Modelling Personalisable Hypermedia : The Goldsmiths Model

February 17, 2003

Abstract

This paper addresses the issue of how hypermedia systems such as the WWW can be endowed with features which allow the personalisation of the interaction process between the hypermedia and the user. The approach taken is unique in formally modelling a rich set of abstract user-initiated personalisation actions which enable individual users to come closer to satisfying their specific, and often dynamic, information retrieval goals.

The model proposed is descriptive, rather than prescriptive, and is cast at a level of abstraction above that of concrete systems exploring current technologies. Such an approach, it is hoped, will allow for user and system-initiated personalisation actions to be studied with greater conceptual clarity than is possible with technology-driven experimentation.

In this paper also describes the development of a personalisable hypermedia system called *PAS*. Developed at Goldsmiths College, University of London, PAS embodies the main concepts underlying the model proposed.

1 Introduction

Research into personalisable hypermedia aims to enhance the functionality of hypermedia systems by making the user interaction process personalisable [3, 6]. This research is motivated by the need (both scientific and commercial) to increase the effectiveness of hypermedia systems as a platform for information retrieving tasks when users have different information goals and histories [4, 5].

Broadly, the approach taken is to endow hypermedia systems with personalisation features which may be initiated by the users or by the system itself. Such personalisation features are assumed to be useful in areas, such as learning, where users have different information seeking goals, histories and preferences. Personalisable hypermedia systems (PHSs) aim to use knowledge provided by (or captured about) specific users to tailor the information and the links presented to each specific user. By applying the knowledge of users, personalisation features can be employed to support user navigation by limiting the options for traversal to information units, tailoring content, suggesting relevant links to follow and providing additional information on links and information units. For the purpose of this paper Personalisation is defined to be the user-initiated tailoring of hyperdocuments.

This paper we first addresses the issue of how to characterise precisely the emergent properties of

hypermedia, thereby making possible a systematic, principled and exhaustive elicitation of the space of possible personalisation actions within hypermedia systems. A rich set of abstract user-initiated personalisation actions are then modelled. These enable users to come closer to satisfying their specific and often dynamic, information retrieval goals.

The rest of the paper is structured as follows: To motivate the contributions of the paper, Section 2 outlines key issues and concepts that underpin our research; Section 3 provides an introduction to the model contributed by this paper; Sections 4 to 5 present the model; Section 6 describes the development of a personalisable hypermedia system, *PAS*, that embodies the main concepts underlying the model; Section 7 compares our results with those obtained by other researchers; Section 8 draws conclusions.

2 Motivation

This section details the motivation for the formal approach taken towards understanding hypermedia personalisation by outlining some of the key issues, concepts and principles underpinning our research.

It is now generally accepted [4] that users of hypermedia systems may differ in their information goals insofar as they may have preferences as to what information is provided and which links are used to navigate the information space. Users may also differ in their histories in that they are likely to have different knowledge of the information contained within the hypermedia system, of the information space and how it may be navigated.

Most of the interaction a user might experience with a hyperdocument is determined by the design decisions that shape the hyperdocument in terms of its content, rendering and navigation possibilities (i.e., its links). As a consequence of the fact that these decisions are unilateral and irreversible (i.e., cannot be overridden by users as they interact with the hyperdocument), it can be said that the designers *own* the hyperdocument.

An approach to overcoming this impediment is proposed by extending hypermedia systems with formally defined personalisation actions that effect a transfer of ownership from designers of the hyperdocument to each of its users, thereby enabling the latter to redesign the former, completely if necessary, according to their specific information goals and histories.

The scope for personalisation actions, it is argued, comprises exactly the emergent properties of hypermedia systems when these are viewed as only loosely coupled to a variety of servers, in each of which the scope for *orthogonal* personalisation actions can also be characterised.

As there is no prior reason to constrain the space of possibilities, one is led to conclude that, in principle, all the decisions that designers make when composing hyperdocuments are potentially within the scope of personalisation actions.

At present, the claim that making hypermedia personalisable increases their efficiency and effectiveness as information retrieval systems suffers from a lack of empirical evidence. For researchers to provide this empirical evidence they must address the question of what is the subject of personalisation actions,

and which personalisation actions could be made available to users.

The challenge is, therefore, how to model, at a suitable level of abstraction, the space of possibilities for personalisation actions that could be made available to users. Furthermore, the choice of personalisation actions we argue should fall out from this abstract model of personalisable interaction and should ultimately be subject to empirical tests for effectiveness gains.

The research reported here aims to respond to these shortcomings by characterising a core of hyperlink-based functionality viewed as a client technology. Personalisation actions are viewed as ranging over entities within this core and as effecting the transfer of ownership from designers of hyperdocuments to their users. Personalisation is defined to be the user-initiated tailoring of hyperdocuments.

3 Introduction to the the Goldsmiths Model

This section describes the architectural assumptions made and the technical approach taken in modelling personalisable interaction with hyperdocuments.

3.1 Architectural Assumptions

A general open architecture for hypermedia systems of the kind depicted in Figure 1 as a simplified data flow diagram[1] is assumed. This architecture reflects the predominant approach to the design of hypermedia systems.

As depicted in Figure 1, it is assumed that a core of hyperlink functionality is a client technology loosely-coupled to (at least) a user-interface server (UIS) and a database server (DBS). The classical example of a UIS is a WWW browser. Among other functions, browsers broker requests and render formal texts (e.g., rendering expressions authored in HTML or XML). Examples of DBSs are Database Management Systems (DBMSs) which support client server architectures (e.g., Oracle).

Broadly, the dynamics associated with Figure 1 are as follows. The UISs capture requests for desired hyperpages. The UISs channel requests for hyperpages into the hypermedia system proper. If a request is for a hyperpage which resides in a remote hypermedia system, then the core of hyperlink functionality interacts with it to obtain the requested hyperpage in the form of a rendering expression that the core can pass back for the UISs to render. If the request is for a local hyperpage (e.g., one which is known to the core) then the latter responds by composing a rendering expression that can be rendered by UISs, possibly after querying one or more DBSs to fetch some or all of the content specified for the requested hyperpage.

[1]The notational conventions should be familiar from classical structural analysis (see, e.g., [22], p. **17**/15), and can be characterised as follows. A square denotes an *external entity*, i.e., a producer or consumer of data that lies beyond the boundary of the function being described. An oval denotes a sub-function of the function being described. An arrow denotes the flow of the data whose name labels the arrow. In diagrams to follow, two parallel lines denote a data store into which data flows and where it is left waiting until it flows out again.

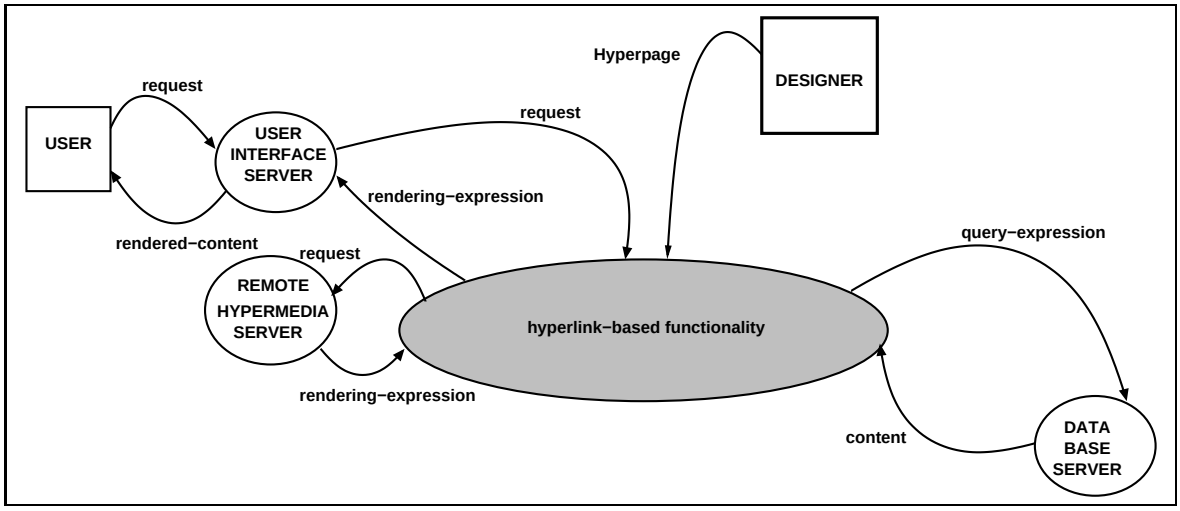

Figure 1: A General, Open Architecture for Hypermedia Systems

Implicit in Figure 1 is the assumption that personalisation actions in hypermedia systems should not, and need not, be compounded with personalisation actions that might be provided by user-interface and database components in hypermedia system architectures. The shaded oval in Figure 1 is responsible for what users experience as hyperlink-based information retrieval. Notwithstanding the fact that users may well want to personalise database and user-interface features, it can be strongly argued that whatever is in the scope for personalisation actions in hypermedia systems resides in the shaded oval.

To model personalisable interaction with hyperdocuments, a model is proposed in which the shaded oval in Figure 1 is partitioned into two regions. The H-region (see Section 4) that models a core of hypermedia functionality and the P-region (see Section 5) that models personalisable hypermedia-based interaction.

3.2 Technical Approach

Technically, the approach taken is to formally model a core of hypermedia functionality as a *composer from specifications*, i.e., what the designer of a hyperpage designs is not a hyperpage, but rather a specification of how to build the hyperpage upon request. Hyperpages are modelled as formal specifications and a formal language is defined for this purpose.

The semantics of hyperpage specifications are given with reference to a formal abstract machine whose operation and instruction set is specified in Section 4.

Personalisation is modelled as the user-initiated process of annotating and rewriting a hyperpage specification into a version thereof that is associated with the user who took that action. It follows that the hyperpages users see rendered may, if they wish, reflect their preferences, shaped by their information goals and their histories.

When personalisation functionality is layered over the core, a designer can annotate a hyperpage in preparation for differences in users' goals and histories. A user can request to personalise not only

such annotations, but the hyperpage specifications as well.

Hyperpage annotations and personalisation requests are modelled as formal specifications and formal languages has been defined for this purpose. Set-theoretic and relational algebraic expressions are used to represent the semantics of personalisation requests.

In summary, the model proposed is an abstract model, as many steps removed from concrete implementations as necessary to allow a systematic, exhaustive investigation of personalisation issues. The model is an open model, insofar as hypermedia systems are viewed as clients of a variety of servers, and in particular of data and user-interface servers. Personalisation involves a transfer of ownership of the process of interaction with a hyperdocument from designers to users. To ensure that the set of personalisation actions is consistent, its elements are induced from the formal definition of the hyperdocuments they act upon. All design decisions are, in principle, within the scope of personalisation actions.

4 The H-Region: Modelling Non-Personalisable hypermedia-Based Interaction

This section introduces a group of functions that model a core of hypermedia functionality referred to as the H-region. Within the H-region users can only request for hyperpages to be rendered. The decisions that the designers of a hyperdocument have made with respect to content, navigation and rendering cannot therefore be overridden.

4.1 The H-Region: A Conceptual Framework

This section presents the conceptual framework underlying the H-region using a bottom-up, constructive approach in which primitive notions are presented before those derived from them.

A *content specification* (C-spec) defines content which is to appear in a hyperpage. A C-spec takes the form of data values (e.g., numbers, text, etc.)[2] or more generally, requests to DBSs (i.e., query expressions which DBSs can evaluate into data values which are served back). A C-spec may be as simple as a number or a string and as complex as a sequence of complex queries which are to be sent to a variety of DBSs, possibly in many different query languages, and using many different client-server protocols.

A C-spec may be associated with a set of *template variables*. Conceptually, a template variable is a place holder for the content denoted by the C-spec it is associated with. This content becomes available after the C-spec is evaluated.

[2]It is assumed that this could, in turn, take the form of a reference to a local data file (containing, e.g., images, sounds, etc.). If this is the case, it is assumed that the UIS knows how to de-reference the name and get hold of the file content for rendering. Under this assumption, values and references to files containing values are considered to be conceptually the same from the point of view of the model.

A *rendering specification* (R-spec) defines how content is to be rendered by a UIS. Every R-spec is paired with one, and only one, C-spec. An R-spec takes the form of formal text in a language which the intended UIS can render, except that this renderable text may be interspersed with template variables. Content associated with template variables may be retrieved via a DBS if necessary. After a C-spec is evaluated, the retrieved content replaces the associated template variable in the R-spec. The overall result is a renderable text (e.g., HTML).

A *chunk* is a pair, the first element of which is a C-spec and the second is an R-spec. A chunk may be associated with two sets of hyperpage identifiers. The first set is referred to as the set of *entry points* to the chunk, the second as the set of its *exit points*.

An entry point enables the hyperpage where the chunk occurs to be referenced in a request. If a chunk has many entry points they are construed as aliases of one another. An exit point enables a chunk to establish a navigable link to the hyperpage denoted by the exit point. In WWW parlance, an entry point can be thought of as a *URL* (Uniform Resource Locator) or as an *anchor* within a hyperpage, and an exit point as a link (e.g., an HREF tag in HTML).

A chunk is best thought of as a building block in the design of a hyperpage. Chunks are to hyperpages as atoms are to molecules.

A *hyperpage* is a sequence of chunks. A *hyper-library* is a store of hyperpages. For the purposes of the model, it is assumed that the implementation of the hyper-library provides the functionality of a modern database system, even a DBS on par with, and perhaps indistinguishable from, those where content is sourced. In particular, it is assumed that there exist mechanisms for scalable retrieval, associative querying, security of access, concurrency, versioning, and transaction control.

Note, there is no need to postulate that hyperpages are further structured to form a *hyperdocument*, i.e., a collection of hyperpages whose formal properties enable certain navigational operations to be performed over it. For example, with reference to an unordered collection and one of its members, one can only request another member. If, however, the collection is known to be totally ordered, then requests for the next and previous member are meaningful.

A *designer* is an author of hyperpages. In this role, the designer is required to decide on the sequence of chunks that make up the hyperpage. This, in turn, involves defining for each chunk its entry points, if any, its exit points, if any, and its C-spec and R-spec. The specification of entry and exit points binds the hyperpage being designed with other hyperpages, not necessarily local ones.

A *user* requests that hyperpages be rendered by interaction with a UIS. Thus, the user might click on a BACK button, or click on a link or type a hyperpage identifier into a GO-TO window. The observable result of processing such a request is the display of the hyperpage as specified by its designers.

4.2 The H-Region: Dynamics of the H-Region

The dynamic behaviour of a hypermedia system instantiating the H-region is now informally described. Figure 2 is a simplified data flow diagram[3] which expands the shaded oval in Figure 1.

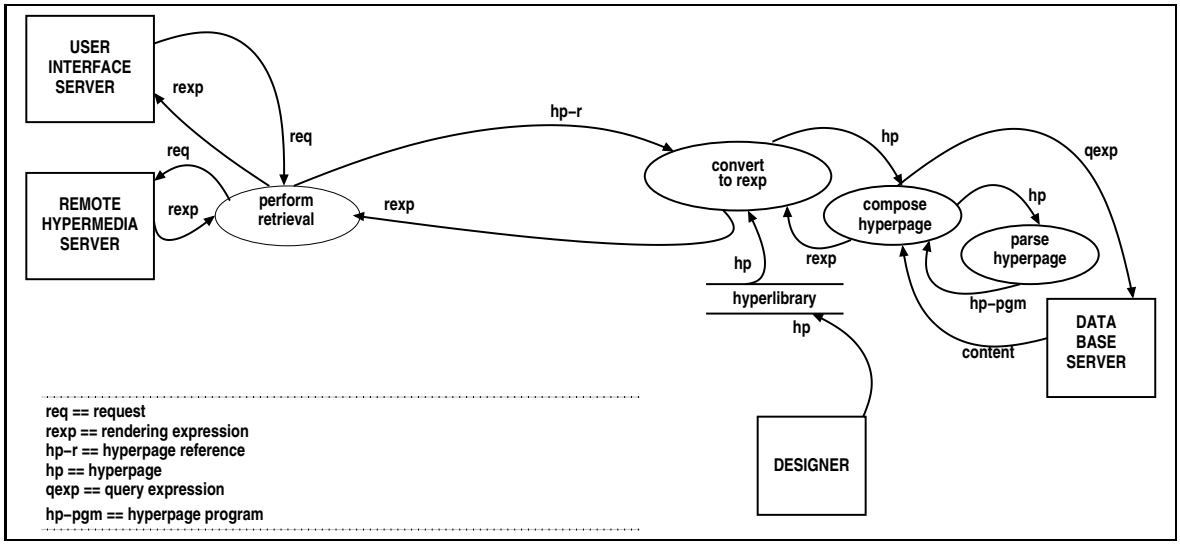

Figure 2: The H-Region

Figure 2 illustrates that designers specify hyperpages and store them in the hyper-library. When a well-formed user request comes from the UIS, the H-region resolves it into either a reference to a hyperpage that resides in a remote hypermedia system, or a reference to a hyperpage that resides in the hyper-library of the hypermedia system proper. In the latter case, the H-region fetches the hyperpage from the hyperlibrary and proceeds to parse and compose it.

4.3 The H-Region: Formal Elements of the H-Region

The following subsection defines the formal elements of the H-region. These include an EBNF grammar for hyperpage specifications and the formal semantics of hyperpage specifications given with reference to a simple abstract machine. Subsection 4.4.1 formally defines this abstract machine.

The EBNF grammar in Figure 3 defines the formal syntax of hyperpage specifications. An example of a hyperpage specification, i.e., a well-formed string in the language defined by the grammar in Figure 3, is given in Figure 4.

Some choices taken in the grammar in figure 3 are as follows. Keywords that introduce component parts are delimited by a pair of curly brackets. For example each chunk is headed by the keyword `chunk` and has its scope delimited by the pair $\{\ldots\}$.

In the example in Figure 4, string values are delimited by single-quote pairs (e.g., `'hello'`). Other values (e.g., integers, characters, etc.) are assumed to be representable as in modern programming languages. Representations of values of primitive types are elements of the terminal class VALUE. Similarly, the terminal class IDENTIFIER from which names for template-variables are drawn is assumed

[3]Notational conventions are the same as those of Figure 1.

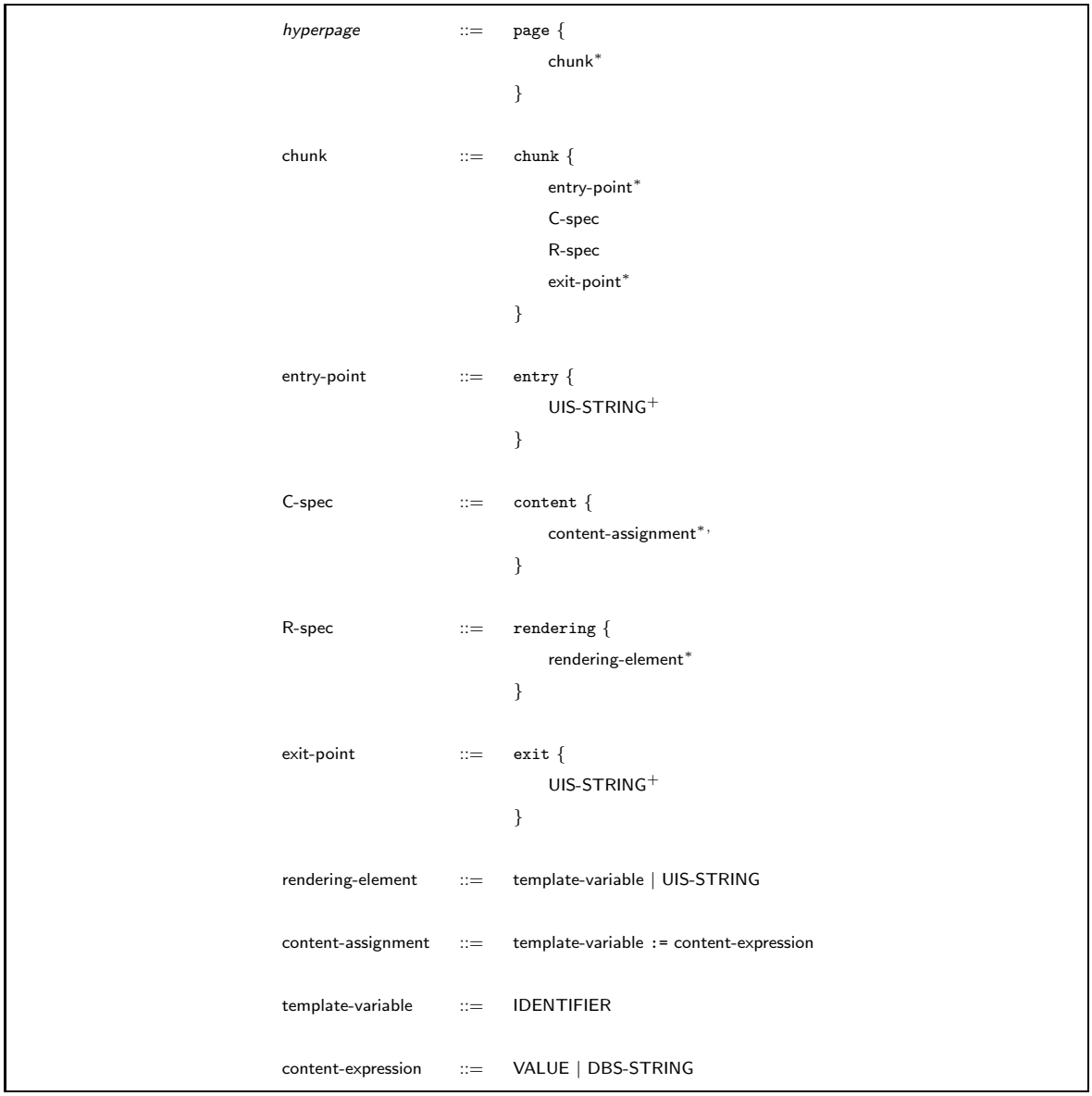

Figure 3: An EBNF Grammar of Hyperpages

to be similar to the class of variable names in modern programming languages, although here there is a preference for naming them with a single `UPPERCASE-LETTER`, possibly followed by an integer, drawn from the end of the Latin alphabet (e.g., `X`, `Y1`, etc.).

The remaining classes of non-terminals are UIS-STRING and DBS-STRING. Their elements are elements of the language which, respectively, the UIS can render and which the DBS can evaluate. In the example above, elements of both UIS-STRING and DBS-STRING are delimited by a pair of square brackets (e.g., `[<I> <B>]`, or `[SELECT * FROM welcome]`) so that they can be distinguished from the context. In the example, the assumption is that the UIS can render HTML and that the DBS can evaluate SQL queries. It may be that these strings may be more complex, e.g., a DBS-STRING may need to indicate a protocol (e.g., ODBC) and a desired server. However, these strings have no H-region specific semantics to allow appropriate languages to be incorporated as required. Therefore their denotations are not further discussed in the model.

A hyperpage, of which Figure 4 is an example, is a specification of the content to be presented to the user and the rendering of this content. The semantics of a hyperpage is an abstract program that when interpreted, retrieves the specified content and constructs the rendering text intended by the designers.

```
page{
    chunk {
      entry     { [<A NAME="welcome"></A>]
                }
      content   { X := 'hello',
                  Y1 := ', world'
                }
      rendering { [<B>] X [</B>]
                  [<B> <I>] Y1 [</I> </B>]
                }
          }
    chunk {
      content   { X := [SELECT * FROM welcome]
                }
      rendering { [<I> <B>] X [</B> </I>]
                }
      exit      { [<A HREF="http://www.gold.ac.uk/"> next </A>]
                }
          }
    }
```

Figure 4: An Example Hyperpage Specification

Note that, at this level of abstraction, a hyperpage specification need not have an identifier beyond the one (e.g., a path name) which allows it to be fetched from the hyper-library. Analogously, the constituent parts of a hyperpage specification can be referred to by their relative position in the hyperpage specification.

The hyperpage in Figure 4 has two chunks, the first of which has one entry point (viz., `<A NAME="welcome"></A>`) and no exit point, while the second has no entry point and one exit point

(viz., `<A HREF="http://www.gold.ac.uk/"> next </A>`). An entry point defines an access point internal to the hyperpage. An exit point defines a hyperlink, i.e., a point at which access may be gained to the corresponding hyperpage.

Every chunk has a possibly empty C-spec, and a possibly empty, R-spec. In the first chunk, the C-spec consists of two assignments, the first assigns the value `', hello'` to the template variable `X` and the second assigns the value `'world'` to the template variable `Y1`.

Content assignments are very similar to standard destructive assignments in imperative languages, hence the choice of the symbol `:=`. The scope of template variables is local to the chunk in which they occur.

The R-spec in the first chunk uses HTML notation to render the content of `X` in boldface and the content of `Y1` in italic boldface. The occurrence of template variables within an R-spec signals that textual replacement operations need to be carried out to obtain a rendering expression that can be returned to the UIS.

The noteworthy feature of the second chunk is in its C-spec where a query expression (`SELECT * FROM welcome`) is specified for the DBS. The results returned by the DBS are assigned to the template variable `X`.

Using simple binding and textual replacement primitives, and assuming the ability to operate client-server protocols, one can define a formal semantics of hyperpage specifications in terms of an extremely simple state-based abstract machine. The composition function in Figure 2 denotes such an abstract machine: it executes the program output by the parser of hyperpage specifications such as the one in Figure 4.

4.4 Formal Semantics of Hyperpage Specifications

In this section the formal semantics of hyperpage specifications is given with reference to a abstract machine whose instruction set is shown in Figure 5.

4.4.1 Hyperpage Abstract Machine

bind	:	IDENTIFIER' $\times$ *location*
evaluate	:	content-expression' $\times$ *location*
compose	:	rendering-expression' $\times$ *location*
replace	:	IDENTIFIER' $\times$ $location_1$ $\times$ $location_2$
append	:	$location_1$ $\times$ $location_2$ $\times$ $location_3$
return	:	*location*

Figure 5: Abstract Machine Instructions for the Hyperpage Composition

In Figure 5, IDENTIFIER', content-expression', rendering-expression' are the denotations of the corresponding syntactic constructs and $location_n$ denotes an address in the memory space of the abstract machine.

Informally, the intended meaning of these instructions is as follows: *bind* allocates memory space at *location* to the template variable whose denotation is IDENTIFIER'; *evaluate* writes onto *location* the value of content-expression'; *compose* writes rendering-expression' onto *location*; *replace* moves the contents of $location_1$ onto $location_2$, replacing each occurrence of IDENTIFIER' in $location_2$ with the contents of the memory space bound to IDENTIFIER' into $location_1$; *append* moves onto $location_3$ the concatenation of $location_1$ and $location_2$; and *return* passes back *location*, which contains the expression to be rendered, to the UIS.

Figure 6 shows the program produced by the parser in an assembly language postulated on the basis of the abstract machine instructions.

```
compose('<A NAME="Welcome"></A>')   →   L1
                          bind(X1')   →   L2
                          bind(Y1')   →   L3
                     evaluate('hello')   →   L2
                    evaluate(', world')   →   L3
compose('<B> L2* </B> <B> <I> L3* </I> </B>')   →   L4
                       replace(L2, L4)   →   L5
                       replace(L3, L5)   →   L6
                          bind(X2')   →   L7
          evaluate('SELECT * FROM welcome')   →   L7
          compose('<B> <I> L7* </I> </B>')   →   L8
                       replace(L7, L8)   →   L9
compose('<A HREF="http://www.gold.ac.uk/"> next </A>')   →   L10
                        append(L1, L6)   →   L11
                        append(L11, L9)   →   L12
                       append(L12, L10)   →   L13
                            return(L13)
```

Figure 6: An Example *program*

In Figure 6, the following notational conventions are adopted. The denotation of a rendering-expression or of a content-expression is a string and a string is enclosed in single quotes. Memory locations are denoted mnemonically by identifiers beginning with an upper-case L and followed by an integer. The mnemonic of a memory location followed by a '*' denotes the address of (i.e., a pointer to) that location. Note that, by abuse of notation, strings may contain references to memory locations. The denotation of a template variable has its scope resolved by appending to it the position of the chunk where it occurs relative to the beginning of the source text.

In summary, the abstract, open model described in this section is one answer to the question of how to characterise the emergent properties of hyermedia systems. It qualifies as one such answer insofar as it does not model any capability that might be inherited from server technologies. The contribution of this lies, in this respect, in providing one such characterisation and doing so explicitly, insofar as the emergent properties defined are unambiguously set in contrast to the properties that such systems can simply inherit from other technologies. In the model, hypermedia systems can be seen to multiply inherit functionality from user-interface and database systems, but they exhibit specific functionality: exactly that which is modelled by the H-region.

The scope for personalisation actions, taking this view, is the interaction emerging from the H-region. This emergence is completely determined by hyperpage specifications. It follows that if one wants to personalise that interaction all that is needed, and all that can be done, is to personalise hyperpage specifications. Once a hyperpage specification is personalised by a user, the functionality of the H-region delivers personalised interaction without requiring any change or redesign, let alone re-implementation.

5 P-Region: Modelling Personalisable Hypermedia-Based Interaction

The P-region comprises a group of functions that are non-disruptively added to the H-region in order to model personalisable hypermedia-based interaction. Within the P-region users can request, annotate or rewrite a hyperpage. The decisions that the designers of that hyperpage have made with regard to content, navigation and rendering of the hyperpage can therefore be overridden by users and this kind of event characterises ownership transfer.

The kinds of personalisation actions modelled are based on annotating and rewriting the hyperpage specifications. Annotation pairs a hyperpage specification with notes of interest to the user and, by doing so, presumes that versioning takes place. Such notes take one of the following forms. Firstly, a note can assign user-specific values to user-generic attributes of interest (e.g., that the level of difficulty of a given page or component part is high, or that 'Email' is a keyword of relevance to a given page). Secondly, a note can specify a rewriting action over the renderable text after it has been composed by the H-region, i.e., after content has been fetched and made ready for display (e.g., to map American into British spelling forms). Rewriting actions on hyperpage specifications allow any part of any hyperpage to be updated.

5.1 The P-Region: A Conceptual Framework

This subsection presents the new concepts introduced by the P-region that form its conceptual framework.

A *note* is a valuation of an attribute of a hyperpage or of one of its component parts. In the model a set of attributes must be specified, but the set of attribute values need not be. In this sense, it can be said that the set of attributes is user-generic and that the set of attribute values is user-specific.

Given an attribute and a hyperpage structure (i.e., an entire hyperpage or a component thereof), a note is an assignment of a value to a given attribute in the scope of that structure. More generally, a note can be viewed as a construct which provides user-specific semantics to some aspect of a hyperpage which may, for example, inform the way preferences are enforced. Notes determine the current view that designers or users have on different aspects of a hyperpage (e.g., what is its subject, what level of complexity is associated with the content, what navigational alternatives are comparable in

content). When notes are attached to hyperpage components then a function in the P-region can be parameterised by their values.

An *annotation* is a sequence of notes associated with a hyperpage.

A *personalisation request* is an editing command over hyperpage specifications that causes a modified version of the hyperpage to be versioned by the user who issued the personalisation request. A personalisation request specifies which hyperpages to personalise and what to transform them into. A personalisation request is therefore a request to override the original decisions of the designers of the hyperpage (and, of course, past expressions of preference by the user). Any design decision can, in principle, be overridden (i.e., a set of personalisation requests can rewrite a hyperpage completely).

All the concepts introduced in Section 4.1 are retained without change, except that users and designers now have more actions that they can perform. A user may also request a hyperpage to be annotated or rewritten. It is also open to the designer to generate the hyperpage with annotations.

A *hyper-library* becomes a store of hyperpages and annotations. It is assumed that a P-region implementation enforces a one-to-one mapping between annotations and hyperpages versioned to the user (or possibly group or users) that made the annotations. This, in turn, assumes a versioning capability, as well as concurrency and transaction control.

This is the conceptual framework underlying the P-region.

5.2 The P-Region: Dynamics of the P-Region

The additional dynamic behaviour of a hypermedia system instantiating the P- and H-regions is now informally defined. Figure 7 is a simplified data flow diagram[4]. P-region functions do not conflict with H-region functions and, indeed, rely on their being exactly as defined in Figure 2, except for the perform retrieval function that now handles more cases and the request data flow and the hyper-library data store which are enriched as already discussed. This match is graphically represented by super-imposition, with the functions and flows that are identical in Figure 2 and Figure 7 drawn with solid lines and named with a lighter typeface.

Figures 7 illustrates that the P-region provides two basic processes: the personalisation of hyperpages by annotation and rewriting and the enforcement over a renderable text of previously expressed preferences (in the form of notes on a hyperpage). In Figure 7, the personalise function embodies the former, while the apply preference function embodies the latter.

The personalisation process starts with a personalisation request which a user conveys to the UIS. A personalisation request specifies a scope (i.e., which hyperpages it acts upon) and the actions (which may simply be annotations) that the user wishes to effect.

The personalise function parses the request into a set of instructions that, when interpreted, retrieve from the hyper-library the hyperpages in scope, carry out the actions specified in the personalisation

[4]Notational conventions are the same as those of Figure 1.

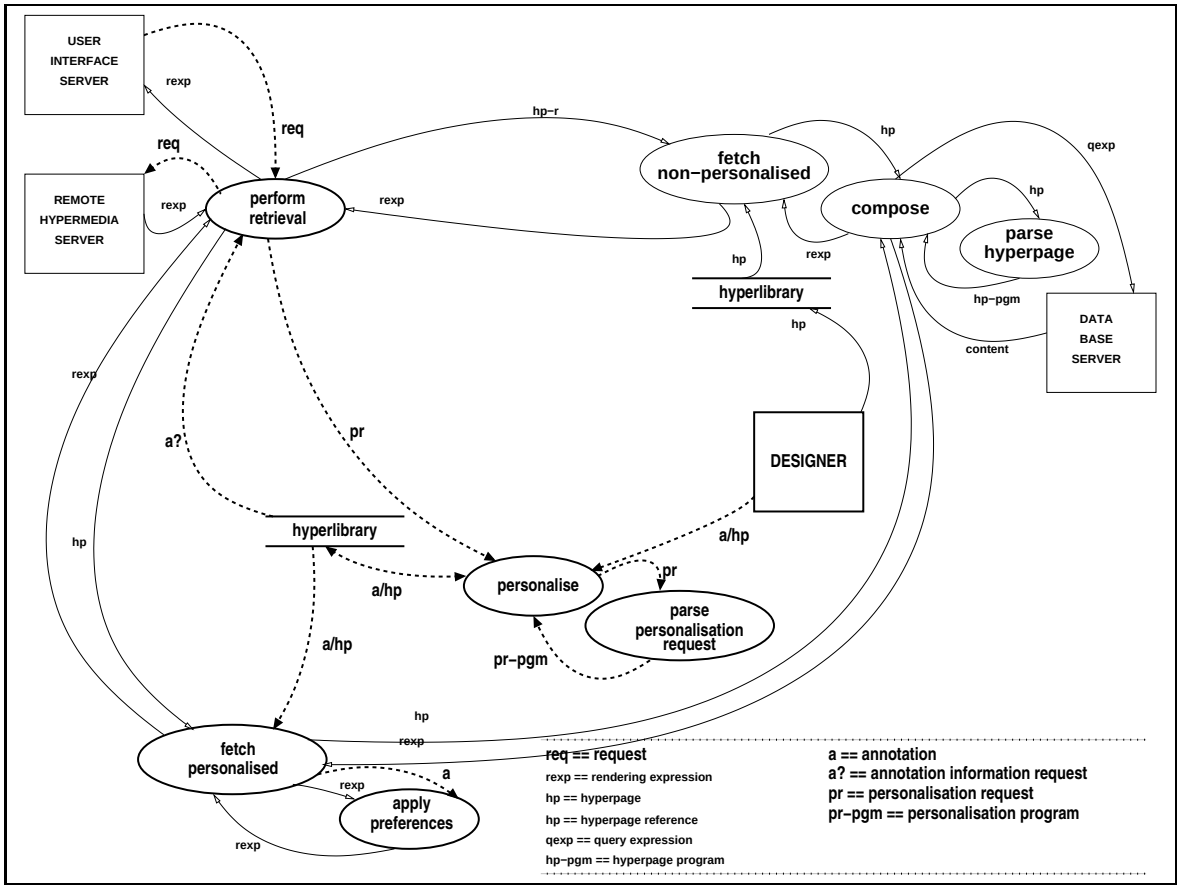

Figure 7: Superimposing the P-Region onto the H-Region

request over the retrieved hyperpages to generate their user-specific versions and write the latter into the hyper-library.

The actions specifiable in a personalisation request are the rewriting and annotating of hyperpages and hyperpage annotations. Thus, a versioned hyperpage may be an edited version, or its pairing with an annotation or both, concomitantly or not.

One possible note is to specify a post-composition rewrite. When one such specification exists for a hyperpage, the P- and H-regions interact under the control of the *perform retrieval* function. Thus, when faced with a data retrieval request, the *perform retrieval* function queries the hyper-library as to whether the required hyperpage is annotated with post-composition rewrites. If the required hyperpage is annotated with post-composition rewrites, then the P-region ensures that after the *composition* function returns the renderable text, the rewriting is effected by the *apply preference* function before a response is dispatched to the UIS.

Other notes cause changes in the behaviour that would not be manifested in the absence of the P-region. For example, given two hyperpages a note may specify that one is an alternative or is comparable to another. This allows requests for alternative and comparable hyperpages to a given one.

5.3 The P-Region: Formal Elements of the P-Region

The formal elements of the P-region are now introduced. These include formal grammars for the specification of personalisation requests and hyperpage annotations. A recursive function that defines the formal semantics of personalisation requests is then introduced. Broadly, this function traverses an abstract syntax tree representation of a personalisation request and generates the corresponding semantic representation as a sequence of assignments and relational algebraic expressions.

5.3.1 A Formal Grammar for the Specification of Hyperpage Annotations

Annotations (and notes) are formally defined in Figure 8. An annotation is a nonempty sequence of notes. A note is either an attribute assignment or a (post-composition) rewriting specification, where the rewriting can be unconditional or conditional on the state of the environment.

Figure 8, the non-terminals *from-regular-expression* and *to-regular-expression* are regular expressions. If a note is an attribute assignment, then it must specify the scope of the assignment. This scope is either the entire page, or one of its component parts. In the latter case, the exact component part is specified by a data structure that completely determines the component part in the page using relative displacements.

The set of attributes to which values can be assigned is fixed for all users, since only then can their semantics be defined and made use of in, e.g., data requests. The set of values which attributes can take might be fixed for all users, but it need not. A fixed set of attributes is modelled for hyperpages, primarily to illustrate their use, however, there is nothing to preclude extensions, or the redefining of this set to an appropriate set of attributes in particular contexts.

In general terms, fixing such domains has the effect of making more explicit knowledge available within the hypermedia system, thereby expanding the range of informed actions that can be taken.

The intended meanings of each of the attributes chosen are: `description` lets users and designers provide an abstract or summary; `keyword` lets users and designers tag content and thereby clarify the information contained along different categories; `level` lets users and designers attach a measure, e.g., of complexity, to the information; `see-as-well` and `see-instead` allow users and designers to define other hyperpages to be, respectively, comparable and alternatives to the hyperpage being annotated; `wherefrom` and `whereto` allow users and designers to provide look-back and look-ahead information so that navigational decisions can be more informed; and `status` lets users and designers record information about the state of the page (e.g., expiry date on time-bound information).

An example of an annotation, i.e., a well-formed string in the language defined by the grammar in Figure 8 is given in Figure 9.

```
annotation          ::=  annotation {
                             note*;
                         }

note                ::=  scope attribute-assignment
                       | rewrite
                       | QUERY-ON-ENVIRONMENT rewrite

scope               ::=  page :
                       | [ page-part ] :

page-part           ::=  relevant-chunk
                       | relevant-chunk ( relevant-single-part )
                       | relevant-chunk ( relevant-multi-part )

relevant-chunk      ::=  shift , chunk

relevant-single-part   |  C-spec
                       |  R-spec

relevant-multi-part ::=  shift , entry-point
                    ::=  shift , exit-point

shift               ::=  any
                       | signed-integer

signed-integer      ::=  INTEGER
                       | sign INTEGER

sign                ::=  +
                       | -

attribute-assignment::=  attribute := attribute_value

attribute           ::=  description | keyword | level
                       | see-as-well | see-instead
                       | wherefrom | whereto
                       | status

attribute_value     ::=  VALUE
                       | entry-point*
                       | exit-point*

rewrite             ::=  scope from -> to

from                ::=  REGULAR EXPRESSION

to                  ::=  REGULAR EXPRESSION
```

Figure 8: An EBNF Grammar of Hyperpage Annotations

```
annotation{
 page                          : level := 1;
 page                          : see-as-well := [http://www.nasa.gov/]
 [any, chunk                ]: keyword := ‘dynamic content’;
 [1, chunk                  ]: keyword := ‘introduction’;
 [2, chunk (C-spec)         ]: keyword := ‘clusters’, keyword := ‘galaxies’;
 [2, chunk (1, exit-point)  ]: level := 1;
 [2, chunk (any, exit-point)]: level := 5;
 [2, chunk (R-spec)         ]: keyword :=‘JPEG’, keyword :=‘greyscale’;
 [3, chunk (C-spec)         ]: keyword := ‘planets’, level := 5;
 [4, chunk (C-spec)         ]: level := 1;
 page                          : ‘‘center’’ -> ‘‘centre’’
 }
```

Figure 9: An Example of a Hyperpage Annotation

5.3.2 A Formal Grammar for the Specification of Personalisation Requests

Personalisation requests allow a user to generate annotations, to update annotation (e.g., those provided at source by the designer), and to update (by versioning) the hyperpage specifications themselves. The EBNF grammar in Figure 10 defines the formal syntax of a personalisation request. Note that this grammar extends (i.e., has shared non-terminals with) the grammars given in Figures 3 and 8.

There are two parts to a personalisation request: the specification of its scope, the semantics of which is the selection of a subset of the hyperpage specifications in the hyper-library (possibly with their accompanying notes and possibly taking into account user-specific versions already stored); and the action to be performed over the hyperpage specifications in the scope.

The specification of scope is indicated by the token `select-page-if`. The set of hyperpage specifications which will be placed in scope contains every hyperpage specification that satisfies the associated selection condition. In the model, selection conditions are built up from one primitive that specifies a regular expression and is true if, and only if, the regular expression occurs in the specified page, page-part or note occurrence. Primitives can be combined using Boolean connectives to form complex selection conditions. There is also a vacuously true selection condition that caters for the need to select an entire set.

The possible actions on hyperpage annotations are: updates on notes (`insert` or `delete`), with which annotations can be built and maintained one by one; projections on sets of notes (`drop-if` or `retain-if`), with which one can filter a set of notes using a selection condition.

The persistence of notes can be made conditional on the environment, in that they may be stipulated to hold only if a condition on the environment holds (or does not hold). Furthermore, as a Note is always associated with at least one hyperpage and the identification of a set of hyperpages is determined by the selection condition, the EBNF grammar need not make a distinction between the parameters associated with the update operators `insert` and `delete`.

Possible actions on hyperpage specifications are: actions on hyperpage structures, which resemble

personalisation-request	::=	action-scope action-list
action-scope	::=	`select-page-if` selection-condition
selection-condition	::=	atomic-selection-condition
	|	`not` selection-condition
	|	selection-condition `and` selection-condition
	|	selection-condition `or` selection-condition
atomic-selection-condition	::=	`true`
	|	atomic-selection-condition-on-page
	|	atomic-selection-condition-on-page-part
	|	atomic-selection-condition-on-note
atomic-selection-condition-on-page	::=	`page` containment-expression
atomic-selection-condition-on-page-part	::=	page-part containment-expression
atomic-selection-condition-on-note	::=	relevant-note containment-expression
relevant-note	::=	shift `,` `note`
containment-expression	::=	`contains` REGULAR-EXPRESSION
action-list	::=	action*
action	::=	`ann-then-do` { annotation-update }
	|	`hp-then-do` { hp-update }
annotation-update	::=	update-operator condition note
	|	update-operator note
	|	note-projection
	|	annotation
update-operator	::=	`insert`
	|	`delete`
condition	::=	`if-not` QUERY-ON-ENVIRONMENT
	|	`if` QUERY-ON-ENVIRONMENT
note-projection	::=	projection-operator selection-condition-on-note
projection-operator	::=	`drop-if`
	|	`retain-if`
selection-condition-on-note	::=	atomic-selection-condition-on-note
	|	`not` selection-condition-on-note
	|	selection-condition-on-note `and` selection-condition-on-note
	|	selection-condition-on-note `or` selection-condition-on-note
hp-update	::=	hp-structure-update
	|	hp-terminal-rewrite
hp-structure-update	::=	`insert` page-part hp-constr
	|	`delete` page-part
	|	projection-operator selection-condition-on-hp-constr
hp-constr	::=	hyperpage

those on notes in that they can be updates or projections based on selection conditions; actions on terminal strings, (i.e., post-composition rewrites).

All actions on hyperpage specifications are destructive and their effect is manifested the next time the H-region composition function is invoked to parse and interpret them. A few examples of personalisation requests that a user might issue are given with comments on their effect.

Example 1

```
select-page-if
  page contains ‘‘2002’’
hp-then-do {
  delete [2, chunk]
  }
```

The personalisation request above applies to all hyperpages that contain the string “2002”. The effect on each selected hyperpage is the deletion of its second chunk.

Example 2

```
select-page-if
  true
hp-then-do {
  retain-if [any, chunk] contains ‘‘2002’’
  }
```

The personalisation request above applies to all hyperpages. The effect on each selected hyperpage is the deletion of any chunk in which the string “2002” does not occur.

5.4 Formal Semantics of Personalisation Requests

This subsection introduces the approach taken to formally defining the semantics of personalisation requests. The approach taken is now briefly described.

A parser has been developed to describe the analysis of the structure and meaning of the language for personalisation requests (See Section 6). The parser consists of a lexical analyser, a syntactic analyser and a semantic analyser. The lexical analyser groups the individual characters of a personalisation request specification into tokens. The syntactic analyser parses the phrase structure generated by the lexical analyser to determine whether the stream of tokens forms a valid personalisation request. If a personalisation request is validated, then the lexical analyser outputs its abstract syntax tree. Semantic analysis is the assignment of meaning. This is achieved by mapping the abstract syntax

tree of a personalisation request into a target language, whose meaning it is assumed the audience (or machine) know.

The meaning of personalisation requests can be understood as requests to manipulate hyperpage specifications and hyperpage annotations. To express the meaning of a personalisation request, it is advantageous to model hyperpage specifications and annotations using a data structure that is well understood and for which there exists a well understood language for manipulating data modelled using such structures. For the purpose of defining the meaning of a personalisation request, the hyper-library has been modelled using the relational data model.

In [24] a recursive function is defined that traverses an abstract syntax tree, represented as a *term* generating the corresponding semantic representation of a personalisation request as a mapping into relational algebra with assignments, examples of which are provided in Subsection 5.4.2.

5.4.1 Hyperpage and Hyperpage Annotation Relational Schemas

This subsection defines the relational schemas used to model hyperpages and hyperpage annotations. Hyperpages and hyperpage annotations are input into a *hyper-library* as text files.

The transformation of hyperpage and hyperpage annotations text files into relation schemas is achieved by retrieving in sequence each hyperpage (or hyperpage annotation) and then analysing its structure. This analysis involves reading each hyperpage from top to bottom and then transforming each part of its structure (i.e., entry-points, C-specs, R-specs and exit-points) into appropriate relations. When reconstructed as relations each part of the structure of a hyperpage (or hyperpage annotation) is given an extra attribute value as a unique identifier. This identifier acts as the primary key for that relation. Identifiers are created in the same sequence as the transformation of hyperpages (or hyperpage annotations). As a side effect these identifiers also hold the ordering in which page parts were transformed and stored into their relations. Note that the definitions of the operations to project, select and join relations are restricted so that they may not affect the ordering of the tuples in the relations used as their parameters.

Hyper-library relation schemas are now defined. Relation names are denoted by a sequence of upper-case SANS-SERIF letters, the corresponding name in lower-case *sans-serif* letters denotes an identifier of an entity modelled by the relation. The primary key of each relation is underlined. Some remarks are provided to increase readability.

PAGE(*page, chunk, shift*)

The relation PAGE has the following attributes: page uniquely identifies a page, chunk identifies a chunk within that page, shift represents the position of that chunk within the page. chunk is a foreign key to CHUNK.

CHUNK(*chunk, entrypointset, c-spec, r-spec, exitpointset*)

In the relation CHUNK the entrypointset attribute identifies the set of entry points for that chunk, c-spec identifies the C-spec for the chunk, r-spec identifies the R-spec for the chunk, exitpointset identifies the set of exit points for the chunk.

ENTRYPOINTSET(*entrypointset, shift, entrypoint*)

In the relation ENTRYPOINTSET the entrypoint attribute uniquely identifies an entry point for that entry point set and is a foreign key to ENTRYPOINT.

ENTRYPOINT(*entrypoint, e_string*)

In the relation ENTRYPOINT the e_string attribute stores an entry point.

C-SPEC(*c-spec,shift, c-element*)

In the relation C-SPEC the c-element attribute uniquely identifies an element within that C-spec and is a foreign key to C-ELEMENT.

C-ELEMENT(*c-element, variable,c_string*)

In the relation C-ELEMENT the variable attribute stores a template variable and the c_string stores a content expression. Recall that a content expression may be either a DBS string or a value.

R-SPEC(*r-spec, shift, r-element*)

In the relation R-SPEC the r-element attribute uniquely identifies an element within that R-spec and is a foreign key to R-ELEMENT.

R-ELEMENT(*r-element, r_string*)

In the relation R-ELEMENT the r_string attribute stores a rendering element. Recall that a rendering element may be either a template variable or a UIS string.

EXITPOINTSET(*exitpointset, shift, exitpoint*)

In the relation EXITPOINTSET the exitpoint attribute uniquely identifies an exit point for that exit point set and is a foreign key to EXITPOINT.

EXITPOINT(*exitpoint, x_string*)

In the relation EXITPOINT the x_string attribute stores an exit point.

NOTE(*note,page, chunk, n_type, scope, shift, lhs, n_string, condition*)

The relation NOTE has the following attributes: note uniquely identifies a note, page identifies the page that note has been associated with and is a foreign key to the relation PAGE, chunk identifies a chunk within that page and is a foreign key to the CHUNK, n_type represents the type of note (i.e.,

attribute-assignment, or rewrite), scope represents the scope of a note which may be a page or a page part and shift represents the position of that page part, lhs stores a note attribute or, in the case of a rewrite note, the regular expression which is the subject of the rewrite, n_string stores the note itself or, in the case of a rewrite note the replacement string, condition stores a query on the state of the environment.

ANNOTATION(*annotation*, *page*)

The relation ANNOTATION has the following attributes: annotation uniquely identifies an annotation, page identifies the page that an annotation has been associated with and is a foreign key to the relation PAGE.

5.4.2 Examples of Personalisation Request Programs

Examples are now given to illistrate that following the successful parsing of the personalisation requests shown in subsection 5.3.2 their semantics may be represented as a mapping into relational algebra with assignments.

In the examples that follow, " foreach $P \in S$ do E endfor" is written to mean that E is performed for each P in the finite set S. " for $i := 1$ to n do E endfor" is written to mean that E is performed with $i = 1$, then with $i = 2$ and so on, up to $i = n$. "$X := Y$" is written to denote the binding of a value (or the value of an expression) Y to a denotation, X. Given two strings s and s', $s \lhd s'$ if s occurs in (e.g., is a substring of) s'.

To generate new primary keys for tuples a function **New** is assumed that takes as a parameter the name of a relation and returns a new primary key for that relation.

To generate the next value in the order defined over an attribute domain a function **Next** is assumed that takes as parameters an attribute name and a relation name and returns the next value for that attribute domain. The legitimate use of the function **Next** is therefore restricted to ordered domains (i.e., numbers).

To insert a set of tuple representations of a hyperpage, hyperpage annotation or hyperpage part into a relation, a function **Insert** is assumed. The **Insert** function allows the union of a relation R with a set t of tuples. The insertion of a set of tuples t into R yields a new relation instance R'. '$R' = \mathbf{Insert}(R, t)$' is written to denote the insertion of tuple t into relation R.

To delete a set of tuple representations of a hyperpage, hyperpage annotation or hyperpage part from a relation a function **Delete** is assumed. The **Delete** function allows the difference between a relation R and a set t of tuples to be obtained. The deletion of the tuple t from the relation R will yield a new relation instance R'. '$R' = \mathbf{Delete}(R, t)$' is written to denote the deletion of tuple t from relation R.

```
select-page-if
  page contains ‘‘2002’’
hp-then-do {
  delete [2, chunk]
  }
```

‘foreach $P \in$ ‘$\pi_{page}(\sigma_{\text{“2002”} \lhd page}\text{PAGE})$’ do

‘SC:=(CHUNK $\star_{chunk}$ $(\pi_{chunk}(\sigma_{page=\text{P} \wedge shift=2}\text{PAGE}))$);

$\mathbf{Delete}$(CHUNK, SC)’

endfor’

Figure 11: Personalisation Program Example 1

```
select-page-if
  true
hp-then-do {
  retain-if [any, chunk] contains ‘‘2002’’
  }
```

‘foreach $P \in$‘$\pi_{page}(\sigma_{*}\text{PAGE})$’ do

‘if ‘$\text{“2002”} \lhd_{\text{chunk}} \text{CHUNK}(\text{CHUNK} \star_{chunk} (\pi_{chunk}(\sigma_{page=\text{P}}\text{PAGE})))$’ = false then

SelectedPage :=$(\sigma_{page=\text{P}}\text{PAGE})$;

$\mathbf{Delete}$(PAGE,SelectedPage)

else

NOP

endif’

endfor’

Figure 12: Personalisation Program Example 2

6 PAS: A Personalisable Hypermedia System

This section describes the development of a personalisable hypermedia system, *PAS*, that embodies the main concepts underlying the model described in this paper.

The aim of PAS is to allow for a principled, systematic empirical study to be carried out, in the future, into the effects of personalisation actions. PAS consists of two major components: an instantiation of the H-region as a WWW-based application for realising dynamically generated hyperdocuments; and an instantiation of the P-region as a parser for personalisation requests.

The semantics of personalisation requests are implemented by a semantic analyser whose input is an abstract syntax tree representation of a personalisation request output by the syntactic analysis phase of the parsing process. The role of the semantic analyser is to generate for a given personalisation request its corresponding representation as a sequence of Standard Query Language (SQL) expressions[5]. Data structures for the storage and management of hyperpages and hyperpage annotations have been implemented using the relational data model and these form the subject of personalisation requests issued as SQL expressions.

The PAS adopts the strategy of separating out into different components the need to support: the functionality afforded by the formal specification language for personalisation requests (back-end component) and; a dynamic style of WWW-based hyperlink interaction (front-end component).

6.1 Architecture of PAS

The architecture of PAS is partitioned into a *back-end component* and a *front-end component*. Given a text file containing a set of personalisation requests, the back-end component performs: lexical analysis; syntactic analysis; and semantic analysis (the generation of SQL statement representations of valid personalisation requests) on this set of personalisation requests. The *front-end component* is responsible for executing SQL statement representations of valid personalisation requests over a database containing hyperpages and hyperpage annotations. The front-end is also responsible for: dynamically retrieving content, mark-ups and hyperpage annotations from the database; dynamically composing that content into a rendering expression; applying post composition rewrites to rendering expressions; and returning rendering expressions to a WWW browser (UIS). As such, the front-end is the active component of PAS and the back-end is the stable component. Figure 13 depicts the architecture of PAS.

6.2 The Back-End Component of PAS

The following Subsection describes the implementation of the processes of lexical analysis, syntactic analysis and semantic analysis of personalisation requests. The parser has been implemented using

[5]SQL is a classical approach to implementing relational algebraic expressions, over a database composed of relations defined using the relational data model.

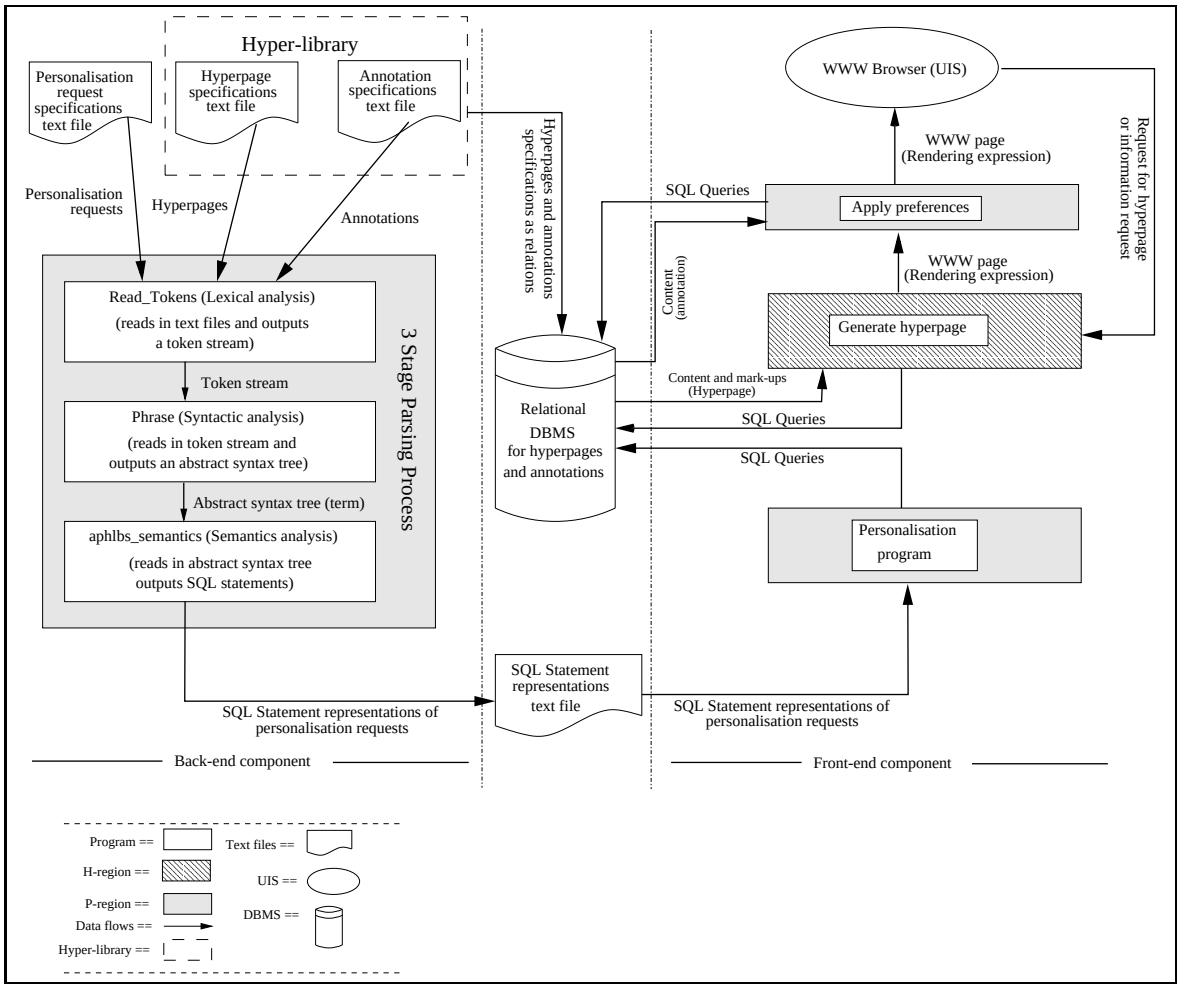

Figure 13: Architecture of PAS

Prolog.

The process of lexical analysis has been implemented as a function, *read_tokens*. The function *read_tokens* takes as input a text file containing a personalisation request and produces as output a comma separated stream of tokens. Figures 14 and 15 show examples of the use of the function *read_tokens*.

```
Given a text file containing the personalisation request

select_page_if
    page contains '2002'
hp_then_do{
    delete[2,chunk]
    }
.
the function read_tokens yields the following token stream

 select_page_if,page,contains,2002,hp_then_do,
'{',delete,'[',2,',',chunk,']','}','.'
```

Figure 14: Personalisation Request and Generated Token Stream: Example 1

```
Given a text file containing the personalisation request

select_page_if
    true
hp_then_do{
    retain_if [2, chunk] contains '2002'
    }
.

the function read_tokens yields the following token stream

select_page_if,true,hp_then_do,'
{',retain_if,'[',2,',',chunk,']',contains,2002,'}','.'
```

Figure 15: Personalisation Request and Generated Token Stream: Example 2

The role of the syntax analyser is to determine if the stream of tokens output of the lexical analyser form a valid sentence in the language's grammar. The process of syntax analysis has been implemented as a function, *Phrase*. The function *Phrase* takes as input a token stream and produces as output an abstract syntax tree. For convenience, an abstract syntax tree may be represented as a term. Figures 16 and 17 show a term representations of the personalisation requests shown in figures 14 and 15.

The remaining step is to assign meaning to the source language program (personalisation request). This is achieved by mapping the term representation of abstract syntax into a formalism (target language) of which the target audience (or machine) knows the meaning. For the purpose of implementing the semantics of personalisation requests, the relational algebraic statements described in section 5.4 are translated into SQL statements. These statements are then stored in a text file so that subsequently, they may be referenced by the personalisation program that executes them *over* the

```
personalisation_request('select_page_if',
   condition(atomic_condition(
      atomic_condition_on_page('page',
         containment_expression('contains', 2002)))),
   action_list(action('hp_then_do',
      hp_update(hp_structure_update('delete',
         page_part(relevant_chunk(shift(signed_integer(+, 2)), 'chunk')))))))
```

Figure 16: Term Representation of Abstract Syntax Tree: Example 1

```
personalisation_request('select_page_if',
   condition(atomic_condition('true')),
   action_list(action('hp_then_do',
   hp_update(hp_structure_update(projection_operation('retain_if'),
   condition_on_hp_construct(atomic_condition_on_hp_construct(
   page_part(relevant_chunk(shift(signed_integer(+, 2)), 'chunk')),
   containment_expression('contains', 2002)))))))
```

Figure 17: Term Representation of Abstract Syntax Tree: Example 2

database of hyperpages and hyperpage annotations to realise a personalisation request.

The process of semantic analysis has been implemented as a function, *aphlbs_semantics*. The function *aphlbs_semantics* takes as input a term representation of an abstract syntax tree for a personalisation request and returns a sequence of SQL statements that are stored in a text file. Figures 16 and 17 show example term representations. The corresponding SQL statement representations are detailed in Figures 18 and 19. Figure 20 depicts the back-end component of PAS as a fragment of a Prolog [11] program code.

In the following subsection, the processes of executing SQL statement representations of personalisation requests and the dynamic generation of personalised hyperpages is described. These processes are the responsibility of the front-end component of PAS.

6.3 The Front-End Component of PAS

The front-end component is comprised of: a personalisation program; a hyperpage generator; and a program to rewrite terminal strings in rendering expressions. All front-end processes have been implemented using *Cold Fusion* (CF) [14].

A CF application is a collection of hyperpage specifications (programs) that are authored in CFML and HTML. CFML provides the functionality to control the behaviour of applications, integrate WWW server technologies and dynamically generate the content and presentation of WWW pages before they are returned to a WWW browser. Broadly, when a hyperpage in a CF application is requested by a WWW browser, it is pre-processed by the CF Application Server. Based on the CFML in the hyperpage, the Application Server executes the application logic, interacts with other server technologies and then dynamically generates an HTML rendering expression that is returned to the WWW browser.

```
INSERT PAGE.page, PAGE.chunk, PAGE.shift INTO SELECTEDPAGES
WHERE PAGE.page IN (SELECT PAGE.page FROM PAGE)
AND (PAGE.page IN                              % select page if

(SELECT PAGE.page FROM PAGE                    % page contains 2002
WHERE PAGE.chunk IN
(SELECT CHUNK.chunk FROM CHUNK
WHERE CHUNK.entrypointset IN
(SELECT ENTRYPOINTSET.entrypointset FROM  ENTRYPOINTSET
WHERE ENTRYPOINTSET.entrypoint IN
(SELECT ENTRYPOINT.entrypoint FROM ENTRYPOINT
WHERE ENTRYPOINT.e_string LIKE ‘‘2002’’))))

OR PAGE.page IN
(SELECT PAGE.page FROM PAGE
WHERE PAGE.chunk IN
(SELECT CHUNK.chunk FROM CHUNK
WHERE CHUNK.cspec IN
(SELECT CSPEC.cspec FROM CSPEC
WHERE CSPEC.c-element IN
(SELECT C-ELEMENT.c-element FROM C-ELEMENT
WHERE C-ELEMENT.c_string LIKE ‘‘2002’’))))

OR PAGE.page IN
(SELECT PAGE.page FROM PAGE
WHERE PAGE.chunk IN
(SELECT CHUNK.chunk FROM CHUNK
WHERE CHUNK.rspec IN
(SELECT RSPEC.rspec FROM RSPEC
WHERE RSPEC.r-element IN
(SELECT R-ELEMENT.r-element FROM R-ELEMENT
WHERE R-ELEMENT.r_string LIKE ‘‘2002’’))))

OR PAGE.page IN
(SELECT PAGE.page FROM PAGE
WHERE PAGE.chunk IN
(SELECT CHUNK.chunk FROM CHUNK
WHERE CHUNK.exitpointset IN
(SELECT EXITPOINTSET.entrypointset FROM  EXITPOINTSET
WHERE EXITPOINTSET.exitpoint IN
(SELECT EXITPOINT.exitpoint FROM EXITPOINT
WHERE EXITPOINT.x_string LIKE ‘‘2002’’))))

DELETE * FROM PAGE                  % delete the second chunk of pages selected
WHERE PAGE.page IN SELECTEDPAGES
AND (PAGE.shift = 2)
```

Figure 18: SQL Representation of Example 1

```
INSERT PAGE.page, PAGE.chunk, PAGE.shift INTO SELECTEDPAGES
WHERE PAGE.page IN (SELECT PAGE.page FROM PAGE)
AND (PAGE.page IN                              % select page if
(SELECT PAGE.page FROM PAGE)                   % true vacuously true

DELETE * FROM PAGE
WHERE SELECTEDPAGES.page IN                    % retain if
(SELECT PAGE.page FROM PAGE                    % second chunk contains 2002
WHERE PAGE.chunk NOT IN
(SELECT CHUNK.chunk FROM CHUNK
WHERE CHUNK.entrypointset IN
(SELECT ENTRYPOINTSET.entrypointset FROM  ENTRYPOINTSET
WHERE ENTRYPOINTSET.entrypoint IN
(SELECT ENTRYPOINT.entrypoint FROM ENTRYPOINT
WHERE ENTRYPOINT.e_string LIKE ‘‘2002’’))))
AND (PAGE.shift = 2)

OR PAGE.page IN
(SELECT PAGE.page FROM PAGE
WHERE PAGE.chunk NOT IN
(SELECT CHUNK.chunk FROM CHUNK
WHERE CHUNK.cspec IN
(SELECT CSPEC.cspec FROM CSPEC
WHERE CSPEC.c-element IN
(SELECT C-ELEMENT.c-element FROM C-ELEMENT
WHERE C-ELEMENT.c_string LIKE ‘‘2002’’))))
AND (PAGE.shift = 2)

OR PAGE.page IN
(SELECT PAGE.page FROM PAGE
WHERE PAGE.chunk NOT IN
(SELECT CHUNK.chunk FROM CHUNK
WHERE CHUNK.rspec IN
(SELECT RSPEC.rspec FROM RSPEC
WHERE RSPEC.r-element IN
(SELECT R-ELEMENT.r-element FROM R-ELEMENT
WHERE R-ELEMENT.r_string LIKE ‘‘2002’’))))
AND (PAGE.shift = 2)

OR PAGE.page IN
(SELECT PAGE.page FROM PAGE
WHERE PAGE.chunk NOT IN
(SELECT CHUNK.chunk FROM CHUNK
WHERE CHUNK.exitpointset IN
(SELECT EXITPOINTSET.entrypointset FROM  EXITPOINTSET
WHERE EXITPOINTSET.exitpoint IN
(SELECT EXITPOINT.exitpoint FROM EXITPOINT
WHERE EXITPOINT.x_string LIKE ‘‘2002’’))))
AND (PAGE.shift = 2)
```

Figure 19: SQL Representation of Example 2

```
aphlbs_read(What, Mode, Answer) :-
    read_tokens(Tokens, _Vars),
    (Tokens = []
     -> Answer = end_of_file
     ;  (( Phrase =.. [What, A, Tokens, []], call(Phrase))
        -> Answer = A,
           nl, write(What),
           write(' parsing succeeded'), nl,
           aphlbs_feedback(Answer, Mode, _Reply),
           aphlbs_semantics(Answer, SQL)
        ;  nl, write(What),
           write(' parsing failed with this token sequence:'),
   nl, nl, write(Tokens), nl, fail
        )
    ).
```

Figure 20: The Back-End Component of PAS

Figure 21 depicts the front-end component of PAS as a collection of CF programs that interact with a CF Application Server.

The *personalisation_program* has been implemented as a CF program. This program retrieves SQL statements stored in a text file and executes these over the relations for hyperpages and hyperpage annotations. Execution is realised via the CF Application Server which provides the functionality to integrate with: local and remote databases; file systems; and files held in directories.

The generate hyperpage program has been implemented as a CF program. On receiving a request for a hyperpage, this program dynamically generates a hyperpage (rendering expression) by executing a sequence of SQL statements. These statements are used to retrieve sequentially the component parts of a hyperpage (i.e., entry points, C-specs, R-specs and exit points) from a relational database comprised of relations over the relational schemas for hyperpages and hyperpages' annotations, as described in subsection 5.4.1. The generate hyperpage program also contains specifications of how retrieved content should be presented.

If a hyperpage (or one of its component parts) is associated with an annotation, then an annotation link[6] is generated to indicate that a particular component part has a note associated with it. To present annotations to the user, a program, *generate annotation*, has been developed. This program retrieves the corresponding notes for a given component part of a hyperpage.

A program to *apply preferences* has been implemented. This program, given a hyperpage in the form of a rendering expression, retrieves for that hyperpage its annotation. If the annotation retrieved specifies one or more post-composition rewrite notes (i.e., has one or more notes, each of which specifies that some terminal symbol be replaced by another terminal symbol), then the apply preferences program executes a series of CFML statements that effect a regular expression search and replace over

[6] An annotation link is a special class of link represented by the image of a pin. When a mouse is moved over the pin, the link is activated and a CF program is run, the purpose of which is to display a subset of notes for a given hyperpage component part.

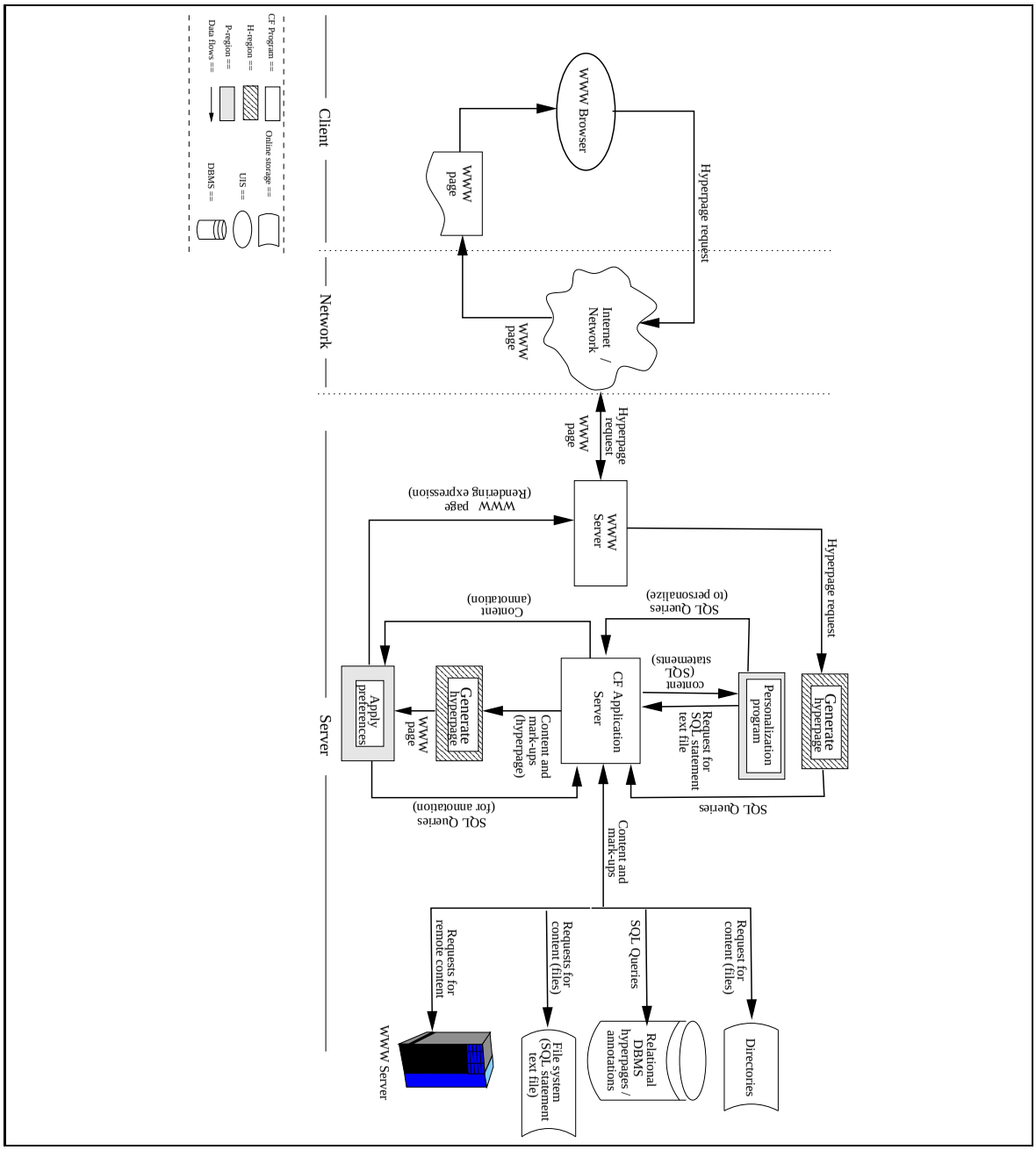

Figure 21: The Front-End Component of PAS

the rendering expression. These statements use as parameters the *from* and *to* regular expressions specified by each post-composition rewrite note. When all specified post-composition rewrites have been executed, the rewritten rendering expression is returned to the UIS that issued the original request for the hyperpage.

6.4 Future Work on PAS

For the purposes of bringing clarity and a greater understanding of the model, we are currently re-specifying the model using a document-centric semi-structured approach supported by a peer-to-peer, service-oriented distributed architecture. This work will allow for an implementation (based on XML and related technologies, and on web services and related technologies) that is modern and can be readily understood by a broader range of systems designers.

7 Related Work

This section compares and contrasts the proposed model with related work on modelling and implementing personalisation in hypermedia systems.

Little research has been conducted into formalising personalisation in hypermedia systems themselves. Work has concentrated on providing personalisation features via data querying mechanisms (e.g., [15]). Although these contributions show how personalisation may be experienced, it can be argued that they merely exploit DBMS functionality rather than accounting for how hyperlink-based interaction can itself be tailored in a principled manner.

Since we have formalised (in [24]) a complete space of possibilities for personalisation actions, the adoption of our model enables a system to support all the personalisation actions described in [4] and many of those in [5]. Implemented systems such as Adaptive HyperMan [21], ELM-ART [8], Hypadapter [16] and AHA [13, 12] can all be represented using our model.

In its aims, our model subsumes the work on the adaptive hypermedia reference model AHAM [12, 13]. However, the explicit aim of the AHAM is to represent implemented techniques hence serving as a primary reference for comparative studies. In contrast to AHAM, the work reported here aims to induce a set of personalisation actions from a formal definition of a core of hyperlink functionality. In this way, we provide a methodological approach to the area that goes beyond the effort embodied in the AHAM.

The work presented herein concurs with that in [20] in which the Munich reference model (MRM) for adaptive hypermedia applications is introduced. However, both the AHAM and MRM models are based upon the notion that hypermedia applications are closed in nature and can be described using the framework of the Dexter Model [23].

The model for the personalisation of hyperdocuments introduced by this paper can be contrasted with the MRM proposed in [20] in that both seek to model personalisable hyperlink interaction. However,

whilst the MRM is a reference model, that characterises the architecture required for hyperlink-based personalisation, the model proposed here characterises a set of formally defined personalisation actions. Furthermore, whilst the model described in the MRM views hyperlink-based systems as comprising user interface and database components, our work views these components as beyond the scope of hyperlink-based personalisation.

In this paper, it is shown how the formal characterisation of hyperpage specifications induces a set of personalisation actions that allows a user to override all the design decisions that a hyperpage specification embodies. It is shown that the P-region provides the functionality needed for personalisation actions without disrupting and, in fact, relying on the functionality of the H-region remaining intact and unaltered.

Specifically the model has shown how all personalisation techniques described in [4] may be formally modelled and therefore understood. For example, the technique of link annotation [9, 10] (the representation of availability of links and the incorporation of visual clues, indicating their status or purpose) can be clearly represented in the model. The notion of not only controlling the visualisation of links but also the ability to manipulate them (i.e., allowing them to be specified according to the appropriateness of the situation [10]) can also be represented.

Classical personalisation techniques, such as the personalisation of content implemented in [2, 17], can be clearly represented within the model. The action of selective content (i.e., hiding parts of information about a particular concept which the user has expressed a wish not to see) can also be modelled. Hyperpage annotations may be used to model comparative and variable content as implementated in [18, 16].

Using the P-region fuctions it also possible to model the incorporation of additional explanations for particular concepts found on a hyperpage. Such functionality is provided by systems such as KN-AHS [19] and Anatom-Tutor [1]. Finally, although not explicitly addressed in the model presented, approaches to pre-requisite content selection (supplementing a request for content with additional content which describes all prerequisite concepts related to the request) and directed guidance [7] could be modelled using the P-region if a topology were introduced which allowed for the notion of a hyperdocument.

8 Contributions and Conclusions

The challenge which the research [24] underlying this paper aims to meet is how to model, at a suitable level of abstraction, the space of possibilities for personalisation actions that could be made available to users of hypermedia systems.

The abstract, additive model for personalisation described aims to answer the question of which personalisation actions could be made available to users. It does this by how, given a conceptual framework such as that which underlies the H-region, it is possible to devise a language for personalisation

whose effects fall out as a consequence of that framework being adopted.

The language for personalisation is induced from the formal definition of the emergent properties of hypermedia systems. It is complete, in the sense that it is expressive enough to override all design decisions, and hence can be said to be an instrument for the transfer of ownership of every aspect of the interaction with a hypermedia system.

Attention is drawn to the fact that even as the detail of the model could be presented in equivalent ways, the methodological procedure of isolating the scope for personalisation, defining it formally and then inducing from that formalisation the set of personalisation actions needed, is a contribution that can be used in other settings, under different assumptions and using alternative conceptualisations of hypermedia systems.

References

[1] I Beaumont. User Modeling in the Interactive Anatomy Aystem ANATOM-TUTOR. *User Models and User Adapted Interaction*, 4(1):21–45, 1994.

[2] C Boyle and A Encarnacion. Metadoc: An adaptive hypertext reading system. *User Models and User Adapted Interaction*, 4(1):1–19, 1994.

[3] P De Bra, P Brusilovsky, and R Conejo, editors. *Aptive hypermedia and Web based systems: Second International Conference, AH 2002*, volume 2347 of *LNCS*, Malaga, Spain, May 29 - 31 2002. Springer.

[4] P Brusilovsky. Methods and Techniques of Adaptive Hypermedia. *User Modeling and User-Adapted Interaction*, 6(2-3):87–129, 1996.

[5] P Brusilovsky. Adaptive Hypermedia. *User Modeling and User-Adapted Interaction*, 11(1-2):87–110, 2001.

[6] P Brusilovsky, A Kobsa, and J Vassileva, editors. *Adaptive Hypertext and Hypermedia*. Kluwer Academic Publishers, Dordrecht, February 1998. ISBN 0-7923-4843-5.

[7] P Brusilovsky and L Pesin. Isis-tutor: An adaptive hypertext learning environment. In *Japanese-CIS Symposium on Knowledge-based software engineering*, pages 83–87, Pereslavl-Zalesski,Russia, 1994.

[8] P Brusilovsky, E Schwarz, and G Weber. ELM-ART: An Intelligent Tutoring System on World Wide Web. *Lecture Notes in Computer Science*, 1086:261–269, 1996.

[9] Peter Brusilovsky and Leonid Pesin. Visual Annotation of Links in Adaptive Hypermedia. In *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems*, volume 2 of *Short Papers: Agents and Anthropomorphism*, pages 222–223, 1995.

[10] Licia Calvi and Paul de Bra. Improving the usability of hypertext courseware through adaptive linking. In *Proceedings of the Eighth ACM Conference on Hypertext*, pages 224–225, 1997.

[11] W Clocksin and C Mellish. *Programming in PROLOG.* Springer-Verlag Berlin and Heidelberg GmbH and Co. KG, 1994. ISBN 3540583505.

[12] P De Bra, Ad Aerts, D Smits, and N Stash. AHA! meets AHAM. In P De Bra, P Brusilovsky, and R Conejo, editors, *Adaptive Hypermedia and Web-based systems: Second International Conference, AH 2002*, LNCS, pages 213–222, Malaga, Spain, May 29 - 31 2002. Springer.

[13] Paul de Bra, Geert-Jan Houben, and Hongjing Wu. AHAM: A Dexter-based Reference Model for Adaptive Hypermedia. In *Proceedings of the 10th ACM conference on Hypertext and Hypermedia*, Darmstadt, Germany, February 21-25 1999. ACM.

[14] B Forta. *Advanced Cold Fusion 4 Application Development.* Macmillan Computer Publishing, 1999. ISBN 0789718103.

[15] Cord Hockemeyer, Theo Held, and Dietrich Albert. RATH — A Relational Adaptive Tutoring Hypertext WWW–Environment Based on Knowledge Space Theory. In Christer Alvegård, editor, *Proc. CALISCE 98*, pages 417–423, 1998.

[16] Hubertus Hohl, Heinz-Dieter Boecker, and Rul Gunzenhaeuser. Hypadapter: An Adaptive Hypertext System for Exploratory Learning and Programming. *User Modelling and User-Adapted Interaction*, 6(2-3):131–156, July 1996.

[17] K Höök, J Karlgren, A Waern, N Dahlback, C Jansson, K Karlgren, and B Lemaire. A glass box approach to adaptive hypermedia. *User Modeling and User Adapted Interaction: Special Issue Adaptive Hypertext and Hypermedia*, 6(2-3):225–261, July 1996.

[18] J Kay and B Kummerfeld. An Individualised Course for the C Programming Language. In *Proceedings of the Second International WWW Conference*, 1994.

[19] A Kobsa, D Müller, and A Nill. KN-AHS: An Adaptive Hypertext Client of the User Modeling System BGP-MS. In *Proceedings of the 4th International Conference on User Modeling*, pages 99–106, 1994.

[20] N Koch and M Wirsing. The Munich reference model for adaptive hypermedia applications. In P De Bra, P Brusilovsky, and R Conejo, editors, *Adaptive Hypermedia and Web based systems: Second qInternational Conference, AH 2002*, LNCS, pages 213–222, Malaga, Spain, May 29 - 31 2002. Springer.

[21] N Mathe and J Chen. User-centered Indexing for Adaptive Information Access. *User Modeling and User-Adapted Interaction*, 6(2-3):225–261, July 1996.

[22] J A McDermid, editor. *Software Engineer's Reference Book.* Butterworth-Heinemann Ltd., 1991. Reprinted 1994, ISBN 0-7506-0813-7.

[23] J Moline, D Benigni, and J Baronas, editors. *Proceedings of the Hypertext Standardization Workshop*, volume 500–178 of *NIST Special Publications*, Gaithersburg, MD 20899 USA, January 1990.

[24] J Ohene-Djan. *A Formal Approach to Personalisable, Adaptive Hyperlink-Based Interaction.* PhD thesis, Department of Computing, Goldsmiths College, University of London, University of London, 2000. Available from http://homepages.gold.ac.uk/djan/jodthesis.pdf.