Running Head: VERIFYING WEB SITE PROPERTIES USING COMPUTATIONAL LOGIC

VERIFYING WEB SITE PROPERTIES USING COMPUTATIONAL LOGIC

João Cavalcanti **David Robertson**

Centre for Intelligent Systems and their Applications
Division of Informatics
The University of Edinburgh
80 South Bridge, EH11HN
Edinburgh, Scotland - UK
{joaoc, dr}@dai.ed.ac.uk

ABSTRACT

The continuing increase in size and complexity of Web sites has turned their design and construction into a challenging problem. Systematic approaches can bring many benefits to Web site construction, making development more methodical and maintenance less time consuming. Computational logic can be successfully used to tackle this problem as it supports declarative specifications and reasoning about specifications in a more natural way. Computational Logic also offers meta-programming capabilities which can be used to develop methods for automated Web site synthesis. This chapter presents an approach to Web site synthesis based on computational logic and discusses in more detail two important features of the proposed approach: the support for property checking and integrity constraint specification and verification.

INTRODUCTION

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

A design principle accepted by many authors is separation between information content, navigation structure and visualisation (Florescu, Levy, & Mendelzon,1998; Schwabe & Rossi, 1995). This idea promotes a better understanding of the data requirements (content), the underlying architecture of the site (navigation) and an appropriate user interface (visualisation). Furthermore it makes maintenance tasks easier as each of those components can be managed separately.

Another design principle is to define a Web site application in a declarative way. This allows a great deal of flexibility, particularly in choosing different visualisations for the same specification. It also supports automated reasoning allowing constraint and property verification against a specification, and in some cases for entire Web sites to be generated from specifications. The degree of automation may vary, from a few page templates being generated to the generation of a complete Web site application.

One motivation for this work is the fact that computational logic has not been well exploited to address the problem of Web site specification and generation. Logics provide a high-level and abstract approach whereby unimportant implementational details can be conveniently postponed until later stages of development. Mappings between some logics and more “concrete” formalisms have been proposed, eg. (Proietti & Pettorossi, 1994). Some logic specifications (although understood in an abstract sense) also can be executed in a procedural style in order to reason mechanically about their consequences (Fuchs, 1992).

As logic programming is intrinsically declarative, it supports both application specification and integrity constraint specification and verification in a more natural way. Symbolic manipulation and meta-programming facilities also make the use of computational logic techniques appropriate for automated Web site synthesis (Robertson & Augustì, 1999).

We have developed an approach to Web site synthesis based on the ideas above (Cavalcanti & Robertson, 2001; Cavalcanti & Robertson, 2002). The main idea behind the proposed approach is to derive a Web site automatically from an application and a related visualisation description. This requires the use of an appropriate formalism and as we already mentioned, logic is used for this task. However, few people feel comfortable using a general purpose logic directly. One way to deal with this issue is by offering a suitable interface to the designer based on a task or domain specific formalism and translate the resulting description into logic expressions. As a result, the synthesiser is organised in three different levels. The architecture of our synthesiser is illustrated in Figure 1.

We begin with a high level description of an application. From the application description an intermediate representation is automatically derived. Constraints are checked against the specification in order to produce a valid intermediate representation with respect to the constraint definitions. Finally, this intermediate representation is combined with a visualisation description to generate corresponding Web site code automatically. Although we give some details of the synthesis process in section 2.4, the main concern of this chapter is to present how our Web site synthesis method supports integrity constraint and property verification.

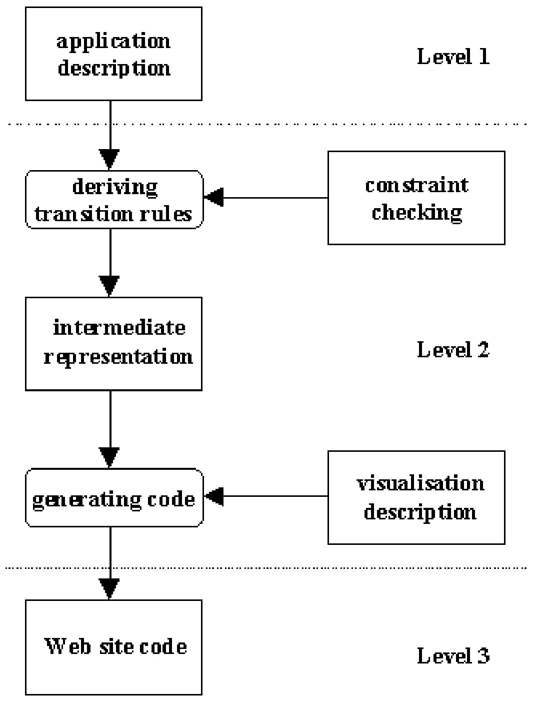


Figure 1: The 3-level approach

Background and Related Work

Various approaches to Web site design and maintenance have been proposed in recent years. There is general agreement amongst these on some core concepts, such as separation between information content, navigation structure and visualisation; declarative specifications, by means of high-level conceptual data models or declarative languages; and automated or semi-automated generation of Web site by means of CASE tools.

The main differences between the different proposals are in the emphasis given to particular aspects of the process. Most works focus on modelling aspects such as Araneus (Atzeni, Mecca & Merialdo, 1998), Strudel (Fernández, et al., 1998), AutoWeb (Fraternali & Paolini, 1998), OOHDM (Schwabe & Rossi, 1995) and WebML (Ceri, Fraternali & Bongio, 1999). Other works are data-driven such as Torii (Ceri, Fraternali & Paraboschi, 1999) and there are also works based on semantic descriptions as OntoWebber (Jin, Decker & Wiederhold, 2001) and SEAL (Maedche, et al., 2001). Our approach is a data-driven approach, which focuses on the generation of different visualisations and on associated Web site maintenance.

Where the research is driven by modelling most approaches are based on traditional conceptual data models, such as the entity-relationship model (ER) and its extensions (Elmasri & Navathe, 1999) or object-oriented data models. In this category we can include Araneus, Torii, WebML, AutoWeb and OOHDM. WebML proposes a structural model compatible with ER, ODMG object-oriented data model and UML class diagrams. AutoWeb is based on the HDM-lite model which is a Web-specific version of HDM. OOHDM is also an object-oriented extension to HDM. Strudel models a Web site as graphs. OntoWebber and SEAL are based on DAML+OIL and RDF, respectively, which are used to define ontologies describing the application domain. Our approach can support different conceptual data models. We advocate the idea of using existing data models, provided the appropriate mapping procedures to our intermediate representation.

Generation of different visualisations for the same specification and personalisation of Web sites are supported by most approaches. Query languages and templates are the main tools used for this purpose. OntoWebber, SEAL, Torii, WebML, AutoWeb and Strudel all support this feature. Our approach provides support for combining declarative descriptions of alternative visualisations with templates in different target languages.

Support for integrity constraints is given by OntoWebber, Torii and Strudel. Fernández, et al., (1999) 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 similar to our work. However their implementation is based on procedural algorithms, contrasting with our logic-based approach.

Our work is distinct by offering a framework for Web site construction based on computational logic. This feature makes it more convenient to introduce reasoning capabilities, also supporting definition of integrity constraints and rules for both navigation and visualisation. Our approach also is highly extensible, either supporting different data sources or different target languages for generating Web pages.

In the following sections we shall present how our Web site development approach supports verification of properties, including integrity constraints checking. In Section 2 we introduce our approach to automated Web site synthesis. Section 3 presents a method for constraint checking in Web sites and Section 4 discusses property checking. Section 5 discusses some future trends and Section 6 presents the conclusion.

AN OVERVIEW OF OUR APPROACH TO WEB SITE SYNTHESIS

A Web site is a collection of pages where each page consists of information content and links. Information content often is described from a database and links correspond to transitions between pages. Our approach supports automatic generation of Web sites and it also offers a framework for property and constraint checking.

A high-level specification of an application must be provided as the starting point of the development process. For this purpose we can use established and well-known conceptual data models, such as the entity-relationship model. Since issues related to mapping an ER model to our intermediate representation language are not in the scope of this chapter, we assume that this task has been already carried out. Details of mapping procedures from an ER schema to our intermediate representation can be found in (Cavalcanti & Robertson, 2002).

Intermediate Representation

The intermediate representation defines navigational paths between all pieces of information that should be presented in the site. This second level allows the definition of a more flexible Web site generation process because it is independent of any particular implementation.

The basic components of our intermediate representation are the *pieces of information*. These are structured pieces of data with additional information about their type and an associated label that allows their reference. Its general format is ***Label(Type, Datum)*** where ***Label*** should be a unique name and ***Type*** is the data type (integer, float, string or image) of ***Datum***. We show in Figure 2 the context-free grammar that defines our pieces of information, using a BNF notation. We use the Prolog conventions (Apt, 1997) for list definition, that is, [Head|Tail], and the empty list “[ ]”.

Info	::=	Label(ScalarType, Datum) |
		Label(ArrayType, ListOfData) |
		Label(table, ListOfLists)
ScalarType	::=	integer | float | string | image | text | null
ArrayType	::=	list | tuple
ListOfData	::=	[(ScalarType, Datum) | ListOfData] | [ ]
ListOfLists	::=	[ListOfData | ListOfLists] | [ ]
Datum	::=	*any datum*
Label	::=	*any string*

Figure 2: Grammar for Pieces of Information

Pieces of information are the basic components of a web site. The label is necessary as a unique reference to pieces of information within a site. The type information, as we shall see below, is needed for the appropriate visualisation of the data. We organise the pieces of information as *display units*, that is, “chunks” of pieces of information comprising a unit. Normally, we equate a web page with a display unit, but different alternatives can be considered, like the sections or chapters of an electronic book. Units of display are represented as

display([Info$_1$, ..., Info$_n$]) where ***Info$_i$*** are as above. They are defined as special kind of Horn clause, ***display([Info$_1$, ..., Info$_n$])*** $\leftarrow$ ***p$_1$(Info$_1$)*** $\wedge$ ***...*** $\wedge$ ***p$_n$(Info$_n$)*** whereby each individual piece of information ***Info$_i$*** is obtained and properly assembled in the expected format via predicate ***p$_i$***. Our display units put together pieces of information and give them a ``handle" by means of which collective references can be made. We provide an intuitive example in Figure 3 of a web site definition using our display units.

```
display([intro(text,Intro), map(image,Map)]) ←
        get_intro(Intro) ∧ get_map(Map)
display([expl(text, Expl), members(list, Members)]) ←
        get_expl(Expl) ∧ get_members(Members)
```

Figure 3: Example of Display Units for Site

Transitions between display units are defined via a binary $\Rightarrow$ operator. This operator connects individual pieces of information with display units. Its general format is ***Info*** $\Rightarrow$ ***DisplayUnit*** where ***DisplayUnit*** is as above. Intuitively, this operator represents a directed connection from ***Info*** to DisplayUnit (but not vice-versa). Since every piece of information belongs to a display unit, all transitions represent links between display units.

Display units can also be linked directly without a particular piece of information involved. In order to represent this kind of transition we have introduced the type ***null*** which denotes a piece of information with no ***Datum***.

Visualisation Description

Visualisation is described by the predicate ***style(Type, Style)***, which associates a data type to a particular pre-defined visualisation style. Examples of styles ***include text, table, enumerated_list, itemised_list***. By associating visualisation styles to data types, visualisation description becomes completely independent from any particular application. Styles are related to specific pieces of code written in a target language, such as HTML. The option to use the data type instead of the piece of information label is to promote an uniform presentation style to all pieces of information of the same data type.

In order to synthesise a complete Web page, generic templates should also be provided. Other details such as font type and size, colours, text margins and alignment are defined in stylesheets. A suitable interface should be offered to the designer in order to input all necessary parameters to the page templates and the related stylesheet.

Currently, a standard CSS stylesheet is automatically generated including definitions provided by the designer. The reason to make use of stylesheets is to keep the representation for our visualisation styles simple. Without a stylesheet, all visual details would have to be included as arguments to the mapping procedure which translates a visualisation style to HTML.

Changes in the visualisation can be done by different means:

- changing a style of a data type (this will affect all pieces of information of that data type);
- changing a stylesheet definition;
- changing page templates.

More importantly, changes in visualisation specifications does not affect content or navigation specifications.

Web Sites as Hypergraphs

A Web site will be defined by its individual pieces of information, the display units and the transitions using the $\Rightarrow$ operator. These components are notational devices to define a special kind of directed hypergraphs (Gallo & Scutell, 1999). The pieces of information comprise the nodes of our hypergraph; the hyperedges are defined by display units and transitions.

Formally, a hypergraph is a pair $\boldsymbol{H = (V, E)}$ where $\boldsymbol{V = \{v_1, ..., v_n\}}$ is the set of nodes and $\boldsymbol{E = \{e_1, ..., e_m\}}$ is the set of hyperedges. In our Web sites, each node $\boldsymbol{v_i}$ is equated with a piece of information. Each edge is a pair $\boldsymbol{e = (h_e, T_e)}$ with $\boldsymbol{T_e \subseteq V}$ and $\boldsymbol{h_e \in V - T_e}$. The display units comprise the sets $\boldsymbol{T_e}$ that can appear in edges. Clearly, if there is a transition ***Info*** $\Rightarrow$ ***DisplayUnit*** in our web site specification, then (***Info, DisplayUnit***) defines an edge.

For example, a research group Web site might have a structure including pages for people, research projects and publications. The transition rules in Figure 4 represent such research group Web site.

```
display([welcome_text(text, WT), contact_details(text ,CD),
    people_link(null, _), research_link(null, _)]) ←
    get_welcome_text(WT) ∧ get_contact_details(CD)

display([people_link(null, _)]) ⇒ display([current_members(list, CM),
    previous_members(list, PM), publication_link(null, _)]) ←
    get_current_members(CM) ∧ get_previous_members(PM)

display([research_link(null, _)]) ⇒ display([project_titles(list, PT)]) ←
    get_project_titles(PT)

display([project_titles(list, PT)]) ⇒ display([project_title(string, T),
    project_abstract(text, A), people_involved(list, PI),
    people_link(null, _), publication_link(null, _)]) ←
    T ∈ PT ∧ get_project_abstract(T, A) ∧
    get_people_involved(T, PI)

display([publication_link(null, _)]) ⇒
    display([publication_list(table, PL)]) ←
    get_publication_list(PL)
```

Figure 4: Example of transition rules for a research group Web site

Figure 5 illustrates the units of display and the navigation structure of the site, corresponding to the transition rules given in Figure 4. The dashed arrows represent links from multiple units of display derived from the same specification as in the transition rule from ***project_titles*** to individual display units for each project.

Paths represent sequences of pieces of information and display units. This is where constraints can be imposed in order to ensure a particular order for presenting information. The hypergraph representation supports reasoning on its structure allowing different properties to be checked, for example, inspecting connectivity or topological features.

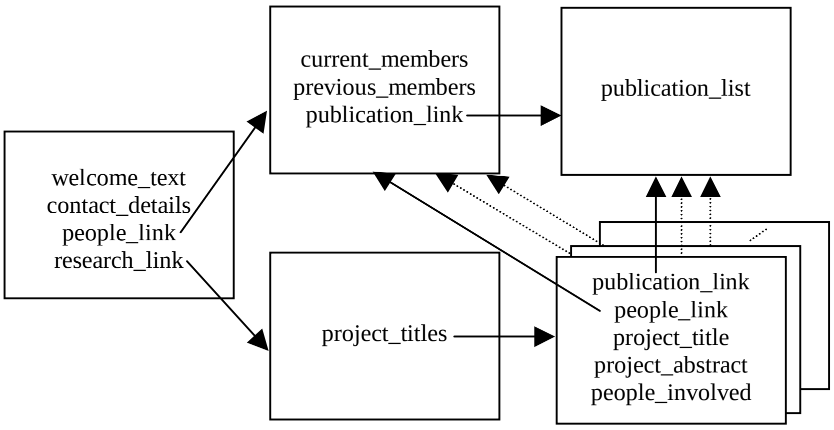


Figure 5: A research group Web site structure

Paths can be written either in terms of display units or pieces of information. Using display units, the number of nodes is reduced. We keep these simpler notations for paths as the original concept of a path within an hypergraph subsumes both. In section 3 the use of paths for specifying and verifying constraints is discussed.

Formally, a path $\boldsymbol{P_{ab}}$ from a piece of information $\boldsymbol{a}$ to $\boldsymbol{b}$ is a sequence $\boldsymbol{P_{ab} = (v_1 = a, e_1, v_2, e_2, ..., e_q, v_{q+1} = b\})}$ where $\boldsymbol{he_1 = a}$ and $\boldsymbol{he_i \in Te_{i-1}, i = 2, ...,q.}$

Although graphs are very useful as an underlying formalism to represent and prove properties of Web sites, this is not a scalable solution. The exponential complexity of computing time to derive paths or to check connectivity between nodes makes this approach prohibitive for extremely large graphs. Nevertheless, it is a practical approach for the Web site domain, given that the size of graphs derived from Web sites usually is not prohibitively large.

Synthesis

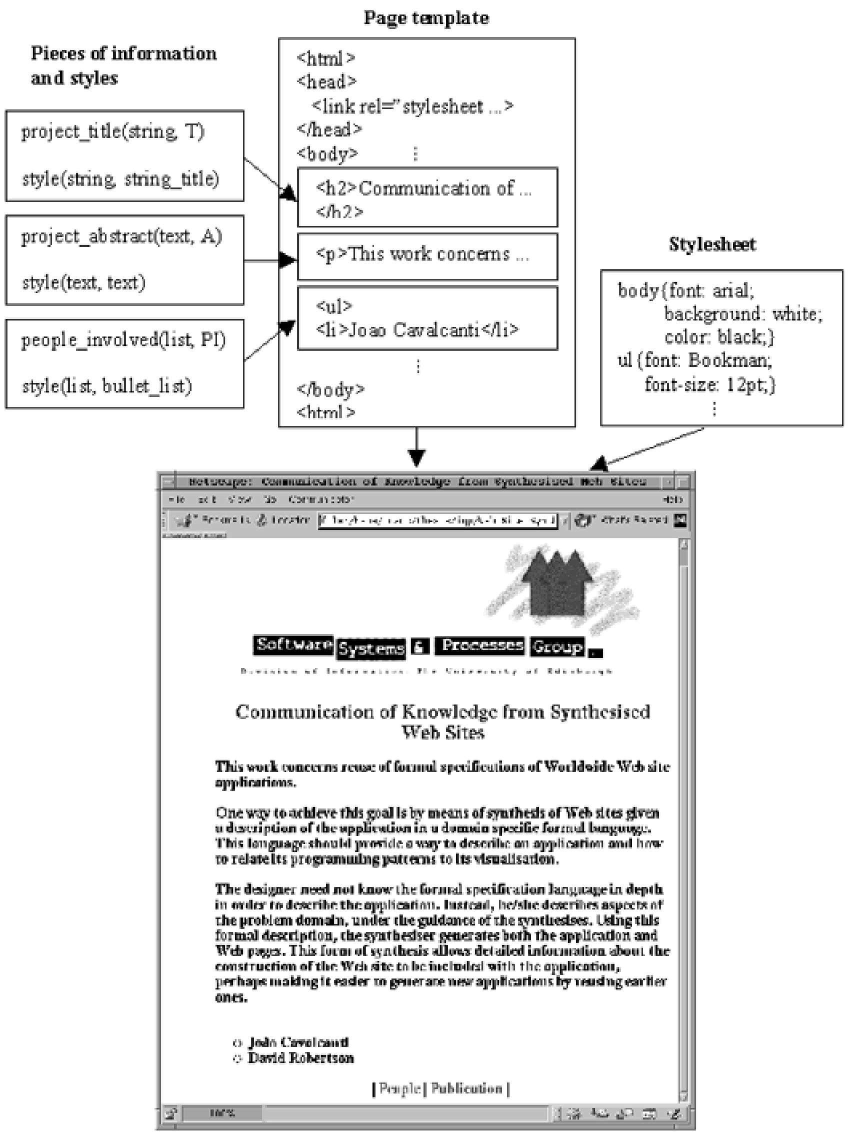
In order to illustrate the synthesis process we show a simple page generation from the following specification extracted from the transition rules presented in Figure 4.

```
display([project_titles(list, PT)]) ⇒ display([project_title(string, T),
        project_abstract(text, A), people_involved(list, PI),
        people_link(null, _), publication_link(null, _)]) ←
        T ∈ PT ∧ get_project_abstract(T, A) ∧ get_people_involved(T, PI)
```

A possible visualisation description for the types of pieces of information ***project_title, project_abstract*** and ***people_involved*** is: ***style(string, title_string), style(text, text), style(list, bullet_list)***.

The synthesis process combines all definitions given above, instantiating pieces of information and by using visualisation style definitions, transforming them into Web site code in a target language, such as HTML. Figure 6 illustrates this process showing a possible rendering in HTML for the transition rule presented above.

Our synthesiser is implemented in Sicstus Prolog using the Pillow library (Cabeza & Hermenegildo, 1997), which provides facilities to convert Prolog terms into HTML.


Page template
Pieces of information and styles
project_title(string, T)
style(string, string_title)
project_abstract(text, A)
style(text, text)
people_involved(list, PI)
style(list, bullet_list)
Stylesheet

```
<html>
<head>
  <link rel="stylesheet ...>
</head>
<body>      ⋮
  <h2>Communication of ...
  </h2>
  <p>This work concerns ...
  <ul>
  <li>Joao Cavalcanti</li>
          ⋮
</body>
<html>
```

```
body{font: arial;
       background: white;
       color: black;}
ul{font: Bookman;
    font-size: 12pt;}
         ⋮
```

Software Systems & Processes Group

Division of Informatics, The University of Edinburgh

Communication of Knowledge from Synthesised Web Sites

This work concerns reuse of formal specifications of Worldwide Web site applications.

One way to achieve this goal is by means of synthesis of Web sites given a description of the application in a domain specific formal language. This language should provide a way to describe an application and how to relate its programming patterns to its visualisation.

The designer need not know the formal specification language in depth in order to describe the application. Instead, he/she describes aspects of the problem domain, under the guidance of the synthesiser. Using this formal description, the synthesiser generates both the application and Web pages. This form of synthesis allows detailed information about the construction of the Web site to be included with the application, perhaps making it easier to generate new applications by reusing earlier ones.

- João Cavalcanti
- David Robertson

| People | Publication |

Figure 6: Example of a Web page synthesis

ENFORCING CONSTRAINTS

One kind of constraint is 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 *immediately after* checkout.

In order to specify constraints in the order of information display, we use two concepts from Transaction Logic (TR) (Bonner & Kifer, 1995), 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 $\boldsymbol{a} \otimes \boldsymbol{b}$ 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. A finite number of paths can be derived following the rules:

1. all paths begin in the same display unit (denoted as the *homepage*);
2. no display unit can be revisited;
3. all paths end in a “leave” display unit, i.e. display units that have no outgoing links.

For example, from the graph presented in Figure 4 the following paths can be derived:

p_1: ***display(homepage)*** $\otimes$ ***display(people)*** $\otimes$ ***display(publications)***

p_2: ***display(homepage)*** $\otimes$ ***display(research)*** $\otimes$ ***display(project)*** $\otimes$ ***display(publications)***

p_3: ***display(homepage)*** $\otimes$ ***display(research)*** $\otimes$ ***display(project)*** $\otimes$ ***display(people)*** $\otimes$ ***display(publications)***

Constraint expressions are similar to path expressions, basically imposing an order to the presentation of information. Another useful concept borrowed 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, if we want the piece of information ***welcome_text*** to be displayed before ***project_title***, the following constraint may be defined:

c_1: $\neg$ ***(path*** $\otimes \neg$ ***display(welcome_text)*** $\otimes$ ***path*** $\otimes$ ***display(project_title)*** $\otimes$ ***path)***

Note that our path expressions should be defined in terms of display units. Since constraint expressions are usually defined for pieces of information, it requires an appropriate membership checking between pieces of information and display units in order to translate one expression into another. Given that a piece of information may appear in more than one display unit, more than one constraint expression based on display units can be derived from an expression based on pieces of information. For example, the expressions above correspond to:

c_1': ¬ (path ⊗ ¬ display(homepage) ⊗ path ⊗ display(project) ⊗ path)

The following table presents some common constraint expressions:

Expression	Interpretation
¬ (path ⊗ ¬ display($Info_1$) ⊗ path ⊗ display($Info_2$) ⊗ path)	Information in $Info_1$ must be displayed before information in $Info_2$.
¬ (path ⊗ ¬ display($Info_1$) ⊗ display($Info_2$) ⊗ path)	Information in $Info_1$ must be displayed immediately before information in $Info_2$.
¬ (path ⊗ display($Info_1$) ⊗ path ⊗ ¬ display($Info_2$) ⊗ path)	Information in $Info_2$ must be displayed after information in $Info_1$.
¬ (path ⊗ display($Info_1$) ⊗ ¬ display($Info_2$) ⊗ path)	Information in $Info_2$ must be displayed immediately after information in $Info_1$.

Data types and values of pieces of information are omitted in path and constraint expressions because they play no role in this kind of constraint checking. As a result constraint and path expressions are simplified, including only information labels.

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, considering the paths and the constraint above, it is possible to conclude that constraint ***c_1'*** is satisfied. Note that the negation in the constraint means that paths which have that pattern are not valid. As none of the paths ***p_i*** can match the pattern, they are all valid.

On the other hand, the following constraint

c_2: ¬ (path ⊗ ¬ display(current_members) ⊗ path ⊗ display(people_involved) ⊗ path)

which maps to

c_2': ¬ (path ⊗ ¬ display(people) ⊗ path ⊗ display(project) ⊗ path)

is not satisfied because path p_3 matches the constraint pattern. Informally, it can be verified that in that path, display unit project is displayed without people being displayed before it.

VERIFYING PROPERTIES

Our intermediate representation comprises a declarative specification of pieces of information and navigation structure (transition rules). This logic specification provides a basis for proving different kinds of properties. Examples of interesting properties of a Web site application are reachability of information, broken links checking and a site navigator.

Information Reachability

If the specification of a Web site involves a large number of pieces of information, an automated check ensuring that a particular information can be reached from the homepage is very useful. This verification can be defined as follows: a given piece of information, ***I***, it is reachable from the homepage if: it is displayed in the homepage itself; it is displayed in a unit immediately linked from the homepage; or it is displayed in a unit which is linked from a reachable unit of display.

Formally, this verification is defined by the following rules:

homepage(H) ← display(H)

reachable(I) ← homepage(H) ∧ I ∈ H

$$\textbf{\textit{reachable(I)}} \leftarrow \textbf{\textit{homepage(H)}} \wedge \textbf{\textit{L}} \in \textbf{\textit{H}} \wedge \textbf{\textit{display([L])}} \Rightarrow \textbf{\textit{display(S)}} \wedge \textbf{\textit{I}} \in \textbf{\textit{S}}$$

$$\textbf{\textit{reachable(I)}} \leftarrow \textbf{\textit{display([IL])}} \Rightarrow \textbf{\textit{display(S)}} \wedge \textbf{\textit{I}} \in \textbf{\textit{S}} \wedge \textbf{\textit{reachable(IL).}}$$

Site Navigator

One example of a Web site property related to the topology of the site graph is the *navigator*. In the vast majority of Web site applications there is a set of pages which should be referenced by all the others. In most cases it is appropriate that links to these pages are grouped together and presented in a standard form in every page of the site. We call this set of links a navigator.

The navigator can be derived automatically by inspecting the topological features of the Web site graph. The simplest case is that of a page which is linked from all other pages of the site. The cardinality of its neighbours set and intersections between other pages neighbours set can help in deciding if a page should be included in the navigator set. Two concepts are used:

1. Backlink counter ***ib(P)***. This refers to the number of links to a page ***P***.
2. Page rank backlink ***ir(P)***. This refers to the weighted sum of backlinks to a particular page ***P***. Formally, the more pages ***P'*** that links to ***P*** having themselves a high ***ib(P')***, greater is the importance of ***P***.

By defining appropriate thresholds to ***ir(P)*** a navigator for the Web site can be derived automatically. For example, using the site representation presented in Figure 4 and supposing we have added a link from each display unit back to the homepage, the backlink counters are ***ib(home)* = 4, *ib(people)* = 2, *ib(publications)* = 2, *ib(research)* = 1 *and* *ib(project)* = 1.**

Note that project actually represents a set of pages with same structure and for that reason it is not necessary to calculate a counter for each page individually.

The page rank backlinks are ***ir(home)* = 6, *ir(people)* = 5, *ir(publications)* = 3, *ir(research)}* = 4 *and ir(project)* = 1**.

In this simple example, if a threshold for ***ir*** is 4, then our navigator includes home, people and research.

Broken Link Checking

Our approach to Web site synthesis makes a distinction between internal and external links. Checking for broken links apply to external links once all internal links are already defined in the transition rules.

In order to facilitate this task a list with all external links can be extracted from the pieces of information definitions and kept in a list. A simple recursion over this list can provide a list containing all broken links:

```
broken([ ], [ ])
broken([Link | MoreLinks], BrokenLinks) ←
        ¬  is_broken(Link)  ∧  broken(MoreLinks, BrokenLinks)
broken([Link | MoreLinks], [Link | MoreBrokenLinks) ←
        is_broken(Link) ∧ broken(MoreLinks, MoreBrokenLinks)
```

Although this is a trivial task it can be very useful for maintenance purposes. It is also possible to automate this task, executing it periodically. As a result maintenance time is reduced. This is an additional facility that illustrates one of the many possible forms of property verification.

FUTURE TRENDS

Given the flexibility and versatility of this logic-based approach, we can envisage its employment specially in knowledge intensive applications. Tools for design, implementation and maintenance of Web-based information systems can benefit from this approach by deploying a computational logic component. Most Prolog systems currently available supports integration with relational databases systems, other programming languages, concurrent programming (threads) and Internet programming. These features combined with the improved performance of modern logic programming systems makes the use of computational logic components viable for commercial applications.

Dynamic pages, i.e. pages that are generated automatically as a result of user interaction, should be considered as an extension of this work. Assuming that the program that generates the pages is correct, an interesting point is what impact this feature causes on the property and constraint verification as presented in this Chapter. We have already carried out some experiments on supporting dynamic Web pages using computational logic (Vasconcelos, et al., 2000).

An interesting issue for further investigation is the role of computational logic in content management systems. There is a need for better and more reasoning capabilities in these systems, as for example user personalisation, finding appropriate relationships between pieces of information and cross-referencing between information content and information providers (people who provides information).

CONCLUSIONS

Computational logic can be applied successfully to specify, synthesise and reason about Web site applications. The approach presented in this Chapter showed that a declarative specification provides a basis for a variety of constraint and property checks. Among the benefits of this approach is the systematic development of a Web site application that joins different levels of description to produce Web sites consistent with a corresponding high level description. As a result, design becomes more methodical and maintenance less time consuming.

This approach has the following main features: declarative specification of an application based on logic; separation between information content, navigation structure and visualisation specification; and automated generation of Web site code from a high level specification.

From the navigation structure description, a site graph and a navigator can be derived. Properties can be checked on the site's intermediate representation. Constraints in the order of information presentation can be enforced by inspecting paths derived from the site graph. The use of logic for both specification and verification of constraints and properties makes this approach very flexible. It supports the addition of a large number of different property and constraint checks without changing the original specification.

Another important feature is that any verification is done on the site specification instead of a particular implementation. This ensures that regardless of the target language used to implement the Web site application all constraints and properties hold. Changes in the visualisation also causes no impact on any constraint or property of navigation structure or information content.

The modular architecture of our synthesiser supports extensions in terms of the conceptual data model used to describe a Web site (other than ER) as well as generation of Web site code in different target languages such as XML schemas, XSL and WML.

Acknowledgements

The first author is on leave from the University of Amazonas (Brazil) and is supported by the Brazilian Government through CAPES grant no. 1991/97-3. This work is also supported by the UK Engineering and Physical Sciences Research Council on the Advanced Knowledge Technologies Interdisciplinary Research Collaboration and on the research grant: “Communication of Knowledge from Synthesised Web Sites”.

References

Apt, K.R. (1997). *From logic programming to Prolog*. Prentice-Hall.

Atzeni, P., Mecca, G. & Merialdo, P. (1998). Design and maintenance of data-intensive Web sites. In proceedings of the international conference on extending database technology (EDBT). Valencia, Spain.

Bonner, A.J. & Kifer, M. (1995, November). Transaction logic programming. Technical report CSRI-323. Computer Systems Research Institute, University of Toronto, Canada.

Cabeza, D. & Hermenegildo, M. (1997). WWW programming using computational logic systems. In proceedings of the workshop on logic programming and the WWW. The WWW6 conference. San Francisco, CA, USA.

Cavalcanti, J. & Robertson, D. (2001). Synthesis of Web sites from high level descriptions. In S. Murugesan & Y. Deshpande (Eds.), Web engineering: Managing diversity and complexity in Web application development. Lecture notes in computer science, 2016. Berlin: Springer-Verlag.

Cavalcanti, J. & Robertson, D. (2002). Synthesis of Web sites based on computational logic. To appear in *Knowledge and Information Systems Journal (KAIS).*

Ceri, S., Fraternali, P. & Bongio, A. (1999, May). Web modeling language (WebML): A modeling language for designing Web sites. In proceedings of the WWW8 conference. Toronto, Canada.

Ceri, S. Fraternali, P. & Paraboschi, S. (1999, September). Data-driven, one-to-one Web site generation for data-intensive applications. In proceedings of 25th international conference on very large data bases (VLDB'99), pp. 615-626. Edinburgh, Scotland, UK.

Elmasri, R. & Navathe, S. (1999). *Fundamentals of Database Systems*, 3rd edition. Benjamin Cummings.

Fernández, M., Florescu, D., Kang, J., Levy, A. & Suciu, D. (1998). Catching the boat with Strudel: Experience with a Web-site management system. In proceedings of SIGMOD conference on management of data. Seattle, WA, USA.

Fernández, M., Florescu, D., Levy, A. & Suciu, D. (1999). Verifying integrity constraints on Web-sites. In proceedings of the 16th international joint conference on artificial intelligence (IJCAI). Stockholm, Sweden.

Florescu, D., Levy, A. & Mendelzon, A. (1998). Database techniques for the world-wide Web: A survey. *SIGMOD Record*, 27(3).

Fraternali, P. & Paolini, P. (1998). A conceptual model and a tool environment for developing more scalable, dynamic and customisable Web applications. In proceedings of the international conference on extending database technology (EDBT). Valencia, Spain.

Fuchs, N.E. (1992). Specifications are (preferably) executable. *Software engineering journal,* 7 (5), 323-334.

Gallo, G. & Scutell, M. (1999). Directed hypergraphs as a modelling paradigm. Technical report TR-99-02. Dipartimento di Informatica, Università di Pisa. Italy.

Jin, Y., Decker, S. & Wiederhold, G. (2001, July). OntoWebber: Model-driven ontology-based Web site management. In proceedings of the first international semantic Web working symposium (SWWS'01), Stanford University. Stanford, CA, USA.

Maedche, A., Staab, S., Stojanovic, N., Studer, R. & Sure, Y. (2001, July). SEAL - A framework for developing semantic Web portals. In proceedings of the 18th British national conference on databases (BNCOD 2001). Oxford, England, UK.

Proietti, M. & Pettorossi, A. (1994). Transformations of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19(20), 261-320.

Robertson, D. & Augustì, J. (1999). *Software blueprints: Lightweight uses of logic in conceptual modelling*. ACM Press, Addison Wesley Longman.

Schwabe, D. & Rossi, G. (1995). The object-oriented hypermedia design model. *Communications of the ACM*, 38(8), 45-46.

Vasconcelos, W., Schwitter, R., Molla, D., & Cavalcanti, J. (2000, October). Implementing prolog-run WWW sites. In the proceedings of the 13th international conference on applications of Prolog (inap). Waseda University. Tokyo, Japan.