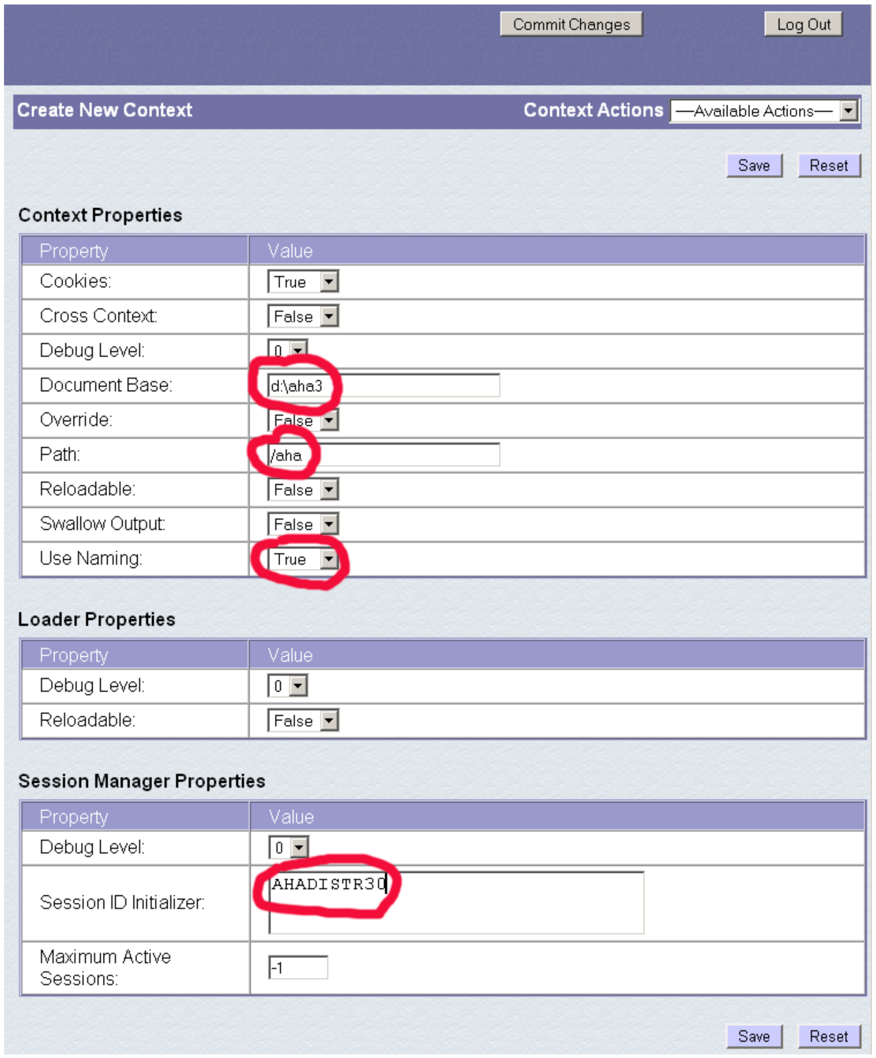
This brings up a menu of available actions of which you should select to create a new context, as shown in Figure 9.



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

Figure 9. Menu with selection to create a new context.

Figure 10 shows the form for creating the AHA! context. The items circled in red are the most important:



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

Figure 10. Form for configuring the AHA! context.

- 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 Section 4.1.3) 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.

4.1.3 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 Section 4.1.4) will not work unless you restart the server.

4.1.4 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 Figure 11.

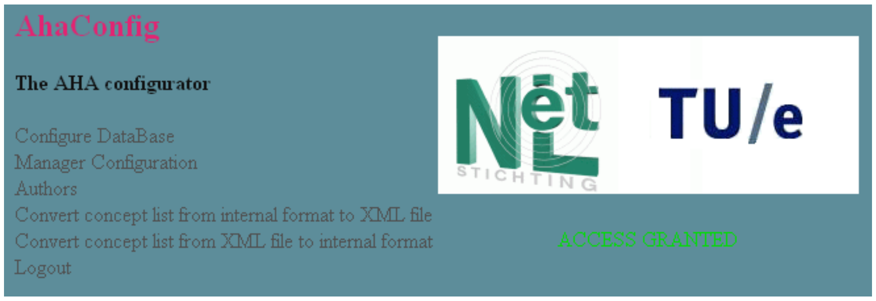
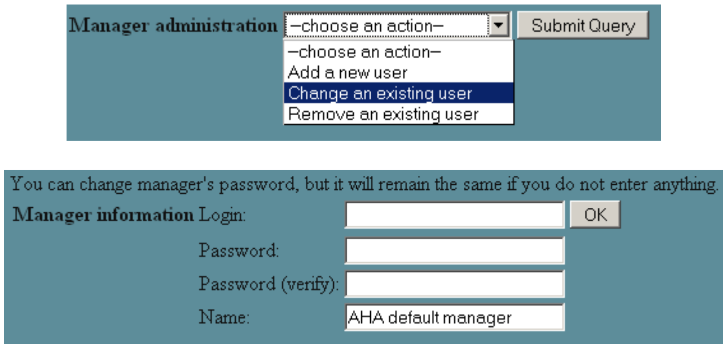


Figure 11. The AHA! configurator

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 Figure 12. 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.



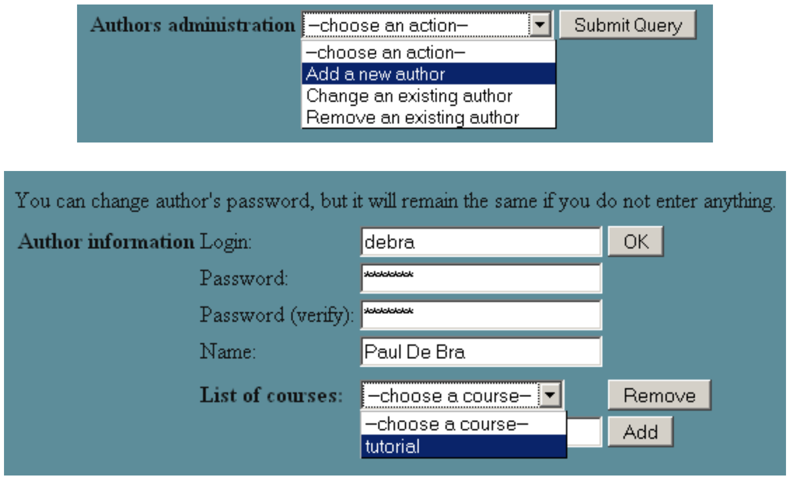
You can change manager's password, but it will remain the same if you do not enter anything.

Manager information	Login:	
	Password:	
	Password (verify):	
	Name:	AHA default manager

Figure 12. 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. Figure 13. 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.


Authors administration –choose an action– Submit Query

–choose an action–
Add a new author
Change an existing author
Remove an existing author

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

Author information	Login:	debra	OK
	Password:	*******	
	Password (verify):	*******	
	Name:	Paul De Bra	
	List of courses:	–choose a course–	Remove
		–choose a course– / tutorial	Add

Figure 13. 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 Section 4.4). 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 Section 4.4) 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.

4.2 AHA! for the end-user

This tutorial is really for authors of adaptive applications, 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.

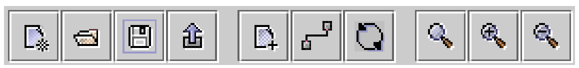
When the user navigates through the browser's history (using the back and forward buttons) the browser may not request the pages from the AHA! server but may use cached versions. This results in pages being shown exactly as the user saw them before, and not in the "new" form that would correspond to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

4.3 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 Section 4.4. 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. Figure 14 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 Figure 14 (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):



[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.

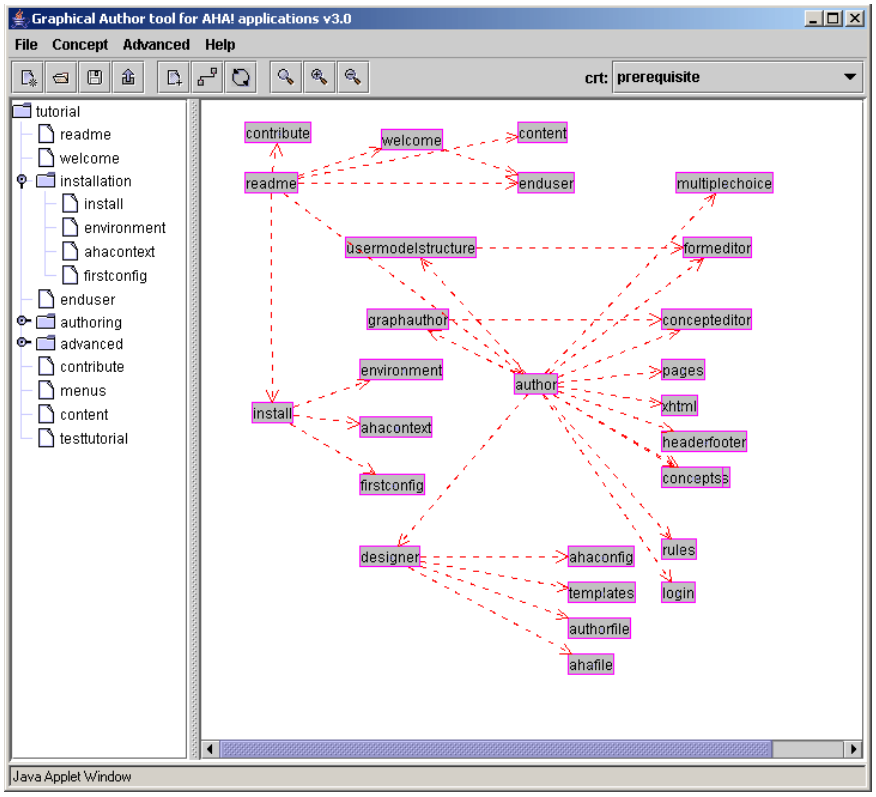


Figure 14. Screen shot of the Graph Author.

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 Figure 14 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. Figure 14 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).

4.3.1 Concept relationships

The *concept relationship graph* is created by dragging concepts from the hierarchy shown on the left (see Figure 14) 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 Figure 14) 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.)

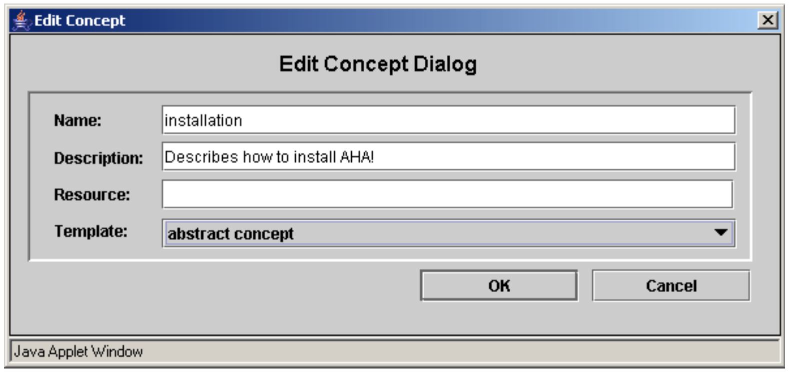
Since prerequisite relationships are most used we will explain how to use them and also how they work. All relationship types in AHA! are translated into *adaptation rules*. The details of these rules are explained in Section 4.9.1. For now all we need to know is that a rule assigns a value to an attribute 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 Figure 5 would have looked a lot more complicated.

4.3.2 Concept templates

Figure 15 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

Figure 15. 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:

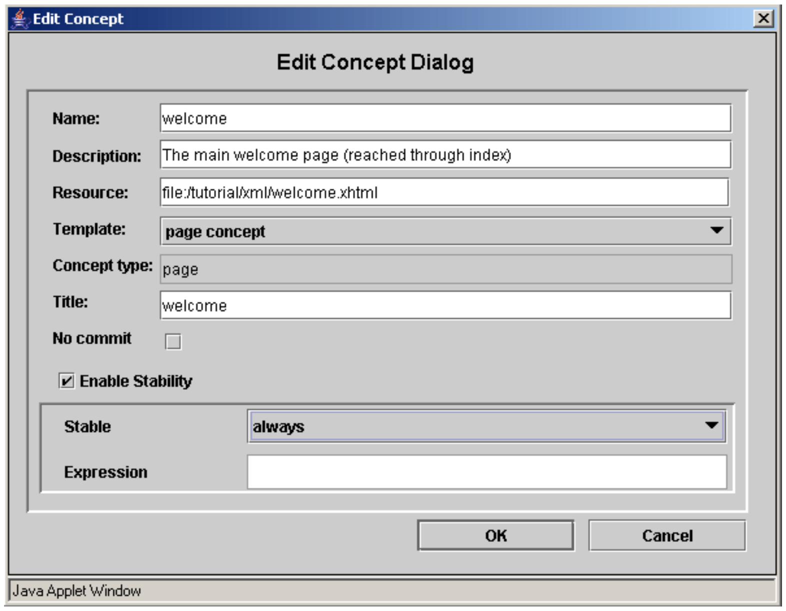
- **access**: When the page is accessed the rules associated with this attribute are executed (first). The attribute only serves the purpose of starting the adaptation rule engine. `<isSystem>true</isSystem>` indicates that this attribute has a special meaning to the AHA! engine. `<isPersistent>false</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 `<is-System>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 Section 4.5), because of `<isChange-able>true</isChangeable>`.
- **visited**: This attribute is used by the system (`<isSystem>true</isSystem>`) to remember (`<isPersis-tent>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. Figure 16 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. Figure 17 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 Figure 17). 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 Figure 17. 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

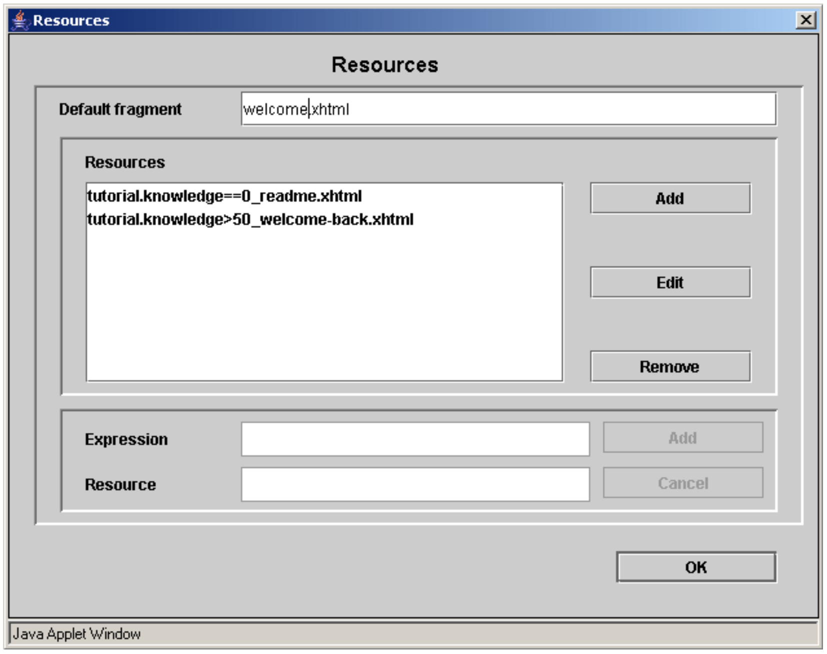
Edit Concept Dialog

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	

Figure 16. The Edit Concept Dialog in advanced mode.



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

Figure 17. Dialog box for performing resource selection.

4.3.3 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 Figure 5), and that cycles are not allowed (`<properties acyclic="true" />`).

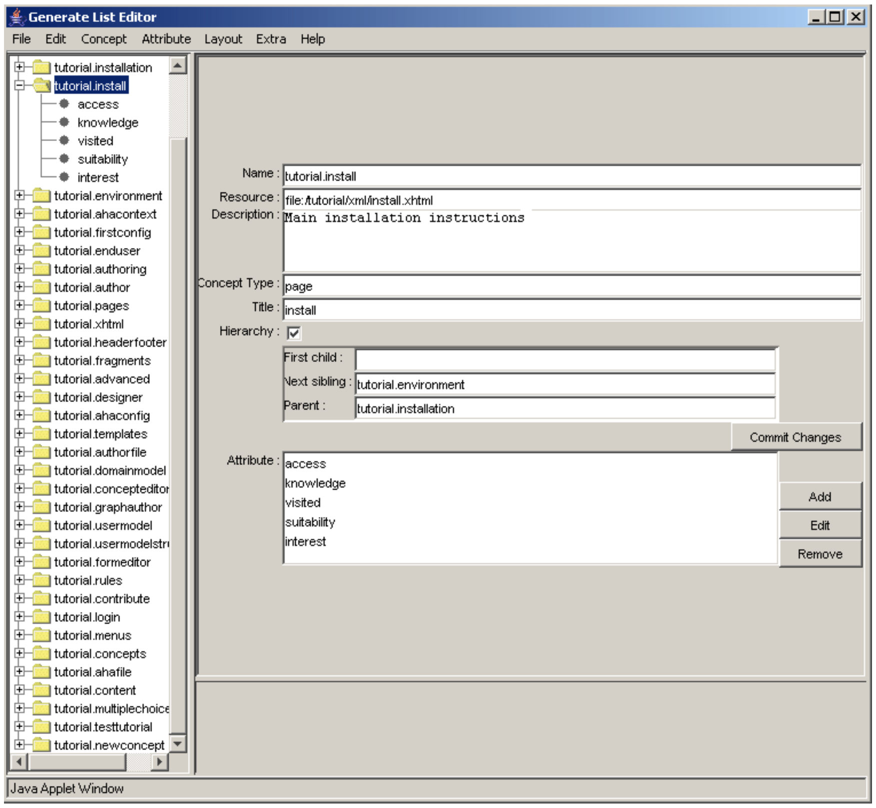
```
<?xml version="1.0"?>

<!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.

4.4 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.) Figure 18 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".



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

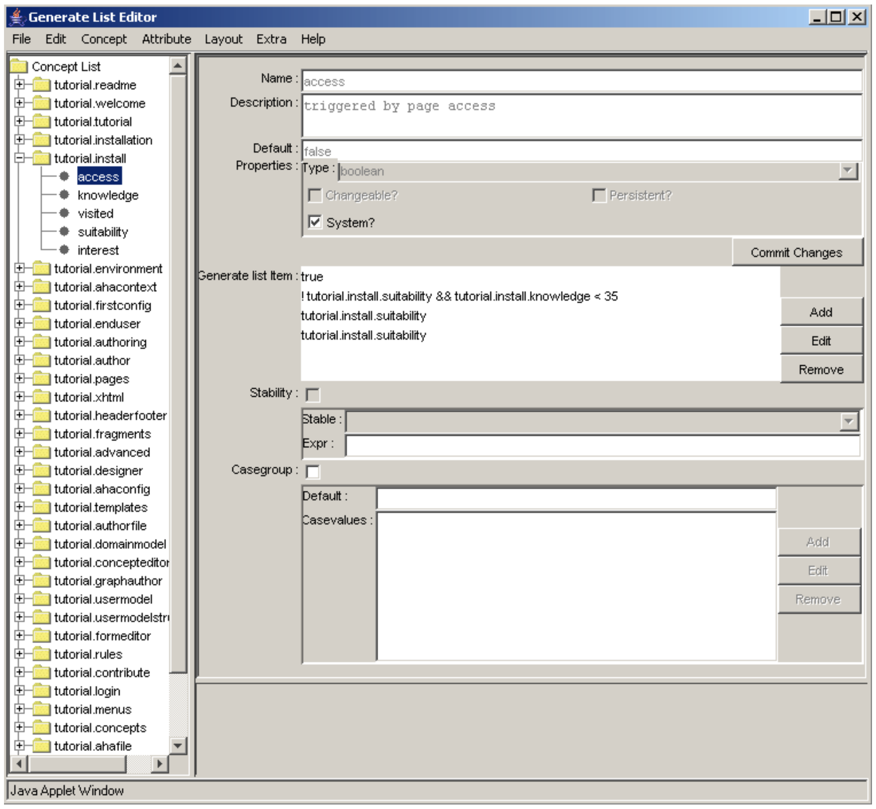
Name : tutorial.install
Resource : file:/tutorial/xml/install.xhtml
Description : Main installation instructions
Concept Type : page
Title : install
Hierarchy : ✓
First child :
Next sibling : tutorial.environment
Parent : tutorial.installation
Attribute : access, knowledge, visited, suitability, interest

Figure 18. The Concept Editor.

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:

- **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 Figure 19 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 Section 4.5 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

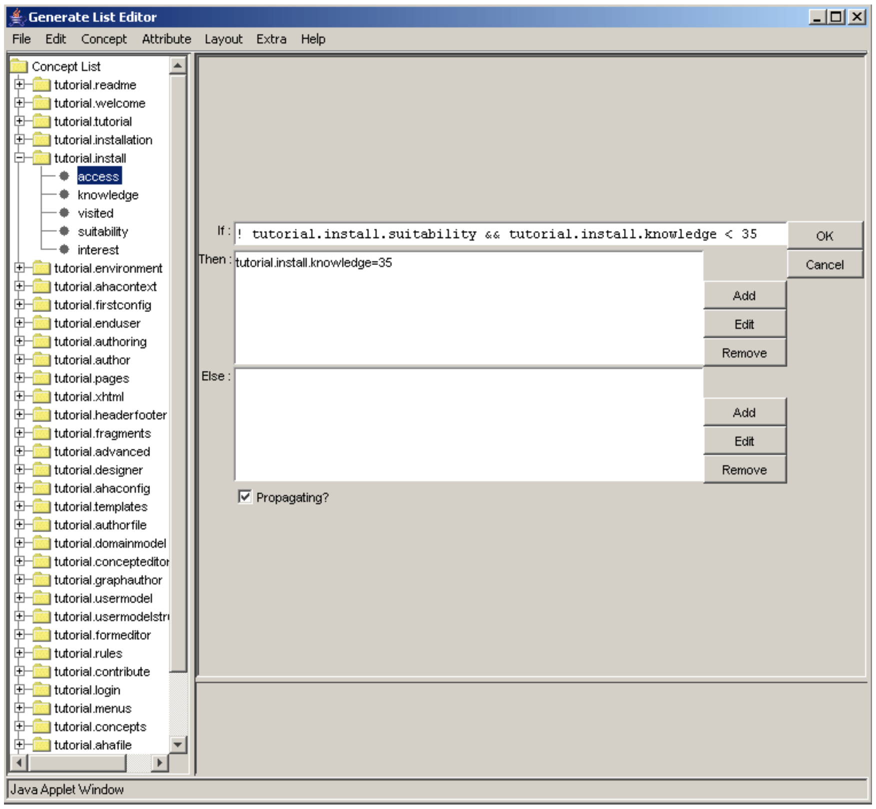
Name : access
Description : triggered by page access
Default : false
Properties : Type : boolean
[ ] Changeable? [ ] Persistent?
[✓] System?

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

Figure 19. 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. Figure 19 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. Figure 20 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 *show-ability*. It is used to conditionally select a resource (file) to include in the presentation in the location of a special <object> tag (See Section 4.8) 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?

Figure 20. Editing an event-condition-action rule in the Concept Editor.

4.5 The Form Editor

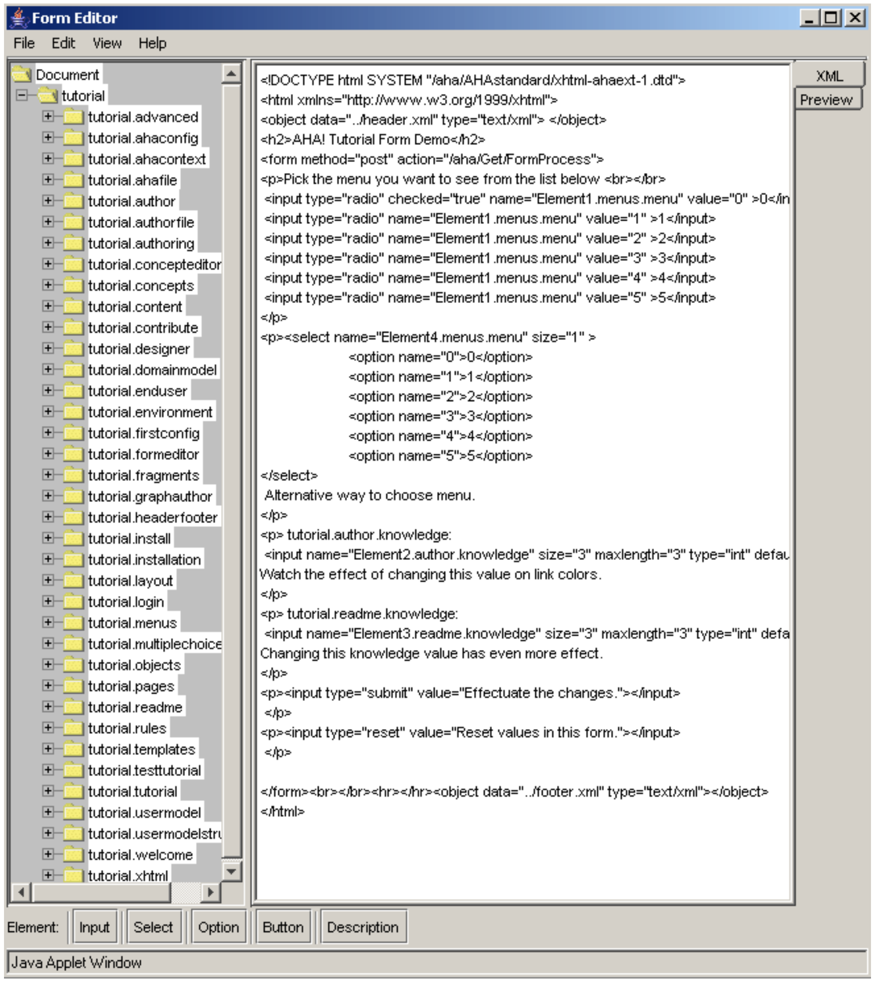
Attributes of concepts defined as “changeable” can be included in a custom-made form. The Form Editor lets you create an (X)HTML form in which form elements for the attributes of concepts are inserted automatically, and in which you create the remainder of the presentation by means of plain HTML code.

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>
```

Figure 21. Form Editor with an example form.

4.6 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.

4.7 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 Figure 16). 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.

4.8 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. Figure 22 shows the "Application Management Tool" (AMt) that makes it easy to copy files (pages and other files) between your PC and the correct directories on the server.

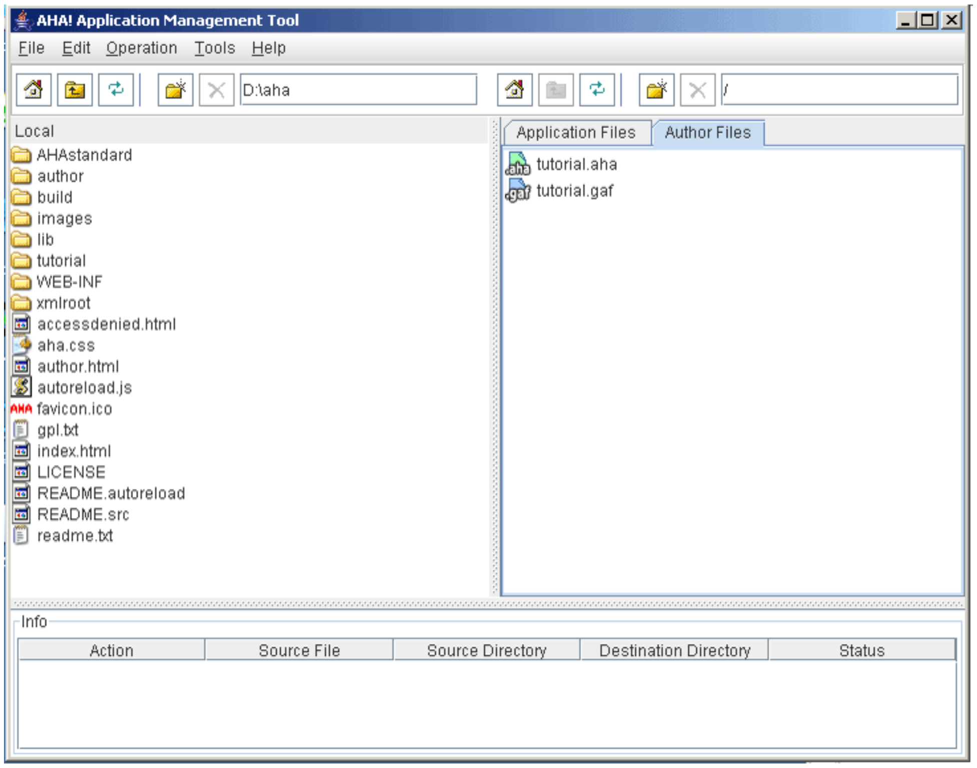


Figure 22. The Application Management Tool

The AMt show an interface that is similar to that of the popular secure shell (ssh) interface. On the left you see the local file system, and on the right you either see the server's file system (in the "Application Files" tab) or the server-side authoring files for the Graph Author and Concept Editor. When you double-click on the ".gaf" files the Graph Author is started automatically and when you double-click on the ".aha" files the Concept Editor is started. It is thus possible to perform all the application creation and maintenance using AMt. The one part that is not supported by special tools in AHA! is the creation of the application's files like pages and images. 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 "unprocessed" links. They are shown using colors that are different from conditional and unconditional links, and when following such links the pages are not processed by the AHA! engine. 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. (Section 4.3.2 explains how to assign resources to concepts.) When a link points to a page the concept definitions are used to find for which concept this page is the default resource.
- 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. Figure 17 shows how to create a “Casegroup” in the Graph Author; Figure 18. does the same for the Concept Editor.

4.9 Advanced topics

4.9.1 Event-Condition-Action rules

In Section 4.3.2 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.

4.9.2 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".

5. Concluding remarks and future work

This tutorial describes why and how to create adaptive Web-based applications. It shows the general architecture of adaptive hypermedia systems and the specific functionality of the AHA! system (version 3.0).

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.
7. De Bra, P., Houben, G.J., Wu, H., AHAM: a Dexter-based reference model for adaptive hypermedia, Proc. of the 10th ACM Hypertext Conference, pp. 147-156, Darmstadt, 1999.
8. Halasz, F., Schwartz, M., The Dexter Hypertext Reference Model. Comm. of the ACM, Vol. 37, nr. 2, pp. 30-39, 1994.