Synthesis of Web Sites from High Level Descriptions

João M. B. Cavalcanti* David Robertson

Institute for Representation and Reasoning
Division of Informatics, The University of Edinburgh
{joaoc,dr}@dai.ed.ac.uk

Abstract. As use of Web sites has exploded, large amount of effort have gone into the deployment of sites but little thought has been given to methods for their design and maintenance. This paper reports some encouraging results on the use of automated synthesis, using domain-specific formal representations, to make design more methodical and maintenance less time consuming.
Key Words: Web site application, computational logic, HTML.

1 Introduction

Web site maintenance has become a challenging problem due to the increase in size and complexity of Web site applications. It often involves access to databases, complex cross referencing between information of the site and sophisticated user interaction.

Web sites applications related to a same domain often share a common pattern. Consider, for example, the Software Systems and Processes Group at Edinburgh (www.dai.ed.ac.uk/groups/ssp/index.html) the research group Web sites of the Artificial Intelligence Research Institute in Barcelona (www.iiia.csic.es) and The Artificial Intelligence Laboratory at MIT (www.ai.mit.edu). Although they look quite different, the underlying application design is very similar, particularly in information content. We would like to exploit these similarities, for example in reusing application components for visualisation designs, thus saving time in application development.

One way to achieve this goal is to separate, formally, the information content of Web sites from their presentational form. Separation of content from visualisation aspects at early design stages is important in order to allow a description its essential content, such as data, operations, information flow within the site and constraints on the information flow. These features are mingled with presentational descriptions if we work directly in HTML [8] code.

A traditional way of representing information processing abstractly is through computational logic. Although logics provide a powerful framework for application description, few people feel comfortable when using them directly as a tool. This problem can be overcome by designing domain or task-specific dialects of

* On leave from University of Amazonas, Brazil.

a logic which are adapted to the informal styles of description used in the application, while also supporting automatic translation to less intuitive formal representations needed for computation. This leads to different levels of representation in which a mapping from higher levels to the lower ones will be required. Hence, the Web site generation process is organised in different levels, from a high level description going through an intermediate representation to the resulting site code.

This work describes an approach to design and maintenance of Web site applications which is based on simple form of computational logic The key feature of the proposed approach is separating the site information content from its presentational form and deriving the Web site code from its content description via automated synthesis. The main benefit of the approach is the ability to work separately on the application and visualisation specifications, allowing updates in the application content without necessarily changing the presentational form and also changing the visualisation of the site without any changes in the specification of the site content.

Parts of this task have been addressed by others. WebMaster [7] is a tool, that can be used for constraint checking on Web Sites based on rules expressed in logic. Strudel [5] is a Web site management system, which generates Web site code from data residing in a database via a SQL-like language. Fernández et al [6] addressed the problem of specifying and verifying integrity constraints on a Web site, based on a domain specific description language and creating a graph structure to represent the site. This is done in a fashion very similar to our work, but we also have added representation and automated generation of operations (CGI programs), which is not considered by Fernández et al.

The next section introduces the 3-level approach, section 3 presents a working example, section 4 describes the problem description language, section 5 presents the synthesis process for the example Web site.

2 A Three-Level Approach

A Web site application is a collection of pages where each page consists of information content and links. Information content often is described from a database. Links correspond to transitions between pages and there are two different ways of making a transition: via a hyper-link between two pages or by an operation call. An operation is a program that may receive some input arguments from the first page and displays the result of its computation in the second page. We implement operations as CGI programs [4]. From this point onwards we refer to hyper-links just as links since we have already made the distinction between them and operation calls in the context of transitions.

Our approach to the design and maintenance of Web sites is based on computational logic. However, since the designer is not supposed to work directly with logic, the synthesiser is arranged in three different levels as illustrated in Figure 1. The high level description should be provided by the designer by means of an appropriate interface. Having this initial description as a starting point, an intermediate representation for the application is built using a domain-specific formal language. The Web site code is automatically generated from the intermediate representation. The key idea in this approach is separating information content from its presentational form. Hence, a description of application components such as data and operations can be produced regardless of its future rendering in a Web browser.

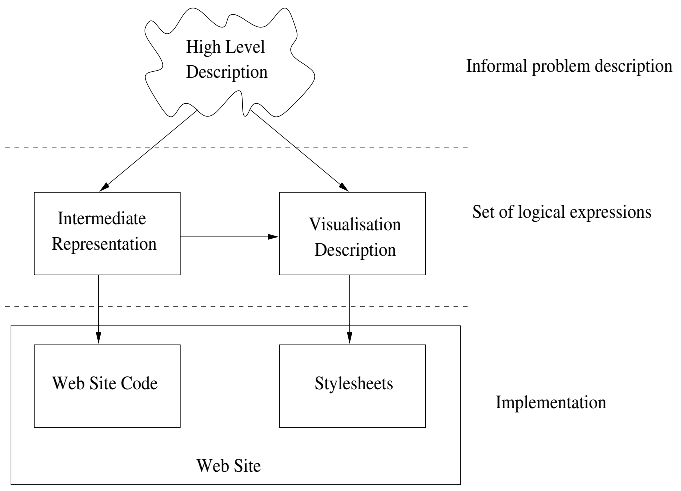

High Level Description
Informal problem description
Intermediate Representation
Visualisation Description
Set of logical expressions
Web Site Code
Stylesheets
Implementation
Web Site

Fig. 1. The 3-Level Approach

Having made this specification in a declarative way, independent from implementation/visualisation details, allows a great deal of flexibility in choosing appropriate styles for visualisation and different implementations for the operations. Our paper concentrates on the left side of the diagram in Figure 1 explaining how the intermediate representation is used in code synthesis. A brief discussion on visualisation issues is given in section 4.4.

3 An Example

As a working example, consider a research group Web site like the ones mentioned in the introduction. The site should display information about the group, such as an introduction about the group aims and activities, members of the group and their publications and projects which have members of the group involved. Note that this is a subset of a real research group Web site which may include more information than that described here.

Formal representation of our example site begins with the domain-specific language used to describe the basic elements of the research group. This is done in conventional predicate logic but it is (equally) easiest to think of this as a database of relevant information. For instance, the database of the research group Web site can be described by the following set of predicates, where the arguments names give a guide to the type of data structure to which each would be instantiated in the actual database.

```
group_aims(Aims).
contact_info(E_mail, Address, Postcode, City).
person(Name, Status).
project(Title, Abstract, People_involved).
publication(Title, Authors, Reference, Year, Abstract).
```

We want to build a Web site structured like the diagram in Figure 2.

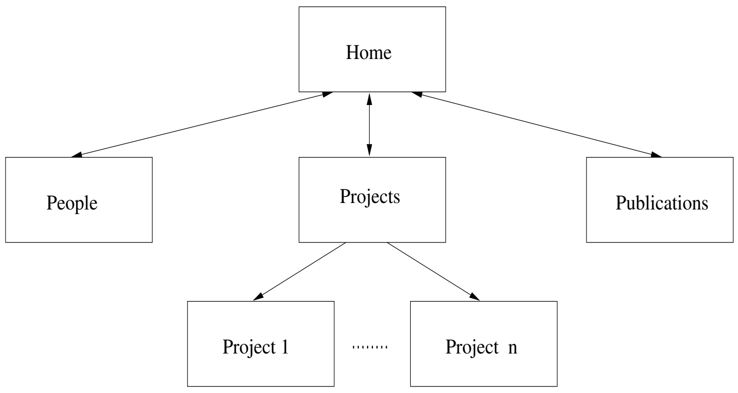


Fig. 2. A general representation for a research group Web site

Each box corresponds to a page of the site and arrows correspond to transitions between pages. Our next step is to describe a possible distribution of data among the pages of the site.

The home page should present the aims of the group and contact information. In the people page we want to show two different lists, one containing the names of the current members and the second with names of previous members of the group. The publications page presents a list with all publications of the group members. A list of project titles is displayed on the project page and for each project there is a specific page presenting its title, abstract and people involved. In the following discussion we shall introduce a formal representation of these requirements and then use these to generate a Web site.

The relation between information displayed on a page and the research group description is expressed by rules. The expression below defines that the group aims and contact information are displayed on the home page.

```
display(home, [group_aims(X), contact_info(Y)]) ←
                group_aims(X) ∧
                contact_info(Y),
```

Changes in display of information are represented by transition rules. The transition from home to the people page, for example, is represented by the following expression:

display(home, [ ]) $\Rightarrow$ display(people, [ ]).

The empty list "[ ]" means that the transition is independent of the information displayed by either page. This sort of expression corresponds to a simple hyper-link.

Similar expressions are used to represent the other pages and transitions. A catalogue of these expressions is given in the next section.

4 Describing Web Site Applications

We define a Web site application in terms of data, operations, transitions and constraints on the transitions. The expressions used for this purpose are formally described here.

4.1 Transitions

Information flow when navigating a Web site can be viewed as a sequence of actions, where each action is the display of a set of information. A transition moves from one page to another. As mentioned earlier, there are two different ways to make a transition: via hyper-links or operation calls. In order to express information flow in terms of actions, an operator and two special predicates were defined. The following tables explain their meaning.

Expression	Interpretation
display(Id, InfoList)	display InfoList at page identified by Id. InfoList is a list with the form [p_1(Info$_1$), P_2(Info$_2$), ..., $p_n(Info_n)$], where each p_i is a predicate and Info$_i$ a variable corresponding to a specific piece of information.
satisfy(p(Arg$_1$, Arg$_2$, ..., Arg$_n$))	operation p can be executed. Each Arg$_i$ is an argument that can be either input or output to p.

Predicates display and satisfy are combined by an additional connective in order to express transitions. Some transitions are conditional, where a condition is defined by a conjunction of predicates. The table below shows different sorts of transition expressions.

Expression	Interpretation
display(Id_1, [ ]) $\Rightarrow$ display(Id_2, [ ]).	A transition from page Id_1 to page Id_2. The empty list means that the transition is independent of any information displayed in either pages.
display(Id_1, $\mathsf{InfoList}_1$) $\Rightarrow$ display(Id_2, [ ]) $\leftarrow$ C.	A transition from page Id_1 to page Id_2 is associated with information in $\mathsf{InfoList}_1$. C is a condition which includes a set of predicates, that is mainly used to retrieve data from the database to instantiate pieces of information.
display(Id_1, [p_1(A)]) $\Rightarrow$ display(Id_2, [p_2(B)]) $\leftarrow$ satisfy(foo(A,B))	A transition from page Id_1 to page Id_2 is done via the execution of operation foo given the input argument A from page Id_1. The result B is displayed in page Id_2.

Using the expressions above, the information flow of our Web site can easily be described, as illustrated by the example below.

display(Id1, [p_1(Info_1)]) $\leftarrow$
    p_1(Info_1).
display(Id1, [ ]) $\Rightarrow$ display(Id2, [ ]).
display(Id1, [ ]) $\Rightarrow$ display(Id3, [ ]).
display(Id2, [p_2(Info_2)]) $\Rightarrow$ display(Id4, [p_4(Info_4)]) $\leftarrow$
    p_2(Info_2) $\wedge$ satisfy(foo(Info_2, Info_4)).

A graphical view for this example is depicted by Figure 3.

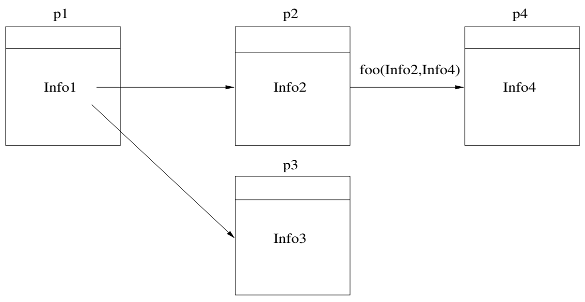


Fig. 3. Information flow

4.2 Operations

A library of parameterisable components is used to build the operations of the application. Each component corresponds to a different type of operation, such as queries, filters, etc, and parameters usually include a name for the operation, input and output arguments. The construction of operations are based on a simplified form of techniques editing [9].

The most common sorts of operation that we have encountered in Web site applications are queries on databases. The following table presents an initial set of components. These are task specific, so more components would have to be added to cover a wider range of tasks. Nevertheless, the small set we have now is surprisingly versatile.

Expression	Interpretation
bl_recursion(P, B, N_s, N_t)	denotes that predicate P defines a recursion over a fact predicate B with the argument position N_s being the starting point of the recursion and N_t the end point.
filter(P, N_d, N_c, test(T))	denotes that predicate P filters elements of a list at argument position N_d by deconstructing that list and constructing a list of chosen elements at argument position N_c. The test used to filter elements is the predicate named T.
query(Q, P, ArgsIn, ArgsOut)	denotes that query predicate Q finds all solutions for predicate P given a list of input argument positions ArgsIn and a list of output arguments positions ArgsOut. These arguments refer to the predicate P which must be a fact.

These expressions work by instantiating a program implementation pattern associated to each sort of component. The parameters serve as an interface to the operation generation process. The resulting instantiated operation is a CGI program.

For example, the definition:

query(pubs_by_year, publication, [4], [1,2,3,5])

corresponds to a query operation, called pubs_by_year, that performs a query on the predicate publication. The query returns values corresponding to the arguments positions [1,2,3,5]. Argument of position 4 is given as input parameter.

The following code correspond to the CGI program (in Prolog) corresponding to this operation:

```
main :-
    get_form_input(F),
    get_form_value(F, year, Y),
    pubs_by_year(Y,PubList),
    show_page(PubList).

pubs_by_year(Y, PubList) :-
    findall([T, A, Ref, Abs], publication(T,A,Ref,Y, Abs), PubList).
```

Predicates get_form_input, get_form_value and output_html are given by the Pillow library [2] which provides facilities for generating HTML code for logic programming systems, including form handlers and Web document parsing.

The predicates above perform the following tasks:

– get_form_input(F): translates input from the form to a list of attributes=value pairs.
– get_form_value(F, Attribute, Value): gets value Value for a given attribute Attribute in list F.
– output_html(T): T is a list of HTML terms that are transformed into HTML code and sent to the standard output. HTML terms have a direct correspondence to Pillow terms.

The first two predicates are used to process the form and get the parameter to call the query operation pubs_by_year. The result (PubList) is given to a specific predicate show_page to generate the HTML code corresponding to the resulting page. The details of this transformation are discussed in section 4.4. A similar pattern is followed to build filtering and recursing operations.

These descriptions are versatile because they support modifications to the implementation (via the program pattern) as long as the interface of the component remains the same.

4.3 Constraints

The constraints considered in our approach are used to enforce an order of information presentation. A very common constraint of this sort appears in electronic commerce Web sites, where information about the purchase and the total amount must be displayed *before* the customer provides the payment information. Similarly, a confirmation of payment must be displayed *after* checkout.

In order to specify constraints in the order of information display, we use two concepts from Transaction Logic (TR) [1], serial conjunction and path, that were adapted to represent the sort of constraints we need. Serial conjunction is used to represent a sequence of actions. This is written in the form $\mathsf{a} \otimes \mathsf{b}$ to define a path formed of action a followed by action b.

In the Web site context a path is simply a sequence of information display. Hence constraints on a Web site can be expressed in terms of valid/invalid paths. Paths can be derived from the site graph, where nodes correspond to pages and edges correspond to transitions. The site graph is easily built by inspecting the transition expressions. We assume that a finite number of acyclic paths can be extracted from the transitions definitions. Figure 4 shows the graph extracted from the transition specification example given on section 4.1.

The simplest path contains a single element which is a display goal, as defined earlier. Hence path expressions are of the form:

$$\mathsf{display}(\mathsf{Id}_1,\ \mathsf{InfoList}_1) \otimes \mathsf{display}(\mathsf{Id}_2,\ \mathsf{InfoList}_2) \otimes \ldots \otimes \mathsf{display}(\mathsf{Id}_n,\ \mathsf{InfoList}_n)$$

Another useful concept taken from TR is a special symbol path which corresponds to a sequence of actions of any length. This concept allows us to write simplified expressions. For example, the expression:

$$\mathsf{path} \otimes \mathsf{display}(\mathsf{Id}_1,\ \mathsf{InfoList}_1) \otimes \mathsf{display}(\mathsf{Id}_2,\ \mathsf{InfoList}_2) \otimes \mathsf{path}$$

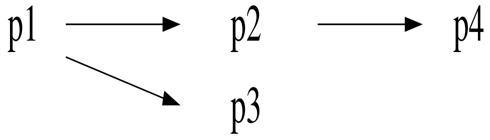


Fig. 4. Site graph

denotes any path that displays the page identified by Id_1 which is immediately followed by page Id_2.

The following table presents some common constraint expressions:

Expression	Interpretation
$\neg$ (**path** $\otimes \neg$ display(p, Info_p) $\otimes$ **path**) $\otimes$ display(q, Info_q) $\otimes$ **path**)	Information Info_p must be displayed before information Info_q.
$\neg$ (**path** $\otimes \neg$ display(p, Info_p) $\otimes$ **path**) $\otimes$ display(q, Info_q) $\otimes$ **path**)	Information Info_p must be displayed immediately before information Info_q.
$\neg$ (**path** $\otimes$ display(p, Info_p) $\otimes$ **path**) $\otimes \neg$ display(q, Info_q) $\otimes$ **path**)	information Info_q must be displayed after information Info_p.
$\neg$ (**path** $\otimes$ display(p, Info_p) $\otimes$ $\neg$ display(q, Info_q) $\otimes$ **path**)	information Info_q should be displayed immediately after information Info_p.

Constraint checking can be done by matching paths expressions with constraint expressions. Note that the special predicate path matches paths of any length. For example, consider the following paths:

p_1: display(Id_1, Info_1) $\otimes$ display(Id_2, Info_2) $\otimes$ display(Id_4, Info_4).
p_2: display(Id_1, Info_1) $\otimes$ display(Id_3, Info_3).

Now, consider the two following constraints:

c_1: $\neg$ (path $\otimes \neg$ display(Id_2, Info_2) $\otimes$ display(Id_4, Info_4) $\otimes$ path).
"Information Info_2 should be displayed before information Info_4".

c_2: $\neg$ (path $\otimes \neg$ display(Id_3, Info_3) $\otimes$ display(Id_1, Info_1) $\otimes$ path).
"Information Info_3 should be displayed before information Info_1".

From the specifications above it is possible to conclude that constraint c_1 is satisfied. Note that the negation of the constraint means that paths which have that pattern are not valid. As paths p_1 and p_2 cannot match the pattern, they are valid.

On the other hand, constraint c_2 is not satisfied because both p_1 and p_2 match the constraint pattern. Informally, it can be verified that in either path information i_1 is displayed without i_3 being displayed before it.

4.4 Visualisation Issues

Visualisation issues are not the main concern in this work, but we summarise here the link to visualisation. Recent technologies for the Web such as XML [10] and style sheets [12] reinforce the idea of separation between Web site content from its presentational form. In this view, visualisation specification is done by style sheets, which describe how the information is presented. There are many activities on defining languages and standards for style sheets, such as CSS [3] and XSL [11].

Our current implementation translates the logic descriptions into HTML via Pillow. Since we have a separate description for the visualisation, this description acts as a style sheet.

We also have a CSS style sheet that defines some presentational attributes, like font type, font size, text color, background color, etc. These attributes are also defined by the site designer and are also part of the visualisation description as simple predicates, that are transformed into the CSS style sheet.

Type information (which appears as part of information content description) is used to associate each piece of information with a particular style of visualisation. For example, from the expression group_aims(X), we can define a style to present X, which can be a bullet list, a table or plain text.

Styles are expressed using definite clause grammar rules that transform a piece of information in a sequence of Pillow terms that are used to produce HTML code. These expressions have the general form:

style(p(A_1, ..., A_n)) –> [T_1, ..., T_m].

where A_i is a specific argument value of predicate p and each T_i is a Pillow term corresponding to the visualisation of A_i. Some examples of the application of these expressions are presented in the next section. The architecture of the system allows replacement of the target languages used to generate the Web site code. Currently we are using HTML and CSS, but XML and XSL could also be used. Changes in the target language do not have any impact on the intermediate representation.

5 Generating Web Site Code

The intermediate representation combined with the visualisation description provide all the necessary information to produce the site code. Three main steps are followed to produce the site code: (1) check constraints; (2) given the intermediate representation, generate pages structure including content, links and operation calls and (3) given visualisation descriptions in style sheets map each page structure into HTML/CSS code.

Here we have the complete transitions specification for the research group Web site, depicted by Figure 2. The following discussion on code synthesis refers to this description.

```
1   display(home, [group_aims(X), contact_info(Y)]) ←
                    group_aims(X) ∧
                    contact_info(Y),
2   display(home, [ ]) ⇒ display(people, [ ]).
3   display(home, [ ]) ⇒ display(publications,[ ]).
4   display(home, [ ]) ⇒ display(projects,[ ]).
5   display(people, [current_members(NamesCurr), previous_member(NamesPrev)]) ←
                    NamesCurr = {$N_1$:person($N_1$,current_member)} ∧
                    NamesPrev = {$N_2$:person($N_2$,previous_member)}.
6   display(people, [ ]) ⇒ display(home, [ ]).
7   display(people, [ ]) ⇒ display(publications,[ ]).
8   display(people, [ ]) ⇒ display(projects,[ ]).
9   display(publications,[pubs_list(Pubs)]) ←
                    Pubs = {[T,AA,R,Year,A]:publication(T,AA,R,Year,A)}
10  display(publications, [ ]) ⇒ display(home,[ ]).
11  display(publications, [ ]) ⇒ display(people, [ ]).
12  display(publications, [ ]) ⇒ display(projects,[ ]).
13  display(projects,[project_titles(AllT)]) ←
                    AllT = {T:project(T,Abstract,People)}.
14  display(projects, [ ]) ⇒ display(home,[ ]).
15  display(projects, [ ]) ⇒ display(people, [ ]).
16  display(projects, [ ]) ⇒ display(publications,[ ]).
17  display(projects,[project_titles(AllT)]) ⇒
18  display(T,[proj_details(T,A,PI)]) ←
                    T ∈ AllT ∧
                    project(T,A,PI).
```

Constraints are checked as described in section 4.3. If the site description conforms with the constraints, the second step is to build a list where each element describes a structure for each page of the site.

The page structure can be in two different forms depending whether the page is a static one or generated by an operation (via a CGI program). For static pages the structure is:

page(Id, ContentInfo, Links, OperationCalls)

where Id and ContentInfo are the same as defined earlier, but ContentInfo is fully instantiated, Links is a list with all page Ids that the current page is linked to via hyper-links and OperationCalls is a list containing operation names and their corresponding input arguments.

For pages resulting from operations, the structure is the following:

program(PId, OperationSpec, ContentInfo, Links, OperationCalls)

where PId is the identifier of the page, OperationSpec is composed of the name of the operation and its input/output arguments. The remaining features are the same as in static page structure.

As an example of page synthesis, we show the complete synthesis process for the people page of the Software Systems and Processes Group Web site at http://www.dai.ed.ac.uk/~joaoc/ssp/ people.html. This is done by using a generic page definition which is instantiated by the specific transition rules for the example via path constraints. Our generic page definition is as follows:

page(PageId, ContentInfo, Links, OperationCalls) $\leftarrow$
    ContentInfo = {I | visible(PageId,I)}
    Links = {L | link(PageId, L)}
    OperationCalls = {Op | op_call(PageId, Op)}

Visible items on the page correspond to any item being displayed on a path. Formally:

visible(PageId, I) $\leftarrow$ path $\otimes$ display(PageId, S) $\otimes$ path $\wedge$ I $\in$ S

Links from a page are those page identifiers which may immediately follow the page on any path. Formally:

link(PageId, L) $\leftarrow$ path $\otimes$ display(PageId,_) $\otimes$ display(L,_) $\otimes$ path

The page structure generated for the example using above definitions is:

page(people, [current_members(['Chris', 'Dave', 'Daniela', 'Jessica', 'Joao', 'Virginia', 'Stefan', 'Yannis']), previous_members(['Steve', 'Renaud', 'Alberto'])], [home, projects, publications], [ ]).

The lists with previous and current members' names are instantiated using rule 5 and having a database containing data for person(Name, Status). Links are defined in transition rules 6, 7 and 8 and there is no operation call for this page.

Finally, the style expressions used to map the people page to HTML code defines the how the information is to be rendered. A general style for all pages of the site is defeined including the group logo image, colors, etc. For example, the styles applied to the member lists are:

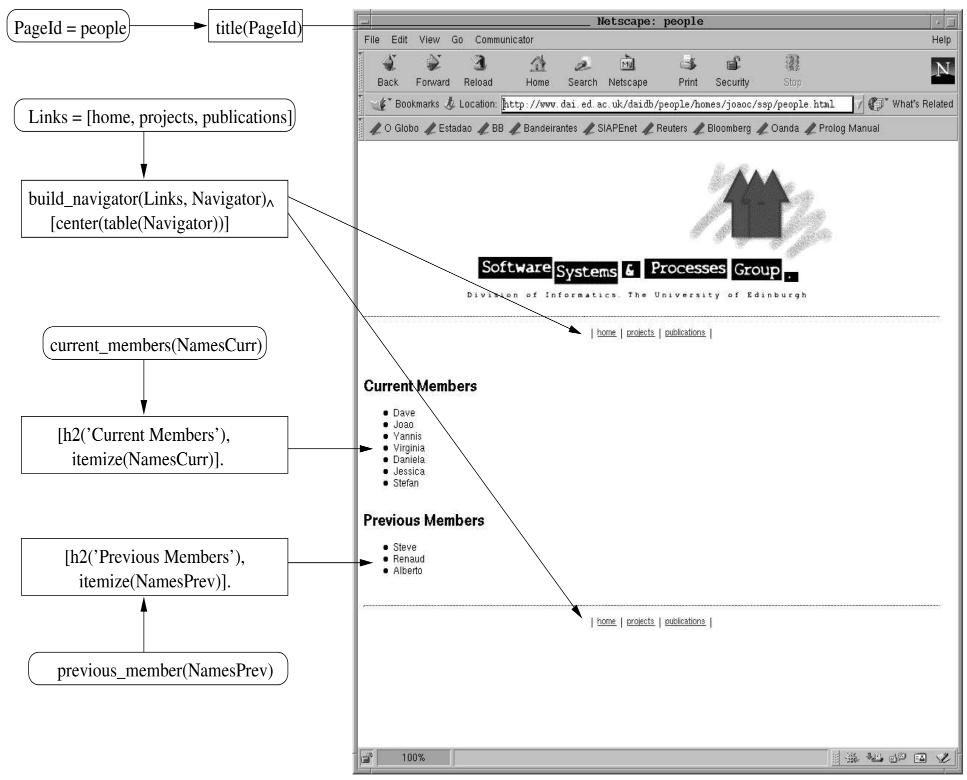
style(current_members(X)) –> [h2('Current Members'), itemize(X)].
style(previous_members(X)) –> [h2('Previous Members'), itemize(X)].

We have also defined a specific style for clustered links, called build_navigator, which given a list of links build a HTML table. The resulting visualisation for this style is shown in Figure 5, which also present the complete synthesis result for the people page. The boxes with rounded corners correspond to the page specifications that come from the page structure and the other boxes show the styles that are applied to them.

All the other pages of the site are generated in a similar fashion. The current synthesiser is implemented in Sicstus Prolog 3.5 and the complete example research group Web site can be visited at http://www.dai.ed.ac.uk/~joaoc/ssp/home.html.

6 Conclusions

A main contribution of this work is to describe a design approach that joins different levels of description to produce Web sites consistent with a corresponding high level description. Although this approach is domain-specific, we observe that a large number of current Web sites have similar domain features.



PageId = people
title(PageId)
Links = [home, projects, publications]
build_navigator(Links, Navigator)∧
[center(table(Navigator))]
current_members(NamesCurr)
[h2('Current Members'),
itemize(NamesCurr)].
[h2('Previous Members'),
itemize(NamesPrev)].
previous_member(NamesPrev)
Netscape: people
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location: http://www.dai.ed.ac.uk/daidb/people/homes/joaoc/ssp/people.html What's Related
O Globo Estadao BB Bandeirantes SIAPEnet Reuters Bloomberg Oanda Prolog Manual
Software Systems & Processes Group.
Division of Informatics. The University of Edinburgh
| home | projects | publications |
Current Members
Dave
Joao
Yannis
Virginia
Daniela
Jessica
Stefan
Previous Members
Steve
Renaud
Alberto
| home | projects | publications |
100%

Fig. 5. Synthesis result: People page

For these, our approach supports different visualisation descriptions for the same site specification. These visualisations might be tailored to different classes of users or to other needs. Changes in the site description without changing the visualisation specification are also supported. This changes the role of site maintenance from HTML hacking to alteration of a much simpler domain-specific problem description, with the site being automatically regenerated from this.

The actual Software System and Processes Group Web Site (which has been in routine use for the past 3 years) is automatically generated from specifications similar to those presented here. The cost of developing the synthesiser for the group site was justified after only a few weeks by the savings in maintenance effort [9]. The site can be visited at http://www.dai.ed.ac.uk/groups/ssp /index.html.

Acknowledgements

The first author is supported by the Brazilian Government through CAPES grant no. 1991/97-3.

References

1. Bonner, A.J. and Kifer, M. Transaction Logic Programming. Technical Report CSRI-323, Computer Systems Research Institute, University of Toronto. November, 1995.
2. Cabeza, D. and Hermenegildo, A. WWW Programming using Computational Logic Systems (and the PiLLoW/CIAO Library). Technical Report, Computer Science Department, Technical University of Madrid, 1997. In: http://www.clip.dia.fi.upm.es/miscdocs/pillow/pillow.html
3. Cascading Style Sheets, level 1. W3C Recommendation December 1996, revised January 1999. In http://www.w3.org/TR/REC-CSS1
4. The Common Gateway Interface. In: http://hoohoo.ncsa.uiuc.edu/cgi/
5. Fernández, M., Florescu, D., Kang, J., Levy, A. and Suciu, D. Catching the Boat with Strudel: Experience with a A Web-site Management System. In SIGMOD Conference on Management of Data, Seattle, USA, 1998.
6. Fernández, M., Florescu, D., Levy, A. and Suciu, D. Verifying Integrity Constraints on Web Sites. Proc. of the 16th International Joint Conference on Artificial Intelligence - IJCAI'99. Stockholm, Sweden, 1999.
7. van Harmelen, F. and van der Meer, J. WebMaster: Knowledge-based Verification of Web-pages. In: Proc. of the 12th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, (IEA/AEI'99), Ali, M. and Iman, I. (eds.), Springer Verlag, LNAI, 1999.
8. HTML - Hyper Text Markup Language. W3C - World Wide Web Consortium. In: http://www.w3.org/MarkUp/
9. Robertson, D. and Augustí, J. Software Blueprints: Lightweight Uses of Logic in Conceptual Modelling, ACM Press, Addison Wesley Longman, 1999.
10. Extensible Markup Language (XML) 1.0. W3C Recommendation, February 1998. In: http://www.w3.org/TR/1998/REC-xml-19980210
11. Extensible Stylesheet Language (XSL) Specification. W3C Working Draft, April 1999. In: http://www.w3.org/TR/WD-xsl/
12. Web Style Sheets. In: http://www.w3.org/Style/