Design Issues for General-Purpose Adaptive Hypermedia Systems

Hongjing Wu, Erik de Kort, Paul De Bra

Eindhoven University of Technology

E-mail: hongjing@win.tue.nl, erik.de.kort@asml.nl, debra@win.tue.nl

ABSTRACT

A hypermedia application offers its users much freedom to navigate through a large hyperspace. For authors finding a good compromise between offering navigational freedom and offering guidance is difficult, especially in applications that target a broad audience. Adaptive hypermedia (AH) offers (automatically generated) personalized content and navigation support, so the choice between freedom and guidance can be made on an individual basis. Many adaptive hypermedia systems (AHS) are tightly integrated with one specific application. In this paper we study design issues for *general-purpose adaptive hypermedia systems*, built according to an application-independent architecture. We use the Dexter-based AHAM reference model for adaptive hypermedia [7] to describe the functionality of such systems at the conceptual level. We concentrate on the architecture and behavior of a *general-purpose adaptive engine*. Such an engine performs adaptation and updates the *user model* according to a set of *adaptation rules* specified in an *adaptation model*. In our study of the behavior of such a system we concentrate on the issues of *termination* and *confluence*, which are important to detect potential problems in an adaptive hypermedia application. We draw parallels with static rule analysis in active database systems [1,2]. By using common properties of AHS we are able to obtain more precise (less conservative) results for AHS than for active databases in general, especially for the problem of termination.

KEYWORDS: adaptive hypermedia, user modeling, adaptation rules, termination, confluence, active databases

INTRODUCTION

Web-based hypermedia systems are becoming increasingly popular as tools for user-driven access to information. They typically offer users a lot of freedom to navigate through a large hyperspace. Authors of hypermedia applications may wish to provide some personal guidance to their users without taking away that navigational freedom (as linear books do):

- A typical hypermedia system presents the same links on a page to all users. Personal guidance requires that the system offer each user (some) personalized links or navigation tools (such as a partial table of contents, a map, or a sorted list of suggested links). The system should thereby take into account what the user read before, what her interests and preferences are, etc.
- When writing pages of a hyperdocument the author must make an assumption about what foreknowledge the user has when accessing each page. The least thing the application should take into account is which pages the user read before. The system can then decide to add some prerequisite explanations, or to omit information that is redundant because the user saw it before (on another page).

Adaptive hypermedia systems perform personalization in these two ways: *adaptive navigation support* and *adaptive content*. The adaptation (or personalization) is based on a *user model* that represents relevant aspects of the user such as preferences, knowledge and interests. The system gathers information about the user by observing the use of the application, and in particular by observing the *browsing* behavior of the user.

Many adaptive hypermedia systems exist to date. The majority of them are used in educational applications, but some are used, for example, for on-line information systems or information retrieval systems. An overview of systems, methods and techniques for adaptive hypermedia can be found in [3]. “Personalized” websites are becoming popular. They typically have a name that starts with “My” (My Yahoo, My Excite, etc.) Some of these systems are only *adaptable*, meaning that the user enters a user profile through a registration form, and the system doesn’t change that profile unless the user explicitly updates it (again through a form). An *adaptive* system performs updates to the user profile automatically by observing the user’s browsing behavior. A primitive form of adaptation is found in systems that log which pages a user accesses, in order to be able to mark pages as “new” or “old” and in order to be able to generate “what’s new” pages. In fact, the traditional Web-browser behavior of changing the color of links from blue to purple after visiting a page is already a form of adaptive behavior.

We have developed a reference model for the architecture of adaptive hypermedia applications: AHAM (for **A**daptive

Hypermedia **A**pplication **M**odel) [7], which is an extension of the Dexter hypermedia reference model [8,9]. AHAM acknowledges that doing "useful" and "usable" adaptation in a given application depends on three factors:

- The application must be based on a *domain model*, describing how the information content of the application or "hyper-document" is structured (using concepts and concept relationships).
- The system must construct and maintain a fine-grained *user model* that represents a user's preferences, knowledge, goals, browsing history and other relevant aspects.
- The system must be able to perform adaptation of the content and link structure, based on the domain model and user model. In order to do so the author must provide an *adaptation model* consisting of *adaptation rules*. The rules define both the process of generating the adaptive presentation and that of updating the user model. An AHS may offer some built-in rules for common adaptation aspects and user model updates. This reduces the author's task of providing such rules. The AHS may also offer a rule language through which authors can define additional rules.

The division into a *domain model* (DM), *user model* (UM) and *adaptation model* (AM) provides a clear separation of concerns when developing an adaptive hypermedia application. Unfortunately, a common shortcoming in many current AHS is that these three factors or components are not clearly separated. The AHAM model advocates the separation of these components in future AHS.

In this paper we focus on *how* an AHS actually performs the adaptation. We define a *rule definition language* and an *adaptation engine* (AE) to execute *adaptation rules*. In previous work [12] we argued that *adaptation rules* should exist at the author level and the system level. (System-defined rules simplify the task that remains for the author.) However, for the analysis of the complete rule system this distinction is irrelevant and therefore not considered in this paper. We describe a *general-purpose rule system* that is more powerful than that of most AHS. As a result, some of the design problems we present may not be present in many AHS with a simpler rule system or a hardcoded adaptation engine. We focus on two design issues:

- We want a powerful rule language through which authors and system designers can specify how different aspects of the domain and user interact to generate an adapted presentation. It becomes difficult to predict whether the interaction between different rules can cause rules to trigger each other indefinitely. In this paper we study the problem of deciding when the rule execution is guaranteed to terminate. In other words, we try to decide which types and combinations of adaptation rules are "safe". Aside from *termination* there is also the issue of efficiency (or "fast" termination), but we do not discuss that topic in this paper.
- Even when the execution of the adaptation rules terminates, the AE may not always produce predictable results. When the same user action under the same circumstances (domain model and user model) is guaranteed to always produce the same result the AHS is said to be *confluent*. We describe an analysis technique to decide whether an AHS is confluent.

This paper is organized as follows. First we briefly recall the AHAM reference model for adaptive hypermedia applications. In the next section we define the rule language associated with AHAM and explain how the rule execution works in AHAM. We then analyze termination and confluence of the AE and we draw parallels with rule execution in active database systems [1,2].

AHAM, A DEXTER-BASED REFERENCE MODEL

In hypermedia applications the emphasis is always on the information nodes and on the link structure connecting these nodes. The Dexter model [8,9] captures this in what it calls the Storage Layer. It represents a *domain model* (DM), i.e. the author's view on the application domain. We use the terms *concept* and *concept relationship* as a generalization of *nodes* and *links*. In adaptive hypermedia applications the central role of DM is shared with a *user model* (UM). UM represents the relationship between the user and the domain model. The user may specify updates to UM (e.g. through forms), and the system also updates UM by tracking the user's browsing behavior.

In order to perform adaptation based on DM and UM an "author" needs to specify how the user's interaction with the AHS influences the presentation of the information from DM. In AHAM [7], this is done by means of an *adaptation model* (AM) consisting of *adaptation rules*. (Note that our earlier work [7] used slightly different terms.) An *adaptation engine* (AE) uses these rules to manipulate link anchors (from the Dexter model's *anchoring* layer) and to generate what the Dexter model calls the *presentation specifications*. Figure 1 shows the overall structure of AHAM (in the same way the Dexter model is depicted in [8,9]).

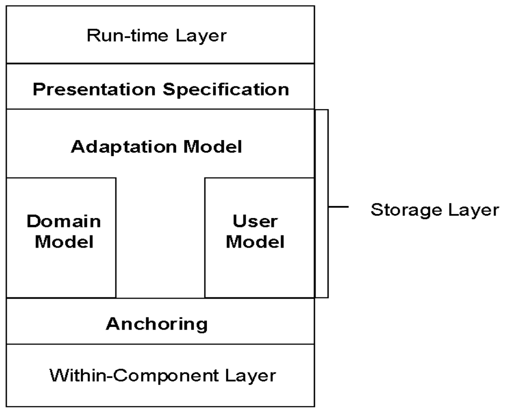

Fig. 1 Structure of adaptive hypermedia applications

The architecture we describe in this paper is only an *abstract* representation of an AHS. Each AHS has some mechanism to decide how to perform adaptation and how to update UM based on the user's actions. AHAM describes this mechanism as consisting of an adaptation engine that is executing adaptation rules. This does not imply that AHAM can only represent AHS that are rule based.

The domain model

The domain model of an adaptive hypermedia application consists of *concepts* and *concept relationships*. Concepts are objects with a unique object identifier, and a structure that includes attribute-value pairs and link anchors. (The remainder of the structure is not relevant for this paper.)

A *concept* represents an abstract information item from the application domain. It can be either an *atomic concept* or a *composite concept*.

- An *atomic concept* corresponds to a fragment of information. It is primitive in the model. This means that its content, attribute- and anchor values have no meaning *within* the model but belong to the "Within-component layer".
- A *composite concept* has a sequence of children (sub-concepts) and a *constructor* function that describes how the children belong together. The children of a composite concept are either all atomic concepts or all composite concepts. A composite concept with (only) atomic children is called a *page*.

The composite concept hierarchy must be a DAG (directed acyclic graph). Also, every atomic concept must be included in some composite concept. Figure 2 illustrates a part of a concept hierarchy.

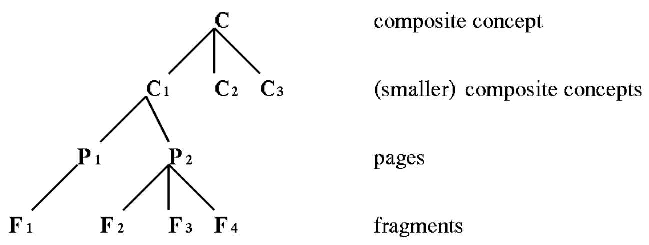

Fig. 2 : Part of a concept hierarchy

A *concept relationship* is an object (with a unique identifier and attribute-value pairs) that relates a sequence of two or more concepts. Each concept relationship has a type. The most common type is the hypertext **link**. In AHAM we consider other types of relationships as well, which play a role in the adaptation, e.g. the type **prerequisite**. When a concept C_1 is a prerequisite for C_2 it means that the user should read C_1 before C_2. It does not imply that there must be a (direct navigational) link from C_1 to C_2. It only means that the system must "somehow" take into account that "visiting" C_2 is not desired before C_1 has been "visited". In AHAM authors and/or system designers can (in theory) define arbitrarily many types of relationships. In fact, one can even imagine an AHS without links (like in spatial hypertext). Figure 3 shows a small example of a set of concepts and concept relationships. In this example we also assume the presence of another type of concept relationship: the **inhibitor**. C_4 "inhibits" C_1, which means that after visiting C_4 it is no longer desirable to visit concept C_1. One way to make the "desirability" of a link destination clear to the user is to use different presentations of link anchors, e.g. by giving them a different color.

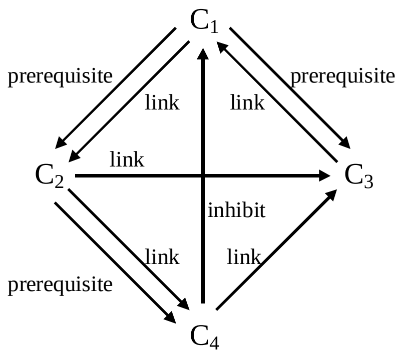

Fig. 3: Example concept relationship structure

Note that a graph presentation like Figure 3 only works for binary concept relationships (such as links, some prerequisites and inhibitors). AHAM allows relationships between a (longer) *sequence* of concepts, in order to be able to describe features like *guard fields* in Storyspace.

Also note that some AHS may represent the concept hierarchy through a **part-of** concept relationship type. The AHA system [6] for instance does not treat the concept hierarchy differently from other concept relationship types. We will also use **part-of** in some examples in this paper.

The user model

A user model (UM) consists of named entities for which we store a number of attribute-value pairs. For each entity there may be different attributes, but in practice most entities will have the same attributes. Therefore, to represent the user model we use a *table structure*, in which for each entity the attribute values for that concept are stored. Most entities in UM represent *concepts* from DM. (Some other entities may represent a user's background, job title, or preferences such as preferred media types, learning style, colors, or also platform specific properties such as screen size or network bandwidth.) In our examples we will often refer to user model attributes that are typical for DM-related concepts, e.g. *knowledge* (about a concept), *read* (to denote that a page or fragment was read by the user) and *ready-to-read* (to denote that the user is "ready" to view information about this concept or to read this page.

The analogy between the structure of the user model and that of database tables suggests that maintaining a user model is very similar to updating a database. The *table* can be considered as a *universal relation instance*, but because different attributes may be appropriate for different entities or concepts it may be a universal instance with null values. (We do not consider the decomposition of this table to obtain a more efficient storage structure in this paper.)

In the sequel we will always consider UM as being the user model for a single user. In this paper we do not discuss adaptation to group behavior.

The user model consists of a *persistent* part and a *volatile* part. For each concept we consider attributes of which the value is maintained, e.g. whether a page has been read, or what the level of knowledge about a concept is. An AHS may opt to recalculate some other attribute values on the fly. Some AHS may verify whether the prerequisites for a concept are satisfied each time that concept is accessed or when a link to that concept is shown, while another AHS may store this information as a *ready-to-read* attribute in UM. In this paper we only consider the user model updates and the adaptation that occur as a result of a single user action (e.g. following a link, accessing a menu, and submitting a form). Therefore it is not relevant to distinguish persistent and volatile attributes.

The adaptation model

The adaptation model (AM) is the key issue in this paper: it describes how the AHS should perform its adaptation. This includes content adaptation, link adaptation and updating the user model. AM is defined as a set of *adaptation rules* that form the connection between DM, UM and the presentation (specification) to be generated. The syntax of adaptation rules is AHS-dependent. In fact, in many AHS a number of rules will be hard-coded into the system and not visible or accessible to authors. In AHAM we must make these rules explicit in order to be able to describe the complete behavior of the AHS.

Let us consider a small example of an AHS in which the relationship between fragments and pages is expressed trough a set-valued attribute "children". In the user model each fragment has a persistent attribute "relevant" to indicate whether it is desirable for this user, and a volatile attribute "pres" to indicate whether the fragment will be included in the presentation of a page that contains it. ("pres" would be part of the *Presentation Specification* layer in the Dexter model.) A rule for accessing a page could then be:

$$< \text{access(P) and } F \in \text{P.children and F.relevant} = \text{true} \Rightarrow \text{F.pres} := \text{show} >$$

This rule only expresses an example of possible behavior of an AHS, in an example of possible rule languages (actually the syntax used in [7]). Instead of hiding undesired fragments the AHS may opt to gray them out, as described in [10]. Instead of describing the relationship between pages and fragments through an attribute of the page we could describe it through an attribute of a fragment, or even as a separate concept hierarchy relationship. We will give more examples, in a more database (SQL)-like syntax, and describe a language with its semantics later.

The adaptation engine

An AHS does not only have a domain model, user model and adaptation model, but also an *adaptation engine*, which is a software environment that performs the following functions:

- It offers generic page selectors and constructors. For each composite concept the corresponding selector is used to determine which page(s) to display when the user follows a link to that composite concept. For each page a constructor is used for building the adaptive presentation of that page. Page constructors allow for dynamic content like a ranked list of links. (Truly generated content is not dealt with explicitly as it is considered part of the within-component layer.)
- It optionally offers a (very simple programming) language for describing new page selectors and constructors. For instance, in AHA [6] a page constructor consists of simple commands for the conditional inclusion of fragments. (AHA has no language for describing page selectors. It only supports links to pages, not to composite concepts.)
- It performs adaptation by executing the page selectors and constructors. This means selecting a page, selecting fragments, organizing and presenting them in a specific way, etc. The adaptation also involves adaptation to links by manipulating link anchors depending on the state of the link (like enabled, disabled and hidden).
- It updates the user model (instance) each time the user visits a page. The engine will change some attribute values for each atomic concept of displayed fragments of a page, of the page as a whole and possibly of some other (composite) concepts as well (all depending on the adaptation rules).

The adaptation engine thus provides the implementation-dependent aspects, while DM, UM, and AM describe the information and adaptation at the conceptual, implementation independent level. Note that DM, UM and AM together thus do not describe the complete behavior of an AHS. The same set of *adaptation rules* may result in a different presentation depending on the *execution model* of the adaptation engine. In the following sections the role of this execution model will be explained. The challenge, of course, is to design an AE that ensures that rule execution always terminates and produces a predictable result. We shall see that this is not always possible.

ADAPTATION RULE LANGUAGE

In the previous section we have described the functionality of AHAM's different parts. In this section we give a more detailed description of the way in which adaptation is expressed, and how it is performed by the adaptive engine.

In the sequel we present an adaptation rule language and assume that this language is available to authors to express the desired adaptation and user model updates. The actual rule system and internal behavior of a "real" AHS needs to be translated to our rule language (and back) in order to infer the actual behavior of the AHS from the behavior of our abstract description in this paper.

To make authoring adaptation rules easier for the author, we provide *generic* rules that apply to all concepts or all concept relationships of a certain type. In addition we provide *specific* rules to describe rules for a specific concept, set of concepts or a specific concept relationship. In a *generic adaptation rule* (bound) variables are used that represent concepts and concept relationships. A *specific adaptation rule* uses concrete concepts from DM or UM instead of variables. Other than that both types of rules look the same. Specific rules always take precedence over generic ones, i.e. a specific rule can be used to override or eliminate a (conflicting) generic rule.

In database literature we find two types of rule (or actually trigger) formalisms that seem useful to describe the behavior of an adaptive engine: *Event Condition Action* (ECA) rules and *Condition Action* (CA) rules. At first sight it seems that ECA rules are best for AHS, because the system always reacts to an event generated by the user (such as clicking on a link anchor). However, after this initial event all rules are just triggered through changes in the user model. By translating the initial event to a simple user model update (updating some attribute value to represent the "click") we can also describe the behavior of an AHS through CA rules.

ECA and CA rules have been studied extensively in the context of active databases [1,2]. ECA rules consist of three "independent" parts: an *event*, a *condition* and an *action*. The semantics of an ECA rule are that when the event is triggered, the condition is checked. If the condition holds, the action will be executed. ECA rules are general rules to describe triggers; their conditions may be true (or false) independent of the cause of the event. The analysis techniques for ECA rules that are known to date [1] are "conservative" in the sense that they take into account the events triggered by an action, but not the conditions. CA rules are much simpler than ECA rules: whenever a rule's condition becomes true that rule's action is executed. Hence, a CA rule is like an ECA rule where the fact that the condition becomes true *is* the *event* that triggers the rule. As already described in [2] many practical applications of ECA rules have the property that a rule's condition becomes true exactly when that rule's event occurs. Such rules are called *quasi-CA* and behave exactly like CA rules.

In this paper we describe adaptation rules for AHS as CA rules. In the next subsections we first present the syntax of our abstract rule language, we then illustrate this language with some examples and finally we describe the execution model for our adaptation rules.

AHAM-CA rule syntax

Since both DM and UM have a structure that consists of objects (like concepts and concept relationships) with an object identity and a set of (named) attributes it appears natural to base our rule language on trigger- and query languages for databases. Doing so also makes it relatively easy to draw a parallel between the properties of our language with CA rules and those of CA rules in active databases, as studied in [2]. In [2] the relational algebra syntax is used, but in order to obtain easily readable rules we base our language on the well-known SQL syntax.

The syntax of the rule system is partially described by the following grammar:

<rule>::=	C→A
C::=	<query>
<query>::=	**select** <name_list>
	where <condition>
A::=	**update** <list_assign>
	{ **where** <condition> }
<condition>::=	<Boolean> | '(' <condition> ')' |
	not <condition> |
	exists <query> | <relationship> |
	<condition> **and** <condition> |
	<condition> **or** <condition> |
	<expr> <eq_op> <expr>
<eq_op>::=	'=' | '≠' | '>' | '<' | '≤' | '≥' | **in**
<expr>::=	<constant> | <name> | <query> |
	<expr> <bmath_op> <expr> |
	<umath_op> <expr>
<relationship>::=	<name> (<name> {, <name>}$^+$)
<bmath_op>::=	'+' | '–' | '∗' | '/' | max | min
<umath_op>::=	'–'
<list_assign>::=	<name> := <expr> {;<list_assign> }*

(Note that we did not define several details like the syntax for constants, names, etc.)

From the context it will always be clear whether a <name> refers to a concept or concept relationship from DM or a concept from UM, so we omit the SQL **from** clause altogether.

In our examples (below) we also use the convention that uppercase names are used to indicate concept variables (used in *generic* rules) whereas lowercase names are used for specific concepts (used in *specific* rules and in *instantiated* rules defined below) and for attributes.

Examples

The examples we give are intended to illustrate our rule language. They are not meant to present generally accepted behavior of AHS (as all systems behave differently).

Example 1

This is the rule example we gave before, but now rewritten in our rule language. The (generic) rule specifies that all relevant fragments of page P will be shown to the user when this page is presented.

C: **select** P.access
  where P.access = **true**
A: **update** F.pres := "show"
  where part-of(F, P) **and** F.relevant = **true**

The meaning of the different clauses in this example is as follows:

1. The rule monitors for a change in P.access because P.access appears in the **select** clause.
2. The rule's action is only executed when the condition in the **where** clause is true for some of the mentioned objects, in this case for some page P.
3. For all fragments F that are part of page P and for which the "relevant" attribute is true the "pres" attribute is set to the value "show". (Note that in a "real" SQL syntax we would need an **exists** and subquery here to check for the existence of the part-of relationship between F and P.)

An interesting aspect of this example is that it shows how information is carried over from the *storage layer* (where the "part-of" relationship in DM and the "relevant" attribute in UM belong) to the *presentation specification layer* (where the "pres" attribute belongs) of the AHAM model.

Example 2
Suppose that the knowledge value of a concept can have the values "not known", "known" and "well known". (In later examples we will also assume the property that "not known" < "known" < "well known".) A concept can be "ready_to_read" or not. Accessing a concept may have the effect described by the following rule:

```
C: select P.access
     where  P.access = true and P.ready_to_read = true
A: update P.knowledge := "well known"
```

Note that while the condition of this rule checks the value of two attributes, the rule is only triggered when the "access" attribute changes. The **select** clause tells the AE to only look for changes to "access". Also, because this rule only changes attribute values for the object (page) that appears in the condition, the action has no **where** clause.

Example 3
Suppose that a concept becomes "ready_to_read" when all its prerequisites are at least "known". We can write a rule to take this into account when a knowledge value changes:

```
C:  select  $C_1$.knowledge
     where $C_1$.knowledge $\geq$ "known"
A:  update $C_2$.ready_to_read := true
     where prerequisite($C_1$, $C_2$) and
          not exists (  select $C_3$
                         where prerequisite($C_3$, $C_2$) and
                                $C_3$.knowledge < "known" )
```

It means that when C_1.knowledge has been changed to "known" (or "well known"), all the concepts for which C_1 is a prerequisite are verified, and for each of them with no more unsatisfied prerequisites the ready_to_read attribute is set to 'true'.

Example 4
As another example let us consider the effect of an "inhibit" relationship that gets activated when a concept becomes "well known".

```
C: select $C_1$.knowledge
     where $C_1$.knowledge = "well known"
A: update  $C_2$.ready_to_read := false
     where inhibit($C_1$, $C_2$)
```

Examples 3 and 4 together already show some of the potential problems with adaptation rules in general: if we allow the arbitrary creation of prerequisite and inhibit relationships and apply these adaptation rules it is possible that two rules are triggered at the same time, one of which tries to set "ready_to_read" to true while the other one tries to set "ready_to_read" to false, for the same concept. Later in this paper we describe how to detect such conflicts.

Users of Storyspace (see www.eastgate.com) will recognize these potential problems: the use of *guard fields* is in fact equivalent to the use of *specific* adaptation rules. Links can become available depending on which nodes the user accessed *or did not access* before. As an author one has to be careful to not disable access to part of the application for some users (unintentionally).

Execution model for AHAM-CA rules
In the previous subsections we presented (part of) the syntax used in the specification of rules, and gave some examples. To determine the actual behavior of the AHS we must describe how the *adaptation engine* actually selects and executes the *adaptation rules*. We call this the *execution model*.

In order to reduce the complexity of the analysis of the rule execution (and at the same time enable optimization of the execution itself) we distinguish a number of *phases* in which rules are executed that have a different purpose. First the adaptive engine AE is started as a result of some user action, e.g. when the user "clicks" on a link. The AE has to perform a series of tasks (sequentially). DM and UM are loaded, and the volatile attributes are initialized through a set of rules ("**i**nitialize **u**ser model" or IU). Then a set of rules will be executed to select the page(s) to display and the fragments to include in that page(s) (phase PS). Another set of rules is used to **u**pdate the **u**ser model (UU). Yet another set of rules determines *how* things will be presented (e.g. link colors, verbose or terse text, plain or grayed out, etc.). We call this phase **g**enerate **a**daptation (GA). The partitioning of rules into these sets and the sequential execution of these sets of rules can be realized through the association of rules with a *priority*. The AE will start by executing rules in IU, which must have the highest priority. Rules in PS come later by giving them a lower priority, and by making sure that these rules do not cause rules from IU to be executed again. Rules from UU and then GA have a still lower priority. (It is easy to write conditions to ensure that rules never trigger a rule from a previously completed phase. In the sequel we will limit ourselves to rules within a single phase and not worry about accidentally invoking rules from another phase.)

As the examples in the previous section already indicate we need to assume that a user's action induces an update that sets the rule execution in motion. The examples use a

(Boolean) attribute "access" for this purpose. Every rule that has P.access in its **select** clause will be triggered by a change in the access attribute of a concept (most likely a page). **(**A specific rule that uses p.access is triggered when the specific page p is accessed.) The condition (**where**) ensures that the rule is only executed when p.access becomes true. The triggered rule(s) will perform updates to attributes of some concepts, thereby triggering other rules, etc. (Note that some rule in a later phase needs to change "access" back to false to ensure that the next access to page p is registered again.)

It is all too easy to define sets of rules that trigger each other indefinitely and thus cause infinite loops in the AE. It is clear that some restrictions are needed to prevent such undesired effects. We first describe some "common sense" constraints that each rule system for AHS should observe:

- The propagation of updates to attribute values within a single concept is not allowed, except when one of the attributes is an "event" attribute such as "access". So when p.access becomes true, a rule may set p.read to true and p.knowledge to "well-known". There is no need to have a rule that is triggered by a change to an attribute like p.knowledge and that updates p.read (because the event that generates the update to p.knowledge can update p.read as well). This constraint eliminates rule sets that generate infinite loops of updates to attributes of a single concept.
- The propagation of updates (to attribute values) from one concept to another in a *generic* rule is only allowed when one of the **where** clauses of the rule contains a relationship between these concepts. A rule that does not satisfy this constraint does not make sense because it would specify arbitrary propagation of updates between unrelated concepts.

It is clear that while the above constraints eliminate certain meaningless rule sets, infinite loops in the rule execution are still possible when there are cycles in the relationships that are used in the adaptation rules. For some common types of relationships, such as *prerequisites* we can require that they do not have cycles. (For *link* relationships we should not enforce this restriction but fortunately links seldom play a role in adaptation rules.) But even when there are no cycles when considering relationships of one type, the interaction between concept relationships of different types may still result in infinite loops. Cycles can also always occur with sets of *specific* concept relationships. In the section on *termination* we show how to detect possible infinite loops in general.

The AE needs to *instantiate* the generic rules before it can apply them. When a generic rule says that "when a page P is accessed P.read is set to true", the AE will actually use a rule for each page p, saying that when page p is accessed p.read becomes true. Also, when a condition contains a concept relationship, the actual instance of the relationship is used to make the rule specific. For instance, let c_1 be a concept and let c_1 be an inhibitor for c_2 and c_3 (i.e. the relationships inhibit(c_1, c_2) and inhibit(c_1, c_3) exist). The rule in example 4 is then instantiated to:

C: **select** c_1.knowledge
  where c_1.knowledge = "well known"
A: **update** c_2.ready_to_read := **false** ;
      c_3.ready_to_read := **false**

The instantiation of generic rules makes it easy to detect the presence of specific rules that are in conflict with some generic rule. The instantiation of the generic rule for the objects that appear in the specific rule is then discarded.

The following pre-process translates a set of generic and specific rules to a set of instantiated rules (Ins_rules).

1) Instantiate all generic rules and add to Ins_rules;
2) For each specific rule override (remove) an instance of a generic rule if both following conditions hold:
   - their conditions are logically equivalent;
   - the sets of attributes that appear in the <list-assign> of action statements are equal.
3) After instantiating all generic rules to specific rules, merge syntactically equivalent rules into one rule.

Note that while conflicting (single) generic and specific rules can be detected, there may still be conflicts that only become apparent when one looks at sequences of rules.

The semantics associated with (instantiated) adaptation rules is as follows: for every rule $C \rightarrow A$, let C^{old} be the set of selected attributes (actually attribute values) of objects for which condition C was true before the event that activated AE. (Initially, when a user model is created for a new user, $C^{old} = \emptyset$ for all rules.) For most rules C^{old} contains 0 or 1 elements because of our instantiation mechanism. For instance, in Example 2 each instantiated rule corresponds to the access of a single page (which is either true or false), and in Examples 3 and 4 each instantiated rule corresponds to the knowledge value for a single concept. Let C^{new} be the set of selected attributes of objects for which condition C is true at some point in time during the rule execution process. Again, in most cases C^{new} will contain 0 or 1 elements.) The rule becomes *active*, i.e. it can be executed, when $C^{new} – C^{old} \neq \emptyset$. More than one rule may become active at once. Most adaptive engines will work sequentially and pick one of the active rules to be executed. Apart from its use to partition the rule set into different *phases*, we can also use priorities to indicate which rule to execute whenever several rules are active. When a rule is executed, the new state (of UM) is used to recalculate C^{old} for that rule. (Otherwise a rule would remain active and fire again and again.) C^{old} remains the same for all other rules, but for some rules C^{new} may change. This may cause some *inactive* rules to become *active* (now having $C^{new} – C^{old} \neq \emptyset$) and some *active* rules (not yet executed) to become *inactive* again (now having $C^{new} – C^{old} = \emptyset$). Rule processing continues until for all rules $C \rightarrow A$ the set $C^{new} – C^{old} = \emptyset$, or in other words until there are no more *active* rules.

The rule execution procedure described above only gives a *conceptual* view of what goes on in the AE. An actual AE will use an optimizer to eliminate the repeated evaluation of rule conditions (calculating C^{new}) that our description might suggest.

TERMINATION AND CONFLUENCE

The partitioning of rules into different *phases* reduces the practical problem of verifying that the rule execution is "well behaved" but theoretically the problem remains just as hard: the rule execution (within a phase) may not terminate, or if it does it may not always produce the same result. These problems are generally called *termination* and *confluence*. They have been studied in a number of fields, including *automata*, *term rewriting* and *active databases*. We base our analysis on the research in active databases[1]. In that research the analysis can only use the database *scheme*. We can obtain more precise results because we perform our analysis on *instantiated rules*, thus on a "database" in which part of the *instance* is known (namely the domain model, and the concepts that occur in the user model). Our analysis is also simpler because we only need to consider modifications (no insertions or deletions).

The following example shows how easily rules can trigger each other indefinitely:

```
C: **select** C₁.attr
     **where** C₁.attr > 0
A: **update**  C₁.attr := C₁.attr  + 1
```

It is clear that once the condition of this rule becomes true the rule will trigger itself indefinitely. Fortunately cases like this are very easy to detect.

The detection of infinite loops becomes more difficult when concept relationships are involved:

```
C: select C1.attr
     where C1.attr > 0
A: update  C2.attr := C2.attr  + 1
     where rel(C1, C2)
```

Here an infinite loop occurs only when there are cycles in the relationship "rel". Because we base our analysis on *instantiated* rules we can actually detect such cycles and thus detect such potential infinite loops.

Although in this example the cyclic nature of a single relationship type causes the infinite loop it is easy to come up with examples in which a combination of relationship types causes infinite loops, even though the relationships of each type (separately) do not contain cycles. Again, such cycles can be detected when using instantiated rules. We can thus improve on the *static analysis* presented in [1,2] for active databases. The analysis is called *static analysis* because it only looks at the database scheme, not the actual instance. In our case we look at part of the instance because we use *instantiated rules*.

When the rule execution terminates, it is still possible that the resulting UM depends on the order in which the adaptation rules are executed. Indeed, the action of a rule can perform updates that *activate* several other rules; one of these is then executed and it may *deactivate* other rules. Should the AE make a different choice as to which active rule to execute first the outcome of the whole rule execution process could be different. When the outcome is independent of the execution order the AHS is said to be *confluent*.

Some AHS are created in such a way that termination and confluence are always guaranteed. The AHA system [6] for instance guarantees termination by only allowing the propagation of monotonic updates and by only supporting attributes with a finite value domain (integers between 0 and 100). When one would remove the restriction of "monotonic updates only" termination would no longer be guaranteed, but infinite loops could theoretically still be detected because AHA has a finite state space. However, even in this case the runtime detection of infinite loops would be impractical because the finite state space is very large. Therefore, even in such cases where one could in theory detect violations of the *termination* property at runtime one will in practice use a static method such as the methods we describe below.

We also show a static analysis method to predict problems with *confluence*. However, as we shall see, confluence can only be guaranteed under severe restrictions for the rules. For the case where confluence is not guaranteed, the AE needs to apply "tricks" to produce predictable results. A common technique is to associate a priority to every rule, concept, concept relationship and/or attribute. However, arbitrary tricks tend to mask problems rather than solve them. We therefore suggest to use execution *phases* and guarantee confluence within each phase.

Generating an Activation Graph

The interaction between adaptation rules is as follows:

Definition 1

Consider two rules $r_i : C_i \rightarrow A_i$ and $r_j : C_j \rightarrow A_j$. Let C_j^{old} denote the result of C_j the last time r_j was evaluated during rule processing and let $C_j^{old} = \varnothing$ if r_j has never been evaluated preciously (see our earlier description of semantics) .

- r_i may *activate* r_j if the execution of action A_i can change UM from a state in which $C_j^{new} - C_j^{old} = \varnothing$ to a state in which $C_j^{new} - C_j^{old} \neq \varnothing$.
- r_i may *deactivate* r_j if the execution of action A_i can change UM from a state in which $C_j^{new} - C_j^{old} \neq \varnothing$ to a state in which $C_j^{new} - C_j^{old} = \varnothing$.

1 Because of lack of space we cannot discuss our choice in detail. The translation of our rule system to the field of term rewriting appears to be very complicated. The use of automata (where nodes are represented by states and links by transitions) is not straightforward in the case of conditional links, because these lead to an explosion of the required number of states.

Following and extending [2] we can describe the behavior of the AHS by using an *Activation Graph* (AG). (We shall look at the issue of *deactivation* later.) In such a graph nodes represent (instantiated) rules and edges indicate that one rule may *activate* the other rule. If there are no cycles in the graph, then rule processing is guaranteed to terminate. The difficulty in this analysis is how to decide whether one rule can activate another rule, i.e. which edges to add to the graph. We must guarantee that every possible activation is represented as an edge in the graph. But we must also try to *only* include edges when this activation is really possible (for some instance of UM).

In [2] a *propagation algorithm* (PA) is described that decides when to include an edge $r_i \rightarrow r_j$ in AG. It does so for so called *active databases*. The main difference between AHS as described in AHAM and active databases as described in [1,2] is that in AHS we only have *updates* to the user model (no *insertions* or *deletions* of concepts). Furthermore, these updates are much more specific because we use *instantiated* rules. (We use the actual instance of DM.) Another difference is that in active databases many different possible (initial) updates must be considered whereas in an AHS the rule execution starts after a very specific type of update that represents a user generated event like clicking on a link anchor. Below we first describe the general propagation algorithm. Then we show how the algorithm can be improved for the use with AHS.

The basic PA described in [2] uses the action of r_i (say A_i) and the condition of r_j (say C_j) to decide whether r_i may activate r_j. The PA of [2] does this by combining the relational algebra expressions of A_i and C_j. Through syntactic analysis of these expressions one can determine whether the update of A_i influences the result of the query in C_j. Because of the use of instantiated rules in AHAM we can easily decide which attributes of which objects may be updated by A_i because they appear in the assignments in <list_assign>, and which (instantiated) rules have a condition that uses these attributes of one or more of these objects (because these appear in the **where** clause of the condition). As an example it is easy to see that the action of the rule in Example 2 can result in the condition of the rule in Example 3 becoming true *for the same concept*. Accessing one page may "generate" the last bit of required prerequisite knowledge for another concept.

In order to ensure that every possible rule activation is represented by an edge in the activation graph the basic version of PA includes $r_i \rightarrow r_j$ in the graph when A_i contains an assignment to an attribute of a concept that appears in C_j. This approach is conservative in a number of ways: the update may be conditional (as in Example 3); A_i may set an attribute value without actually changing it; and even when A_i changes an attribute value that is use in C_j it may not change $C_j^{new} – C_j^{old}$.

There are a number of ways in which the basic PA can be improved, while remaining conservative:

1. Properties from the value domains of attributes can be used to determine that an update changes the value to something that the condition can accept. If A_i sets a knowledge attribute to "not known" and C_j checks for knowledge $\geq$ "known", we know that r_i will not activate r_j. (This cannot be done in general, but for many domains this is easy, e.g. for integers with simple expressions and for enumerated types.)
2. In [2] it is already remarked that we can also include C_i in the process: when action A_i is executed it must be executed in a situation in which C_i was satisfied. This can possibly provide some information needed to see whether C_j will be satisfied after executing A_i.
3. Since we know the "event" that starts the AE we can start the construction of the activation graph at the concept with the initial "event", e.g. the concept for which the "access" attribute becomes true. We can thus improve on the above step by taking into account all the conditions of the rules on a path from the rule that is triggered first to the rules r_i and r_j for which we are investigating the possible activation.
4. A final improvement can be obtained by constructing a separate activation graph for each possible "event" that starts the AE. Indeed, for every link anchor we can construct an activation graph that represents the possible rule executions that are a consequence of a user clicking on that link anchor.

Termination analysis of CA rules

Recall the mechanism for rule processing of CA rules. The semantics of CA rules are such that, after each execution of a rule r, r becomes active again only if new attribute values for objects make the condition become true (i.e. $C^{new} – C^{old} \neq \emptyset$). The *Activation Graph* constructed by PA contains an edge $r_i \rightarrow r_j$ if A_i may add elements to C_j^{new}. The basic PA is conservative because it assumes $C_j^{old} = \emptyset$. (Some of the optimizations may detect the guaranteed presence of elements in C_j^{old}.) The PA also assumes that when the assignments in A_i *can* make $C_j^{new} – C_j^{old}$ become non-empty they *do*. Therefore the following holds:

Theorem 1:
If there are no cycles in AG then the rule execution for the given adaptation AM is guaranteed to terminate.

Confluence analysis of CA rules

Assume that rule execution terminates, i.e. there are no cycles in the Activation Graph. We wish to determine if every possible rule execution, given the same input event, DM and UM, is guaranteed to have only one final state.

If we look at the basic (unoptimized) PA, we see that it creates edges $r_i \rightarrow r_j$ for each pair of rules such that A_i assigns values to attribute of objects that appear in C_j. The effect of A_i can be the *activation* of r_j as well as the *deactivation* of r_j. AG as obtained through the basic PA is thus an *Activation Graph* as well as a *Deactivation Graph* (say DG). By using different heuristics than for AG we can obtain an optimized PA for constructing DG.

Definition 2:
Consider two rules $r_i : C_i \rightarrow A_i$ and $r_j : C_j \rightarrow A_j$. The actions A_i and A_j *commute* if, for all possible instances of UM, the execution of A_i followed by A_j and the execution of A_j followed by A_i produce the same new instance of UM.

Lemma 1: [2]
Two distinct rules r_i and r_j commute if: (1) r_i cannot activate r_j; (2) r_i cannot deactivate r_j; (3) condition (1) and (2) with i and j reversed; (4) r_i's action and r_j's action commute.

To guarantee the commutativity of two rules r_i and r_j, we need to verify the conditions (1)-(4) in the above lemma. Condition (1) can be seen from the *Activation Graph*: an edge $r_i \rightarrow r_j$ indicates that r_i may activate r_j. Condition (2) can be seen from the (very similar) *Deactivation Graph*. For condition (3) we only reverse the role of i and j. For condition (4) we (conservatively) check whether A_i and A_j do not assign values to the same (attributes of) objects.

The AE may have a choice between several (more than two) active rules at any time. So there may be many possible rule execution sequences. In [2] it is shown that:

Theorem 2:
A rule set R is confluent if all pairs of rules in R commute.

In [2] Baralis and Widom argue that this theorem cannot really be improved upon: without any restriction on the order in which rules are executed confluence can only be guaranteed if all the rules that need to be executed are activated by the initial event, and not by each other. Of course, rules are allowed to activate rules that belong to a later *execution phase*. Thus, the phases we introduced do serve a purpose in guaranteeing confluence.

CONCLUSION AND FUTURE WORK

We have proposed an abstract rule (or trigger) language to specify which adaptation and user modeling functionality an AHS offers. We have described the semantics of rules and how an adaptive engine executes the rules. We characterize AHAM rules as Condition-Action rules and used some results from [2] for analyzing termination and confluence. Our analysis methods are less conservative than those of [2] because we use *instantiated* rules (that use the instance of the *domain model*) and *execution phases*.

In actual adaptive hypermedia systems authors will not write all these rules by hand. An AHS will have part of the adaptive behavior built in. We have made this behavior explicit in order to provide a framework for comparing different AHS (and possibly translating applications from one AHS to another).

Our future work will consist of the creation of user-friendly authoring tools (e.g. as an addition to the AHA system [6]) that let authors define how adaptation is to be done in their system. Such authoring tools can offer support in the form of analysis tools. The authoring environment can warn an author when rule execution (during end-user browsing activity) may not terminate or may generate unpredictable results.

Acknowledgement
We wish to thank the anonymous reviewers for their helpful suggestions and their requests for additional explanations and examples.

REFERENCES

1. Aiken, A., Widom, J., Hellerstein, J.M. Static Analysis Techniques for Predicting the Behavior of Database Production Rules. ACM Transactions on Database Systems, Vol. 20, nr. 1, pp. 3-41, 1995.
2. Baralis, E., Widom, J. An algebraic approach to static analysis of active database rules. ACM Transactions on Database Systems, Vol. 25, nr. 3, pp. 269-332, 2000.
3. Brusilovsky, P. Methods and Techniques of Adaptive Hypermedia. User Modeling and User-Adapted Interaction, 6, pp. 87-129, 1996. (Reprinted in Adaptive Hypertext and Hypermedia, Kluwer Academic Publishers, pp. 1-43, 1998.)
4. Baralis, E., Ceri, S., Paraboschi, S. ARACHNE: A tool for the analysis of active rules. Proceedings of the Second International Conference on Applications of Databases, pp. 68-81, (Santa Clara, December 1995).
5. Ceri, S., and Widom, J. Deriving production rules for constant maintenance. Proceedings of the Sixteenth International Conference on Very Large Data Bases, Pages 566-577, (Brisbane, Australia, August 1990).
6. De Bra, P., Calvi, L. AHA! An open Adaptive Hypermedia Architecture. The New Review of Hypermedia and Multimedia, pp. 115-139, 1998.
7. De Bra, P., Houben, G.J., Wu, H. AHAM: A Dexter-based Reference Model for Adaptive Hypermedia. Proceedings of ACM Hypertext'99, pp. 147-156, (Darmstadt, 1999).
8. Halasz, F., Schwartz, M. The Dexter Reference Model. Proceedings of the NIST Hypertext Standardization Workshop, pp. 95-133, 1990.
9. Halasz, F., Schwartz, M. The Dexter Hypertext Reference Model. Communications of the ACM, Vol. 37, nr. 2, pp. 30-39, 1994.
10. Hothi, J., Hall, W. An Evaluation of Adapted Hypermedia Techniques Using Static User Modeling. Proceedings of the Second Workshop on Adaptive Hypertext and Hypermedia, pp. 45-50, 1998.
11. Paton, N.W., Díaz, O. Active Database Systems. ACM Computing Surveys, Vol. 31, nr. 1, pp. 63-103, 1999.
12. Wu, H., De Bra, P., Aerts, A., Houben, G.J. Adaptation Control in Adaptive Hypermedia Systems, Proceedings of the International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems, LNCS 1892, pp. 250-259, Springer Verlag, (Trento, Italy, August 2000).