A Computational Logic Approach for Web Site Synthesis and Management

João Cavalcanti

Doctor of Philosophy
Centre for Intelligent Systems and their Applications
Division of Informatics
University of Edinburgh
2003
Abstract

This thesis concerns the development of an approach for Web site synthesis. Previous work has demonstrated the feasibility of domain-specific methods for Web site synthesis using computational logic. We are particularly interested in extending this idea, investigating domain-independent methods whilst still being able to produce practical Web site applications. The main contribution of this thesis is to propose a design approach that joins different levels of description to produce Web sites consistent with a corresponding high level description.

It is now widely accepted that the separation of content, navigation structure and visualisation is a good Web engineering practice. This feature promotes a better understanding of the requirements of the application and facilitates maintenance as each component can be changed often with little or no impact on the other components. In addition, methods for Web site design and construction should support declarative specifications, integrity constraint verification and automated or semi-automated generation of Web site by means of CASE tools.

Computational logic is particularly suited to this task given its natural support for declarative specifications and automated reasoning. Content, navigation and visualisation specifications are all defined under the same formalism as well as programs for synthesis and integrity constraint verification. As a result, the integration of specification and program is performed more naturally than with procedural methods.

Our approach is based on a three-level architecture, composed of a high-level specification, intermediate representation and target language. The high-level specification corresponds to the description of a Web site application. We use an entity-relationship model for this purpose and a mapping procedure derives a corresponding intermediate representation automatically. The intermediate representation describes display units, which are sets of pieces of information, and how to navigate among them. A separate visualisation description relates each piece of information to a particular presentation style and templates to display units. From the intermediate representation and visualisation description a complete Web site is generated automatically. Experiments with HTML/CSS, XML/XSL and WML as target languages have been successfully carried out.
The core of our approach is the intermediate representation. An important feature is its independence from any particular implementation, making this approach very flexible. It forms a basis for different kinds of reasoning about the application, including property and constraint checking. It also supports definitions of logic-based agents that are constructed as part of the synthesis of the Web site specification and can be employed to automate the maintenance of parts of the site. Data-intensive Web sites, such as Web portals and e-commerce sites can all benefit from this development approach as it makes design more methodical and maintenance less time consuming.
Acknowledgements

First I would like to dedicate this thesis to my wife Vilza, who gave me full support and lived through this PhD experience with me.

Big thanks to my supervisor Dave Robertson who was always there to help with insightful advice and suggestions which made a great difference to my work. Also a big thank-you to Wamberto Vasconcelos, my second supervisor, for all support and comments on my work.

Many friends I met during the past four years made my stay in Edinburgh a wonderful time. Alberto Castro, Marco Carvalho, Márcio and Ana Brandão, Daniela Carbogim and José Pizani, Jason Baldridge and Fernanda Aranha, Virginia Biris-Brilhante and Elias Biris, Bob and Jill Knox, Beatriz Paiva and Colin Grant, Gisela and Roney, Clauirton Siebra and Natasha Lino, Marcelo and Leslie, “Cheers” to you all.

A special thanks to my friends of the SSP Group: Chris Lin, Marco Schorlemmer, Stefan Daume, Yannis Kalfoglou, Renaud Lecoeuche, Steve Polyak, Siu-Wai Leung, Chris Walton, Jessica Chen-Burger. I couldn’t have asked for a better group.

Finally, thanks to my colleagues from the Department of Computer Science of the Federal University of Amazonas for taking care of it while I was away. A special thanks to Edjard Mota for being a great landlord. I also would like to acknowledge the support of CAPES, under grant no 1991/97-3.
Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.
# Table of Contents

1 Introduction and Background  

1 Motivation and Context  
1.1 What is the Problem with Web Site Development  
1.2 What is the Problem with Web Site Maintenance  
1.3 Our Hypothesis  
1.4 Problems Addressed in this Thesis  
   1.4.1 Thesis Organisation

2 Concepts and Terminology  
2.1 Approaches to Web site design and construction  
   2.1.1 Declarative Specifications  
   2.1.2 Information Content, Navigation Structure and Visualisation  
2.2 Automated Synthesis  
2.3 The Role of Computational Logic in Web Site Synthesis  
2.4 Specific questions addressed by this Thesis

3 Related Work  
3.1 Methods for Web Site Construction  
3.2 Logic Programming and the Internet  
3.3 Ontologies and the Semantic Web  
3.4 Maintenance and Property Verification  
3.5 Discussion
## An Approach for Web Site Design and Implementation

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Web Site Synthesis: System Architecture</td>
<td>37</td>
</tr>
<tr>
<td>4.1</td>
<td>A 3-level Architecture for Web Site Synthesis</td>
<td>38</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Level 1 - Conceptual Design</td>
<td>40</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Level 2 - Intermediate Level</td>
<td>41</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Level 3 - Implementation</td>
<td>42</td>
</tr>
<tr>
<td>4.2</td>
<td>Web Design Issues</td>
<td>43</td>
</tr>
<tr>
<td>4.3</td>
<td>Summary</td>
<td>44</td>
</tr>
<tr>
<td>5</td>
<td>High-Level Specification of Web Site Applications</td>
<td>47</td>
</tr>
<tr>
<td>5.1</td>
<td>High-level Description of Web Sites</td>
<td>48</td>
</tr>
<tr>
<td>5.2</td>
<td>Application Description using an Entity-Relationship Model</td>
<td>50</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Working Example</td>
<td>51</td>
</tr>
<tr>
<td>5.3</td>
<td>Deriving Intermediate Representations from ER</td>
<td>53</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Representing ER Schemas in First-order Logic</td>
<td>54</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Entity-based Intermediate Representation</td>
<td>56</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Instance-based Intermediate Representation</td>
<td>58</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Query-based Intermediate Representation</td>
<td>64</td>
</tr>
<tr>
<td>5.4</td>
<td>Summary</td>
<td>66</td>
</tr>
<tr>
<td>6</td>
<td>The Intermediate Representation</td>
<td>67</td>
</tr>
<tr>
<td>6.1</td>
<td>Pieces of Information</td>
<td>68</td>
</tr>
<tr>
<td>6.2</td>
<td>Display Units</td>
<td>71</td>
</tr>
<tr>
<td>6.3</td>
<td>Operations</td>
<td>72</td>
</tr>
<tr>
<td>6.4</td>
<td>Transitions</td>
<td>78</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Interpretation of Transition Rules</td>
<td>80</td>
</tr>
<tr>
<td>6.5</td>
<td>Summary</td>
<td>83</td>
</tr>
<tr>
<td>7</td>
<td>Visualisation</td>
<td>85</td>
</tr>
<tr>
<td>7.1</td>
<td>Styles for Pieces of Information</td>
<td>86</td>
</tr>
<tr>
<td>7.2</td>
<td>Page Templates</td>
<td>89</td>
</tr>
</tbody>
</table>
10.2 Using Agents for Web Site Maintenance ................. 144
10.2.1 A Multi-Agent System Architecture for Web Site Maintenance 147
10.3 Example of Maintenance Agents .......................... 150
10.4 Summary ................................................. 153

IV Discussion and Prospects ................................. 155

11 Evaluation .................................................. 157
11.1 Evaluation: Number of Specifications V Number of Visualisations 158
11.1.1 Specifying Large Commercial Web Sites ............... 165
11.2 Support for Web Site Synthesis Features ................. 169

12 Conclusion .................................................. 173
12.1 What We Have Achieved .................................. 174
12.2 Continuing our Work .................................... 177
12.2.1 Supporting other High-Level Specifications ............ 177
12.2.2 Operations ............................................. 178
12.2.3 Personalisation ....................................... 179
12.2.4 Using Logic in Content Management Systems ........ 179

A Accident Reporting Web Site .............................. 181

B SSP Web Site ............................................... 185

C Synthesiser .................................................. 189

D An Implementation of a CGI-Prolog Interface ............. 191

E TeLA - A Temporal Logic for Agent Specification and Execution 197
E.1 TeLA Syntax .............................................. 197
E.2 TeLA Semantics .......................................... 198
E.3 A Meta-Interpreter for TeLA .................. 200
E.3.1 Reenactment .................................. 203
# List of Figures

2.1 A general Architecture for CMS ........................................... 16  
2.2 Example of parameterisable components .......................... 19  

4.1 Synthesiser architecture options ........................................ 37  
4.2 The 3-level approach ......................................................... 39  
4.3 Supporting multiple source and output ............................. 46  

5.1 Accident Reporting ER Diagram .......................................... 52  
5.2 Definition of pieces of information corresponding to entities .......... 57  
5.3 Definition of display units corresponding to entities ............. 58  
5.4 Transition rules for binary relationships 1:1 and 1:N ........... 59  
5.5 Binary relationship N:N and its transition rules ................ 60  
5.6 Transitions derived from generalisation hierarchies ............ 61  
5.7 Information Structure - Entity-based ................................ 62  
5.8 Definition of pieces of information corresponding to entities and relationships N:N ................................................. 62  
5.9 Transition rules for binary relationships 1:1 and 1:N ........... 63  
5.10 Transition rules for binary relationships N:N .................... 63  

6.1 Grammar for Pieces of Information ..................................... 69  
6.2 Definition of pieces of information .................................... 70  
6.3 Pieces of Information and Display Units ............................ 71  
6.4 Example of an operation ................................................... 75  
6.5 Sample Transitions .......................................................... 79  
6.6 A Web site example .......................................................... 80
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1</td>
<td>A Research Group Web Site Organisation</td>
<td>185</td>
</tr>
<tr>
<td>D.1</td>
<td>CGI-Prolog Interface</td>
<td>191</td>
</tr>
<tr>
<td>D.2</td>
<td>Shopping List Application</td>
<td>196</td>
</tr>
<tr>
<td>E.1</td>
<td>Context-Free Grammar for TeLA</td>
<td>198</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th>3.1</th>
<th>Features of approaches to Web site synthesis</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Architecture levels and description components relation</td>
<td>45</td>
</tr>
<tr>
<td>5.1</td>
<td>Concept correspondence between data models and Web sites</td>
<td>49</td>
</tr>
<tr>
<td>5.2</td>
<td>Aircraft table definition</td>
<td>53</td>
</tr>
<tr>
<td>5.3</td>
<td>Flight table definition</td>
<td>53</td>
</tr>
<tr>
<td>6.1</td>
<td>Components for defining operations</td>
<td>74</td>
</tr>
<tr>
<td>8.1</td>
<td>Example of aircraft data</td>
<td>100</td>
</tr>
<tr>
<td>8.2</td>
<td>Example of incident reports data</td>
<td>100</td>
</tr>
<tr>
<td>8.3</td>
<td>Example of incident data</td>
<td>101</td>
</tr>
<tr>
<td>9.1</td>
<td>Common constraint expressions</td>
<td>137</td>
</tr>
<tr>
<td>11.1</td>
<td>Compatibility between data types and visualisation styles</td>
<td>162</td>
</tr>
</tbody>
</table>
Part I

Introduction and Background
Chapter 1

Motivation and Context

This thesis concerns the development of an approach for Web site construction and maintenance using computational logic. The proposed approach includes property and constraints verification and logic-based agents that are constructed as part of the synthesis of the Web site specification and can be employed to automate the maintenance of parts of a Web site.

This thesis is directly related to the emerging area of Web engineering, which concerns disciplined and systematic approaches to development, deployment and maintenance of Web-based systems and applications [Ginige and Murugesan, 2001]. In recent years Web engineering has attracted the attention of researchers from different areas such as software engineering, hypermedia, human-computer interaction, among others. However, few have addressed the problems related to Web site management using logics. This thesis presents a novel way of automating Web site construction, as logic is used for specification, property verification, synthesis and the maintenance processes.

A motivation for this work is the fact that computational logic has not been well exploited to address the problem of Web site specification and construction. Logics provide a high-level and abstract approach whereby unimportant implementational details can be conveniently postponed until later stages of development. This allows the designer to focus on the requirement analysis and design of the application, rather than on the mechanics of producing the Web site.

As logic programming is intrinsically declarative, it supports both applica-
tion 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 appropriate for automated Web site synthesis [Robertson and Agustí, 1999].

1.1 What is the Problem with Web Site Development?

The Web has become a standard media for publishing information. With the advent of e-commerce, the Web has also been used for the commercialisation products, attracting more attention from commercial organisations. This caused a very quick expansion in the use of the Web. As the volume and complexity of information presented on the Web increased, now some large Web sites contain thousands of pages and pieces of information. Web sites also often involve access to databases, complex cross referencing between information within the site and sophisticated user interaction. Web site development has become a challenging problem, demanding methods and tools for assisting their design, construction and maintenance.

Much attention has been given to the deployment of Web sites but little thought has been given to methods for their design and maintenance. Many Web developers often use ad hoc approaches without rigour or systematic techniques. This can cause serious problems specially for maintenance which only gets worse with scaling-up.

Recent technologies such as Java Server Pages - JSP [Hall, 2000], Active Server Pages - ASP [Weissinger, 2000] and XML [W3C, 2002a] are now being widely used in commercial applications and many regard them as the solutions for the problems mentioned above. The simple use of these technologies, however, does not solve problems such as high cost of prototyping, explicit page linking, difficulty on reasoning and extracting knowledge from the application specification and the need for the Web designer to deal directly with database access. They are in reality, tools upon which methodologies for Web site construction are built. Many of the proposed approaches for Web site construction discussed in Chapter 3 make use of one or more of these technologies.

Many Web sites have out-of-date content because it is too time-consuming, com-
1.1. What is the Problem with Web Site Development?

Most Web tools are designed for technical people, not average users. Organisations may have to rely on non-technical staff to add and update content on their Web sites in order to solve the bottleneck problem with their IT departments. This poses a challenging problem for the research and development of methods and tools that solve those problems whilst allowing their use by clerical staff.

This is particularly true for data-intensive Web sites. These sites are subject to constant updates, raising the cost of site maintenance. As this cost usually is recurrent, a design method that facilitates maintenance is valuable. This is the sort of Web site we have in mind for our approach to Web site synthesis.

It is important to make a distinction between Web site applications appropriate to our style of generation and others which for our approach is not appropriate. For example, others view Web design as an authoring problem rather than an application development. This makes the designer concentrate on aesthetics and visualisation design instead of application requirements and maintainability. Although for some sorts of application visualisation is the main concern, this thesis is not concerned with visual design or aesthetics of Web sites. The approach proposed here is content-driven, where the visual design of Web pages is not directly assisted by our method. Nevertheless, the proposed approach offers facilities for generating multiple visualisations and maintenance of the presentation of Web sites.

Another type of Web site application are those with high rates of access, possibly receiving hundreds of thousands of hits per minute. For these sites continuous availability and high performance are the critical factors that drive the development process [Challenger et al., 2001]. Examples of such Web sites are the 2000 Olympic Games, the 2002 World Cup and the Grand Slam tennis tournaments. This is also not the sort of Web site application our approach targets, because their design is primarily driven by performance constraints, having different requirements compared to other types of Web sites.

Traditional software engineering methods for software development are not well-suited for Web site applications, though many of them can be adapted in some way to address this problem. The main reasons are [Murugesan et al., 2001]:

- Web-based systems are document-oriented, containing static or dynamic Web
Chapter 1. Motivation and Context

pages.

- Web site applications usually involve multiple and heterogeneous information sources. Some of these sources can include non-structured information, such as simple files and text.

- The nature and characteristics of the Web as an application medium are different from traditional software environments. Most Web-based applications are viewed by means of browsers, which have standard presentation and capabilities.

- The individuals involved in the development of Web sites have distinct backgrounds and skills beyond computing.

- One of the main requirements for a successful Web site is that it should aesthetically pleasing, emphasising visual creativity and use of multimedia elements for the presentation of information.

It is possible that some of these features are also found in other software systems. However, the combination of all of these features is very unlikely to be found in a single software system other than Web-based systems.

Clearly, there is a demand for methods and tools for Web site development that take into account the unique features of the Web, the operational environment of Web-based systems, the different classes of users and the diverse skills and knowledge of people involved in creating and maintaining Web sites.

In our view, Web site applications can benefit from systematic approaches to development that make design more methodical and maintenance less time consuming. One way to tackle this problem is via automated synthesis, automatically deriving a Web site from a high-level application description [Cavalcanti and Robertson, 2000].

1.2 What is the Problem with Web Site Maintenance?

Maintenance involves a constant verification and evaluation of a Web site content, navigation, visual design and performance, making sure the site is technically ready
for visitors.

Keeping content fresh and up-to-date is the most common problem with Web sites. In addition new content must be added, mistakes must be corrected and links must be checked to ensure they work properly. Other common tasks faced by Webmasters are [Friedlein, 2000, Developers, 2002, Lowe and Arevalo-Lowe, 2002]:

- text and image update to a page
- page to be posted
- troubleshooting a problem with the sites functionality, which may require reprogramming
- new content to be formatted and added
- new functionality to be added to the site
- interface, programming or design changes

Performing these tasks manually is tedious and time-consuming. Our approach automates those maintenance tasks which are recurrent and can be described by parameters, as for example addition of new information, updating information and links verification.

Other maintenance problems are checking for browser compatibility, finding and fixing spelling, grammar, and coding errors. These specific problems are not addressed by our approach.

Our approach concerns maintenance of content, navigation and visualisation, although with a varying degree of support to each one of them. Content is maintained by updating the Web site database or by means of maintenance agents. Navigation maintenance requires modifying the transitions in the site specification and visualisation maintenance can be performed by updating pieces of information styles, page templates or page style sheets. Details on how our approach support these maintenance tasks are given in Chapters 7 and 10.

According to an interview with an experienced Webmaster [Journal, 1999], load time is the most common problem related to performance. It is critical to optimise
load time because sites that take more than 20 seconds to load may lose up to 50% of their visitors. Web sites that load much information and graphics on to each page make their usability and functionality difficult to understand. This might also affect the design of the Web site requiring maintenance on the site content, navigation and visualisation.

One of the key maintenance problems to the site's navigation structure is finding and fixing broken links [Developers, 2002, Lowe and Arevalo-Lowe, 2002]. These include not only links to other sites, but the links internal to the site which defines the navigation structure of the Web site. Links can be checked manually, however if the Web site is large with many internal and external links, this can be very time-consuming.

Our approach tackles this problem with two different approaches, by making a distinction between internal and external links. Internal links are guaranteed to be well-defined by the Web site specification. Property verification is also used as a means to ensure that a Web site is well-defined and free from navigational errors. This is explained in Chapter 9.

An automatic verification of external links is also provided by our approach. This procedure can be run from time to time and the resulting list of broken links is reported to the Webmaster. How often this check is performed depends on the frequency that the Web site is updated. For instance, in a human resource Web site links must be checked frequently because jobs are added several times a week. An educational site may only need to be checked for link problems each time a new course is included on the Web site.

Another issue related to navigation structure is keeping navigational consistency. If the navigation controls are not consistent throughout the site, visitors might become confused and lost [Friedlein, 2000]. This means that every Web page must use the same navigational controls, in the same place, and with the same graphics. For this aim our approach supports the notion of the site navigator, which is a set of internal links to a group of pages. This concept allows the designer to define a set of relevant links for composing the navigator and choosing a particular visualisation design which remains the same in any page of the site. More details about the site navigator is given
1.3. Our Hypothesis

in Chapter 9.

With respect to visualisation, consistency is one of the key issues for achieving usable interaction design. Users get very annoyed when they move between pages on a Web site and find drastically varying visualisation designs at every page. When all interface elements look and function the same, users feel more confident using the site because they can transfer their learning from one subsite to the next rather than having to learn everything all over again for each new page. Although our approach does not address this issue directly, a brief discussion on how to verify visualisation properties is presented in Chapter 9.

1.3 Our Hypothesis

The main contribution of this thesis is to propose a design approach that joins different levels of description to produce Web sites consistent with a corresponding high level description.

Our assumption is that by using computational logic for specifying and synthesising Web sites, the proposed approach will be very flexible, allowing a variety of property and constraint verification, generation of different visualisations for the same specification, support for multiple source of data and facilitating maintenance by automating repetitive tasks.

We believe that computational logic is well suited to tackle this problem because of its support to a uniform view of data and computation, allowing reasoning with both specification and program via meta-programming.

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 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
Chapter 1. Motivation and Context

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.

1.4 Problems Addressed in this Thesis

We want a design method that produces a Web site consistent with a specification and facilitates maintenance more generally. Updating data should become a clerical task and modifications in navigation structure or visualisation should not involve further programming, although inevitably it still requires some technical expertise. We also verify what relations can be drawn between the specification components and how the synthesis process benefits from the use of logic.

The main questions to be answered by this thesis are:

- What are the advantages and benefits of using computational logic to address the problem of Web site synthesis?
- How general is the proposed method for Web site synthesis and how does it compare to domain-specific methods?

1.4.1 Thesis Organisation

This thesis is organised as follows:

**Part I** Chapter 2 presents the main concepts and the terminology used in this thesis. Chapter 3 presents a discussion on related work.

**Part II** In this part the core of our approach for Web site synthesis is presented. In Chapter 4 we present the 3-level architecture of our Web site synthesis sys-
1.4. Problems Addressed in this Thesis

tem. Chapter 5 discusses the use of a conceptual data model for the high-level representation of Web sites and its subsequent translation into our logic representation. Chapter 6 presents how we represent content, navigation structure and visualisation of Web sites using logic. Chapter 7 discusses visualisation issues and how we represent the presentation styles of content and Web pages. Finally, Chapter 8 presents the details of the synthesis process.

Part III Here we present two additional features of our approach. Chapter 9 presents how we define and verify Web site properties and constraints. Chapter 10 presents an agent architecture used for automating some maintenance tasks, built on top of our synthesiser architecture.

Part IV In this final part we present in Chapter 11 an empirical evaluation of our method, highlighting the potential benefits of the proposed approach. Chapter 12 presents the final remarks and discusses some future work.
Chapter 2

Concepts and Terminology

When talking about Web sites, different terminology is used depending on the background of the people involved. For that reason, in this section we define some important concepts and the terminology which are referred to throughout this thesis.

A Web site is defined as 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, which are either based on a particular piece of information that links to another page or they are “free” links between pages (no particular piece of information is involved).

The term Web site application denotes a Web site which includes dynamic pages. These are pages generated on-the-fly, resulting from some computation which also involve some user interaction. We call these computation operations. Operations are usually implemented using technologies such as Java applets [Arnold et al., 2000], Active Server Pages (ASP) [Weissinger, 2000] or the Common Gateway Interface (CGI) programs [NCSA, 2002]. In this thesis we use the term Web site application when referring to this kind of Web site and simply Web site otherwise.

In contrast to dynamic pages we have static pages, which are regular pages not resulting from operations. These we also refer to simply as pages or Web pages. The term static Web site then, refers to a Web site in which all its pages are static and Dynamic Web sites are those in which some of its pages are dynamic.

We use the term transition to denote the existence of a direct connection between two pages. This can be done via a link or an operation. The set of transitions in a Web
site defines the *navigation structure* of the site. It defines all the possible paths that a user can follow within the Web site. The navigation structure is often represented as a graph, where nodes correspond to pages (or pieces of information) and edges correspond to transitions. This graph is also known as the *site graph*.

When the content of a Web site requires constant updates, this site is referred to as a *data-intensive Web site*. Another feature of this sort of Web site is the high workload of maintenance. Sometimes new requirements of data also force changes in visualisation, however in most cases content update has no impact on the visualisation style. This also illustrates the need for separating content from visualisation. The approach to Web site construction proposed in this thesis is particularly concerned with data-intensive Web sites. We address the problem of specification, automated construction and maintenance in a way that is most suitable for this sort of Web site, although it can also be successfully applied to other kinds of Web sites.

Finally we use the term *visualisation* to refer to all details concerning the visual layout of Web pages. It also refer to pieces of information styles and page style sheets. These define how a particular piece of information is rendered, its text font, size and colours among other details, such as graphics, logos and their arrangement on the pages.

### 2.1 Approaches to Web site design and construction

Depending on the approach used, the cost of designing and maintaining a Web site can vary. Without a systematic approach to Web site construction, a site developer performs this task by writing HTML files by hand (possibly using a structure editor). The effort to produce the site this way increases with the size and complexity of the application. The main sink of effort is maintenance, which can be very time consuming and tedious because information content and presentational details are mingled. It also involves direct manipulation of source files or program code which requires technical skills.

An alternative approach is to develop programs which automatically synthesise a Web site given a specification of an application. This approach reduces maintenance
cost and in some cases it is possible to allow non-technical personnel to update the content of the site via an appropriate interface. However, changes in the navigation structure or visualisation can be hard as this requires modifications in the program that generates the Web site. If changes of this nature are frequent, this approach shifts the problem from Web page maintenance to synthesiser maintenance.

Having an integrated system for capturing, storing and managing information is fundamental to most of today’s organisations to improve and enhance their operations. This scenario gave rise to Content Management Systems (CMS), which covers a wide range of applications from managing large number of documents through millions of pieces of text-based information and graphics for public Web sites [Hackos, 2002]. The range of software tools is still wide and confusing, each one offering different capabilities. Nevertheless, most CMS provide facilities for Web authoring and design through a templating approach.

Although content management goes beyond design and construction of Web sites, the core of any CMS system is the organisation and automatic transformation of pieces of information from different sources into Web pages. The approach proposed in this thesis resides in this realm. We address the problem of representing Web sites using a high-level representation language and transforming that into a live Web site. The proposed approach also addresses the problem of maintaining Web sites, by proposing a semi-automated method which can alleviate the demand on the technical staff and allowing non-technical people to post content to the Web site directly.

A general architecture for a Web synthesis system is depicted in Figure 2.1. Users may involve both technical and non-technical staff as Webmasters, department staff and other content contributors, who interact with the system to supply information content for publication and perform maintenance tasks. The content usually resides in a database and Web pages are automatically constructed. This is the sort of system that our method for Web site synthesis supports.

### 2.1.1 Declarative Specifications

Others have addressed the problem of design and construction of Web sites. Although our work is distinct in some aspects, which are discussed in the next chapter, most
work in this area has pointed to advantages of using declarative specifications as the starting point of the synthesis process.

Declarative specifications provide a better way for describing applications because they are independent from implementation details. Providing a high-level description of a problem independent from any particular implementation allows the designer to concentrate on the application description rather than on the mechanics for producing the Web site [Florescu et al., 1998, van Harmelen and Fensel, 1999, Cavalcanti and Robertson, 2002]. This makes the design of the content infrastructure of a Web site tidier and easier to detect errors.

Logic is naturally declarative and specifications although defined separately, they can coexist and be integrated easily since they are all defined under the same formalism. As a result a complete Web site specification, including content, navigation and visualisation, can also serve as basis for a variety of kinds of reasoning and property checking on that specification. We discuss in more detail the role of logic in Section 2.3.

The use of declarative specifications naturally leads to the separation of content specification from the visualisation details. This idea is crucial for a robust and flexible Web synthesis system. The next section discusses the details of this concept.
2.1. Approaches to Web site design and construction

2.1.2 Information Content, Navigation Structure and Visualisation

Every Web site application can be described as three separate components [Schwabe and Rossi, 1995]: information content, navigation structure and visualisation. Information content refers to data to be displayed on the Web pages. Navigation structure defines the organisation of the site and how items of information are related to each other. Finally, visualisation concerns how the information will be presented on the Web pages comprising the site.

The content of a Web site usually correspond to data items of the application database or they can correspond to text or other simple files. The navigation structure as explained earlier, is defined by the links and operations of the Web site. A Web site visualisation is defined in terms of page templates and information styles.

The common notion of a Web page is the rendering of a page in a particular language, such as HTML [Committee, 2002]. In these Web pages, content and visualisation specifications are mingled. If the Web site is relatively large, maintenance becomes a hard and very time consuming task. For that reason, the idea of separation between information content, navigation structure and visualisation became a design principle accepted by most researchers in the field [Florescu et al., 1998].

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. The problem now is how to represent appropriately each component of a Web site application in a way that they can be used for synthesis.

Combining this concept with declarative specifications has not been fully exploited in the context of Web site synthesis [Florescu et al., 1998]. We believe that the use of logic can bring all the different components together under the same formalism. Although each component should be specified and managed separately, defining all of them using the same formalism avoids the problem of integrating different concepts coming from different formalisms. Computational logic can be effectively used for this task, simple facts and rules forming the specification and logic programs acting on the specification to generate the Web pages.
2.2 Automated Synthesis

Automation is necessary because manual methods often result in the wrong person performing the wrong task (e.g. a programmer posting content to the Web rather than an editor) or in people being distracted from their real work by having to perform repetitive tasks.

The sheer volume of information presented in some Web sites made them practically impossible to construct and manage without the aid of some sort of CASE tool. If the information changes frequently requiring constant update, as in data-intensive Web sites, the problem is even more complex.

We have been using a domain-specific synthesiser for our research group Web site at Edinburgh (http://www.dai.ed.ac.uk/groups/ssp/index.html) for about 4 years. Maintenance is done simply by updating its declarative specification. Although this has proved cost-effective it is limited to the generation of a particular visualisation and navigation structure. Although there is a declarative specification of the site, any substantial change to the visualisation style requires modification in the synthesiser program.

This draws the attention to the issue of how such a synthesis method can be generalised and still generate practical Web sites. If the synthesiser is very general it is likely to require specialist expertise to control it. On the other hand, if the synthesiser is very specific it will only be able to produce Web sites for a narrow range of applications. We want to find a balance between these two extremes.

One way to achieve domain independence is to apply the paradigm of design using parameterisable components [Robertson and Agustf, 1999]. The idea is to create a domain-independent library of components that are used to describe pieces of information and visualisation styles, which are instantiated according to the parameters given. For example, visualisation components are defined to transform a piece of information in different presentational forms. This idea is illustrated in Figure 2.2.
2.2. Automated Synthesis

Figure 2.2: Example of parameterisable components
2.3 The Role of Computational Logic in Web Site Synthesis

A strong 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. The logic programming community has been focusing mainly on information retrieval techniques [Prendinger, 1997, Loke and Davison, 1996, Clark and Lazarou, 1997]. Other recent work have proposed some limited use of logic, mainly for specification or property verification [van Harmelen and van der Meer, 1999, Jin et al., 2001]. We believe that computational logic can play a major role in addressing the problem of specifying and synthesising Web sites. Our approach applies computational logic to support a uniform view of data and computation, allowing reasoning with both specification and program via meta-programming.

Declarative specifications can form a basis to automated synthesis, changing the focus from the mechanics of producing Web sites to specification of applications [Cavalcanti and Robertson, 2001, Florescu et al., 1998]. Providing a high-level description of a problem independent from any particular implementation allows the designer to concentrate on the application description rather than on the mechanics for producing the Web site.

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 and 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].

By using a separate declarative specification, all implementation details are detached from the specification. This facilitates both the initial design and future modifications in the definition of the Web site.

As logic programming is essentially 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 compu-
tational logic techniques appropriate for automated synthesis of Web site, since most synthesis steps are basically transformation of logic terms in terms of some target language, such as HTML.

As benefits of this approach, Web sites are constructed consistent with an specification and maintenance is facilitated more generally. Updating data should become a clerical task requiring no technical expertise. Modifications in navigation structure or visualisation should not involve further programming, facilitating these tasks as well. Maintenance can be done by means of a simple message passing interface between the Webmaster and maintenance agents. These agents can reduce dramatically maintenance cost by avoiding the need of human intervention. Chapter 10 gives the details of how we address the problem of Web site maintenance using task-specific agents.

Computational logic, in summary, gives us the support for declarative specifications and procedural execution, necessary for the synthesis process. However, logic is more than just a vehicle for the representation and synthesis. It makes easier the integration of all parts of a Web site specification and it adds reasoning power to the system, allowing the definition and verification of constraints and properties. It also serves as a basis for automating maintenance tasks as we shall see in the following chapters.

2.4 Specific questions addressed by this Thesis

Now that more detail about the proposed approach for Web site has been given, we can also specify more specific questions that are answered along the thesis.

- What formalism can be used for the high-level specification of Web sites which is suitable for Web designers and allows a straightforward translation to logic?
- How to represent content, navigation and visualisation using logic?
- What are the implications of using logic for this sort of automated synthesis?
- How do we specify and verify properties and constraints on a Web site specification?
• How can maintenance tasks be automated?

• How does maintenance benefit from the proposed method?
Chapter 3

Related Work

This chapter presents a survey of methods, techniques and other related work on automated synthesis and the use of computational logic to address problems on the Web, discussing the different approaches proposed.

Research on methods for automated synthesis of Web sites is very recent, the majority of work being only published in the last four years. Although, this subject has attracted the attention of researchers of different areas, such as database, software engineering and multimedia, there are still few published results. One explanation for this fact is the fast pace in which Web technologies are evolving. As a result, techniques proposed a few years ago may become obsolete and surpassed by the advent of new technologies. An example of this is the AutoWeb system [Fraternali and Paolini, 1998] which has evolved into a new approach, the ToriiSoft system [Ceri et al., 1999], incorporating new technologies such as XML. Section 3.1 presents related work in this specific area.

Another line of related work concerns the integration of logic programming with the Web and the use of ontologies for representing Web site applications. This is discussed in Sections 3.2 and 3.3. Web site maintenance and property verification are discussed in Section 3.4 and finally Section 3.5 presents an overall comparison of our approach to Web site synthesis with the other approaches presented in this chapter.
3.1 Methods for Web Site Construction

Various approaches to Web site design and maintenance have been proposed in recent years. There is general agreement amongst these on some core concepts [Florescu et al., 1998]:

- Separation between information content, navigation structure and visualisation.
- Declarative specifications, by means of high-level conceptual data models or declarative languages.
- Automated or semi-automated generation of Web site by means of CASE tools.

Below we describe related approaches to Web site construction:

- **Araneus** [Atzeni et al., 1998]
  This is one of the database related approaches, in which the starting point of a Web site design process is the application database conceptual design phase using the entity-relationship model. The approach is particularly suitable for data-intensive Web sites. Visualisation is designed using a page-oriented model (Navigation Concept Model - NCM), which describes the Web pages in a conceptual level. From NCM, a logical hypertext design based on the Araneus Data Model (ADM) is derived. Finally, a specific language (Penelope) is used for the generation of HTML Web pages from ADM specifications. In the Araneus approach, database and hypertext design are separate, corresponding to the separation between content and navigation structure design. This approach also gives attention to maintenance, providing tools for Web site reorganisation.

- **OntoWebber** [Jin et al., 2001]
  OntoWebber is an ontology-based approach designed for the generation of data-intensive Web sites, in particular Web portals. It address the problem of multiple and heterogeneous sources of data by defining an integration and articulation layers, responsible for the resolving syntactic and semantic differences between the different data sources. The composition layer includes a specification of
a Web site view based on the Web site modelling ontologies. Ontologies are described using DAML+OIL [W3C, 2001], a semantic mark-up language for Web resources. A generation layer process queries on the site specification to produce the Web pages.

- **OO-H** [Gómez et al., 2001]

  The OO-H method is an object-oriented approach which is based on a UML-compliant model. This approach extends traditional UML [UML, 1999] modelling of static and dynamic features of an application with a navigation access diagram (NAD) and abstract presentation diagram (APD) use, respectively, to model the navigation structure and visualisation of the Web site. This approach supports the generation of Web sites in multiple target languages (HTML, XML and WML).

- **OOHDM** [Schwabe and Rossi, 1995]

  The Object-Oriented Hypermedia Design Methodology is a model-driven approach which uses an object-oriented data model with the same name (OOHDM). This methodology introduces models for description of hypermedia applications, where the main constructs are entity, links and menu (used to represent access structures). The design process is organised in conceptual data design, navigation design, interface design and implementation. The main focus of this approach is on the specification of the Web site. The approach lacks an automatic synthesis component, as the implementation phase is not integrated with the approach by means of CASE tools.

- **SEAL** [Maedche et al., 2001]

  This is an approach for developing semantic portals. It is based on an underlying ontology describing the Web site. The main components of this approach are the knowledge warehouse (data repository), represented in the form of F-Logic [Kifer et al., 1995] statements, and the inference mechanism, which uses the Ontobroker system [Decker et al., 1999] for performing inferences. It also includes a navigation module for modelling the links between pieces of information, a
query module for constructing Web pages based on queries on Ontobroker, a personalisation module for specifying preferences for the page content and a template module used to generate HTML Web pages.

- **Strudel** [Fernández et al., 1998]

This is an earlier work on Web site synthesis. It has a particularity of using graphs as the modelling paradigm, in contrast with most approaches which are based on conceptual data models. The key feature in this approach is the separation between content and visualisation. The system also supports the integration of heterogeneous sources of data. The design process comprises two main steps: definition of the data that will be available at the Web site by means of queries using a site-definition query language (StruQL). The result of the evaluation of queries is the site graph which represents both the site content and structure. The second phase of design is the definition of the presentation of the data, using Strudel's HTML-template language.

- **ToriiSoft** [Ceri et al., 2000]

ToriiSoft is a tool for generation of Web sites based on the WebML modelling language. WebML is a conceptual notation for specifying Web sites, which supports a high-level description separated in four parts: content (structural model), the pages that compose the Web site (composition model), links (navigation model) and customisation features (personalisation model). All concepts in WebML are associated with a graphic notation and with underlying XML schemas.

The main differences between these proposals are in the emphasis given to particular aspects of the Web construction process. Most approaches focus on modelling aspects (Araneus, OO-H, OOHDM and Strudel), some are data-driven (ToriiSoft) and others are based on semantic descriptions (OntoWebber and SEAL). 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.
or object-oriented data models, such as UML. In this category we can include Araneus, OO-H, OO-HDM and ToriiSoft. Araneus uses the ER model. OO-H is based on an extension of the UML model. OO-HDM is also an object-oriented extension of HDM [Garzotto et al., 1993]. ToriiSoft proposes a structural model compatible with ER, ODMG object-oriented data model and UML class diagrams. Strudel models Web sites 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 in first-order logic.

Support for heterogeneous data sources is offered by OntoWebber and Strudel. Some limited support (for relational database systems only) is offered by ToriiSoft. This issue is not addressed by the other approaches.

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, OO-H, SEAL, Strudel and ToriiSoft all support this feature. Our approach address this issues by giving support for combining declarative descriptions of alternative visualisations with templates in different target languages.

Another interesting feature is the support for integrity constraints, which is given by OntoWebber, Strudel and ToriiSoft.

Maintenance issues are not explored deeply by most approaches, although many claim support for it or offer some degree of automation, such as Araneus and OntoWebber, by means of rules defined as triggers.

Table 3.1 presents a summary with the related work mentioned in this chapter and their main features. The features described in the table are:

- **Paradigm**: the paradigm of the approach for Web site construction is usually based on modelling formalisms, such as the entity-relationship model (model-driven) or object-oriented data models. Web pages can also be constructed from queries on a database (query-based) or from ontological descriptions (ontology).

- **Model(s)**: what are the conceptual data models used by the method for concep-
tual descriptions of Web sites.

- **dyn**: support for generation of dynamic pages
- **+inp**: support for multiple heterogeneous input sources
- **vis.**: generation of different visualisation, including different target languages
- **pers.**: support for personalisation of Web pages
- **maint.**: support for automation of maintenance tasks
- **prop.**: support constraint and property verification

<table>
<thead>
<tr>
<th>Method</th>
<th>Paradigm</th>
<th>Model(s)</th>
<th>dyn</th>
<th>+inp</th>
<th>vis</th>
<th>pers</th>
<th>maint</th>
<th>prop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Araneus</td>
<td>model-driven</td>
<td>ER, NCM</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>OntoWebber</td>
<td>ontology</td>
<td>DAML+OIL</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>OO-H</td>
<td>object-oriented</td>
<td>UML</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>OOHDM</td>
<td>object-oriented</td>
<td>OOHDM</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>SEAL</td>
<td>ontology</td>
<td>F-logic</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Strudel</td>
<td>query-based</td>
<td>graph</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>ToriiSoft</td>
<td>model-driven</td>
<td>WebML</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Our Approach</td>
<td>logic-based</td>
<td>First-order</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 3.1: Features of approaches to Web site synthesis

### 3.2 Logic Programming and the Internet

It can be argued that the Web is causing a paradigm shift in the nature of computing, moving away from stand-alone systems towards distributed, highly interactive systems, that achieve much of their power from the benefits they receive from interacting with other such systems. Logic programming seems to be well suited to meet the needs of this new computing paradigm. Opportunities exist for information retrieval, knowledge discovery and Web application modelling. However, opportunities have also been created for Web site synthesis.
Cabeza and Hermenegildo (1997) discussed the issues involved in creating Web applications using logic programming systems. They also describe the Pillow system which is a library of logic programming components for the Web. It supports the generation of HTML documents from logic terms, including HTML forms and their associated handlers. It also offers facilities for accessing and parsing Web pages and automatic logic programming code downloading for local execution using generic browsers. Pillow is extensively used in our approach to translate the logic predicates we use to represent a Web site into HTML.

In [Loke and Davison, 1996], an integration between logic programming and the Web is proposed, by incorporating programmable behaviour and state into Web pages. The behaviour of a Web browser is changed by a logic programming module which intercepts all user requests for processing before generating and sending the result back to the browser.

Boley (1997) discussed the use of knowledge bases on the World Wide Web, in which Horn logic is used for Web publication. The main idea of this proposal is to provide a declarative Web representation which serves as a basis for logic programming applications, where a search engine is the main application discussed.

Tarau (1997) discussed the integration of logic programming with Web technologies, such as Java [Arnold et al., 2000] and VRML [VRML Architecture Group, 1996]. In [Tarau et al., 1999] details of the mapping rules from Prolog to VRML are defined, allowing the creation of Prolog-based virtual worlds for distributed group-work over the Internet.

In [Clark and Lazarou, 1997] a multi-agent system architecture is proposed to address the problem of distributed information retrieval. In this work the notion of a collection of information sites is used to model the Web. Each information site corresponds to a logical clustering of actual Web sites and for each of them an information source agent and an extractor agent are defined. The extractor agent as the name suggests, extracts relevant information from an information site and passes it in a attribute-based representation to the information source agent. This agent is then responsible for handling and answering user queries using the attribute-based formatted information. The query language used is a variant of Prolog.
These work have the common theme of extending the notion of Web page with the logic programming paradigm. With the exception of Tarau (1997), the main application is information retrieval for the Web. Although they bring the reasoning capabilities of logic programming to the Web environment, these approaches have to address performance issues which makes it difficult to match those of traditional algorithmic approaches.

### 3.3 Ontologies and the Semantic Web

The extensions proposed in the work discussed in the previous section are external to the Web environment, usually implemented via Prolog-CGI modules which are integrated with the Web site. A different approach to extend the Web with reasoning capabilities is by extending the HTML language, which makes query processing easier. However, the Web pages are no longer standard HTML pages and in some cases browsers are not capable of displaying them without the appropriate processing modules.

SHOE [Luke et al., 1997] is one approach that extends the HTML language by incorporating annotated semantics. SHOE allows authors to add semantic content to web pages and to relate this content to common ontologies that provide contextual information about the domain. The main application is once again information retrieval on the Web.

In the same line of this work, the use of XML as the basis for defining semantics for the information presented on the Web gave birth to a new area known as the Semantic Web [Decker et al., 2000, Berners-Lee et al., 2001]. The semantic Web concerns the abstract representation of data on the Web, based on ontology specification formalisms, such as RDF [W3C, 2002b]. The W3C standards proposed for the semantic Web is posed to drive research and commercial initiatives in this area, including future Web synthesis systems, which might support the generation of semantic-annotated Web pages.

Ontologies are becoming more popular with the focus changing from information retrieval to knowledge representation for the Web. [Staab et al., 2000] discusses
an ontology-based approach for building and maintaining Web portals. The ontology serves as a semantic backbone for accessing knowledge on the portal, supporting queries and inference.

A similar approach is the Ontobroker project [Decker et al., 1999], which also views the Web as a knowledge base and uses ontologies for representing semantic information. The system provides means for defining ontologies, queries and annotating HTML Web pages with ontological information. It also offers a webcrawler that collects information from the Web and an inference engine to support query processing based on the defined ontology.

3.4 Maintenance and Property Verification

Although most of the proposals are incorporated in some approaches for Web site construction, such as Araneus and OntoWebber, a few work have addressed the problem of Web site maintenance.

[Sindoni, 1998] proposes an approach to enforce consistency between page content and database state. A specific algebra is proposed for defining views and views updates, which uses the Araneus Data Model [Atzeni et al., 1998] as the reference model. Views updates are used by an algorithm that automatically updates the Web site from a set of changes on the application database. Maintenance is performed by removing or generating the appropriate sets of pages.

WebMaster [van Harmelen and van der Meer, 1999] is a knowledge-based approach to the verification of Web page content. Semantic mark-up using XML is used for describing Web page content against which constraints are verified. Constraint rules are specified using the general logic expression:

$$\forall X[\exists Y \land_i p_i(X_k, Y_l)] \rightarrow [\exists Z \land_j q_j(X_k, Z_m)]$$

where $X, Y, Z$ are sets of variables and $p_i$ and $q_i$ are binary predicates. The system provides a graphical interface for the definition of rules. This avoids the need for the Web designer specifying rules directly using logic.

An approach for verifying integrity constraints on Web sites is proposed by [Fernández et al., 1999]. Regular-paths expressions are used for specifying constraints
over a Web site represented as a graph, where nodes correspond to pages or pieces of information. The verification of constraints is based on expressions of the form \( X \rightarrow R \rightarrow Y \) where \( R \) is a regular-path expression and \( X \) and \( Y \) are terms. The atom \( X \rightarrow R \rightarrow Y \) is satisfied in a labeled directed graph by each pair of nodes \( X,Y \) for which there is a path from \( X \) to \( Y \) that satisfies the regular-path expression \( R \).

### 3.5 Discussion

The proposed approach to Web site synthesis is novel by offering a framework for Web site construction based on computational logic. This feature makes it more convenient to introduce reasoning capabilities, supporting definition of integrity constraints (an issue also addressed by OntoWebber, ToriiSoft and Strudel) and rules for both navigation and visualisation. Our approach also is flexible, supporting different data sources and different target languages for generating Web pages. Logic also facilitates property checking, which is difficult to accomplish with procedural approaches.

Our approach also integrates all aspects of the specification, synthesis and maintenance of Web sites using computational logic. Almost all work on Web site synthesis use different formalisms to represent different parts of the specification. Although this is not crucial to the success of a method, using fewer formalisms facilitates the integration of different components, allows extensions to the method more easily (using the same formalism) and offer the possibility of a unique view of the whole specification.

Although we use a different conceptual data model for representing the high-level representation, this representation is also translated to logic, making it fully compatible with the remaining representations. This makes easier to integrate or reason about aspects from different parts of the Web site specification (pieces of information, navigation structure and visualisation).

Automating maintenance tasks is also a distinctive feature of our method. Few people have addressed this problem via automation. Furthermore, the few existing proposals require technical expertise to alter parts of the Web site, which is not necessary in our approach to Web site maintenance.

We can summarise the main features of the proposed approach as the following:
3.5. Discussion

- the use of logic to describe Web site applications;
- separation between information content, navigation structure and visualisation;
- support for producing different views on the information;
- support for different target languages;
- support for integrity constraint and property verification
- automation of maintenance tasks.
Part II

An Approach for Web Site Design and Implementation
Chapter 4

Web Site Synthesis: System Architecture

Defining an architecture for a Web synthesis system involves an analysis of the features that the system will include and how these features relate to each other. Figure 4.1 presents architecture options that can be used.

![Diagram](image)

a) Simple synthesiser architecture

b) Synthesiser architecture with templates

c) Complete synthesiser architecture

Figure 4.1: Synthesiser architecture options
Architecture a) is suitable for domain-specific applications where all application and visualisation details are embedded in the synthesiser program. It also is suitable for application with static data, i.e. applications that do not require constant update in the information presented. Clearly, it is the least flexible architecture as changes in the data structure or in the visualisation requires modifying the synthesiser program. This approach also has the disadvantage of requiring trained people to work with it.

Option b) adds the notion of separating data from visualisation descriptions, which allows changing the visualisation without further programming. This option is suitable for applications which need constant modification in the presentation or require alternative visualisations for the same information. However, changes in the data structure, for example changing the number of attributes of a table in the application database, still require modifying the synthesiser because it is still tied with a particular data structure. Another disadvantage of this approach is the need for people with specific technical skills for maintaining the Web site. Most Web site authoring systems fall in this category.

By introducing an independent component for application description, architecture c) brings all the advantages of using a conceptual model to Web site applications. The synthesiser becomes independent from the actual data structures, supporting changes in both the database and visualisation descriptions. This kind of architecture has been used in recent research concerning Web site synthesis and there are already some commercial applications, such as content management systems, following this concept [Hackos, 2002].

Note that the technical skills mentioned above refer to the use of programming languages used to implement a Web site synthesiser and mark-up languages, such as HTML, for the rendering of the Web pages.

4.1 A 3-level Architecture for Web Site Synthesis

One of the main features of the approach proposed here is to use logic to represent and synthesise a Web site automatically. Logic descriptions would correspond to the application description component as presented in Figure 4.1 c. However, few people
feel comfortable using a general purpose logic directly. One way to deal with this issue is to offer a suitable interface to the designer based on a popular conceptual model and translate the resulting description into logic formulae.

This introduces an intermediate representation, which hides the logic representation details from the Web designer. More importantly, the intermediate representation will form a basis for all reasoning performed on the pieces of information. As a result our synthesiser architecture is organised in three levels, as illustrated in Figure 4.2.

We begin with a high-level description of an application. From the application description, an intermediate representation is automatically derived. The final step involves access to the application database in order to instantiate all pieces of information accordingly and transform the intermediate representation into Web pages. It is assumed that a database is available with standard means to access data in order to instantiate pieces of information described in the application description. These
transformation processes are detailed in Chapter 8.

It is also possible to generate different intermediate representations from the same high-level description. Since from each intermediate representation many visualisations can be produced, the actual number of Web site instances derived from one application description can become very large.

Ideally the system should be able to offer the Web designer different mappings from the high-level description combined with many visualisation options. Chapter 11 discusses the implications of deriving multiple intermediate representations and visualisations.

4.1.1 Level 1 - Conceptual Design

This level contains a high-level description of a Web site application.

Conceptual data models are powerful tools for representing data, relations between data and integrity constraints with the advantage of being independent from any implementation. They are the natural candidate to be used at this level.

Rather than proposing yet another conceptual data model for modelling Web site applications, our approach uses existing data models and defines mapping procedures for translating the data model constructs into our logic-based intermediate representation. This is important in making the proposed approach extensible, supporting multiple high-level description sources.

Since there is no dominant data model for modelling Web sites our choice is based on popularity and ease of use. The entity-relationship (ER) model [Chen, 1976] is used as an example of one conceptual data model for the high-level description. A detailed discussion about transforming ER schemas into the intermediate representation can be found in Chapter 5. The ER model and its extended versions [Elmasri and Navathe, 1999] are a popular choice for modelling applications with an underlying relational database. [Feyer and Thalheim, 1999] also propose the use of the ER model for designing Web sites.

Other conceptual data models can also be used, such as the OOHDM [Schwabe and Rossi, 1995], which is an object-oriented data model for designing hypermedia applications. The popular UML [UML, 1999] can also be used for modelling
Web site applications. Although it possess a large number of concepts, a well-defined sub-set of UML can be used as most of object-oriented data models. This approach is proposed by [Gómez et al., 2001]. In any case, the crucial requirement for incorporating such models in our approach is translating data model concepts into first-order predicate logic. This allows the definition of appropriate mapping procedures to the intermediate representation.

Other possibilities for the high-level description rather than conceptual data models are ontological descriptions [Jin et al., 2001, Maedche et al., 2001] and relational database schemas. Although it might be possible to use both, this thesis does not address the implications of using such formalisms.

Visualisation description is not supported by the majority of data models. For this reason, a separate component defining how each piece of information should be presented and how a Web page should be visualised must be provided. Chapter 7 discusses the details of this component.

Integrity constraints can be derived from a conceptual data model. However some important constraints are particular to Web site applications, for instance the order of presentation of information. More detailed discussion follows in Chapter 9.

### 4.1.2 Level 2 - Intermediate Level

This level includes the intermediate representation and the database of an application. At this level all reasoning processes take place including verification of integrity constraints, creation of an intermediate representation and the synthesis process to derive a Web site.

The intermediate representation includes separate descriptions for information content, navigation structure and visualisation design.

A mapping procedure from the application description to the intermediate representation can be either automated or semi-automated. In the latter, the system interacts with the Web designer where decision points are met. We only describe a fully-automated synthesis method, since user interaction during design raises other issues not addressed in this thesis.

The intermediate representation can also be seen as an additional layer of knowl-
edge about the application. One of the main advantages of using logic is that all aspects of a Web site application are represented in the same formalism. This eliminates the semantic gap which usually occurs when using different formalisms to describe different components of an application. In addition, reasoning can be performed on each component alone or on all components at the same time. This second level also allows the definition of a more flexible Web site generation process because it is independent of any particular implementation.

It is also in this level where integrity constraints are verified. It is important to note that integrity constraints and other properties are verified against the intermediate representation, rather than on particular instances of a Web site. This ensures that all constraints and properties hold on any Web site derived from the same intermediate representation. An example of properties are reachability of information and verification of correctness of Web site specifications. Chapter 6 presents a detailed description of the intermediate representation.

4.1.3 Level 3 - Implementation

The third level corresponds to an instance of a Web site composed of Web pages written in a target language. In order to generate the Web site pages, the intermediate representation is combined with a visualisation description to generate a corresponding Web site. This transformation is detailed in Chapter 8.

Note that content and navigation (links) usually are represented by the same language. The recent advent of style sheets gave rise to languages such as CSS [W3C, 1999] for defining particular presentation styles for the content of a Web site. This is a step forward in the direction of separating content from presentation details. However, the use of HTML still requires additional visualisation specifications. For example, Web page templates still have to be defined.

CSS is one of the style sheet languages proposed as a standard by the W3C. It is used to define and control presentation details, such as colours, text fonts, text size, alignments, margins, etc. By associating visual specifications to HTML elements tags, it is possible to control the presentation form of individual pieces of information and the overall look of a Web page.
4.2 Web Design Issues

Although CSS is proposed as a standard the current browsers offer variable support for CSS, resulting in slight different visualisations of a Web page depending on the browser used. It is expected that new Web browser versions offer full support to CSS style sheets. In http://www.w3.org/Style/CSS/#browsers there is a comprehensive list of browsers describing their current support to CSS.

Our approach makes use of style sheet languages together with other target languages for the implementation of the Web pages. Target languages considered include HTML [Committee, 2002], XML schemas [W3C, 2002a] and WML [WAP Forum, 2001] for WAP enabled mobile devices [WAP Forum, 2002].

WAP is a recent technology that allows interactive content to be requested and delivered to mobile devices, such as phones and handheld PCs. Web pages can be handled by the WAP protocol if they are written in WML (Wireless Markup Language). We have carried out some experiments with the synthesis of Web sites written in WML.

4.2 Web Design Issues

It is not the intention of this thesis to dwell upon stylistic design issues, there is already considerable work on Web design [Ivory et al., 2001]. Nevertheless a few points are worthy of mention. The visualisation specification includes page templates and style definitions for each piece of information. As this specification is separate from content and navigation specifications, the designer can focus only on visual aspects.

The idea of CSS style sheets is to override the presentation of HTML elements based on its tags. This is not an ideal solution as misuse of styles can lead to even more complicated and error prone Web site visualisation. An example of bad visualisation design concerns changing the colour and font of links represented by the tag <ahref = ... > which should be of colour blue and underlined according to design standards. Another example is changing the font size of tags <h1>, <h2>, etc, which are expected to follow a standard font size.

Web site synthesis methods whilst promoting separation of visualisation design from content design cannot guarantee the quality of design. However, well known standards could be enforced by some rules, for example by preventing or alerting to
changes in styles of particular tags, like the ones mentioned above.

Our approach can help on detecting and solving a few design issues:

- orphan pages: by verifying a property that every page must have at least one link to the homepage of the Web site.

- navigation support: by suggesting a set of pages as the site navigator, which is included in every page of the Web site in a standard visualisation style.

- use of frames: this is very often described as bad design. To name a few problems, it is confusing for users, printouts become difficult and the predictability of user actions is lost. Our approach does not support the definition of frames.

There are other bad design practices that our approach is not prepared to detect, for instance the extensive use of animated graphics, bad design of external components such as multimedia material, CGI programs, Java scripts and VRML.

### 4.3 Summary

This chapter presented the various architecture options for Web site synthesis systems. The features included in our approach led naturally to a 3-level architecture. Features that had a direct influence on the definition of our system architecture include separation between information content, navigation structure and visualisation; support for producing different views on the information; and support for integrity constraint and property verification.

Assuming that an ER model is used for representing a Web site application and HTML is the target language, Table 4.1 shows the correspondence between the architecture levels and the three description components of a Web site.

This separation of levels allows a very flexible design process as different solutions can be explored in the implementation level before deciding for a final result. Independence between application and visualisation specifications gives us the ability to produce different Web sites by combining the same application specification with different visualisation descriptions.
4.3. Summary

<table>
<thead>
<tr>
<th>Content</th>
<th>Navigation</th>
<th>Visualisation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1</td>
<td>Entities and attributes</td>
<td>Relationships and cardinalities</td>
</tr>
<tr>
<td>Level 2</td>
<td>Pieces of information</td>
<td>Transition rules</td>
</tr>
<tr>
<td>Level 3</td>
<td>HTML elements</td>
<td>HTML links</td>
</tr>
</tbody>
</table>

Table 4.1: Architecture levels and description components relation

The architecture is flexible enough to support the addition of different data models and target languages. This can be exemplified by replacing the ER and HTML concepts in Table 4.1 with other data model and target language corresponding concepts. Figure 4.3 illustrates the extensible architecture supporting multiple sources and multiple target languages.
Figure 4.3: Supporting multiple source and output
Chapter 5

High-Level Specification of Web Site Applications

Conceptual data models and high-level formal languages have successfully been used for modelling and designing a variety of software applications. As an engineering tool they provide a common formalism to describe aspects of an application. Data models and formal languages can also be used as a means of communication between members of a development team and, in some cases, with the user [Brodie, 1984].

As explained in the previous chapter, the proposed method requires as a precursor to Web site generation a model of the task or domain for which the site is to be generated. The language in which this model is described should be as general as possible, so it may be widely applied, but it must also be accessible to application-specific engineers, which requires some commitments to accepted engineering notation. One way of reconciling this tension between generality and specificity is to follow a standard view of a class of Web design applications. One such standard view is to provide an information system built around one or more databases with an accompanying navigation structure. This has many similarities with traditional non-Web systems allowing the use for the Web domain of a standard notation for data modelling, such as the Entity-Relationship Model (ER) [Chen, 1976].

The ER model was chosen due to its popularity and widespread use in application modelling, although other conceptual data models can also be used. The ER model
has been highly successful for modelling database applications and gave rise to a large number of extended versions [Teorey et al., 1986, Elmasri and Navathe, 1999]. A common type of Web site application is the one in which information resides in a database and Web pages provide a particular view on data. An ER schema of that sort of application can also be used to derive such views, which can be materialised as Web pages [Cavalcanti and Robertson, 2002].

This chapter discusses the use of conceptual data models for specifying Web site applications, in particular the ER model. It also presents an automated method for deriving Web site applications from entity-relationship schemas.

## 5.1 High-level Description of Web Sites

The benefits of using high-level models and languages for conceptual modelling are widely accepted as they may (when used appropriately) provide an appropriate representation of the application properties and enable a better understanding of application requirements. In addition, a high-level specification is independent of implementation details, supporting multiple solutions and allowing many implementation decisions to be postponed.

Therefore, it is not surprising that new conceptual data models and languages for Web site applications have been proposed, such as the Navigation Conceptual Model (NCM) [Atzeni et al., 1998], HDML-lite [Fraternali and Paolini, 1998] and WebML [Ceri et al., 2000]. However, given the large scope of Web site applications and their close relation to other software applications, existing conceptual data models also can be used for modelling Web sites. One reason for using existing data models is to avoid the need for the Web designer to learn yet another formalism.

Having a large variety of conceptual data models available, the nature of the application determines which modelling paradigm is more appropriate. If the Web site is data-intensive with the information residing in a database, then the ER model or any of its extensions can be used. Applications that include user interaction, usually by means of forms and external programs for processing user input, would benefit from data models that support modelling of dynamic aspects (operations). This cat-
5.1. High-level Description of Web Sites

Category includes object-oriented data models, such as UML [UML, 1999]. There are applications in which the navigation structure (the way to reach information) is very important. In this case, models capable of modelling links between pieces of information are needed. This type of modelling is often used in hypermedia applications [Schwabe et al., 1995].

Although conceptual data models can be applied to Web site application design, they are clearly restricted to content and navigation modelling. They do not support Web page visualisation design. As a result, most approaches to Web design separate completely the content and navigation descriptions from visualisation specification. In fact this is good design principle, since each component can be designed and maintained separately. However, if the formalisms used for modelling each component are different, integration of the concepts of the different models might become problematic. Table 5.1 shows the correspondence between concepts of data models and Web sites.

<table>
<thead>
<tr>
<th>Data Model/Web Site</th>
<th>Pieces of Info</th>
<th>Links</th>
<th>Visualisation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ER</td>
<td>entity/attribute</td>
<td>relationship</td>
<td>–</td>
</tr>
<tr>
<td>OOHDM</td>
<td>class/attribute</td>
<td>relationship</td>
<td>ADV-Chart</td>
</tr>
<tr>
<td>Relational</td>
<td>relation/attribute</td>
<td>foreign key</td>
<td>–</td>
</tr>
<tr>
<td>UML</td>
<td>class/attribute</td>
<td>association</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 5.1: Concept correspondence between data models and Web sites

Note that the OOHDM (Object-Oriented Hypermedia Design Model) [Schwabe and Rossi, 1995] approach supports visualisation design. However, this is by means of a separate model, the Abstract Data Views - ADV, which relates to navigation objects. These views are expressed in different diagrams called ADV-charts. This separation between data and visualisation modelling is the usual solution adopted by most approaches to Web design. It also appears in the OO-H method [Gómez et al., 2001] and the Araneus design methodology [Atzeni et al., 1998]. This also is the way followed by our approach to Web site synthesis.
5.2 Application Description using an Entity-Relationship Model

The Entity-Relationship Model was originally proposed by Chen [Chen, 1976]. Due to its popularity, especially in the database community, many extensions have been proposed leading to a large number of slightly different data models [Teorey et al., 1986, Hainaut, 1990]. This chapter uses the Enhanced Entity-Relationship model (EER), defined by Elmasri and Navathe [Elmasri and Navathe, 1999]. The EER model adds the concepts of specialisation, generalisation and categories.

The EER model can be appropriately used either as a basis for creating a Web interface for an existing database application or for developing a new application in which the information are kept in a database.

The main concepts of the EER model are:

- **Entity**: an object or concept that is relevant to the application
- **Relationship**: a meaningful association among entities.
- **Attribute**: properties that define the characteristic of an entity or relationship. Properties are used to distinguish one entity instance from another.
- **Generalisation Hierarchy**: defines a superclass/subclass relation between entities. It refers to the process of minimising the differences between entities by identifying their common features creating a single *general* entity.
- **Specialisation Hierarchy**: also defines a superclass/subclass relation between entities. It highlights the differences between members of an entity by identifying their distinguishing characteristics creating distinct specific entities.

Since the pieces of information are assumed to be kept in a database (usually a relational database), the correspondence between the ER constructs and its corresponding database items must be known. This information can be extracted from the mapping procedure from an ER diagram into a relational database schema. There are standard methods for mapping an ER diagram into a relational database schema. We follow the procedures proposed in [Elmasri and Navathe, 1999]. Briefly, the mapping rules are:
5.2. Application Description using an Entity-Relationship Model

1. Entities

For each entity a relation is created including its attributes, except those multi-value and structured attributes.

2. Multi-value and structured attributes

These are translated into separate relations including the identifier of the entity they belong to as a foreign key.

3. Relationships are mapped according to their cardinalities:

   (a) 1:1 Relationships

   The identifier of one of the entities is included in the relation representing the other entity as a foreign key. If the minimum cardinality in one of the sides is zero then its identifier is included in the relation representing the “one” side.

   (b) 1:N Relationships

   The identifier of the entity in the “one” side is included in the relation representing the entity in the “many” side. Note that the min cardinality does not affect this mapping.

   (c) N:N Relationships

   A separate relation is created to represent the relationship including both entities identifier and any relationship attributes.

4. Specialisation and Generalisation Hierarchies

   Additional relations are created for each sub-entity, including the parent entity identifier as a foreign key.

5.2.1 Working Example

Throughout this thesis a Web site for an accident reporting application is used as a working example. The Web is an attractive medium for accident reporting application domain because it allows easy access to information both to experts and the general
public. It is also a data intensive Web site, as new reports or updates on existing reports are very frequent. Figure 5.1 shows an ER diagram describing an accident reporting application.

The application concerns incidents that have happened during scheduled flights (cargo or passenger). Some information about the aircraft and the pilot are also of interest. Incidents are related to events which can be of three different types: descriptions (weather conditions, flight level, etc), nature of problem (mechanical failure, collision, fire, etc) or actions (by the crew or instructions by the air traffic control).

Given that existing techniques for mapping an ER diagram into a corresponding relational schema are available [Elmasri and Navathe, 1999], we assume that a relational database is created. As an example, the corresponding relational database schema to entities aircraft, flight and the relationship fly (including attributes and data types) is presented by tables 5.2 and 5.3. It also includes indication of foreign key attributes where appropriate. Note that relationship fly is represented by attribute aircraft_registration in table flight.

The ER schema presented in Figure 5.1 is the starting point to the Web site synthesis and all examples in the following sections will refer to the data items defined there.
5.3 Deriving Intermediate Representations from ER Schemas

Given that an ER schema is provided by the Web site designer our task is to define a corresponding intermediate representation automatically. A fully automated process at this step is ideal because the details of the intermediate representation are completely hidden from the Web site designer.

It is possible to derive different intermediate representations from one ER diagram, allowing the construction of different Web sites for the same application. This is an interesting solution for rapid application prototyping as the only interaction with the site designer is the ER diagram input and style choices for pieces of information and
Web pages. A standard navigation structure and a layout for visualisation are also automatically produced.

In the following sections, three different intermediate representations which are called entity-based, instance-based and query-based are presented. The first defines a Web page for each entity, placing all instances of that entity together in a single Web page. The second option defines a Web page for each instance of an entity. The third option allows for a combination of the two previous approach creating Web pages corresponding to the result of queries on the application database. Links between pages are derived from relationships between entities.

These mapping procedures are appropriate to data-intensive Web sites, where the content usually resides in a database. These particular mapping methods are also suggested because of the popularity of the Web view of a relational database modelled with the ER model.

5.3.1 Representing ER Schemas in First-order Logic

In order to transform an ER schema into our intermediate representation, we use a first-order logic representation for the ER model. This representation comprises a set of predicates each corresponding to an ER concept. They are:

- **entity**($E, \text{Attr}, K$)
  
  where $E$ is the name of an entity, $\text{Attr}$ is a list of attribute definitions with the form $\text{Attr} = [(A_1, T_1), \ldots, (A_n, T_n)]$, where $A_i$ are attribute names and $T_i$ are their corresponding data types. $K$ is the key attribute(s), represented as a list of attribute names with the form $K = [A_1, \ldots, A_n]$ in which $A_i \in \text{Attr}$. The key is a set of attributes that identifies uniquely an instance of an entity.

  A simplified form of this predicate entity/1 is used to check whether the given argument denotes an entity. It is defined as $\text{entity}(E) \leftarrow \text{entity}(E, \_, \_)$.

- **relationship**($R, E_1, C_{\text{min}_1}, C_{\text{max}_1}, E_2, C_{\text{min}_2}, C_{\text{max}_2}, \text{Attr}$)
  
  where $R$ is the relationship name, $E_i$ are the entities involved and $C_{\text{min}_i}$ and $C_{\text{max}_i}$ are the minimum and maximum cardinalities for the participation of the
corresponding $E_i$ entity. $Attr$ is a set of attributes just like in the definition in entity/3.

We have also defined a simplified form relationship/1 in the same way as entity/1, that is, relationship($R_i$) ← relationship($R_{i-1}$, $\ldots$).

- **generalisation($SE$, $G$)**

  where $SE$ is a “super entity” and $G$ is a list of sub-entity names. The generalisation concept refers to a set of entities which have common attributes, which are put together to create a single “generalised” entity.

- **specialisation($SE$, $S$)**

  This definition is complementary to generalisation/3. The creation of a specialisation hierarchy starts with an entity which is categorised in different specialised entities, by identifying specific attributes for each category. Here the differences are highlighted whereas in the generalisation concept the emphasis is on the common attributes that forms the “super entity”.

Using this notation, part of the ER schema presented in Figure 5.1 is defined as:

entity(aircraft,
    [(model, string), (registration, string), (no.of_engines, integer),
     (type_of_engines, string), (year_of_manufacture, integer)],
    [registration])

relationship(fly, aircraft, 1, n, flight, 1, 1, [(date, date)])

Some additional auxiliary predicates are used for deriving intermediate representations in the next two sections. Some of these predicates correspond to the EER model concepts and they are self-explanatory. Other predicates are used to instantiate values. Details on the specification of pieces of information and transitions will be given in Chapter 6. The predicates are:

- **get_all_instances($E$, Info)**: Info is instantiated to a structure including all instances of entity $E$. A relationship can also be used in place of an entity.
Chapter 5. High-Level Specification of Web Site Applications

An example of a result for get_all_instances(aircraft, Info) is presented below. Note the correspondence of the values to the attributes defined in Table 5.2.


- get_instance(E, K, Info): Info is instantiated to one instance of entity E. Alternatively, the key attribute(s) K is provided for instantiating the corresponding attributes in Info. A relationship identifier can also be used in place of the entity identifier E. For example, get_instance(aircraft, ['N784UA'], Info) would result in:

  Info = info(aircraft, tuple, ['Boeing 777-200B', 'N784UA', 2, 'P&W PW 4090 turbofan engines', 1997])

- instantiate_val(E, Attr, Var): instantiate the variable Var with an instance value of attribute Attr belonging to entity E. A relationship can also be used in place of an entity. For example, in instantiate_val(aircraft, model, Var) the result can be:

  Var = 'Boeing 777-200B'.

5.3.2 Entity-based Intermediate Representation

We separate the definition of the intermediate representation in three parts: pieces of information, display units and transitions. Pieces of information correspond to the content of a Web site. They are defined by predicate info(L, T, D) where L is a label, T is data type and D is the actual value of the information. Display units, which give origin to Web pages, are defined by predicate display(D, SL) where D is a label and SL is a set of labels of pieces of information. Finally, transitions are denoted by the \( \Rightarrow \) operator. Transitions expressions have the form \( L_1 \Rightarrow L_2 \) where \( L_i \) is a label of a piece of information or a display unit. This is the notation used throughout this and the following sections. The intermediate representation is detailed in Chapter 6.
In the entity-based approach, pieces of information are defined by using predicate get_all_instances/3, as described in the previous section. All pieces of information are defined with data type table to take into account that they represent an entity (or a relationship N:N) including all their instances. The rules are described in Figure 5.2.

\[
\begin{align*}
\text{info}(E, \text{table}, \text{Info}_E) & \leftarrow \\
& \quad \text{entity}(E) \land \\
& \quad \text{get_all_instances}(E, \text{Info}_E) \\
\text{info}(R, \text{table}, \text{Info}_R) & \leftarrow \\
& \quad \text{relationship}(R, \ldots, n, \ldots, n, \ldots) \land \\
& \quad \text{get_all_instances}(R, \text{Info}_R)
\end{align*}
\]

Figure 5.2: Definition of pieces of information corresponding to entities

For each entity and relationship N:N a display unit is created including all its instances in a page as described by Figure 5.3. This is done by including the corresponding piece of information as defined by the previous rules, presented in Figure 5.2. The display unit identifier, E-page, is simply to make a distinction from the piece of information which has the same name of the entity.

Additionally, a simple fact is defined for keeping the relation between the entity and its display unit. This fact has the form: display_unit.label(E, E-page) where E is an entity name and E-page is the corresponding display unit label.

Transition rules related to relationships depend on the cardinality constraint defined. Relationships N:N require the construction of separate Web pages whereas 1:1 and 1:N relationships are represented by links between the related entities Web pages. Figure 5.4 shows the transition rules derived from 1:1 and 1:N relationships. Only two rules are necessary to define both incoming and outgoing transitions between the display units that represent entities participants in 1:1 and 1:N relationships. Note that there is a link only from the entity on the “1” side of the relationship.

Rules for N:N relationships are presented by Figure 5.5. This is different from the
previous case given that there is a display unit explicitly representing the relationship. In this case, the display units corresponding to the entities involved are linked to and from that display unit. A similar rule applies to n-ary relationships.

Transition rules corresponding to ternary and n-ary relationships are similar to those related to binary N:N relationships. For hierarchies of generalisation and specialisation, transitions are defined between the parent entity and its sub-entities. The corresponding rules to create these transitions are defined in Figure 5.6.

Applying these rules in the ER diagram described in Figure 5.1, the resulting structure is illustrated by Figure 5.7. Note the arrows representing the bi-directional links. Also note the display unit including piece of information $Info_{described}$ by which corresponds to a N:N relationship.

### 5.3.3 Instance-based Intermediate Representation

In this intermediate representation specification, a page is created for each instance of an entity. Transitions now are defined in terms of relationships between pieces of information and display units. Although, they follow a similar pattern to the entity-based intermediate representation, for each entity or relationship instance there will be an individual page. To take that into account, pieces of information are defined with data type tuple and predicate $get\_instance/3$ is used instead of $get\_all\_instances/3$. 

```-prolog
\[display(E\_page, [E]) \leftarrow\]
\[\text{entity}(E) \wedge\]
\[\text{info}(E, \_\_\_\_\_)\]
\[display(R\_page, [R]) \leftarrow\]
\[\text{relationship}(R, \_\_\_\_\_\_, n, \_\_\_\_\_, n, \_\_\_\_) \wedge\]
\[\text{info}(R, \_\_\_\_\_\_\_)\]
```

Figure 5.3: Definition of display units corresponding to entities
Figure 5.4: Transition rules for binary relationships 1:1 and 1:N

Figure 5.8 presents the rules that generate the pieces of information.

The rules used to generate display units are the same as those defined in the entity-based approach (presented in Figure 5.3). However, a different result will be achieved by the synthesis process, given that the pieces of information involved are of the type tuple. Usually there are many instances for one specification of a piece of information, which results in the creation of many different pieces of information, each corresponding to one entity instance. The value of the key attribute is appended to the page identifier in a later stage in order to make a distinction among the various instances of a same entity.

The specification of transitions are also based on the relationships defined in the ER schema but their definition follows slightly different rules from those defined in the previous approach. The rules for deriving transitions from relationships 1:1 and 1:N are shown in Figure 5.9.

Relationships N:N, again require the definition of four rules in order to create the transitions. Transitions are defined from the two pieces of information, representing
Figure 5.5: Binary relationship N:N and its transition rules
5.3. Deriving Intermediate Representations from ER Schemas

Figure 5.6: Transitions derived from generalisation hierarchies
Figure 5.7: Information Structure - Entity-based

![Diagram of information structure](image)

Figure 5.8: Definition of pieces of information corresponding to entities and relationships N:N

\[
\text{info}(E, \text{tuple}, \text{Info}_E) \leftarrow \\
\quad \text{entity}(E) \land \\
\quad \text{get}\_\text{instance}(E, \_\_, \text{Info}_E)
\]

\[
\text{info}(R, \text{tuple}, \text{Info}_R) \leftarrow \\
\quad \text{relationship}(R, \_\_, n, \_\_, n, \_\_) \land \\
\quad \text{get}\_\text{instance}(R, \_\_, \text{Info}_R)
\]

the two entities involved in a relationship, to the display unit representing the relationship. Similarly, transitions are defined from the piece of information representing the relationship to the two participants entities. Figure 5.10 presents the definition of these rules.

Rules for deriving transition rules from generalisation and specialisation hierarchies are similar to those presented in Figure 5.6 with the same modifications presented for the creation of transition rules from relationships.

Note that the only difference to the entity-based rules is that transitions are of the form: Piece of Information $\Rightarrow$ Display Unit. Entity-based transition rules involve
5.3. Deriving Intermediate Representations from ER Schemas

Figure 5.9: Transition rules for binary relationships 1:1 and 1:N

```
E1 ⇒ E2.page ←
    display_unit.label(E2, E2.page) ∧
    relationship(., E1, → 1, E2, → _)
E2 ⇒ E1.page ←
    display_unit.label(E1, E1.page) ∧
    relationship(., E1, → _, E2, → 1, _)
```

Figure 5.10: Transition rules for binary relationships N:N

```
E ⇒ R.page ←
    display_unit.label(R, R.page) ∧
    relationship(R, E, _ n, _ n, _)
E ⇒ R.page ←
    display_unit.label(R, R.page) ∧
    relationship(R, _ → _, n, E, _ n, _)
R ⇒ E.page ←
    display_unit.label(E, E.page) ∧
    relationship(R, E, _ n, _ n, _)
R ⇒ E.page ←
    display_unit.label(E, E.page) ∧
    relationship(R, _ → _, n, E, _ n, _)
```
only display units. Another important distinction is that these transition expressions generate many instances of links. This requires an evaluation of the actual instance values of the pieces of information in order to establish the correct links between them and the display units defined in the transitions. This is detailed in Chapter 8.

### 5.3.4 Query-based Intermediate Representation

This approach combines the two previous approaches allowing pieces of information to include all instances of an entity or single instances. Additionally it provides a means to filter unwanted information, preventing them from being presented on the Web site. However, this approach requires the designer to specify the filtering criteria. This might decrease the level of automation but increases the level of customisation of the Web site.

A straightforward way to implement this method is to associate SQL queries [Elmasri and Navathe, 1999] to the instantiation of pieces of information. Given that SQL is a popular and standard language for manipulating relational databases, it is reasonable to use it instead of creating another query language. The existence of many integration tools with programming languages, including Prolog, also facilitates the implementation of this approach. However, it also requires that the designer has some knowledge of SQL in order to specify the queries.

The example below illustrates a piece of information derived from entity aircraft, in which only aircrafts constructed before 1970 are included. In addition only attributes model and registration are part of the piece of information definition.

\[
\text{info(old_aircraft, table, T) } \leftarrow \\
T = \text{select model, registration} \\
\text{from aircraft} \\
\text{where year_of_manufacture < 1970}
\]

Definition of display units have to be made manually, since the pieces of information created from queries may be put together in one display unit. The organisation of pieces of information in display units becomes a design choice.
Transitions can still be automatically derived, since they are based on the concept of foreign keys. An additional simple fact can be derived from the piece of information specifications to describe the relation between them and their original entities or relationships. For example predicate assoc(PI, E) represents such relation, where PI is a piece of information label and E the name of an entity or relationship.

Another fact can record the presence of a foreign key and to which entity or relationship it refers to. For example predicate foreign_key(PI, FK, E), where PI is defined as above, FK is the foreign key attribute name and E is the referred entity or relationship.

With these two additional predicates we are able to automatically define transitions. For example, consider another piece of information as follows:

```
info(recent_flights, table, T) ←
    T = select date, flight.no, origin, destination, aircraft_registration
        from flight
        where date > 1999
```

Using the additional predicates assoc/2 and foreign_key/3 we can establish the relation between pieces of information recent_flights and old_aircrafts as defined earlier:

```
assoc(old_aircrafts, aircraft)
assoc(recent_flights, flight)
foreign_key(recent_flights, aircraft_registration, aircraft)
```

This information is derived from the ER schema presented in Figure 5.1, where the relationship fly between entities aircraft and flight has cardinality 1:N. This causes the definition of a foreign key in the table corresponding to entity flight making a reference to table aircraft. This is also illustrated in Tables 5.2 and 5.3.

Now that we are able to record and derive information the origins of query-based pieces of information, the following rule can be used to define transitions between them:

```
PI1 ⇒ PI2 ←
    foreign_key(PI1, _, E) ∧
    assoc(PI2, E)
```
A transition is defined between pieces of information PI1 and PI2 if there is a foreign key in PI1 referencing some entity E and piece of information PI2 is associated to that entity.

### 5.4 Summary

In this chapter we have discussed the use of high-level specifications as the starting point of Web site synthesis. We have chosen an entity-relationship model to use for that purpose. The use of high-level modelling formalisms requires a translation of the concepts of the formalism used to our intermediate representation.

We presented three transformation rules from an ER schema into our intermediate representation:

1. entity-based, where each entity or relationship is represented in a tabular structure in a single display unit;

2. instance-based, where each instance of an entity or relationship is placed in an individual Web page;

3. query-based, where the pieces of information are result of queries on the application database.

The result of those transformations is the specification of a complete intermediate representation including pieces of information, display units and transitions.
Chapter 6

The Intermediate Representation

This chapter presents the details of the intermediate representation for the information content of a Web site. It also defines navigational paths between the pieces of information that should be presented in the site.

The main challenge is to define appropriate logic expressions that represent a Web site application accurately and also give support to reasoning about the specification. The intermediate representation thus is the core of the proposed approach, upon which computations are performed in order to infer knowledge, verify properties and constraints.

The proposed approach prescribes a high-level description as the starting point of the synthesis process. For this reason, the intermediate representation is actually hidden from the Web designer, avoiding the need for manipulating the logic expressions directly. This allows more choices for defining logic expressions that represent a Web site application. One option is to provide a high-level description of a Web site, such as an entity-relationship schema and all the remaining details are taken care of by the synthesiser, as discussed in Chapter 5. An alternative is to offer a suitable user interface (visual, menu-based, etc) which can be used to obtain all the parameters necessary to proceed with the synthesis. Most of the concepts introduced in the intermediate representation are based on parameterisable components, which facilitates this task.

Although it is possible to translate a high-level representation of a Web site directly onto Web pages, we decided on this second level of representation because it brings additional benefits to the synthesis method. It allows the definition of a more
flexible Web site generation process because it is independent of any particular implementation. As a result all reasoning and other computation can be performed on the specification before committing to a particular visualisation style or implementation language. It makes possible to derive different Web sites from the same specification, depending on a variety of criteria such as classes of users, accessibility constraints or visualisation requirements.

The intermediate representation has undergone many developments during this research project. Previous work [Cavalcanti and Robertson, 2000, Cavalcanti and Robertson, 2002, Cavalcanti and Vasconcelos, 2002] presented earlier versions, showing how these concepts evolved over time. The following sections describe in detail each component of the final intermediate representation.

### 6.1 Pieces of Information

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.

A piece of information can directly correspond to a database item or it can be derived from a database as a result of queries. These are represented as additional facts or simple rules. The general format of a piece of information is

\[
\text{info}(\text{Label}, \text{Type}, \text{Datum})
\]

where

- Label is a unique identifier for the piece of information within the Web site specification.
- Type is a pre-defined data type.
- Datum describes a cluster of information associated with the label.

Figure 6.1 presents a context-free grammar for pieces of information, using a BNF notation. The Prolog conventions [Apt, 1996] are used for lists definition, that is, \([\text{Head}/\text{Tail}]\), and the empty list "[\]".


6.1. Pieces of Information

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

Figure 6.1: Grammar for Pieces of Information

Labels often correspond to attributes names of an entity (in an entity-relationship schema) or a table of a relational database. Labels may also be used as additional information about the data item, giving specific context to the information. Types are needed for the appropriate visualisation of the data as will be explained in Chapter 7. The pieces of information of a Web site must conform to the syntax above and are defined as Horn clauses. Any logic programming (Prolog) construct is allowed as the body of the Horn clauses defining a piece of information.

Compound information is supported by the array types and a table type. Type tuple includes elements of different scalar types. Type list is a tuple in which all elements have the same data type and table is a list of tuples.

For instance, Figure 6.2 shows the definition of some pieces of information based on the application example described in Section 5.2.1. They correspond respectively to an aircraft model, an instance of an aircraft including all its attributes and a table with all instances of aircraft. In this example constants start with a lower case letter and variables start with a capital letter, following Prolog's convention [Apt, 1996].

The technical details necessary to access a database should be included in the specification of a piece of information. This brings the benefit that only a piece of information needs to "know" what is the information source and how to access it. Throughout this thesis we use simple facts to represent database items, for simplicity reasons. Pred-
Chapter 6. The Intermediate Representation

\[
\begin{align*}
\text{info(aircraft._model, string, Model)} & \leftarrow \text{aircraft(Model, _, _, _)} \\
\text{info(aircraft._instance, tuple, [Model, Reg, NoEngines, TypeEngine, Year])} & \leftarrow \\
& \quad \text{aircraft(Model, Reg, NoEngines, TypeEngine, Year)} \\
\text{info(all._aircraft, table, T)} & \leftarrow \\
& \quad T = \{[\text{Model}, \text{Reg}, \text{NoEngines}, \text{TypeEngine}, \text{Year}] | \\
& \quad \text{aircraft(Model, Reg, NoEngines, TypeEngine, Year)}\}
\end{align*}
\]

Figure 6.2: Definition of pieces of information

icate aircraft presented in Figure 6.2 is an example of such database item.

Since each piece of information also includes the specification of how to obtain data items from a particular source, multiple sources of data is naturally supported by our approach. Information sources might include relational databases, text files, etc. Details about access to external sources are only needed for defining pieces of information. All further references to pieces of information are made via their labels.

Pieces of information are the building blocks of our approach to Web site construction. They represent the information content and define how to assemble that information from a particular source. They may also represent access to other pieces of information which will be achieved in the synthesised Web site by constructing hyperlinks to other pages as will be explained in Chapter 8.

Instantiation of pieces of information is performed only during the synthesis process. This allows all reasoning and property verification to be performed on the site specification, rather than on a particular instance of a Web site. The actual site may grow considerably depending on the number of instances of pieces of information. This would make constraints and properties difficult, if not impossible to check.
6.2 Display Units

Pieces of information are organised in display units, that is, "chunks" of pieces of information that should be presented together. Normally a Web page corresponds to such a display unit, but different alternatives can be considered, like the sections or chapters of an electronic book. Display units are represented as

\[
display(\text{DisplayLabel}, \{\text{InfoLabel}_1, \ldots, \text{InfoLabel}_n\})
\]

where

- \( \text{DisplayLabel} \) is a unique identifier for the display unit;
- \( \text{InfoLabel}_i, 1 \leq i \leq n \), are labels of pieces of information as previously defined.

Our display units define groups of information and provide them with a "handle" by means of which collective references can be made. This is a notational device to organise pieces of information but how to group them is a design issue. However, this task can be (partially) automated by using a conceptual data model to describe information content and deriving display units from it.

Figure 6.3 provides an intuitive example of display units of a Web site specification using pieces of information and display units. We shall use \( \text{Info} \in \text{Display} \) when \( \text{display}((\text{Display}, \text{SetInfo})) \) and \( \text{Info} \in \text{SetInfo} \).

| info(introduction,text,'Welcome to the web-site...') |
| info(reports,table,R) ← get_reports(R) |
| info(aircrafts,table,A) ← get_aircraft(A) |
| display(home,{introduction}) |
| display(reports_page,{reports}) |
| display(aircrafts_page,{aircraft}) |

Figure 6.3: Pieces of Information and Display Units
6.3 Operations

Dynamic Web sites have the ability to present information generated “on the fly”. One way to do this is to present an HTML form and trigger an operation after user input. Thus, operations are programs that perform some computation and present their results in a Web page. The proposed approach also supports the specification and automated generation of operations.

In order to implement the programs that correspond to the application operations, we make use of the CGI (Common Gateway Interface) technique [NCSA, 2002]. The CGI programs are written in Prolog which in turn are associated with HTML forms defined in the Web pages. Details of the architecture created to support Prolog-CGI programs is presented in [Vasconcelos et al., 2000].

The main concept behind our approach to operation synthesis is that of parameterisable components [Robertson and Agustí, 1999]. In this technique, simple parameters instantiation of pre-defined components is used to generate the code of a Prolog program. Parameterisable components are defined in terms of standard programming patterns which are kept in a component library.

The general format of an operation specification is:

\[
\text{op(Name, Type, InputArgs, OutputArgs, ExtraArgs)}
\]

where

Name is a unique identifier of the operation;

Type is a pre-defined operation type;

InputArgs is a list of input arguments which correspond to fields of a Web page form;

OutputArgs is a list of pieces of information, which are presented in the resulting display unit.

Target Predicate is the name of a predicate corresponding to a database item or a piece of information that the operation acts on (searching, updating, etc).
6.3. Operations

The type of an operation is used to select a programming pattern which will be used to synthesise the CGI program. A library with many programming patterns (components) each related to one operation type is created.

Table 6.1 presents an example of components. We include the most common operations which are those used for manipulating information in the application database, by searching, adding, updating and removing information. We also describe an additional operation to illustrate how other types of operation can be described and synthesised using this approach. Many different kinds of operation can be added to this library of components using similar descriptions. Note that operations do not change the navigation structure of a Web site. Conceptually they represent a transition from a piece of information to a Web page in the same way as a hyperlink. The difference is that an operation requires arguments that should be provided by the user of the Web site and all visualisation specification necessary to render the resulting Web page must be embedded in the CGI program. Since all the information needed for synthesising a Web page is available in the intermediate representation and in the visualisation description the operation can be automatically synthesised.

Operations can also be implemented using programming languages other than Prolog. This is facilitated by the fact that in a transition rule only the operation interface is necessary, giving details about the name and type of operation and its arguments. This feature promotes independence from a particular programming language and other implementational details.

The following steps illustrate the specification and creation of an operation. It defines an operation aircraft_by_year that searches for all aircraft constructed in a given year.

1. Specification

   op(aircraft_by_year, search, [aircraft.year], [aircraft.list], aircraft)

   This specification defines an operation called aircraft_by_year of type search, having the piece of information aircraft.year as input argument and aircraft.list as the output argument. The last argument aircraft corresponds to the entity that is the object of the query.
<table>
<thead>
<tr>
<th>Type</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>denotes a search operation which finds all solutions for a target predicate $P$ given some of its arguments instantiated. The arguments of predicate $P$ are used to construct a Web form in order to receive user input. Predicate $P$ must be a fact.</td>
</tr>
<tr>
<td>add</td>
<td>denotes an operation that adds a new fact to the knowledge base. The input arguments provide the values.</td>
</tr>
<tr>
<td>update</td>
<td>denotes an operation that updates the application database items corresponding to target predicate $P$. Two forms are constructed: one for inputting the arguments of $P$ with the current values for update and another for inputting the new values.</td>
</tr>
<tr>
<td>remove</td>
<td>denotes an operation that removes a fact from the knowledge base. The input arguments provide the values for matching the instance to be removed.</td>
</tr>
<tr>
<td>filter</td>
<td>denotes an operation that filters elements of a list and constructs another list with elements that pass the test defined by target predicate $P$. $P$ must have arity one and be defined beforehand.</td>
</tr>
</tbody>
</table>

Table 6.1: Components for defining operations
6.3. Operations

Note that input and output arguments must also be defined as pieces of information, in this case `aircraft_year` and `aircraft_list`. The reason is to provide visualisation information given by the data type of the pieces of information involved and to maintain uniformity of representation throughout the Web site specification. The specification of the two pieces of information involved are, for instance:

\[
\text{info}(\text{aircraft}\_\text{year}, \text{integer}, \_)
\]

\[
\text{info}(\text{aircraft}\_\text{list}, \text{table}, \_).
\]

Since the input argument results in a form item and the output argument is a result of the operation execution, the piece of information specification is a simple fact. There is also no need for a variable (third argument) which is denoted by `\_`. The data type information is important for visualisation purposes.

Figure 6.4 illustrates the two Web pages created from the specification of operation `aircraft_by_year`, the first with the form for receiving user input (`aircraft_year`), and the other with the resulting piece of information.

![Figure 6.4: Example of an operation](image)

2. Selecting a programming pattern

The type of the operation (in our example this is “search”) selects the appropriate component from the library of operations. One of the advantages of this approach is that the actual implementation of the operation can be changed without any impact to the specification as long it keeps the same number and type of arguments.
As an example, the expressions below present descriptions of some of the library of general operation components. The underlined terms are those which are replaced by the arguments from the operation specification.

- **Search operation**

\[
\text{SearchOp(InputArgs, Result)} \leftarrow \\
\text{Result} = \{ [\text{Args}] \mid \mathcal{P}(\text{Args}) \}
\]

where \text{SearchOp} is the name of the operation, \text{InputArgs} are the arguments given by the user, \mathcal{P} is the predicate related to an entity or relationship used for retrieving data from the database. The arguments of \mathcal{P}, denoted by \text{Args}, are defined in the ER model describing the Web site. Similarly, other programming patterns are defined as follows.

- **Add operation**

\[
\text{AddOp(InputArgs, Result)} \leftarrow \\
\text{add.to.DB}(\mathcal{P}(\text{InputArgs}), \text{Result})
\]

where predicate \text{add.to.DB}/2 inserts an element described by predicate \mathcal{P} in the application database. The result is simply a message describing the success of failure of the operation.

Operations update and delete are similar, only replacing predicate \text{add.to.DB} with \text{update.DB} and \text{delete.from.DB}, respectively. These predicates implements all the necessary machinery for performing a database operation.

- **Filtering operation**
6.3. Operations

\[
\text{FilterOp}([],[])
\]

\[
\text{FilterOp}([H|T1],[H|T2]) \leftarrow \\
\quad \text{Test}(H) \land \\
\quad \text{FilterOp}(T1,T2)
\]

\[
\text{FilterOp}([H|T1],T2) \leftarrow \\
\quad \neg \text{Test}(H) \land \\
\quad \text{FilterOp}(T1,T2)
\]

where \text{FilterOp} is the name of the filtering operation, \text{Test} is the name of a predicate used to filter the elements of the input list of items. The input argument must be a list of items, such that it can unify with the list denoted in the first argument.

3. Synthesising the program

An operation is synthesised by instantiating the parameters with the appropriate terms and variables. An example of a generated program corresponding to the search operation \text{aircraft\_by\_year} defined above specification is:

\[
\text{aircraft\_by\_year}(\text{Year}, \text{Result}) \leftarrow \\
\quad \text{Result} = \{([\text{Model}, \text{Reg}, \text{NoEng}, \text{TypeEng}, \text{Year}] \mid \\
\quad \quad \text{aircraft}(\text{Model}, \text{Reg}, \text{NoEng}, \text{TypeEng}, \text{Year})\}
\]

Note that the underlined terms that appeared in the programming pattern defined earlier are all replaced with the appropriate arguments.

Style, page template and other visualisation information concerning the pieces of information involved are embedded in the CGI program.

The site graph is not affected by the presence of operations in the intermediate representation. However, it is not part of the methodology to guarantee correctness of the
program that implements an operation. As a result the designer must verify the appropriate implementation of operations and its conformance to the interface specified in transition rules.

Although dynamic pages are supported by our Web site synthesis approach we concentrate only on the generation of static pages in the remainder of this thesis in order to keep the examples simple.

6.4 Transitions

Site navigation can be seen a sequence of actions, where each action is the display of a set of pieces of information in a unit of display, possibly followed by a transition to another set of pieces of information. Hence, transition rules represent the navigation structure of a Web site.

Transitions between portions of a Web site are specified by a binary “⇒” operator. Its general format is

\[ \text{Origin} \Rightarrow \text{DisplayUnit} \]

where \( \text{Origin} \) is either a piece of information label, a display unit label or an operation identifier and \( \text{DisplayUnit} \) corresponds to a label of a display unit. Intuitively, this operator represents a link from \( \text{Origin} \) to \( \text{DisplayUnit} \) (but not vice-versa). This creates three different kinds of transition:

1. \( \text{Info} \Rightarrow \text{DisplayUnit} \). In this case the piece of information \( \text{Info} \) will also be a link to the display unit. It means that wherever \( \text{Info} \) appears it always points to the same display unit.

2. \( \text{DisplayUnit}_1 \Rightarrow \text{DisplayUnit}_2 \). This sort of transition represents those in which there is no particular piece of information associated with the link. In this case a link is created in \( \text{DisplayUnit}_1 \) as a simple reference to \( \text{DisplayUnit}_2 \).

3. \( \text{Operation} \Rightarrow \text{DisplayUnit} \). This transition correspond to a dynamic page denoted by \( \text{DisplayUnit} \) which is created as a result of the execution of the operation. It usually involves user interaction by means of forms in order to receive the required input arguments.
A Web site will be defined by its individual pieces of information, operations, display units and transitions among these forged via the “⇒” operator. Figure 6.5 shows examples of transitions, linking pieces of information and display units presented in Figure 6.3.

\[
\text{home} \Rightarrow \text{reports.page} \\
\text{home} \Rightarrow \text{aircrafts.page} \\
\text{report} \Rightarrow \text{aircrafts.page}
\]

Figure 6.5: Sample Transitions

The initial page, also known as the homepage, needs to be identified explicitly because it is the starting point of all navigation paths. The homepage is defined by a simple predicate homepage(D) in which D is a label of a display unit. Usually the Web designer have to inform the system which page is the “homepage” as it would be very difficult and costly to determine the homepage automatically. When deriving a Web site from an ER diagram a standard homepage is created automatically avoiding this problem. This definition is used in the synthesis process as will be explained in Chapter 8.

There are some restrictions on the specification of transition rules. For example, pieces of information appearing in a transition rule must also be part of at least one display unit. This prevents the creation of orphan pages, i.e., pages that cannot be reached from the homepage. It also helps detecting mistakes on the specification, such as referring to a label of a non-defined piece of information or defining a link from a piece of information to a dynamic page. Since a dynamic page is generated as a result of an operation execution, it does not exist before the actual execution of the operation takes place.

The navigation structure of a Web site is defined by inspecting subset relations between pieces of information, display units and operations on the left and display units on the right side of transition expressions. As an example, the following display units and transition rules define the Web site of Figure 6.6.

\[
\text{display}(d1, \{a,b\})
\]
6.4.1 Interpretation of Transition Rules

Transitions of the kind Info $\Rightarrow$ DisplayUnit often require relating an instance of Info to some other piece of information in DisplayUnit. As an example, consider the following transition

aircraft_registration $\Rightarrow$ accident_report

which specifies that the piece of information aircraft_registration links to the display unit accident_report. However it does not define explicitly what is the relation between instances of aircraft_registration and instances of the pieces of information included in accident_report. There are two possible interpretations: any instance of aircraft_registration links to one display unit including all reports, or each instance of aircraft_registration links to individual corresponding display units of accident_report.
The correspondence between a piece of information and a display unit is actually made by checking corresponding pieces of information in the display unit. This relation determines whether all instances of the piece of information link to only one display unit or each instance links to a separate display unit. There are different types each of which are resolved by inspecting the relation between the piece of information and the display unit in a transition rule. These types are explained below.

1. \( I \Rightarrow D \land I \in D \)

   This is the simplest case where piece of information \( I \) is present in the target display unit. Hence, each instance of \( I \) links to an individual instance of \( D \) which includes the same instance of \( I \).

2. \( I \Rightarrow D \land \neg \text{compound type}(I) \land \exists I' \cdot I' \in D \land \text{compound type}(I') \land I \in I' \)

   where \( \text{compound type}(I) \) denotes that \( I \) is either a list, tuple or table. The relation \( I \in I' \) only applies when
   \( \neg \text{compound type}(I) \land \text{compound type}(I') \).

   This transition denotes a link between a simple piece of information \( I \) and a display unit that includes a compound piece of information in which one of its elements corresponds to \( I \).

3. \( I \Rightarrow D \land \text{compound type}(I') \land \exists I \cdot I \in D \land \neg \text{compound type}(I) \land I \in I' \)

   This transition denotes a link between an element of a list, tuple or table to a display unit containing a corresponding simple piece of information.

Returning to the example transition (aircraft_registration \( \Rightarrow \) accident_report), assuming that aircraft_registration is of type string and accident_report is a tuple, this transition corresponds to a type 2 transition. Figure 6.7 illustrates an example of this sort of transition.

Similarly, Figure 6.8 illustrates a type 3 transition, in which each element of a table links to a separate page that presents a corresponding value.

Although the specification is written in terms of labels, sets of labels and data types, the actual synthesis is done in terms of instances. This means that one display unit may
Figure 6.7: Page links example from a type 2 transition

Figure 6.8: Page links example from a type 3 transition
correspond to more than one Web page (with the same structure) and a transition rule may result in the creation of more than one link. Hence, pieces of information, display units and transition rules are notational devices for specifying sets of information, pages and links with common structure.

The transition types presented in this section give an idea of how the synthesiser is able to discern the different kinds of transitions and how to use the actual values of pieces of information to link information. More details about the synthesiser is given in Chapter 8.

6.5 Summary

In this chapter we presented the details of the intermediate representation. The components are the pieces of information, display units, operations and transitions.

Pieces of information describe the data items that will be presented in the Web pages. One piece of information can appear in more than one display unit and can also generate many instances from one specification.

Display units correspond to a conceptual notion of Web page, which is described simply as a set of pieces of information. One display unit can actually generate many Web pages, if one of its pieces of information has more than one instance. This concept is also important to separate visualisation details of Web pages which are only addressed in the synthesis process.

Operations are defined simply as a form of transition from one piece of information to a display unit. However, its realisation is unique as the piece of information involved is represented by a form to receive user input and the display unit generates dynamic Web pages, which are constructed as a result of some computation. We have implemented a Prolog-CGI architecture [Vasconcelos et al., 2000] for the implementation of our operations.

Transitions require a careful explanation in order to clarify their intended meaning and interpretation. We have defined transitions as connections between pieces of information or display units with other display units. The implementation of a transition is a link. One transition can also generate many links depending on the number of
instances of the pieces of information or display units involved.
Chapter 7

Visualisation

An important principle in our approach to Web site synthesis is the separation between content and visualisation. This feature is important in allowing definition and control over visualisation features without affecting the information content specification which includes pieces of information, display units and transition rules. The previous chapters have described how to model data and navigation aspects of a Web site application. This chapter explains how to relate those components to a visualisation specification.

In the proposed approach, visualisation is specified at three different levels: pieces of information styles, page templates and style sheets. The reason for separate descriptions is to support a high degree of control over the presentation of information and to facilitate maintenance. One of the premises of our approach with respect to visualisation is to allow changes in the presentation style of pieces of information individually and also in the overall appearance of a Web page. This gives extra flexibility in defining appropriate presentation styles for the Web site, enabling a Web designer to change and adjust each part separately.

Since Web pages are usually written in mark-up languages such as HTML [Committee, 2002], WML [WAP Forum, 2001] and a variety of XML-based languages [W3C, 2002a], in order to define a visualisation style for a Web site, it is necessary to relate components of our intermediate representation to elements of at least one of these target languages.

Often it is possible to relate a component of the intermediate representation to
more than one visualisation style. For example, a piece of information which includes a list of names can be displayed either as a bullet list or a table of names. This is the key to generating completely different-looking Web sites from the same specification. More importantly, changes in the visualisation styles have no impact in the application specification.

The following sections give details of how to define the visualisation style for pieces of information, Web page templates and how to maintain them.

7.1 Styles for Pieces of Information

In order to relate information content with a particular visualisation we use the data types defined in section 6.1. Similarly, a set of visualisation styles is defined. Examples of such styles include text, table, enumerated.list, itemised.list. By associating visualisation styles to data types, visualisation description becomes independent from any particular application.

A specific visualisation can be associated to data types using a simple predicate that relates a data type to a presentation style.

\[
\text{style}(\text{Type}, \text{Style})
\]

where Style is an identifier of a pre-defined visualisation style written in a target language and Type is a data type.

The reason to use data types is to maintain uniformity of presentation for all pieces of information of the same data type. Alternatively, the label of pieces of information may be used in order to define an exclusive style for individual pieces of information.

The concept of parameterisable components is once again used, for each visualisation style is related to a parameterisable piece of code, for instance in HTML. By defining a library of these components different options of presentation styles are made available.

The Pillow library [Cabeza and Hermenegildo, 1997] provides facilities for translating Prolog terms into HTML or XML terms. Pillow facilitates this transformation by offering ready-made components. Some examples of Pillow components are presented below. Figure 7.1 illustrates the method used to relate pieces of information to
visualisation styles and transform it into HTML code.

![Diagram of visualisation styles and HTML code process]

**Figure 7.1: Applying styles to pieces of information**

The data type of a piece of information is used to select the related visualisation style, then the selected style is applied to the instantiated information and finally transformed into HTML code via the Pillow library. Below we present a list of some of the Pillow constructs used here.

- **image(Addr, Atts)** includes an image of the address Addr. Atts is an optional list of attributes.
- **itemize(items)** creates a list of bulleted items.
- **enumerate(items)** creates an list of numbered items.
- **begin(Tag, Atts)** starts an HTML environment Tag with attributes Atts.
- **end(Tag)** ends an HTML environment Tag.
- **start_form(Addr, Atts)** specifies the beginning of a form. Addr is the URL of an operation (CGI program) which will handle the form.
• end_form specifies the end of a form.

• checkbox(Name, State) Specifies an input of type checkbox with name Name. State can be either “on” or “off”.

• input(Type, Atts) Specifies an input of type Type and an optional list of attributes Atts. Type can be text, hidden, submit, reset, etc.

In fact, the style identifiers are used to select one of the Pillow predicates which produces a piece of HTML code that corresponds to the chosen style. As an example consider a list of aircraft and their reports dates, represented by the piece of information info(aircraft_list, table, AN). It can be displayed, for instance, either as a bullet list or a table according to the style specifications style(list, bullet_list) or style(list, table). Note that not only the piece of information aircraft_list is affected by this specification, but all pieces of information of type table.

The transformation of the piece of information aircraft_list into a bullet list written in HTML is done by invoking the Pillow predicate itemize(AN), where variable AN must be an instantiated list of items. Invocation of Pillow predicates is done by our apply_style/2 predicate which uses its first argument to select the appropriate Pillow predicate. Although we use the same variable name (AN) to denote that it corresponds to the same data item defined in aircraft_list, the appropriate binding and instantiation of variables is done during the synthesis process, in accordance with the Prolog rules of unification and scope of variables [Apt, 1996]. This will be explained in the next chapter.

The definition of this piece of information and examples for its style specifications are:

\[
\begin{align*}
\text{info(aircraft_list, table, AL)} \\
\text{style(table, bullet_list)} \\
\text{style(table, simple_table)}
\end{align*}
\]

Figure 7.2 illustrates the process of transforming the piece of information aircraft_list into a presentational form.
7.2. Page Templates

Figure 7.2: Applying a style to piece of information aircraft_list

Figure 7.3 presents two possible renditions for the piece of information aircraft_list, resulting from the application of styles bullet_list (Figure 7.3a) and simple_table (Figure 7.3b).

7.2 Page Templates

Page templates are necessary to define the overall look of Web pages. Templates in this context correspond to diagrammatic descriptions, which should define a general layout and positions for placing pieces of information.

The benefit of having page templates is to provide a standard style to the whole Web site. It is commonly believed that a well-designed presentation of Web pages makes navigation easier and facilitates the overall use of the Web site by the users [Ivory et al., 2001].

There are different ways of defining a page template, ranging from simple text editors to sophisticated authoring tools which assist the designer in defining Web pages. The proposed approach does not require the use of any particular technique for defining the page templates. However there is a need for specific mappings from our intermediate representation. This can be done by defining special “tags” in the page template.
corresponding to labels of pieces of information, which will be replaced with the actual pieces of information values during the synthesis process.

The relation of a Web page template to an intermediate representation is defined by the labels of display units. For each display unit a page template must be assigned to allow its final rendering in a target language. Optionally, a unique template can be assigned to the whole Web site.

The approach allows for various forms of automated process for the generation of a visualisation, from using simple standard templates to a semi-automated generation of Web pages in which the designer has the option to intervene and change some aspects of the Web site presentation (we describe only the fully automated process, though). Figure 7.4 illustrates an example of a page template with constant visual elements (logo and bars) and a special tag \texttt{<aircraft/>} denoting the placement of that piece of information. These special tags must correspond to existing pieces of information and are used only to indicate where the corresponding piece of information should be placed on the Web page.
7.3. Style Sheets

The need for separating information from visualisation gave rise to the concept of style sheets. Style sheets describe how documents are presented on screens, in print, or perhaps how they are pronounced. The World-Wide Web Consortium (W3C) has actively promoted the use of style sheets on the Web. The Style Activity within W3C has produced several W3C Recommendations such as CSS1, CSS2, XPath, XSLT [W3C, 2002c].

Style sheets provide extensive stylistic control over the presentation of Web pages. Visualisation details such as font type, font size, colours, text alignment are all defined in a style sheet.

Style sheet languages work by overriding the presentation of HTML or XML elements with its own definitions. However, the degree of support for this technology vary depending on the Web browser used. As a result, a Web page might be seen slightly different if displayed on different Web browsers or even on different versions of the same browser. Although style sheets depend on browser vendors to achieve their full potential, the benefits of using a style sheet outweigh its disadvantages.

CSS [W3C, 1999] is a style sheet language widely implemented in current Web browsers. For that reason it is adopted as the style sheets language in our approach. An intuitive example of a CSS style sheet is presented in Figure 7.5.

Figure 7.4: Example of a page template
Figure 7.5: Example of a CSS style sheet
7.3. Style Sheets

This style sheet defines text fonts (arial and times), size (14pt) and the text colour (blue) and a background colour (white). These parameters are associated to the HTML tag body. It also specifies that images (tag img) are aligned to the right of the pages and defines specific colours and a text font for tables (represented by tags table and td).

All pages that include any of the tags above will conform to the specification of that style sheet. As a result, it becomes easy to change a particular visualisation of an element, for example the style of all tables presented in any page of the Web site can be updated by simply changing the styles defined for the tags table and td. Figure 7.6 presents the resulting Web page after applying the style sheet defined in Figure 7.5 to the Web page that includes a table presented in Figure 7.3. Note the changes in the text colour, font type, size and table colour.

![Figure 7.6: Example of a stylised Web page](image-url)

A suitable interface should be offered to the designer in order to input all necessary parameters to the style sheet. Currently, a standard CSS style sheet is automatically generated including definitions for each style.

Style sheets make possible the specification and modification of visual elements of a Web site without changing any other specification, including pieces of information styles and page templates. Updating a style sheet affects all corresponding elements of a Web site, hence it maintains the standard presentation for the site, facilitating
maintenance.

The reason to make use of style sheets is to keep the representation for our visualisation styles simple. Without a style sheet, details such as colours and fonts would need to be included as arguments to the mapping procedure to translate a visualisation style to HTML.

Given that the visualisation specification is separated into three different parts, pieces of information style, page templates and style sheets, there are different ways to change the presentation of information. Below we present the alternatives for visualisation maintenance:

1. Changing the style associated with the piece of information data type. In this case all pieces of information of the same data type will also have their style changed.

2. Changing the data type of a piece of information. As a result, the information is displayed according to the new data type style.

3. Changing the page templates. This is used for rearranging the positions of pieces of information.

4. Changing the CSS style sheet. This allows modifying colours, font types and sizes, margin alignments, etc.

7.4 Other Visualisation Issues

The proposed approach supports generation of many visualisations for the same specification. This is an important feature that is becoming increasingly important as usage of Web sites becomes more specialised and is attracting wider audience.

Visualisation requirements vary depending on the application domain. Nevertheless, there are some general requirements which are supported by our method for Web site synthesis. Examples of such requirements are:

- Multi-language Web sites
Templates and pieces of information styles are defined once for any language. Maintenance of the visualisation remains simple by changing only the templates, pieces of information styles and style sheets.

- Support for different classes of users

Some information might be presented in different ways depending on the user capabilities or permissions. For example some users might have limited visualisation capabilities such as those using mobile devices.

Another interesting example of this sort of application is presenting Web sites for visual impaired users. In this case, audio style sheets or special visualisation styles with large text fonts can be defined.

- Personalised Web sites

Personalisation of Web sites is also supported by our method. Basically it would involve defining an individual visualisation specification for each user, which might include aspect of the two previous points.

Multimedia applications such as Web-based games or e-commerce applications among others might have different visualisation requirements which are not supported by the proposed approach. In order to support such applications, technologies for presenting multimedia information, such as VRML [VRML Architecture Group, 1996], Macromedia Flash [Macromedia, 2002] and Java applets [Arnold et al., 2000], must be integrated in some way with the Web site synthesis method. Given that the proposed method is a data centred approach it requires additional research in order to verify how multimedia applications could benefit from this approach to Web site synthesis.

7.5 Summary

We have described in this chapter the visualisation specification. It is separated in three components:

1. Styles for pieces of information, which defines the presentational form of pieces of information based on their data types.
2. Page templates, defining a general layout of Web pages, including static visual elements and the arrangement of the pieces of information.

3. Style sheets, which defines visual details for both pieces of information and pages such as colour schemes, text fonts, among others.

We also have discussed how maintenance is facilitated by this approach by supporting the generation of different visualisations for the same specification.
Chapter 8

Automated Synthesis of Web Sites

The synthesis process is described in detail in this chapter, discussing the major decisions concerning the definition of the synthesiser.

An important choice for the definition of the synthesiser is the level of automation, the decision whether to implement a fully automated or a semi-automated synthesis method. Automation requires the system to be robust enough to cope autonomously with the various situations and different routes to follow when synthesising a Web site. In contrast, a semi-automated synthesiser has many decisions taken by the user, simplifying the implementation, but this interaction must be meaningful in human terms.

Here we propose a fully-automated synthesis method. It starts by translating an application description (entity-relationship schema) into an intermediate representation. From the intermediate representation a Web site is constructed. Figure 8.1 presents an overall description of the synthesis process.

Each box in Figure 8.1 represents the separate components of our approach to Web site synthesis. They are:

$\Sigma$ - Application description

$V$ - Visualisation description

$IC$ - Integrity constraints

$I$ - Intermediate representation
Formally we describe the process of deriving an intermediate representation from an application description as:

$$\Sigma, \text{IC} \mapsto \text{I}$$

Similarly the generation of Web pages from an intermediate representation corresponds to:

$$\text{I, V} \mapsto \text{W} \text{ given IC}$$

It is often possible to generate more than one intermediate representation from one application description, as we discussed in Chapter 5 different translations from an ER diagram into intermediate representations. This can be described formally as:

$$\Sigma \mapsto \{l_1, \ldots, l_n\}$$

Similarly, each intermediate representation can be combined with different visualisation descriptions:

$$\{l_1, \ldots, l_n\} \times \{V_1, \ldots, V_m\} \mapsto \{W_{[1,1]}, \ldots, W_{[n,m]}\}$$
In order to illustrate a complete synthesis process, an example of a simplified accident report Web site is given. This site presents information about aircraft and their corresponding incident reports, which is described in the following section.

8.1 An Example Scenario

The example presented here is based on the The United Kingdom Air Accidents Investigation Branch (AAIB) at http://www.aaib.dtlr.gov.uk/. The AAIB is responsible for the investigation of civil aircraft accidents and serious incidents within the UK. Its Web site presents a monthly publication of bulletins concerning aviation incidents since 1996. This is an interesting example of a data intensive Web site as bulletins are constantly added to the site or updated upon new evidence or information found about an incident. Currently, it has approximately 2000 reports.

Other accident reporting sites such as the National Transportation Safety Board (http://www.ntsb.gov/) and the Aviation Safety Network (http://aviation-safety.net) also have a similar organisation for presenting accident reports.

Given the high volume of information presented on these sites, an automated synthesis method facilitates maintenance and provides a basis for producing different visualisations for the reports.

The information presented in a large Web site application usually resides in the application database. The main information concerning an accident reporting application consists of data about aircraft and the reports themselves. Figure 8.2 presents the ER diagram that describes part of this application:

![Figure 8.2: ER diagram for accident reporting application example](image)

As incidents might involve more than one aircraft and the same aircraft may also be involved in more than one incident the relationship incident has maximum cardinality N:N. Aircrafts are described by their model, registration, number and type of
Chapter 8. Automated Synthesis of Web Sites

<table>
<thead>
<tr>
<th>model</th>
<th>reg.</th>
<th>no.eng.</th>
<th>engine type</th>
<th>year</th>
<th>photo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boeing 747-236B</td>
<td>G-BDXP</td>
<td>4</td>
<td>Rolls-Royce RB211 turbofan</td>
<td>1988</td>
<td>747.jpg</td>
</tr>
<tr>
<td>Douglas C47 Dakota</td>
<td>G-AMPZ</td>
<td>2</td>
<td>Pratt &amp; Witney R1830-92 piston</td>
<td>1944</td>
<td>dc3.jpg</td>
</tr>
</tbody>
</table>

Table 8.1: Example of aircraft data

<table>
<thead>
<tr>
<th>no.</th>
<th>report date</th>
<th>report text</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01/03/00</td>
<td>The aircraft had departed from Belfast Aldergrove Airport on an IFR night cargo flight to Coventry ...</td>
</tr>
<tr>
<td>2</td>
<td>01/05/02</td>
<td>The crew and aircraft were assigned to operate an 0545 hrs scheduled flight from Glasgow to Manchester ...</td>
</tr>
<tr>
<td>3</td>
<td>12/05/02</td>
<td>The flight departed from Boston USA and was inbound to London Heathrow Airport (LHR) ...</td>
</tr>
</tbody>
</table>

Table 8.2: Example of incident reports data

Engines, year of construction and a photo. Reports include a number, date and the textual description. A date for each incident is also recorded (attribute of the relationship incident). Tables 8.1, 8.2 and 8.3 give an example of a relational database for the accident reporting application which will be used throughout the synthesis description.

8.2 Mapping an Entity-Relationship Schema into an Intermediate Representation

The first step of the synthesis process is to create automatically a specification for pieces of information, display units and transition rules from an entity-relationship diagram.

There are different ways of transforming an ER schema into a corresponding in-
8.2. Mapping an Entity-Relationship Schema into an Intermediate Representation

<table>
<thead>
<tr>
<th>report no.</th>
<th>aircraft reg.</th>
<th>incident date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>G-AMPZ</td>
<td>17/12/99</td>
</tr>
<tr>
<td>2</td>
<td>G-RJXA</td>
<td>06/09/01</td>
</tr>
<tr>
<td>3</td>
<td>G-BDXP</td>
<td>12/12/00</td>
</tr>
</tbody>
</table>

Table 8.3: Example of incident data

Intermediate representation. In this section the three different approaches described in Chapter 5 are applied on the application example described in the previous section. Entity-based rules create pieces of information as structured lists including all instances of an entity or relationship in a single display unit. Instance-based rules define each instance of an entity as an individual piece of information which is displayed in separate display units. Finally, query-based rules generate pieces of information as result of queries, allowing filtering the information that will be presented on the Web site.

The goal of these transformations is to create an intermediate representation specification which will be used in the subsequent stages of the synthesis process. For simplicity, we concentrate on the generation of the aircraft and report Web pages as described in the previous section.

8.2.1 Entity-based Transformation

In this transformation, for each entity a piece of information is created with the entity name as its label and having data type table in order to include all instances of that entity. A display unit is also defined including only that piece of information.

Transitions are created according to the relationships in the ER schema. Regardless of the cardinality of the relationship, a transition is defined between the display units that correspond to the entities involved.

Some supporting predicates are used in this sort of transformation as described in Chapter 5. The following list describes those predicates:

- \( \text{get.all.instances}(E,ls) \): used to create a structure \((ls)\) including all instances of an entity \(E\).
• entity(E): succeeds if the argument is the name of an entity.

• relationship(R,E1,Cmin1,Cmax1,E2,Cmin2,Cmax2,Attr): defines a relationship R between entities E1 and E2. Cardinalities Cmin1, Cmax1, Cmin2, Cmax2 correspond to minimum and maximum cardinalities for the participation of E1 and E2 in R. Attr is a list of attributes, which may be empty if there are none.

• display_unit_label(E, L):
  this is a fact of the form display_unit_label(E, E_page) where E is an entity name and E_page is the corresponding display unit label. This is necessary in order to keep the relation between an entity and its display unit.

Below we describe the generation process of an intermediate representation corresponding to the ER diagram presented in Figure 8.2. It comprises three steps each one responsible for creating pieces of information, display units and transitions.

1. Pieces of Information

Given that entity-based transformation arranges all instances of an entity into one piece of information, this must be defined as a table data type. The following rule describes the specification of pieces of information from entities:

\[
\text{info}(E, \text{table}, \text{Info}) \leftarrow \\
\quad \text{entity}(E) \land \\
\quad \text{get.all.instances}(E, \text{Info})
\]

Relationships of cardinality N:N also require the definition of a piece of information. This piece of information includes the key attributes from the entities involved and relationship attributes, if any is defined:

\[
\text{info}(R, \text{table}, \text{Info}) \leftarrow \\
\quad \text{relationship}(R, \ldots, n, \ldots, n) \land \\
\quad \text{get.all.instances}(R, \text{Info})
\]
As an example, the resulting piece of information for entity aircraft is:

\[
\text{info(aircraft, table, Info)}
\]

where

\[
\text{Info} = [
\begin{array}{c}
\text{['Boeing 747-236B', 'G-BDXP', 4, 'Rolls-Royce RB211 turbofan', 1988, '747.jpg'],}
\text{['Douglas C47 Dakota', 'G-AMPZ', 2, 'Pratt & Witney R1830-92 piston', 1944, 'dc3.jpg'],}
\end{array}
\]

Note that values of \text{Info} come from the database, illustrated by Tables 8.1 and 8.3.

In a similar way, pieces of information \text{info(report, table, Info)} and \text{info(incident, table, Info)} are created from entities \text{report} and relationship \text{incident}, respectively.

2. Display Units

In order to create display units the following rule is used:

\[
\text{display(D, [E])} \leftarrow \\
\text{entity(E)} \land \\
\text{display_unit.label(E, D)}.
\]

Note that the piece of information included in the display unit has the same name of the entity. The resulting display units for this example application are:

\[
\text{display(aircraft.page, [aircraft])}
\]

\[
\text{display(report.page, [report])}
\]

\[
\text{display(incident.page, [incident])}
\]

Additionally, facts representing the relation between the entity (or relationship) with the display unit are created, as for example:
Chapter 8. Automated Synthesis of Web Sites

3. Transitions

The third step is to create the transitions using the rules:

\[
\text{E}_\text{page} \Rightarrow \text{R}_\text{page} \leftarrow \begin{align*}
\text{relationship}(R, E, \rightarrow, n, \rightarrow, n, \rightarrow) \wedge \\
\text{display} \_\text{unit} \_\text{label}(E, E\_\text{page}) \wedge \\
\text{display} \_\text{unit} \_\text{label}(R, R\_\text{page})
\end{align*}
\]

\[
\text{E}_\text{page} \Rightarrow \text{R}_\text{page} \leftarrow \begin{align*}
\text{relationship}(R, \rightarrow, n, E, \rightarrow, n, \rightarrow) \wedge \\
\text{display} \_\text{unit} \_\text{label}(E, E\_\text{page}) \wedge \\
\text{display} \_\text{unit} \_\text{label}(R, R\_\text{page})
\end{align*}
\]

\[
\text{R}_\text{page} \Rightarrow \text{E}_\text{page} \leftarrow \begin{align*}
\text{relationship}(R, E, \rightarrow, n, \rightarrow, n, \rightarrow) \wedge \\
\text{display} \_\text{unit} \_\text{label}(E, E\_\text{page}) \wedge \\
\text{display} \_\text{unit} \_\text{label}(R, R\_\text{page})
\end{align*}
\]

\[
\text{R}_\text{page} \Rightarrow \text{E}_\text{page} \leftarrow \begin{align*}
\text{relationship}(R, \rightarrow, n, E, \rightarrow, n, \rightarrow) \wedge \\
\text{display} \_\text{unit} \_\text{label}(E, E\_\text{page}) \wedge \\
\text{display} \_\text{unit} \_\text{label}(R, R\_\text{page})
\end{align*}
\]

With these rules, a transition is created in both directions between the display units that represent the relationship and the participant entities. In the application example the transitions created are:

- \text{aircraft \_page} \Rightarrow \text{incident \_page}
- \text{report \_page} \Rightarrow \text{incident \_page}
- \text{incident \_page} \Rightarrow \text{aircraft \_page}
- \text{incident \_page} \Rightarrow \text{report \_page}
8.2.2 Instance-based Transformation

This approach creates a piece of information for each instance of an entity and a display unit is defined including only that piece of information. Although instance-based pieces of information usually correspond to many instances the specification is unique because the structure of the piece of information is the same. This keeps the intermediate representation concise. The synthesiser is responsible for instantiating each instance and create the appropriate separate Web pages for each one of them. This will be explained in detail in Section 8.3.

Instance-based transformation rules need to deal with the database instance level. This means that access to actual values is necessary in order to define relationships between instances. Again, some supporting predicates are used to allow the manipulation of instances and values from the database. These predicates are described below.

- get_instance(E, K, I): used to create a structure (I) including one instance of an entity E. Alternatively, a value for the key attribute K can be given to retrieve a particular instance with that key value.

- foreign_key(E1, FK, E2): succeeds if there is a foreign key in the relation corresponding to entity E1 referencing the relation corresponding to entity E2.

1. Pieces of Information

The pieces of information in the instance-based intermediate representation have data type tuple. This is the only difference from the previous approach. The rules for creating pieces of information are:

\[
\text{info}(E, \text{tuple}, \text{Info}_E) \leftarrow \\
\quad \text{entity}(E) \land \\
\quad \text{get\_instance}(E, \_\_\_ \text{Info}_E)
\]

\[
\text{info}(R, \text{tuple}, \text{Info}_R) \leftarrow \\
\quad \text{relationship}(R, \_\_\_ \_\_ n, \_\_\_ \_\_ n, \_\_\_) \land \\
\quad \text{get\_instance}(R, \_\_\_ \text{Info}_R)
\]
Chapter 8. Automated Synthesis of Web Sites

and the pieces of information created are:

\[\text{info(aircraft, tuple, Aircraft\_Info)}\]
\[\text{info(report, tuple, Report\_Info)}\]
\[\text{info(incident, tuple, Incident\_Info)}\]

Note the use of predicate get\_instance/2 instead of predicate get\_all\_instances/2 which was used in the entity-based approach. Predicate get\_instance/2 returns one instance of a given entity or relationship. For example, one instance of entity aircraft is:

\[\text{AirCraft\_Info} = ['Boeing 747-236B', 'G-BDXP', 4, 'Rolls-Royce RB211 turbofan', 1988, '747.jpg']\]

2. Display Units

The creation of display units is done using the same rules as in the entity-based approach. As a result, the display units created are:

\[\text{display(aircraft\_page, [aircraft])}\]
\[\text{display(report\_page, [report])}\]
\[\text{display(incident\_page, [incident])}\]

Although the specification is identical to the previous approach, recall that each piece of information is defined with data type tuple which causes the creation of individual pages for each instance of the pieces of information, as we shall see later.

3. Transitions

Transitions are created according to the relationships defined in the ER schema. The cardinality of the relationship has a direct effect in the resulting transitions, as explained in Chapter 5. Here we present the rules for mapping relationships N:N into transitions expressions. The rules are:
8.2. Mapping an Entity-Relationship Schema into an Intermediate Representation

1. \( E \Rightarrow R_{\text{page}} \leftarrow \)
   \( \text{display}_\text{unit}_\text{label}(R, R_{\text{page}}) \land \)
   \( \text{relationship}(R, E, -, n, -, n, -) \)

2. \( E \Rightarrow R_{\text{page}} \leftarrow \)
   \( \text{display}_\text{unit}_\text{label}(R, R_{\text{page}}) \land \)
   \( \text{relationship}(R, -, n, E, -, n, -) \)

3. \( R \Rightarrow E_{\text{page}} \leftarrow \)
   \( \text{display}_\text{unit}_\text{label}(E, E_{\text{page}}) \land \)
   \( \text{relationship}(R, E, -, n, -, n, -) \)

4. \( R \Rightarrow E_{\text{page}} \leftarrow \)
   \( \text{display}_\text{unit}_\text{label}(E, E_{\text{page}}) \land \)
   \( \text{relationship}(R, -, n, E, -, n, -) \)

These rules result in the following transitions:

Using rule 1: aircraft \( \Rightarrow \) incident.page

Using rule 2: report \( \Rightarrow \) incident.page

Using rule 3: incident \( \Rightarrow \) aircraft.page

Using rule 4: incident \( \Rightarrow \) report.page

8.2.3 Query-based Transformation

This sort of transformation allows the designer to choose parts of the information from the database to display on the Web site, by specifying SQL queries. It is also one way of combining the entity-based and instance-based approaches for arranging information. It also allows the creation of pieces of information including data from more than one entity.
The two previous approaches create Web sites based on entities and relationships including all available data. There are cases, however, where not all information in the database can be displayed in a Web site. Selecting the appropriate information to display also offers another flexibility for constructing a Web site.

To achieve that, the designer should be able to select which entities, relationships, attributes and which instances will be part of the intermediate representation. Designers can specify their preferences for the information to be extracted from the database via a suitable interface. One representation for this specification has the form:

\[
\text{query(PI, E, Attr, Cond, Type)}
\]

where PI is a label of a piece of information, E is the name of an entity or relationship, Attr is a subset of the attribute set of E and Type is a data type. Cond refers to a list of conditions for selecting specific instances of entity E based on actual values of their attributes. Conditions must have the form:

\[
A \text{ Op } V
\]

where A is one attribute of entity E, Op is a comparison operator such as <, > or = and V is a scalar value.

This specification is actually translated into SQL queries in order to retrieve the information. For example, the following expressions define a Web site for the aircrafts and reports application where only aircrafts constructed before 1970 and reports made since year 2001 are included. The attribute names refer to those illustrated in Tables 8.1 and 8.2 in Section 8.1.

\[
\text{query(old_aircraft, aircraft, [model, reg], [year < 1970], table)}
\]

corresponding to the following SQL query:

```sql
select model, reg
from aircraft
where year < 1970
```

As another example, the specification

\[
\text{query(new_report, report, [no, report.date, report.text],}
[report.date > '31/12/2000'], tuple)
\]
corresponds to the SQL query:

```sql
select no, report.date, report.text
from report
where report.date > '31/12/2000'
```

For each query statement a piece of information and a display unit are created. Transitions are derived using the same rules as in the previous approaches since the original entities and relationships are defined in each piece of information specification. The resulting intermediate representation for this example is the following:

1. **Pieces of information**

   The pieces of information specification below are illustrated in a Prolog-like notation in order to show the selection of attributes and instance values.

   ```prolog
   info(old_aircraft, table, OA) ←
   OA = \{[M,R] | aircraft(M,R,Y,_,_) ∧ Y > 1970\}
   info(new_report, tuple, [N,D,T]) ←
   report(N,D,T) ∧ D > '31/12/2000'
   ```

2. **Display Units**

   The display units are derived from each piece of information. In order to achieve a fully automated method we create one display unit for each piece of information created. Although this might be limiting possible different designs, it is one simple way to implement the query-based approach. The rules are the same as those used in the two previous approaches.

   ```prolog
display(old_aircraft_page, [old_aircraft])
display(new_report_page, [new_report])
```

Instead of using predicate display_unit.label/2 to relate each display unit with the original entity, we have to use the display unit and query definitions. The reason for this is the fact that pieces of information no longer have the same name of
their originating entities. For example, the following rule can be used to find out which entity originated a display unit:

\[
\text{relates}(\text{DU, E} \leftarrow \text{display}(\text{DU, [PI]}) \land \text{query}(\text{PI, E, \_, \_})
\]

3. Transitions

Transitions are derived applying similar rules used in the instance-based approach. We use predicate \textit{query/5} to refer to the original entity and relationship names, necessary to perform the creation of transitions.

1. \textit{PI} \Rightarrow \textit{R\_page} \leftarrow
   \begin{align*}
   & \text{query}(\text{PI, E, \_, \_}) \land \\
   & \text{display}(\text{R\_page, [RI]}) \land \\
   & \text{query}(\text{RI, R, \_, \_}) \land \\
   & \text{relationship}(\text{R, E, \_, n, \_, n, \_})
   \end{align*}

2. \textit{PI} \Rightarrow \textit{R\_page} \leftarrow
   \begin{align*}
   & \text{query}(\text{PI, E, \_, \_}) \land \\
   & \text{display}(\text{R\_page, [RI]}) \land \\
   & \text{query}(\text{RI, R, \_, \_}) \land \\
   & \text{relationship}(\text{R, \_, n, E, \_, n, \_})
   \end{align*}

3. \textit{RI} \Rightarrow \textit{E\_page} \leftarrow
   \begin{align*}
   & \text{query}(\text{RI, R, \_, \_}) \land \\
   & \text{display}(\text{E\_page, [PI]}) \land \\
   & \text{query}(\text{PI, E, \_, \_}) \land \\
   & \text{relationship}(\text{R, E, \_, n, \_, n, \_})
   \end{align*}

4. \textit{R} \Rightarrow \textit{E\_page} \leftarrow
   \begin{align*}
   & \text{query}(\text{RI, R, \_, \_}) \land \\
   & \text{display}(\text{E\_page, [PI]}) \land \\
   & \text{query}(\text{PI, E, \_, \_}) \land 
   \end{align*}
relationship($R, \rightarrow, n, E, \rightarrow, n, \rightarrow$)

Assuming a piece of information incident corresponding to the relationship with the same name, examples of transitions created with the above rules are:

Using rule 1: old_aircraft $\Rightarrow$ incident_page

Note that in this case, $PI = old\_aircraft$, $R\_page = incident\_page$, $E = aircraft$, $RI = incident$ and $R = incident$.

Using rule 2: new_report $\Rightarrow$ incident_page

Using rule 3: incident $\Rightarrow$ old_aircraft_page

Using rule 4: incident $\Rightarrow$ new_report_page

8.3 The Synthesis Process

The core of the synthesis process is the manipulation of the intermediate representation in order to generate a Web site. This process comprises three main steps: instantiating data, creating links and applying a visualisation style to produce the Web pages. The generation process is basically a structure expansion procedure where each step produces a structure which is expanded by the subsequent steps until the final translation to a target language. This process is detailed in the following sections.

8.3.1 Instantiating Data

Instantiation of data is straightforward as the rules are already defined in the specification of pieces of information. The creation of links, however, requires a careful analysis of the transition rules. There are different types of links which are identified by the type of the piece of information defined in a transition. This distinction of link types is necessary for applying visualisation styles and making references to target Web pages correctly. In some cases, it is also necessary to inspect the actual values of pieces of information involved in a transition. Therefore links can only be created after the instantiation of all pieces of information. This process is detailed in Section 8.3.2.
Formally, the synthesiser is defined as:

\[
\text{create}(\text{Site}) \leftarrow \\
\text{gen\_pages}(P) \land \text{gen\_links}(P, L) \land \text{gen\_visualisation}(P, L, \text{Site})
\]

First, a set of page structures \((P)\) is created including all instantiated pieces of information and grouping them according to the definition of display units. \(L\) denotes a set of links derived from the instantiated page structure \(P\), by predicate \(\text{gen\_links}/2\). Finally, given the page structure \(P\) and the links definition, predicate \(\text{gen\_visualisation}/3\) creates the Web pages according to the visualisation specification and chosen target language.

A page structure has the form \(\text{page}(\text{Id}, \text{Is})\) where \(\text{Id}\) is a display unit identifier and \(\text{Is}\) is its set of instantiated pieces of information. Recall that each piece of information is a triplet \((L, T, D)\) where \(L\) is a label, \(T\) is a data type and \(D\) is an instantiated data item.

More than one page structure can be created from one display unit specification. This happens when there is more than one instance of a piece of information belonging to a display unit. The creation of a set of page structures is defined as:

\[
\text{gen\_pages}(P) \leftarrow \\
P = \left\{ \text{page}(\text{Id}', \text{Is}') \mid \\
\begin{align*}
\text{display}(\text{Id}, \text{Is}) \land \\
\text{instantiate\_info}(\text{Is}, \text{Is}') \land \\
\text{page\_id}(\text{Id}, \text{Id}')
\end{align*}
\right\}
\]

\[
\text{instantiate\_info}(\text{Is}, \text{Is}') \leftarrow \\
\text{Is}' = \{ (l, T, l') \mid \text{info}(l, T, l') \land l \in \text{Is} \}
\]

In the specification above \(\text{Id}\) is a display unit label, \(\text{Is}\) is its list of labels of piece of information and \(\text{Is}'\) denotes the corresponding set of instantiated pieces of information. As its name implies, predicate \(\text{instantiate\_info}/2\) creates \(\text{Is}'\) from a given \(\text{Is}\). Predicate \(\text{page\_id}(\text{Id}, \text{Id}')\) is necessary for generating unique identifiers for the pages, given that more than one page can be created from the same display unit.

As an example, taking the intermediate representation created from the entity-based transformation in Section 8.2.1, the resulting page structure is:
8.3. The Synthesis Process

\[ P = \{ \text{page(aircraft.page,} \]
\[ \text{\{aircraft, table, ["Boeing 747-236B", 'G-BDXP', ...]}) \}, \]
\[ \text{page(report.page,} \]
\[ \text{\{report, table, [1, '01/03/00', 'The aircraft had departed from ...', ...]}} \), \]
\[ \text{page(incident.page,} \]
\[ \text{\{incident, table, [1, 'G-AMPZ', '17/12/99', ...]}} \} \} \]

Note that the page structure includes as many instances of each piece of information that exist in the application database. An example of a page structure with multiple instances is presented below. This sort of structure results from the instance-based transformation. Note the page identifier, generated by predicate page.id/2.

\[ P = \{ \text{page(aircraft.page001,} \]
\[ \text{\{aircraft, table, ["Boeing 747-236B", 'G-BDXP', ...]}} \), \]
\[ \text{page(aircraft.page002,} \]
\[ \text{\{aircraft, table, ["Douglas C47 Dakota", 'G-AMPZ', ...]}} \), \]
\[ \text{page(aircraft.page003,} \]
\[ \text{\{aircraft, table, ["Embraer EMB-145EP", 'G-RJXA', ...]}} \} \} \]

8.3.2 Creating Links

As mentioned earlier the creation of links is not a simple task. Although the starting point is only the transition rules, it is usually necessary to inspect the actual values of the pieces of information involved in order to define the link correctly.

Links introduces the concept of link anchor, which refers to a piece of information or a simple label that links to another page. In practice the anchor is presented as a “clickable” element in a Web page that leads to another page. If the link anchor is a piece of information it has implications to the synthesis process. For example, if the link anchor is a list of items, it is possible that each item in the list points to individual pages. The target page is defined by matching the values of the pieces of information in the target page with the link anchor. Depending on the number of instances this can be an expensive operation. This happens because the intermediate representation defines structures as meta-data. This means that for each display unit more than one page can
be created and consequently for each transition very often there are more than one link created.

Nevertheless, in order to achieve a fully automated Web site generation, the synthesiser must be responsible for finding instances and matching values. An alternative would be to define pieces of information and transitions which always have only one instance, equating display units to pages and transitions to links. However, that would make the specification potentially very large and as a result property checking would be very difficult as pages with same structure would be defined in separate rules. One of the main reasons for the intermediate representation to be defined as meta-information instead of instances is to maintain the specification of a Web site concise and to facilitate property verification.

Recall that transitions are represented in the intermediate representation by the operator “⇒”. Transition expressions are of the form Origin ⇒ Target, where Origin corresponds either to a display unit identifier or to a label of a piece of information and Target denotes an identifier of a display unit. In order to generate links, more information is needed to represent a link than simply origin and target. During the synthesis, links among parts of the web site are represented as link(Origin, Anchor, Type, Target) where

- Origin is a label of a display unit.
- Anchor is the link anchor which can be either a label of a piece of information or the label of a display unit.
- Type corresponds to the data type of the Anchor if that is a piece of information, otherwise the label standard is used. The data type facilitates the application of visualisation style and value matching, specially in cases of lists, tuples and tables.
- Target is a label of a display unit.

The distinction of link types is necessary for the appropriate value matching and translation to a target language, such as HTML. Below we describe the different kinds of links.
• **standard link**: links in which Origin is also an identifier of a display unit, corresponding to transitions of the form \(D_1 \Rightarrow D_2\) where both \(D_1\) and \(D_2\) are labels of display units. This is the simplest form of link. It does not involve any particular piece of information and usually is rendered as a simple reference to the target page.

• **simple type link**: refers to links where Origin is a label of a piece of information of a simple data type (integer, string, etc). This results in a reference from the piece of information to the target page.

• **list type link**: refers to links where Origin is a label of a piece of information having data type list. This refer to transitions of the form \(P_1 \Rightarrow D\) where \(P_1\) is the label of a piece of information and \(D\) is the label of a display unit. Each item in the list can point to a different page. This sort of link requires an inspection in the list items matching those with data items on the target pages.

• **tuple type link**: similar to the previous type in which one attribute of a tuple will be the actual link anchor. That attribute corresponds to a foreign key defined in the database or it matches a value of a piece of information on the target page.

• **table type link**: this is a combination of the two previous types. Each row in a table correspond to a tuple, hence one of the attributes defined as the link anchor. Each attribute in the different rows can point to different target pages.

Although there are five different types of links, only two different rules are necessary to create the link structure, implemented by two predicates standard_links/1 and info_links/1. The final link structure is created by predicate gen_links/1 defined below. The resulting structure is a union of the sets of structures created by standard_links/1 and info_links/1.

\[
\text{gen_links}(P, L) \leftarrow \\
\quad \text{standard_links}(Ls) \land \\
\quad \text{info_links}(P, Li) \land \\
\quad L = Ls \cup Li
\]
Predicate standard\_links/1 creates standard links having the link anchor as the same target identifier. This is the sort of links that are used mostly in the entity-based transitions, where each transition are of the form DisplayUnit$_1$ $\Rightarrow$ DisplayUnit$_2$. The specification of this predicate is:

\[
\text{standard\_links}(Ls) \leftarrow \\
Ls = \left\{ \text{link}(\text{Origin}, \text{Target}, \text{standard}, \text{Target}) \mid \text{Origin} \Rightarrow \text{Target} \land \right. \\
\left. \text{display}(\text{Origin}_{\_}) \right\}
\]

A second rule creates links which have pieces of information as anchors. It makes a thorough verification on instances of pages and also on the actual values of pieces of information in order to determine the correct target. This rule is used to create links from transitions of the form PieceOfInformation $\Rightarrow$ DisplayUnit.

\[
\text{info\_links}(P, L) \leftarrow \\
L = \left\{ \text{link}(\text{Origin}, \text{Anchor}, \text{T}_1, \text{Target}) \mid \text{Anchor} \Rightarrow \text{Target} \land \right. \\
\left. \text{info}(\text{Anchor}, \text{T}_1, \text{V}_1) \land \right. \\
\left. \text{display}(\text{Origin}_{\_}, \text{Is}_1) \land \right. \\
\left. \text{Anchor} \in \text{Is}_1 \land \right. \\
\left. \text{page\_id}(\text{Origin}_{\_}\text{Origin}) \land \right. \\
\left. \text{display}(\text{Target}_{\_}, \text{Is}_2) \land \right. \\
\left. \text{info}(\_\_\_, \text{T}_2, \text{V}_2) \in \text{Is}_2 \land \right. \\
\left. \text{match\_value}(\text{V}_1, \text{T}_1, \text{V}_2, \text{T}_2) \land \right. \\
\left. \text{page\_id}(\text{Target}_{\_}\text{Target}) \right\}
\]

This rule works as follows:

1. get a transition Anchor $\Rightarrow$ Target$\_\_\_;$

2. get data type \text{T}_1 and value \text{V}_1 of a piece of information having Anchor as its identifier;

3. determine \text{Origin} as the page to which belongs the piece of information identified by Anchor;
4. get the set of pieces of information Is belonging to the display unit with identifier Target;

5. get data type T2 and value V2 of a piece of information belonging to Is;

6. check if V1 and V2 match;

7. if match_value succeeds, page_id/2 returns the unique identifier Target for the target page.

Matching values depends on the data type, for example matching a tuple with a table requires inspection of the attributes actual values.

In order to illustrate the definition of links we use the transition between aircraft and incident page. In the entity-based approach, all links are standard resulting from grouping all instances of an entity in a single display unit. A more interesting example is that same transition defined in the instance-based approach. In that approach, both pieces of information aircraft and incident are defined having data type tuple. Using the rules defined above to create the link structures, the result is:

\[
\text{link(aircraft\_page}_1, \text{aircraft, tuple, incident\_page}_3) \\
\text{link(aircraft\_page}_2, \text{aircraft, tuple, incident\_page}_1) \\
\text{link(aircraft\_page}_3, \text{aircraft, tuple, incident\_page}_2)
\]

These three expressions correspond each to an instance from the database, as presented earlier in Tables 8.1 and 8.3.

Note the label of pages which now correspond to individual instances. The numbers are just a means for illustrating labels that identify uniquely each page.

If the piece of information aircraft is defined as a table then the resulting expressions are:

\[
\text{link(aircraft\_page, aircraft, table, report\_page}_1) \\
\text{link(aircraft\_page, aircraft, table, report\_page}_2) \\
\text{link(aircraft\_page, aircraft, table, report\_page}_3)
\]

Note that from the same page aircraft\_page there are three links each one linking to a different page. Each link is defined on an element of the table. Table and list types
require a further inspection of the value used as the anchor for targeting the correct page.

Figure 8.3 illustrates the result as Web pages for the tuple and table type links respectively.

![Figure 8.3: Examples of tuple and table types links](image)

Figure 8.3: Examples of tuple and table types links

Figure 8.4 illustrates page structures with links resulting from the transition derived from entity-based transformation. This example shows instances of aircraft and incident instantiated and related via attribute aircraft registration, which is the one that has a matching value on the other page.

### 8.3.3 Applying Visualisation Styles

The last step of the synthesis process is applying a visualisation style to the page structures created so far. Visualisation style is defined in terms of the target language, which creates the Web site. In this example HTML is used as the target language.

Visualisation specification involves styles for the pieces of information, page templates for the display units and a style sheets for the pages. Style specifications are simple definitions as the following expression:

\[
\text{style(\text{table}, \text{table.style})}
\]

The style label table.style is not very informative, as it is meant to be used simply
as a style selector. Styles are pre-defined bits of code in a target language which will produce a specific visualisation style for a piece of information. Therefore it is necessary to make some assumptions on the structure of the information upon which the style will be applied. In this case the data type table indicates that the piece of information is a structure consisting of a list of lists. It is also possible to offer the designer a list of styles compatible with the piece of information to choose from. In the current solution, given that we propose a fully automated synthesiser, standard visualisations are associated with the pieces of information, although different visualisations can be attained by backtracking.

The piece of code below illustrates how the style \texttt{table.style} is applied to a piece of information. Items \texttt{Val}_{i,j} denote the data items that will be placed on those positions. These data items come from the pieces of information in the page structures.

\begin{verbatim}
< table >
  < tr >< td > Val_{1,1} < /td >< td > Val_{1,2} < /td > ... < td > Val_{1,n} < /td >
  :  
\end{verbatim}
Chapter 8. Automated Synthesis of Web Sites

A page template must also be provided to the Web pages, defining the overall look of pages. It only includes visual elements and special tags defines the places for the pieces of information and other elements created in the synthesis process. Figure 8.5 illustrates a simple page template used for the application example.

![Page Template](image)

Figure 8.5: Page template for the accident reporting Web site

The final synthesis step consists of a simple mapping on the set Pages combining each piece of information with its visualisation style. The resulting WebPages is a set of pages written in a particular target language. The following definitions describe how the mapping is done: visualise/2 builds the set of WebPages using the elements

\[
\text{visualise}(\text{Pages}, \text{WebPages}) \leftarrow \\
\text{WebPages} = \left\{ \begin{array}{l}
\text{WebPage} \\
\text{Page} \in \text{Pages} \land \\
\text{visualise}_\text{page}(\text{Page}, \text{WebPage})
\end{array} \right. \\
\text{visualise}_\text{page}(\text{page}(\text{Is}', \text{Ls}), \text{WebPage}) \leftarrow \\
\text{apply}\_\text{styles}(\text{Is}', \text{IsStyled}) \land \\
\text{transform}\_\text{links}(\text{Ls}, \text{TLs}) \land \\
\text{page}\_\text{template}(\text{IsStyled}, \text{TLs}, \text{WebPage})
\]
of Pages, via visualise_page/2. Each Page gives rise to a WebPage by first applying an adequate style to the contents of the page (apply_styles/2) and translating the links onto a suitable format (transform_links/2). The results of the styled page and transformed links are then used to assemble WebPage using the available set of templates (page_template/3).

One important aspect of the synthesis process is the complete independence between the visualisation generation and the information content and links generation. This allows, for instance, altering the visualisation by simply changing the visualisation specification, as for example,

\[ \text{style(table, text_style)} \]

This definition generates an alternative presentation for the same aircraft page, using the same transition rule set. Another way to change the visualisation is by defining different page templates. Specification of pieces of information and transitions remain the same.

Finally, we present examples of the resulting Web pages corresponding to entity aircraft. Figure 8.7 illustrates the result of the entity-based approach.

Figure 8.6 presents the resulting page for one instance of aircraft. For all other instances of aircraft similar pages are created.

8.4 Summary

In this chapter we have described the synthesis process, which includes derivation of intermediate representations from an ER diagram, instantiation of data, creation of links and the creation of Web pages.

We also presented three approaches for deriving intermediate representations from ER diagrams, namely entity-based, instance-based and query-based. Each of the approaches organises pieces of information, display units and transitions in different ways, resulting in the construction of different Web sites.

Creation of links has proven to be a complex task, especially deriving links between individual instances of pieces of information and display units. This requires inspecting the actual data values in order to find the correct related instances.
Chapter 8. Automated Synthesis of Web Sites

Figure 8.6: Aircraft page: Instance-based and tuple visualisation

Figure 8.7: Aircraft page: Entity-based and table visualisation
8.4. Summary

Applying visualisation styles to create the Web pages is also separated into three different steps: applying styles to pieces of information, combining these with page templates and creating style sheets.

The synthesis process is very flexible, as the combination of the different intermediate representations with visualisation specification can result in a large number of different presentations for the same application description. It is also extensible as new intermediate representations and visualisation styles can be easily added to the system.
Part III

Additional Features
Chapter 9

Integrity Constraints and Properties

The 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.

Computational logic can be successfully used to tackle this problem as it supports declarative specifications and reasoning about specifications in a more natural way. This chapter discusses in more detail two important features of the proposed approach: the support for property checking and integrity constraints specification and verification.

Constraints and properties are checked against the intermediate representation, which represents a Web site specification, avoiding verifying particular instances of a Web site. This reduces significantly the size of the specification making the verification rules less computationally expensive to run. Constraints and properties focus on the information and navigation structure specification, affecting the organisation of the pieces of information in display units and the transitions among them.

The goal of these verifications is to produce a valid intermediate representation with respect to the constraint definitions, where some are related to errors and other are related to application specific properties.

Visualisation descriptions are defined apart from the intermediate representation, hence they do not have any effect on the majority of properties and constraints discussed in this chapter. Nevertheless, some specific visualisation properties can be verified separately as it is presented in section 9.3.
Different formalisms have been investigated for specifying and verifying Web site properties. The required condition is that the formalism can be easily and appropriately specified in first-order logic. Hypergraphs and transaction logic are the two formalisms adopted, from which a sub-set of their complete theories are used to suit our needs. The concepts used from both formalisms are detailed in the following sections.

9.1 Web Sites as Hypergraphs

The components of the intermediate representation are notational devices to define a special kind of directed hypergraph [Gallo et al., 1992]. 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 \( H = (V, E) \) where \( V = \{v_1, \ldots, v_n\} \) is the set of nodes and \( E = \{e_1, \ldots, e_m\} \) is the set of hyperedges. In our Web sites, each node \( v_i \) is equated with a piece of information. Each edge is a pair \( e = (h_e, T_e) \) with \( T_e \subseteq V \) and \( h_e \in V - T_e \). The display units comprise the sets \( T_e \) that can appear in edges, thus

\[
I \Rightarrow D \wedge \text{display}(D, Ps) \iff (I, Ps) \in E
\]

where \( I \) is a label of a piece of information, \( D \) is a display unit label and \( Ps \) is a set of pieces of information. We restrict our specifications to those hypergraphs that only allow one-to-many hyperedges and only one hyperedge should leave a node. More specifically, it is possible to have a connection between a piece of information and a display unit (with many pieces of information) but this connection should be unique.

Figure 9.1 illustrates a hypergraph in a graphical notation and its corresponding formal definition. The sets correspond to display units and each node corresponds to a piece of information.

A Path \( P_{ab} \) from piece of information \( a \) to piece of information \( b \) in a hypergraph is a sequence \( P_{ab} = (v_1 = a, e_1, v_2, e_2, \ldots, e_q, v_{q+1} = b) \), where \( a \in T_{e_1}, h_{e_q} = b \) and \( h_{e_i} \in T_{e_{i+1}}, i = 1, 2, \ldots, q-1 \).

Clearly, loops make property checking not feasible as an infinite number of paths can be derived from a graph. For that reason only acyclic paths are considered for constraint checking. A finite number of acyclic paths can be derived following the
9.1. Web Sites as Hypergraphs

rules:

1. all paths begin on pieces of information of a root display unit (denoted as the homepage);

2. no display unit can be revisited;

3. all paths end in pieces of information belonging to a leaf display unit, i.e. display units that have no outgoing links.

Figure 9.2 shows an example of paths extracted from a hypergraph.

Hyperpath is another important concept from hypergraphs used here. A hyperpath $P_{aD}$ from node $a$ to the set $D$ is a minimal acyclic sub-hypergraph containing both node $a$ and the nodes in $D$, such that each node, with the exception of the nodes in $D$, has exactly one incoming hyperarc. The concept of hyperpath is one way for proving reachability of display units. It answers questions such as “is it possible to reach display unit $D$ from piece of information $I$?”. For example, Figure 9.3 presents a hyperpath $P_{bD_2}$ extracted from the hypergraph presented in Figure 9.2.

Although hypergraphs 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 ap-

$$H = \{(a,b,c,d,e,f,g,h), \{(a, \{d, e\}), (b, \{f, g\})\}\}$$

Figure 9.1: Illustration of a hypergraph
path₁ = {a, (a,D2), d}
path₂ = {a, (a,D2), e}
path₃ = {b, (b,D3), f, (f,D2), d}
path₄ = {b, (b,D3), f, (f,D2), e}
path = {b, (b,D3), g}

Figure 9.2: Illustration of acyclic paths

Figure 9.3: Example of a hyperpath

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. 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. Other examples of interesting properties of a Web site application are broken links checking and a site navigator.

9.2 Web Site 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. 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. Other examples of interesting properties of a Web site application are broken links checking and a site navigator.
We further require that some basic properties hold in the representation of a Web site in order to ensure that the Web site specification is well defined. For example, all pieces of information and display units must be uniquely defined, that is the identifying label in pieces of information and display units can occur only once within a web site. Below we present a list of these properties and the rules to verify them.

1. A desirable property of our web sites is that all its pieces of information should be part of a display unit. More formally, we have

\[ \forall I, T, R \exists D, Is \text{ info}(I, T, R) \rightarrow \text{display}(D, Is) \land I \in Is \]

2. A complementary restriction ought to be imposed on the display units: they must all consist only of defined pieces of information, that is,

\[ \forall D, I, Is \exists T, R (\text{display}(D, Is) \land I \in Is) \rightarrow \text{info}(I, T, R) \]

3. Transitions can only be forged between an existing piece of information and an existing display unit – additionally the piece of information must not be one of the pieces of information of the display unit. This property can be formally stated as

\[ \forall I \forall D(I \Rightarrow D) \rightarrow (\text{info}(I, T, R) \land \text{display}(D, Is) \land I \notin Is) \]

4. Uniqueness of information units. Each information unit has to have a unique definition. It is not possible to have two information units with the same label, formally

\[ \forall I \exists T_1, T_2, V_1, V_2 (\text{info}(I, T_1, V_1) \land \text{info}(I, T_2, V_2)) \rightarrow T_1 = T_2 \land V_1 = V_2 \]

5. Uniqueness of display units. Each display unit has to have a unique definition. It is not possible to have two display units with the same label.

\[ \forall D \exists Is_1, Is_2 (\text{display}(D, Is_1) \land \text{display}(D, Is_2)) \rightarrow Is_1 = Is_2 \]
6. Uniqueness of transition anchor. If a piece of information is the anchor of transitions, it must always link to the same display unit.

$$\forall I \exists D_1, D_2 (I \Rightarrow D_1 \land I \Rightarrow D_2) \rightarrow D_1 = D_2$$

### 9.2.1 Information Reachability

A very important property of any web site is the *reachability* of pieces of information from a given initial display unit. Reachability can be best understood as the transitive closure of our transitions. Formally, $I \Rightarrow D$ if

1. $I \Rightarrow D$
2. $I \Rightarrow D_t \land I_t \in D_t \land I_t \Rightarrow D$ for some $D_t$ and $I_t$.

Case i) depicts direct transitivity; case ii) shows indirect transitivity via an intermediate display unit $D_t$. The initial display unit is the point of access offered to anyone willing to visit the web site. If we assume that an initial display unit exists as homepage($D$), then a given piece of information $I$ is reachable from the homepage if

1. it is displayed in the homepage itself;
2. it is displayed in a unit immediately linked from the homepage; or
3. it is displayed in a unit which is linked from a reachable unit of display.

Note that steps 2 and 3 denote exactly the transitive closure of transitions as defined above. Hence, reachability of a piece of information $I$ can be defined formally as

$$\exists D \text{ homepage}(D) \land I \in D \land \exists D' \exists I' \text{ homepage}(D) \land D' \neq D \land I' \in D' \land I' \Rightarrow D' \land I \in D'$$

### 9.2.2 Site Navigator

A Web site property related to the topology of the site graph is the site navigator. In the 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 \( \text{ib}(P) \). This refers to the number of links to a page \( P \).

2. Page rank backlink \( \text{ir}(P) \). This refers to the weighted sum of backlinks to a particular page \( P \). Formally, \( \text{ir}(P) = \sum_i \text{ib}(P_i) \) where \( P_i \) are pages that links to \( P \). Informally this concept means that the more pages \( P' \) that links to \( P \) having themselves a high \( \text{ib}(P') \), greater is the importance of \( P \).

By defining appropriate thresholds to \( \text{ir}(P) \) a navigator for the Web site can be derived automatically. For example, using the site representation presented in Figure 9.4, the backlink counters are \( \text{ib}(\text{homepage}) = 4, \text{ib}(\text{books}) = 3, \text{ib}(\text{cds}) = 3, \text{ib}(\text{book\_page}) = 2, \text{ib}(\text{cd\_page}) = 2, \text{ib}(\text{shopping\_cart}) = 5, \text{ib}(\text{payment}) = 1 \) and \( \text{ib}(\text{confirmation}) = 1 \).

Note that \( \text{book\_page} \) and \( \text{cd\_page} \) actually represent a set of pages with same structure (each page for a book or cd instance) and for that reason it is not necessary to calculate a counter for each instance of a page individually. The page rank backlinks are:

\[
\begin{align*}
\text{ir}(\text{homepage}) &= 10, \\
\text{ir}(\text{books}) &= 11, \\
\text{ir}(\text{cds}) &= 11, \\
\text{ir}(\text{book\_page}) &= 5, \\
\text{ir}(\text{cd\_page}) &= 5, \\
\text{ir}(\text{shopping\_cart}) &= 11
\end{align*}
\]
Chapter 9. Integrity Constraints and Properties

\[
\text{ir(payment)} = 5
\]
\[
\text{ir(confirmation)} = 1
\]

In this example, if a threshold for \( \text{ir} \) is 9, then our navigator includes homepage, books, cd’s and shopping_cart.

A specific visualisation for the navigator is defined in the same way as any other piece of information using predicate style.

\[
\text{style(navigator, navigator_style)}
\]

where \( \text{navigator\_style} \) defines a specific presentation style for the navigator.

Once a navigator is defined it is also treated as a piece of information and will be presented in all pages generated. The visualisation of the navigator depends on the style defined for it and its place in a page depends on the layout of the page template.

9.2.3 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:

\[
\text{broken}([], [])
\]

\[
\text{broken}([\text{Link} | \text{MoreLinks}], \text{BrokenLinks}) \leftarrow
\]

\[
\neg \text{is\_broken(Link)} \land
\text{broken}(<\text{MoreLinks}, \text{BrokenLinks})
\]

\[
\text{broken}([\text{Link} | \text{MoreLinks}], [\text{Link} | \text{MoreBrokenLinks}) \leftarrow
\]

\[
\text{is\_broken(Link)} \land
\text{broken}(\text{MoreLinks}, \text{MoreBrokenLinks})
\]

where predicate \( \text{is\_broken/1} \) includes all the machinery needed for accessing the URL of a Web page and verifying the response from the HTTP server.
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.

### 9.3 Visualisation Properties

Some simple verification can be performed on the visualisation specification, allowing the detection of common mistakes or encouraging the use of some design standards. Properties can be used to ensure that the visualisation specification is complete and correct. Two properties of this sort are:

- all display units must have at least one associated template

\[ \forall D \exists T \ (\text{display}(D, I_s) \land \text{template}(D, T)) \]

- all pieces of information must have a corresponding visualisation style

\[ \forall I \exists S \ \text{style}(I, S) \]

Other properties concerning the visualisation specification involve verifying the compatibility of data type and its style. There are some styles which are more appropriate to some specific data types. Simple data types such as strings and integers clearly cannot be displayed using the same style used for a table and vice-versa. Displaying a table or a tuple as text paragraphs is not recommended either, although it is possible.

Other sorts of verification can be done in order to promote a better visualisation of information. Once the visualisation styles and templates are defined, simple verifications can be made to ensure the site design follows some standard. For instance, a large number of page templates used might indicate a problem as a uniformity of presentation of information is usually desired in order to make navigation in the site easier for users.

These checks can result in recommendations and alerts to the designer as they might not necessarily indicate mistakes in the Web site design.
9.4 Representing Constraints using Transaction Logic

The set of acyclic paths extracted from the hypergraph are used to check constraints on the order of presentation of information. In order to specify the constraints we make use of some concepts borrowed from Transaction Logic (TR) [Bonner and Kifer, 1995]. TR is a declarative formalism for specifying and executing procedures that change a logical theory, for example a knowledge base.

TR is also an extension of predicate logic, whose transactions can be expressed naturally as logic programs. TR has a “Horn” version with both declarative and procedural semantics. This feature makes TR suitable and compatible with our approach to Web site construction, in which logic programming is the main tool for both specification and execution of synthesis.

Another reason for using TR is that it provides a rich language for expressing constraints on sequences of actions. In a Web site context actions correspond to visiting a Web page or visualising a particular piece of information and sequences relate to the information presentation ordering, as the paths extracted from the Web site hypergraph.

We use two concepts from transaction logic, serial conjunction and path, that were adapted to represent the sort of constraints we need. Serial conjunction is a logical connective used to represent a sequence of actions. This is written in the form $\phi \otimes \psi$ meaning “do $\phi$ then do $\psi$”.

This also defines a path formed of action $\phi$ followed by action $\psi$. The simplest path contains a single element which is a label of a piece of information, as defined in Section 6.1. Hence path expressions are of the form:

$$\text{Info}_1 \otimes \text{Info}_2 \otimes \ldots \otimes \text{Info}_n$$

The second useful concept taken from TR is a special symbol $\text{path}$ which corresponds to a sequence of actions of any length. Formally this concept is defined by $\text{path} \equiv \phi \lor \neg \phi$, for any formula $\phi$. This concept allows us to write simplified expressions. For example, the expression $\text{path} \otimes \text{Info}$ specifies a path which ends in $\text{Info}$. Similarly, $\text{Info} \otimes \text{path}$ specifies a path which starts with $\text{Info}$ and $\text{path} \otimes \text{Info} \otimes \text{path}$ defines a path which passes through information $\text{Info}$.

The most interesting expressions are those which relate two pieces of information
9.4. Representing Constraints using Transaction Logic

<table>
<thead>
<tr>
<th>Constraint Expression</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \neg (\text{path} \odot \neg \text{Info}_1 \odot \text{path} \odot \text{Info}_2 \odot \text{path}) )</td>
<td>( \text{Info}_1 ) must be displayed before ( \text{Info}_2 ).</td>
</tr>
<tr>
<td>( \neg (\text{path} \odot \neg \text{Info}_1 \odot \text{Info}_2 \odot \text{path}) )</td>
<td>( \text{Info}_1 ) must be displayed immediately before ( \text{Info}_2 ).</td>
</tr>
<tr>
<td>( \neg (\text{path} \odot \text{Info}_1 \odot \text{path} \odot \neg \text{Info}_2 \odot \text{path}) )</td>
<td>( \text{Info}_2 ) must be displayed after information in ( \text{Info}_1 ).</td>
</tr>
<tr>
<td>( \neg (\text{path} \odot \text{Info}_1 \odot \neg \text{Info}_2 \odot \text{path}) )</td>
<td>( \text{Info}_2 ) must be displayed immediately after ( \text{Info}_1 ).</td>
</tr>
</tbody>
</table>

Table 9.1: Common constraint expressions

defining an order in which the information must be presented if they appear in any path of the Web site. For example the expression:

\[
\text{path} \odot \text{Info}_1 \odot \text{Info}_2 \odot \text{path}
\]

denotes any path that presents piece of information \( \text{Info}_1 \) immediately followed by \( \text{Info}_2 \). Since a path is simply a sequence of information, constraints on a Web site can be expressed in terms of valid/invalid paths. Paths can be derived from the site hypergraph, as explained in the previous section. Table 9.1 presents some common constraint expressions.

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 pieces of information labels. Constraint checking can be done by matching paths expressions with constraint expressions. This is explained in the following section.

The designer is the one responsible for defining the appropriate constraints. A simple user interface can be constructed to help building constraint expressions as only information labels are needed. Standard expressions such as those presented in the table above can also be used to guide the definition of constraints.
Chapter 9. Integrity Constraints and Properties

9.5 Enforcing Constraints

As explained earlier, 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. Web sites with complex navigational structures can benefit of this sort of constraint checking for preventing errors.

In order to exemplify the application of constraints, we introduce an example of an e-commerce application. This application concerns an online retailer of books and CDs. Figure 9.4 presents the organisation of the site showing display units and the transitions.

![Diagram of e-commerce application example](image)

Figure 9.4: E-commerce application example

Pages books and cds present a list of the books and cds available in stock. The book_page and cd_page contain information about a particular book or cd and they are linked to each other in cases where the cd is related to a book. The shopping_cart presents the list of items selected for purchase and the total amount to be paid, then payment collects payment details and confirmation as expected shows a confirmation of purchase.

For simplicity we assume one piece of information in each display unit and we will
use the same name of the display unit as the label of the piece of information. In a real situation these labels must be different and other information can also be included in these display units.

The site hypergraph can also be seen in Figure 9.4, since each display unit has only one piece of information. The paths extracted are:

\[ p_1 = \text{homepage} \otimes \text{books} \otimes \text{book\_page} \otimes \text{shopping\_cart} \otimes \text{payment} \otimes \text{confirmation} \]

\[ p_2 = \text{homepage} \otimes \text{books} \otimes \text{book\_page} \otimes \text{cd\_page} \otimes \text{shopping\_cart} \otimes \text{payment} \otimes \text{confirmation} \]

\[ p_3 = \text{homepage} \otimes \text{cds} \otimes \text{cd\_page} \otimes \text{shopping\_cart} \otimes \text{payment} \otimes \text{confirmation} \]

\[ p_4 = \text{homepage} \otimes \text{cds} \otimes \text{cd\_page} \otimes \text{book\_page} \otimes \text{shopping\_cart} \otimes \text{payment} \otimes \text{confirmation} \]

The following expression illustrates an example of a constraint:

\[ c_1 = \neg (\text{path} \otimes \neg \text{payment} \otimes \text{confirmation} \otimes \text{path}) \]

This constraint defines that piece of information payment must appear immediately before confirmation. Considering the paths and the constraint above, it is possible to conclude that constraint \( c_1 \) is satisfied. Note that the special predicate \( \text{path} \) matches paths of any length and the negation in the constraint means that paths which have that pattern are not valid. As none of the paths \( p_j \) can match the pattern, they are all valid. On the other hand, the following constraint

\[ c_2 = \neg (\text{path} \otimes \neg \text{book\_page} \otimes \text{path} \otimes \text{shopping\_cart} \otimes \text{path}) \]

specifies that the shopping cart must be presented before the book page. Given path \( p_2 \) constraint \( c_2 \) is not satisfied. Informally, it can be verified that in that path, \( \text{shopping\_cart} \) is displayed without \( \text{book\_page} \) being displayed before it. If that is the case, display units \( \text{book\_page} \) and \( \text{cd\_page} \) could not be linked to each other.

### 9.6 Summary

We have investigated and implemented straightforward means to verify the properties discussed in this chapter. The complexity of some of these checks is, as one would
Chapter 9. Integrity Constraints and Properties

expect, exponential in the worst case. However, it must be pointed out that property checks are performed on the specification of the web site and not on the web site itself.

Verifying constraints and properties against a specification instead of a particular instance of a Web site is an important feature of the proposed approach. It allows the design of consistent Web sites guaranteeing that the desired properties hold in any instance of the Web site created from the same intermediate representation. Our specifications show relationships among pieces of information and display units which stand for abstractions of possibly thousands of actual parts of the web site, once it is synthesised. By ensuring that the properties above hold at the specification level, then we need not worry about checking the actual contents of the web site, where the exponential complexity of the checks can become a problem. It also supports checking constraints and properties of Web sites which include dynamic pages, since the intermediate representation only describes page structures.

Hypergraphs and transaction logic are the two formalisms used to support the definition of constraints and properties. Hypergraphs have a direct correspondence to our intermediate representation, making it suitable for representing a Web site. Transaction logic provides us the means to represent and reason about paths, which are used to define the constraints presented here.

Although the focus is on the navigation structure, some properties on the visualisation specification are also discussed. Compatibility of data types, presentation styles and page templates can be verified in order to avoid errors when defining a visualisation for a Web site. This is another advantage of separating information content and navigation specification from visualisation details.

The properties presented here although useful to most application areas, are not exhaustive. There might be many other interesting properties, specially related to specific application domains. Nevertheless, the specification and proof of these properties can follow the same approach presented in this chapter.
Chapter 10

Web Site Maintenance using Agents

In this chapter we introduce a proposal for representing and extracting agents from the intermediate representation for Web site maintenance. This idea comes from the fact that the intermediate representation forms a knowledge base of a Web site application. All information (and meta-information) about the site is represented in a declarative format. Therefore it allows reasoning on a Web site specification and derivation of further knowledge about that specification [Cavalcanti and Vasconcelos, 2002].

Manual Web site maintenance is tedious and time consuming. Since this is the activity that accounts for most of the running costs of a Web site, automating part of maintenance is useful for reducing the overall costs. Most maintenance tasks are also repetitive and systematic, usually referring to updating information content or visualisation of the site. Given the repetitive nature of these tasks, a vocabulary of maintenance tasks can be specified, defining specific operation types and frequency.

Modelling task frequency requires some form of time representation. Furthermore, since the original specification (intermediate representation) is defined using first-order logic, integration of an agent architecture with the proposed approach is easier if the agents are also defined using logic. For that reason we have chosen to employ a formalism that supports logic-based specification of agents, making it compatible with our intermediate representation.

As a result of this approach for Web site maintenance, maintenance costs are reduced as many tasks are completely automated, requiring no human intervention. A multi-agent architecture is also defined as described in the following sections.
10.1 Specifying Maintenance Agents using a Temporal Logic

TeLA (Temporal Logic of Assertions), proposed by Vasconcelos [Vasconcelos, 2001], allows for the specification and execution of a large class of computations. TeLA can be used as a programming language, allowing the specification of initial values, terminating conditions, etc.

TeLA has its origins from MetateM a language for modelling reactive systems [Fisher, 1993], which supports the notion of concurrent and communicating objects. The language incorporate executable temporal logic for the implementation of objects. TeLA can be seen as a scale-down version of MetateM.

Our personal contribution is in applying an existing temporal logic and interpreter to a problem for which it had not originally been designed. TeLA was not designed for specifying Web-based agents and as a more general formalism, it includes more features than we need for specifying and executing our maintenance agents. Nevertheless, TeLA is suitable for our needs mainly because of its state-based specification and execution, which simplifies the definition of agents. TeLA is implemented in Prolog, which also makes it compatible with our synthesiser and facilitates the integration of the maintenance agents with our Web site specifications (intermediate representation).

One of the basic TeLA concepts is the state of computation. Each agent has an initial state to begin its execution. From that initial state other states are reachable depending on the proof of logic expressions. The sequence of states during the execution of an agent comprises its history, which can be used, for instance, for identifying problems and errors.

Since our maintenance agents are simple, involving a reduced number of distinct states, we have changed the implementation of the original state mechanism, introducing a circular list for keeping the states of an agent execution. A parameter defines the maximum number of states that will be kept in the list at a particular time. This solves the problem of keeping an infinite number of states, given that the execution of an agent is a continuous process.

Agents are defined as a pair $\Sigma = (\Gamma, \Pi)$, where $\Gamma$ is a set of temporal assertions and
10.1. Specifying Maintenance Agents using a Temporal Logic

\( \Pi \) is an associated logic program. \( \Gamma \) defines the temporal properties that \( \Pi \) must hold during its execution. Temporal assertions are the basic construct of TeLA, which have the form:

\[
\psi \Rightarrow \bigcirc \phi
\]

where \( \psi \) is a conjunction/disjunction of positive or negative literals, and \( \phi \) is a basic temporal formula.

The support for temporal logic operators and the notion of state of computation makes TeLA particularly suitable for specifying the Web maintenance agents described in this chapter. The simple structure of our maintenance agents can be described by state transitions diagrams, which is one way to describe the behaviour of an agent. For example, Figure 10.1 illustrates the generic behaviour of maintenance agents.

![Figure 10.1: Generic behaviour of maintenance agents](image)

Given the simple behaviour of maintenance agents, only a subset of TeLA features is necessary for the specification and execution of our maintenance agents, which facilitates their implementation. As an initial example, the specification below defines an agent for performing a periodic maintenance task.

\[
\Sigma = \left\{ \begin{array}{c}
\text{start} & \Rightarrow & \bigcirc \text{check} \\
\text{check} \land \text{receive(I)} \land \neg I = [] & \Rightarrow & \bigcirc \text{update(I)} \\
\text{check} \land \text{receive([])} & \Rightarrow & \bigcirc \text{sleep} \\
\text{update(I)} \land \text{perform_update(I)} & \Rightarrow & \bigcirc \text{sleep} \\
\text{sleep} & \Rightarrow & \bigcirc \text{check} \\
\text{receive(I)} & \leftarrow & \ldots \\
\text{perform_update(I)} & \leftarrow & \ldots
\end{array} \right\}
\]
In this example "[ ]" denotes absence of information for updating. The temporal assertions on the right-hand side of the operator \( \Rightarrow \) define the states of the computation, that is check, \( \text{update}(I) \) and sleep. State sleep corresponds to the time the agent remains inactive. This time corresponds to the frequency of the maintenance task, which is defined prior to the agent deployment as part of the Web site specification.

The example above describes an agent that updates a piece of information \( I \) at intervals of time. This specification actually corresponds to a generic skeleton of a maintenance agent. The different maintenance tasks replaces the definition of \( \text{perform}_\text{update}(I) \), according to the specification of the maintenance agent defined in the intermediate representation. The types of maintenance task and frequency are detailed in section 10.2.

The \( \bigcirc \) operator is the next time temporal operator. \( \bigcirc p \) denotes that \( p \) is true in the next state of the computation. Hence a formula \( q \Rightarrow \bigcirc p \) means that \( p \) is true in the next state of execution if the assertion \( q \) is true at the current state.

Note that the \( \bigcirc \) operator is the only temporal operator used for specifying a maintenance agent, although other operators could also be employed. The reason is that we have no intention to prove temporal properties of an agent specification at this point. The use of only one operator also keeps the agent specification simple and facilitates automatic generation of this sort of agents.

A natural extension is to allow the use of all TeLA temporal (modal) operators in the specification of an agent. This includes operators such as \( \diamond p \) (eventually \( p \)) and \( \square p \) (always \( p \)). A richer temporal specification would support the verification of properties of an agent specification, including past and future actions.

### 10.2 Using Agents for Web Site Maintenance

Maintenance agents are very specialised. For that reason, few parameters are necessary for creating the maintenance agents. In order to perform its task, each agent must know the labels of the pieces of information for which it is responsible. In addition, the type of maintenance task to be performed and its frequency should also be specified. A simple vocabulary of maintenance actions is defined as:
10.2. Using Agents for Web Site Maintenance

- add: the agent will only add information to the Web site.
- update: denotes that the agent will replace the instance of one or more pieces of information.
- delete: the agent will only remove information from the Web site.

Frequency is defined as one of daily, weekly, monthly, occasionally or it can also be a specific date. This vocabulary can be extended, as long as suitable definition for its “meaning” is also included. This argument defines the time the agent will remain inactive between tasks, represented by the clause sleep, as explained in Section 10.1. Its implementation corresponds to a system call suspending the execution of the agent for the specified time. We have decided not to include the time parameter in the TeLA representation of an agent to avoid the problem of introducing real-time operators in a linear interval-based temporal logic. This problem is well-known and is discussed in [Manna and Pnueli, 1996]. Therefore, our decision is a pragmatic way to introduce a procedural behaviour and representing real time within this temporal framework.

Agents are specified by a list of labels of pieces of information, a type of action (from the list of actions above) and the frequency (from the list of frequency above) of this action. These parameters are defined as part of the predicate

\[
\text{agent}(\text{LabelList}, \text{ActionType}, \text{Frequency}).
\]

Section 10.3 presents an example of maintenance agents specification.

The predictability of maintenance tasks allows us to define a (potentially extensible) vocabulary of actions and frequency. For each action we can associate pre-defined portions of code that will be instantiated to specific tasks, thus minimising the effort of programming agents. Agents will thus be synthesised (and started up) “on-the-fly”, that is, as the specification is used to infer/synthesise the site. The action type and frequency are values to be used in the customisation of the pre-defined code.

In principle, any programming language can be used to represent the code. We have employed TeLA as the underlying formalism. The use of logic for this purpose allows all components of our architecture to coexist in one single notation that can be easily integrated to the intermediate representation used in our Web site synthesis approach.
Other specific tasks can also be given to agents such as periodical checks of broken links and property verification if there is any change in the original transition specification. These sorts of agent do not require any parameter or additional specification, since the complete code is standard. Nevertheless they follow the same template as illustrated in Section 10.1. An example of such agent for verification of broken links is defined below:

\[
\begin{align*}
\text{start} & \Rightarrow \text{check_links}(L) \\
\text{check_links}(L) \land L = [] & \Rightarrow \text{sleep} \\
\text{check_links}(L) \land L \neq [] & \Rightarrow \text{take_action}(L) \\
\text{take_action}(L) \land \text{send_msg}(\text{WebMaster}, L) & \Rightarrow \text{sleep} \\
\text{sleep} \land \text{wait}(T) & \Rightarrow \text{check_link}(-)
\end{align*}
\]

The agent starts by checking the links. If there are no broken links, denoted by the empty list [], the agent moves to state sleep. Otherwise, the agent sends a message to the Webmaster (via e-mail for example) and then moves to state sleep. State sleep simply involves performing a wait procedure, which suspends the execution of the agent for a period of time T. This is one of the parameters included in the agent definition. After the execution of \text{wait}(T), the agent returns to state \text{check_links}(L).

Definition of predicate \text{check_links}/1 involves creating a list with all external links of a Web site and calling predicate \text{broken}/2 (defined in Section 9.2.3). \(L\) is a list of broken links which can be passed over to the Webmaster for correcting the links.

Some further advantages of using our temporal logic are the possibility of proving properties of the pre-defined portions of code and the relatively small size of the actual code (say, in comparison with C++ or Java). This architecture proves to be very flexible as generation of Web pages in different target languages can be achieved by defining one publisher agent for publishing information in each target language. The maintenance agents, however, remain the same.
10.2.1 A Multi-Agent System Architecture for Web Site Maintenance

With maintenance agents, we have added an extra layer of functionality to our specifications. Together with pieces of information, units of display and visualisation styles, we offer the possibility of interspersing the information and knowledge representation with abstract descriptions of agents. At this point we investigated two options for designing our multi-agent system architecture:

1. each agent is responsible for both parts of the intermediate representation and the visualisation specification concerning the pieces of information and display units involved. This alternative gives each agent the sole responsibility for acquiring information for updating, performing maintenance and publishing the resulting Web page(s) in a target language.

2. An additional special purpose agent, named publisher is introduced. It is responsible for mapping intermediate representation specifications into Web pages, according to the visualisation specification. In this case, the maintenance agents are only responsible for updating pieces of information and creating a new piece of information using the same intermediate representation language.

The advantage of approach (1) is that agents are more specialised and there is no communication between agents, since all the knowledge necessary for a maintenance task is already part of the agent specification. However, this makes each agent specification more complex and larger than those of the second alternative. There is also the need for replicating parts of the visualisation specification for all agents that maintain pieces of information with the same data type or page template. In addition, if the visualisation changes all agents have to be restarted in order to update their knowledge on visualisation specification.

The second option was chosen because it follows the principle of separating visualisation from content specification and for making the agents simpler, although the publisher agent may include a large specification depending on the size of the visualisation specification. One clear advantage of this approach over the previous one is,
for instance, if the visualisation is changed only the publisher agent has to be restarted. Figure 10.2 illustrates the maintenance process according to this option.

Upon receipt of a message with the expected parameters, a maintenance agent performs its task by updating a subset of the current intermediate representation $SI$ transforming it into an updated version $SI'$, which includes the actual values of pieces of information. $SI'$ is sent via a message passing mechanism (represented by dotted arrows in our diagram above) to the publisher agent to be transformed onto Web pages $W$ written in a target language. In our diagram solid arrows represent the agent's source of information and the dashed arrows correspond to transformations $I \rightarrow SI'$ and $SI' + V \rightarrow W$. Transformation $SI \rightarrow SI'$ is dependent on the type of maintenance task. It involves data manipulation only, the underlying page structure being the same as described in Chapter 8. Transformation $SI' + V \rightarrow W$ follows the same rules for application of a visualisation style. This transformation was also described in Chapter 8.

Our agents are run as independent processes which exchange messages by means of the Linda process communication mechanism available in SICStus Prolog [Intelligent Systems Laboratory, 2000]. Linda [Carriero and Gelernter, 1989] is a mechanism for asynchronous process communication. It consists of an area, the tuple space, which is shared by different processes. Each agent can store data in the tuple space and perform operations such as write, read and delete. The Linda concept has proven very successful due to its simplicity and flexibility. SICStus Prolog offers a Linda library including built-in predicates which allow the implementation of programs that run as independent processes and which are able to exchange data via the tuple space.
10.2. Using Agents for Web Site Maintenance

Given a specification of a site, we can extract its agents and start them up as a set of SICStus Prolog Linda client-server processes. We show in Figure 10.3 a diagrammatic representation of how this proposal was implemented [Cavalcanti and Vasconcelos, 2002].

![Diagram of agent processes](image)

**Figure 10.3: LPadL**

A list of agents \((\Pi_1, \tilde{G}_1), \ldots, (\Pi_n, \tilde{G}_n)\), where \(\Pi_i\) is its Prolog program (i.e. temporal formula plus proof procedure) and \(\tilde{G}_i\) is the initial query, is input to LPadL (Launching Pad for Linda) a tool that implements the described architecture. Each agent is initialised as an independent Prolog (client) process \(Ag_i\) – this activity is represented in the diagram by the dotted arrows. A preliminary sequence of steps is required to “connect” each process to a Linda server process (central Sv box in diagram above). This server process is responsible for managing the tuple space and is automatically generated and started up as a background job. The management of the tuple space is done by Prolog itself.

Each pair \((\Pi_i, \tilde{G}_i)\) will give rise to a process \(Ag_i\). However, some initial steps are required before a process can actually interact with the tuple space. Designers need not worry about this: such a sequence of steps is done automatically by LPadL at the onset of the process life. At the end of the execution of each process a procedure to close the connection has to be followed. Again, this has been automated in LPadL.
10.3 Example of Maintenance Agents

In this section we describe an example of the use of maintenance agents, illustrating the specification of agents to maintain an accident report Web site, which has been used as an example in previous chapters. Specifying maintenance agents involves the following steps:

1. identify the relevant pieces of information and assign them to maintenance agents.
2. define the frequency of the maintenance task (daily, weekly, etc).
3. define the maintenance task (add, update, delete).
4. extract the piece of information specification and the display units and transitions in which it participates.
5. create the agent accordingly.

Based on the steps above we have defined two agents: agent $A_{g1}$ is responsible for updating existing information about accident reports and agent $A_{g2}$ adds information about new aircrafts. Their specification are:

1. agent($A_{g1}$, [report], update, occasionally)
2. agent($A_{g2}$, [aircraft], add, monthly)

The extract of the intermediate representation used for this example is described below. It includes definitions of the pieces of information involved report and aircraft, the display units where they appear and the transitions these pieces of information are involved.

1. Pieces of information

   info(aircraft, tuple, A) ← get_aircraft_info(A)

   info(report, tuple, R) ← get_report_info(R)

2. Display Units
10.3. Example of Maintenance Agents

\begin{align*}
\text{display(aircraft\_page, [aircraft])} \\
\text{display(report\_page, [report])}
\end{align*}

3. Transitions

\begin{align*}
\text{aircraft} & \Rightarrow \text{report\_page} \\
\text{report} & \Rightarrow \text{aircraft\_page}
\end{align*}

Frequencies of actions define how often an agent is required to perform a maintenance task. This also makes possible for an agent to be proactive. Instead of waiting for messages, every time the agent resumes its execution it can send messages to the Webmaster asking for the expected parameters to complete its task.

The parameters expected by each agent are in most cases, instantiated pieces of information. For example, the second agent for the maintenance of aircraft information expects a message with a tuple corresponding to the definition of an aircraft. This have to include the attributes model, registration, number of engine, engine type and year of construction. Given that information the agent automatically creates a new instance of the piece of information aircraft such as:

\text{info(aircraft, tuple, <'Embraer 170', 'PP-XJA', 2, 'General Electric CF34-8E', 2002>)}

This specification is sent to the publisher agent who transforms it into the target language, according to the style definition for the data type tuple and creates the Web page.

Formally, the maintenance agent for the piece of information aircraft is described in TeLA as follows:

\begin{align*}
\text{start} & \Rightarrow \text{check\_info(Info)} \\
\text{check\_info(Info)} \land \text{Info = "Flag"} & \Rightarrow \text{dead} \\
\text{check\_info(Info)} \land \text{Info = []} & \Rightarrow \text{sleep} \\
\text{check\_info(Info)} \land \text{Info \neq []} & \Rightarrow \text{add(Info)} \\
\text{add(Info)} \land \text{assemble\_info(Info)} \land \text{send([add, Info])} & \Rightarrow \text{sleep} \\
\text{sleep} & \Rightarrow \text{check\_info(Info)}
\end{align*}
We have enclosed the left and hand sides of the formulae in boxes to improve visualisation. The formulae show an agent that keeps an account of its current state, one of the set \{start, check_info(Info), add(Info), sleep, dead\}, and the conditions that ought to hold in order for the states to change.

In this example, the agent verifies if a piece of information, denoted by state check_info(Info). Once the message is received there are three possible state transitions:

1. a flag is received indicating termination of execution and the agent moves to state dead;
2. there is no information available (Info = []) and the agent moves to state sleep;
3. an information is received for adding to the Web site and the agent moves to state add(Info).

In state add(Info) the agent assemble the information (performed by predicate assemble_info/1) as described earlier and sent it to the publisher agent. In state sleep the agent suspends its execution for the time defined in its specification.

Note that the specification of the agent for updating reports is identical to the one above. The difference is on the specification of the maintenance task, in this case, replacing state add(Info) with update(Info) and the message to the publisher agent has format send([update,Info]), instead of send([add,Info])

Similarly, the publisher agent is described below:

\[
\begin{align*}
\text{start} & \Rightarrow \text{check_info(Info)} \\
\text{check_info(Info) \land Info = "Flag"} & \Rightarrow \text{dead} \\
\text{check_info(Info) \land Info = []} & \Rightarrow \text{sleep} \\
\text{check_info(Info) \land Info \neq []} & \Rightarrow \text{visual(Info)} \\
\text{visual(Info) \land apply_visual_style(Info)} & \Rightarrow \text{sleep} \\
\text{sleep} & \Rightarrow \text{check_info(Info)}
\end{align*}
\]

One way of maintaining a Web site using our maintenance agents architecture is illustrated by Figure 10.4.

The steps involved in the completion of a maintenance task are:
1. user interacts with agents, for instance via e-mail, sending the information for publication in the expected format.

2. agents ($Ag_1$ and $Ag_2$) verify the existence of messages periodically, for example using predicate $check\_info/1$ as described in the maintenance agent code. The frequency in which the agent verify messages is part of the agent specification. Once a message is received, the agents assembles the corresponding pieces of information in an appropriate internal representation and send them to the publisher agent ($Ag_p$) for publishing.

3. The publisher agent receives the pieces of information and publishes them according to their visualisation styles.

10.4 Summary

In this chapter we have described a method for automation of Web site maintenance. The approach exploits TeLA, an executable temporal logic, as the underlying formalism to specify and execute maintenance agents. Agents can be constructed as part of the
Web site synthesis or can be added in later stages. Since TeLA is implemented in Prolog, it is compatible with our intermediate representation and is easily integrated with our Web site synthesis approach.

A maintenance agent is responsible for a set of piece of information and perform pre-defined tasks whenever requested. Alternatives to interact with agents is a simple messaging mechanism or e-mail. Parameters define the piece of information, type of task and frequency of the task that an agent performs. The publisher agent is responsible for publishing all information altered by maintenance agents, applying the appropriate visualisation styles. This approach keeps a clean separation between content and visualisation management.
Part IV

Discussion and Prospects
Chapter 11

Evaluation

One of the main benefits we claim of our approach to Web site synthesis provides is saving on maintenance costs. We have been using a domain-specific synthesiser for our research group Web site at Edinburgh (http://www.dai.ed.ac.uk/groups/ssp/index.html) for about 4 years [Robertson and Agustí, 1999]. Maintenance is done simply by updating its declarative specification. Although this has proved cost-effective it is limited to the generation of a particular visualisation and navigation structure. Although there is a declarative specification of the site, any change to the visualisation style requires modification in the synthesiser program.

Our Web site synthesiser gives the possibility of creating different Web sites. This is possible because more than one intermediate representation can be generated from one high-level specification. Combining that with the generation of different visualisations for the same intermediate representation, the number of possible different Web sites is even higher. This is the main feature that saves maintenance time, as no programming or time-consuming coding is necessary to produce alternative visualisations.

In traditional approaches the initial cost to produce a Web site is lower than using synthesisers. This happens because traditional methods such as authoring systems, results in rapid deployment of a Web site since the construction involves direct coding. However, the lack of an underlying declarative specification makes maintenance cost much higher because every alteration to the visualisation of the site requires altering Web page code and may also involve programming.

The initial cost of constructing a Web site using synthesisers is higher given the
need for writing specifications using formalisms such as an entity-relationship diagram and the additional definitions necessary for the synthesis of a Web site. However, altering a declarative specification (if sufficiently simple and domain-specific) is much simpler and less expensive, requiring no technical expertise. This can be done by means of simple user interfaces to gather the appropriate specification alterations. This also offers the possibility of non-technical people being able to perform maintenance tasks.

We estimate that the break-even point, when the cost of using synthesisers becomes lower than the cost of using traditional methods is when the Web site construction is complete. From that point onwards traditional methods tends to require a lot of attention of technical staff for the maintenance of the Web site whereas the use of synthesisers simplify this task becoming clerical.

Our approach makes maintenance cost even lower than using the domain-specific synthesiser, given the extra capabilities for automating maintenance tasks and the generation of alternative visualisations.

In order to demonstrate how that is achieved, we evaluate our method in two different ways. Firstly an empirical analysis is presented on the advantages of having declarative specifications, in particular the intermediate representation, and the number of specifications necessary for generating different Web site presentations. Secondly, the main features that should be supported by Web site synthesisers are discussed, showing how our approach addresses them.

Other traditional evaluation metrics, such as hits per page and load time are not considered because the proposed approach focus on the specification of Web sites, not on its use.

11.1 Evaluation: Number of Specifications V Number of Visualisations

Independence between application and visualisation specifications gives us the ability to produce different Web sites by combining the same application specification with different visualisation descriptions. Formally, given a high-level representation $\Sigma$, a
number of different intermediate representations \( I \) can be generated. By combining \( I \) and a visualisation specification \( V \), a Web site \( W \) can be automatically synthesised. The ideal situation, depicted in Figure 11.1a, is that every intermediate representation is compatible with any visualisation specification. This means that a particular visualisation specification can be reused for producing different Web sites. For instance, from 3 intermediate representations and 3 visualisation specifications it is possible to produce 9 different Web sites for the same application.

However, this is not always the case as some visualisation specifications may not be compatible with an intermediate representation. It is expected that our system produces a smaller number of practical and useful visualisations compared to the ideal case. This is shown in Figure 11.1b. Nevertheless the number of visualisations generated should still be greater than in the worst case, depicted in Figure 11.1c, where there is only one visualisation produced from each specification. There is an empirical issue of how close we get to the ideal of Figure 11.1a in practice.

The number of possible different visualisations has a direct relation to the compatibility between the data types and the visualisation styles. That will determine whether a visualisation specification can be applied to a given intermediate representation. Recall the mapping procedures presented in Chapter 5. There we have defined translations from an entity-relationship diagram (the high-level description \( \Sigma \)) into three different intermediate representations:

1. **entity-based**, where a display unit containing all instances of each entity is created. The data type used to represent pieces of information resulting from this mapping is table, in which the instances of an entity are represented in a tabular structure.

2. **instance-based**, where each instance of an entity is presented in separate display units. For that reason, the data type used in the definition of the pieces of information is tuple.

3. **query-based**, where pieces of information are created as a direct result of queries. This mapping procedure produces pieces of information of various data types, such as text, string, list as well as table and tuple. The data type of pieces of
Chapter 11. Evaluation

Figure 11.1: Number of specifications $V$ number of visualisations
information depends on the result of the associated query. Therefore, query-based intermediate representations allow for a larger number of visualisation derivations compared to the previous two approaches.

Now we have to evaluate the visualisation styles and how they relate to the data types mentioned above. In this sort of analysis we have to define the target language as the styles result in pieces of code written in that language. In this evaluation we have chosen HTML as the target language because it is the most common mark-up language for the Web. It is also important to point out that we do not take into account the variations of visualisation that are possible to achieve by using style sheets. One way to verify if a visualisation specification is compatible with an intermediate representation is to answer the question "Is it possible to apply style S to piece of information of type T?"

The common visualisation elements in HTML [Committee, 2002] are string, text, list and table. Strings and text do not require any special HTML tag, although tag <P> can be used for organising a text in paragraphs. Lists are represented by tags such as <OL> for ordered lists and <UL> for unordered lists and tables by tag <TABLE>. A compatibility table is presented in Table 11.1, showing the possible visualisation renditions for each data type. Compatibility is marked with a √. Non-compatibility is represented by the symbol × and a ? indicates that the style can be applied to that data type although the visualisation might not be very useful, for instance when text is presented in a table style.

From the compatibility table we can see that data type table is only compatible with one style. That means intermediate representations derived by the entity-based approach have only one possible visualisation. Data type tuple, which is mostly associated to the instance-based approach, can be presented using three different styles. Query-based intermediate representations can range from one presentation style, if it has only pieces of information of type table, to more than three, if it has at least one piece of information of type tuple and another one of type list. It is safe to assume that at least one piece of information of type tuple should be present, because this is the usual data type associated with individual instances. Figure 11.2 illustrates this derivation process, having seven different visualisations in total.
Chapter 11. Evaluation

<table>
<thead>
<tr>
<th>Style / Data Type</th>
<th>string style</th>
<th>text style</th>
<th>list style</th>
<th>table style</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>string</td>
<td>✓</td>
<td>?</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>text</td>
<td>?</td>
<td>✓</td>
<td>×</td>
<td>?</td>
</tr>
<tr>
<td>list</td>
<td>×</td>
<td>?</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>tuple</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>table</td>
<td>×</td>
<td>?</td>
<td>×</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 11.1: Compatibility between data types and visualisation styles

![ER Diagram](image)

Figure 11.2: Number of different visualisations generated from intermediate representations

If more intermediate representations and visualisation specifications are defined, the number of different Web sites that can be produced will be greater. For example, from 4 intermediate representations and 5 visualisation specifications (9 specifications in total), 20 different Web sites can be constructed in total, assuming total compatibility between intermediate representations and visualisation specifications. Traditional Web site construction methods require the same number of specifications as the number of Web sites produced. The reason is the lack of separation between application and visualisation specifications.

The ability to combine a Web site application specification to different visualisation
11.1. Evaluation: Number of Specifications

Number of Visualisations

163

description dramatically reduces maintenance effort. This feature can also be used for personalisation of Web sites for classes of users or individual users. This is clear even in the small example presented in Chapter 7.

For example, consider the piece of information aircraft, with data type tuple. It is defined as info(aircraft, tuple, A) where A is a tuple including attributes model, registration, number of engines, engine type, construction year and photo. Assuming that this piece of information is the result of an entity-based mapping procedure, it can be presented with three different styles, illustrated by Figures 11.3, 11.4 and 11.5.

![Figure 11.3: Aircraft page: text style](image)

Although this is a simple example, it demonstrates the simplicity to change a piece of information style. Each of the style specifications which generate the Web pages illustrated by Figures 11.3, 11.4 and 11.5 are defined using predicate style/2 as defined in Chapter 7. This predicate associates a piece of information label with a style label, which corresponds to a parameterisable component defined in the target language. For instance, the styles specifications for the examples presented are:

```
style(aircraft, text_style)
```
Chapter 11. Evaluation

Figure 11.4: Aircraft page: list style

Figure 11.5: Aircraft page: table style
11.1. Evaluation: Number of Specifications V Number of Visualisations

style(aircraft, list.style)

style(aircraft, table.style)

Although it is possible to combine a set of transition rules with different visualisation definitions, there are visualisations more appropriate to each kind of application. However this issue is subjective making it very difficult to decide automatically which visualisation should be applied. In the example presented in Section 5.2.1 the table-based visualisation is more appropriate to entity-based rules than text-based presentation. For instance-based rules both visualisations are appropriate.

11.1.1 Specifying Large Commercial Web Sites

The examples presented so far are small Web sites with no more than 200 pages. Although the number of pages and links usually is a measure of complexity, the basic principle of our approach is to work on the specification of the site instead of the Web page instances.

In our intermediate representation, the display units correspond to sets of pages with same structure. As a result, the size of a Web site specification using our method has no direct relation to the number of Web pages. Instead, it is directly associated with the number of different sets of pieces of information grouped in display units. We illustrate this idea by presenting a partial specification of the Amazon.com Web site using our intermediate representation.

Amazon.com (http://www.amazon.com) is a huge online retail store. It is organised in different categories of items, each one organised in sub-categories. Examples of these categories are:

- Books: with over 35 categories of books, such as Children’s Books, History and Literature Fiction;
- Electronics with over 30 categories;
- over 20 categories of music;
- over 25 categories of DVDs.
For simplicity, in the example below only the book and payment sections are detailed. The reason for presenting a partial specification is that all categories have similar structure: items organised in sub-categories. For instance, books are organised in categories which are also classified into sub-categories which are also classified into specific topics. From a specific topic one can obtain a list of books and from that list an individual book page is presented.

Operations allow a user to have direct access to selected information. For example, operation search allows a user to get a list of books, given key words (eg. book’s title or author’s name). A shopping cart keeps the items selected for purchase and the user can add or remove items from the cart using operations add_to_cart and delete_from_cart. To finalise a purchase the user goes to checkout where payment details are collected and finally the user can confirm the whole transaction or return to the shopping cart.

1. **Pieces of Information**


   info(book_subjects, list, [SL])

   info(specific_subject, list, [SS])

   info(specific_topic, list [ST])

   info(book_list, list, [BL])

   info(cart, tuple, [item, Price, Qty])

   info(payment_details, tuple, [CardType, Number, Exp_Date, Name_on_Card, Shipping_Address])

   info(purchase_details, tuple, [Items, PaymentMethod, TotalPrice])

   info(search_arg, form, [SearchArg])

   info(promotion, tuple, [Title, Cover, Price])

   info(top_sellers, table, [Title, Cover, Price])

   info(recommended_books, table, [Title, Cover, Price])
2. **Display Units**

   display(welcome, [nav1, nav2, nav3, search, promotion, new_future.releases])
   display(books, [nav1, nav2, nav4, book.subjects, new_future.releases, recommended.books, top.sellers])
   display(book.category, [nav1, nav2, nav4, specific.subject, recommended.books])
   display(book.sub.category, [nav1, nav2, nav4, specific.topic, recommended.books])
   display(book.list.page, [nav1, nav2, nav4, book.list])
   display(book.details, [nav1, nav2, nav4, book, add.to.cart])
   display(search.result, [nav1, nav2, nav4, book.list])
   display(shopping.cart, [nav1, nav2, cart, proceed.to.checkout, delete.from.cart])
   display(check.out, [cart, get.payment.details])
   display(confirm.purchase, [purchase.details, confirm])
   display(confirmation.page, [nav1, nav2, purchase.details])

   and similar definitions for electronics, dvd, toys_and_games, home_and_garden, computer_and_video_games and cars.

3. **Transitions**

   welcome ⇒ books
   book.subjects ⇒ book.category
   specific.subject ⇒ book.sub.category
   specific.topic ⇒ book.list.page
   book.list ⇒ book.details
   search ⇒ search.result
   add.to.cart ⇒ shopping.cart
   shopping.cart ⇒ check.out
   get.payment.details ⇒ confirm.purchase
   confirm ⇒ confirmation.page
confirmation_page ⇒ shopping_cart

delete_from_cart ⇒ shopping_cart

shopping_cart ⇒ books

Note that transitions made via the navigators are omitted for space reasons.

4. Operations

\[ \text{op(search, query, [search_arg], [book_list])} \]
\[ \text{op(add_to_cart, add, [book], [cart])} \]
\[ \text{op(delete_from_cart, remove, [book], [cart])} \]
\[ \text{op(get_payment_details, add, [payment_details], [purchase_details])} \]
\[ \text{op(confirm_purchase, add, [], [purchase_details])} \]

5. Navigators

navigator(nav1, [cart, wish_list, your_account, help])

navigator(nav2, [welcome, your_store, books, electronics, dvd, toys_and_games, home_and_garden, computer_and_video_games, cars])

navigator(nav3, [international, top_sellers, target, todays_deals, sell_your_stuff])

navigator(nav4, [search, your_store, bestsellers, magazines, corporate_accounts, e-books, new_used_textbooks, used_books])

This partial specification is very close to a specification which our synthesiser can use to generate a Web site automatically. Definitions of pieces of information, display units and transitions should not differ much from the specifications presented here. Operations were accommodated in the types (query, add, etc) discussed in Section 6.3, although their implementation might be more complex than expected. This issue requires further investigation. Extending the operation library with new types of operation is one way to solve this problem.
11.2 Support for Web Site Synthesis Features

In this section, we discuss how each of the features of Web site synthesis methods are supported by our method. A discussion on the support of these features by other methods was presented in Chapter 3.

1. Paradigm

Our method can be defined as a logic-based approach to Web site construction. This is a novel approach as all the known methods are either model-driven, object-oriented, ontology-based or query-based. A few methods apply some use of logic to verify properties and to specify rules (triggers) for automating maintenance tasks.

Our approach also proposes the use of conceptual data model, such as the entity-relationship data model, as we have explained in Chapter 5. In our approach conceptual data models are only used as the starting point of the synthesis process and all the relevant concepts are translated into logic. The reason to do this is to hide details of logic expressions from Web designers, as most people may not feel comfortable with the direct manipulation of logic specifications.

However, no approach proposes the use of logic as the main formalism upon which the synthesis process is defined. The use of computational logic appears to be useful for Web site synthesis, mainly because of its support for declarative specifications and reasoning capabilities, which allows for property and constraint verification.

2. Support for Generation of Dynamic Pages

The proposed approach supports the creation of Web sites with dynamic pages. This feature was presented in Chapter 6. The specification of operations is an integral part of the intermediate representation, modelled as another type of transition.

An operation “links” one piece of information to a page. This piece of information is usually represented as a form which receives user input. The resulting page is actually a result of the computation performed by the operation. We have
Chapter 11. Evaluation

implemented a limited set of operation components in order to demonstrate this concept. Nevertheless, this library of operation components can be extended.

The current implementation of operations still requires further investigation, which basically involves refining the operation specification notation, extending the component library and use of technologies for the implementation, other than CGI. These issues are discussed in Chapter 12.

3. Support for Heterogeneous Input Sources

The definition of pieces of information conceal the details of file or database access. The definition of a piece of information, for instance

\[ \text{info(aircraft, tuple, A) \leftarrow get_aircraft.data(A)} \]

allows for the definition of access to different input sources. In the expression above, the predicate \( \text{get_aircraft.data(A)} \) can include details for accessing simple files or SQL queries to a relational database. Therefore, each piece of information specification might relate to a different data source.

4. Generation of Different Visualisations

Our method is particularly strong in this area, offering a very flexible way of combining different visualisation styles to one Web site specification. This feature was discussed in detail in the previous section.

There are other ways for producing different visualisations using our method: changing the display units templates and changing the Web pages style sheets. Display unit templates define the general layout of the corresponding Web pages and style sheets are used to define colours, text fonts, font sizes among other presentation definitions. These issues were discussed in Chapter 7.

Additionally, different target languages can also be used for producing alternative visualisations, such as XML and WML for WAP-enabled devices. Again, the same issues of different styles and page template apply to other target languages.

5. Personalisation of Web Pages
Although we have not exploited this feature in detail in our method, personalisation can be achieved by two means:

(a) Personalisation of information: this involves the definition of customised pieces of information each one, filtering the desired information to be presented to the user. This concept can also be used to support classes of users with different levels of authorisation access to information. The implementation of this feature in our method is similar to the query-based approach to derive intermediate representations.

(b) Personalisation of presentation: this involves producing specific visualisations for individual users or classes of users. One way to support customisation of presentation styles is to allow styles to be associated directly to pieces of information labels, instead of data types. For example, styles for the piece of information info(aircraft, tuple, A) can be defined as:

\[
\text{style(aircraft, style}_1) \\
\vdots \\
\text{style(aircraft, style}_n)
\]

where each style\(_i\) is defined for different users.

6. **Automation of Maintenance Tasks**

This feature was detailed in Chapter 10 where we presented logic-based agents to automate recurrent maintenance tasks. Tasks were defined in terms of pieces of information, which were assigned to agents responsible for their maintenance. Maintenance tasks are defined as one of add, update or delete each one having a periodicity defined in terms of time. This set of maintenance task is also extensible. This is a novel approach as few other work support automation of maintenance tasks. To our knowledge, only the OntoWebber system [Jin et al., 2001] offers facilities to automate parts of maintenance.

7. **Constraint and Property Verification**

This is also another feature not well exploited by other methods to Web site synthesis. Our approach supports the specification and verification of constraints
on the order of presentation of pieces of information. Properties are mainly used for verifying correctness of specifications and reachability of information.

This is also one of the strong points of our approach, since Web site specification, constraints and properties are all defined using logic, which promotes an easy integration of all these parts.
Chapter 12

Conclusion

In this thesis we have presented a logic-based approach for the automated synthesis and maintenance of Web sites. Our approach consists of a representation for the information and knowledge comprising the content of the site, plus means to use this representation for synthesising an actual site, that is, synthesis by reasoning.

Content is represented initially by a high-level description, for which we have used an entity-relationship model. Intermediate representations are derived from the high-level representation which forms a knowledge base describing content and navigation structure of a Web site. Content is described by pieces of information and their organisation in display units and transitions describe the navigation structure. A separate component of our proposal contemplates visual aspects of the site, including styles for the pieces of information, page templates and style sheets for the display units. Web pages are produced by combining an intermediate representation with a visualisation specification, translating the result into a mark-up language such as HTML.

Dynamic pages, i.e. pages that are generated automatically as a result of user interaction, are also supported via a parameterisable component library with common database operations such as search, add, update and delete. This library is extensible allowing for the addition of many task-specific operations. We have implemented these operations using a Prolog-CGI mechanism.

Separation between information content, navigation structure and visualisation allows maintenance tasks to be performed individually on each of those components without any impact on the others. There is no need to change the synthesis process.
Additionally, we offer means to automate some of the tasks involved in the maintenance of sites: the specification of the site can be "annotated" with executable agents that can perform such tasks.

Our approach is particularly suitable for generating data-intensive Web sites, since we have emphasised the use of an underlying database management system for storing the Web site content. This is also facilitated by the easy integration of first-order logic and relational representations. Most Prolog systems currently available support 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.

Given the flexibility and versatility of this logic-based approach, we can envisage its employment especially 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 for specification and synthesis.

## 12.1 What We Have Achieved

We have addressed different aspects of Web sites using logics. Computational logic is used as a common formalism for the specifications, for specifying the synthesis process and even for the implementation of the maintenance agents. This eliminates the semantic gap between specification and implementation that usually occurs in approaches that use different formalisms to represent separate parts of specification. Another important advantage is that our specifications can be used in many different forms with distinct purposes, such as property and constraint checking and generating alternative organisations of content and different visualisations.

Logic is not just a vehicle for synthesising Web sites. Although we have employed a variety of concepts from formalisms such as entity-relationship model, hypergraphs, transaction logic and TeLA, all of them can be seamlessly integrated to our intermediate representation via appropriate translations. As a result, the synthesiser acts on a single
12.1. What We Have Achieved

underlying logic representation. Logic also adds reasoning power to the system and gives us natural support for declarative specifications. Integrity constraint and property checking are easily integrated with all specifications. Maintenance is facilitated by the declarative descriptions and logic-based agents can also be used to automate some repetitive maintenance tasks.

Our approach for Web site synthesis is particularly strong on the following points:

- Logic-based declarative specification of Web sites

  Our approach is based on logic descriptions of Web site applications. This makes easier to separate details of the synthesis process from the conceptual level. This encourages the designer to concentrate on the application description rather than on the mechanics for producing a Web site.

  Only the specification of the site should be changed in order to change the Web site, either its structure, visualisation or both. There is no need to change the synthesis program.

  Declarative specifications of Web site applications offer possibilities to apply many automated synthesis techniques to reduce the effort of design and maintenance.

  Complexity is reduced in the specification phase, because fewer specifications are needed to produce different Web sites. For example, one visualisation style can be applied to different intermediate representations producing different Web sites.

- Generation of different visualisations

  From one specification, different visualisations can be generated. This can be useful for many purposes such as providing specific visualisation for different classes of users and personalisation.

  Fine tuning of visualisation of pieces of information and Web pages is done via CSS style sheets. This allows a simpler description of visualisation styles and supports changes in the style sheet without necessarily changing the style.
Additional visualisations are possible by changing the target language, for example from HTML/CSS to XML/XSL. We have carried out experiments with these languages and with WML, which is used in WAP-enabled mobile devices. This feature also makes the approach very flexible and adaptable to new technologies.

Changing the visualisation of an existing Web site is also easy and can be done in different ways, as follows:

- changing the visualisation style of a data type, which in turn changes the visualisation of all pieces of information of that type;
- changing the data type of a piece of information, which changes the visualisation for that particular piece of information;
- changes in the general visual layout of the site by altering the page templates, without necessarily changing the styles of individual pieces of information;
- changing the style sheet which changes presentational details such as colours and text fonts.

- Property and constraint checking

As we have shown, desirable properties in a site can be automatically checked, such as references to non-existing units, multiple references from one same piece of information, pieces of information or display units that are not referred to, and so on.

The representation of a Web site as a hypergraph allows for formal verifications on the site structure, for example reachability of pieces of information.

The constraints addressed in this thesis are those concerning order of presentation of information, which might be important for certain applications such as e-commerce Web sites.

We do not claim an exhaustive study and implementation of property checking. Nevertheless, the properties and constraints implemented in our synthesis system demonstrate the feasibility and the power of using computational logic for this task.
12.2. Continuing our Work

- Automated maintenance

This is one of the distinctive features of the proposed approach. By deploying agents to manage parts of the Web site, some maintenance tasks can be automated, saving time and facilitating the Work of the Webmaster as non-technical people can supply the information directly for publishing.

Automatically generated agents perform recurrent maintenance tasks with minimal human intervention as only the appropriate parameters must be supplied to the agents. As a result, the effort to produce a Web site is reduced and maintenance cost is low.

- Extensibility

The system can be extended basically in two ways: supporting other high-level formalisms other than the entity-relationship model and adding other target languages for generating the Web pages. In order to extend the system, we have to define the mapping procedures to translate a data model concepts into our intermediate representation (first-order logic) and from the intermediate representation to a new target language.

12.2 Continuing our Work

During the development of our approach to Web site synthesis, we have identified some issues which require more thoughts and investigation. What became clear is that a research project involving the Web never really ends. One can always find one or other aspect to improve or to investigate in our Web site synthesis approach. We leave here some of the issues we have identified as potential extensions or developments.

12.2.1 Supporting other High-Level Specifications

An entity-relationship is employed as the high-level description of Web site applications. This choice was based on its simplicity and popularity, in particular for database applications. However, there are other formalisms that can replace the ER model or be added as alternative representations, such as:
• Ontological descriptions: this sort of representation is suitable for knowledge-intensive applications or for those applications which contains information relating to a variety of concepts. Translating ontological descriptions to our intermediate representation should be straightforward.

• UML: this conceptual data model has become very popular in recent years, for software design. Many models have been defined as sub-sets of the complete UML specification, targeting specific applications. A few research on the use of UML for representing Web site applications have already been carried out [Gómez et al., 2001], however we believe that integrating that with our logic-based approach can bring additional advantages, as property verification.

• XML schemas: XML Schemas express shared vocabularies and allow programs to carry out rules made by people. They provide a means for defining the structure, content and semantics of XML documents. Translations from XML schemas to our logic expressions are also straightforward. One of the potential advantages of using XML for specifying Web site applications is the existing facilities for transformation of XML schemas into other representations.

12.2.2 Operations

Our current representation and implementation of operations is limited. We only offer components for implementing standard database operations (search, add, update and delete) although it is possible to add new components. A further investigation is necessary to improve the specification of operations.

Another issue concerns the technology used to implement operations. We have carried out experiments on supporting dynamic Web pages using by means of a CGI-Prolog interface [Vasconcelos et al., 2000]. There are many technologies available, such as JSP, ASP and PHP scripts which may be used instead of CGI. However, their use is likely to pose additional requirements for the specification and synthesis of operations.
12.2. Continuing our Work

12.2.3 Personalisation

The display units can be automatically transformed onto alternative representations of Web pages. Personalisation concerns the support for user-defined specifications for filtering relevant content and particular visualisations.

Personalisation is partially addressed by our approach, given the facilities for generating different visualisations. However a detailed investigation on this issue is necessary to achieve a full support for personalisation of Web sites for individual users. For example, our approach lacks the support for user-defined templates, style sheets and pieces of information style. This can be implemented, for instance, via a Web-based interface.

12.2.4 Using Logic in Content Management Systems

An interesting issue for further investigation is the use 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). To our knowledge, no existing content management system offers all these features or uses logic for this purpose.

We believe that computational logic modules can be successfully integrated with content management systems to perform tasks such as property and constraint verification and to support declarative specifications in the same fashion as in our approach.
Appendix A

Accident Reporting Web Site

Here we present a complete example of the synthesis process of an Accident Reporting Web Site.

The relational representation represents the relations of the application database. Expressions of the form "Attribute1 references Relation", indicate foreign keys, which are references from an attribute (Attribute1) in the current relation to the key attribute of another relation denoted by Relation.

The transformation from the entity-relationship/relational representation to the intermediate representation is based on the instance-based transformation rules presented in Section 5.3.3.

1. Entity-Relationship Representation

entity(aircraft,
    [(model, string), (registration, string), (no_of_engines, integer),
    (type_of_engine, string), (construction_year, integer), (photo_file, string],
    key([registration]))

entity(flight,
    [(number, integer), (date, date), (type_of_flight, string), (crew, integer),
    (passengers, integer]),
    key([number, date]))
entity(pilot,
    [(licence_no, integer), (licence_type, string), (age, integer),
    (flying_experience, integer)],
    key([licence_no]))

entity(incident,
    [(incident_ref_no, integer), (date, date), (time, integer), (location, string),
    (injured_crew, integer), (injured_passengers, integer),
    (nature_of_damage, string), (information_source, string)],
    key([incident_ref_no]))

entity(event,
    [(event_id, integer), (desc, string)],
    key([event_id]))

relationship(fly, aircraft, 1, n, flight, 1, 1, [])
relationship(assigned, flight, 1, 1, pilot, 1, n, [])
relationship(happen, flight, 1, 1, incident, 1, 1, [])
relationship(described_by, incident, 1, n, event, 1, n, [(time_stamp, integer)])

2. Relational Database Representation

aircraft(model, registration, no_of_engine, type_of_engine,
    construction_year, photo_file)

flight(number, date, type_of_flight, crew, passengers,
    aircraft_registration, pilot_licence)
aircraft_registration references aircraft
pilot_licence references pilot

pilot(licence_no, licence_type, age, flying_experience)
incident(incident_ref_no, date, time, location, injured_crew, injured_passengers, 
nature_of_damage, information_source, flight_number)
flight_number references flight

event(event_id, desc)

described_by(incident_number, event_id)
incident_number references incident 
event_id references event

3. Intermediate Representation

4. Pieces of Information

info(aircraft, tuple, [M,R,NE,TE,Y,P]) ← aircraft(M,R,NE,TE,Y,P)
info(flight, tuple, [N,D,TF,C,P,AR,CLN]) ← flight(N,D,TF,C,P,AR,CLN)
info(pilot, tuple, [LN,LT,A,FE]) ← pilot(LN,LT,A,FE)
info(incident, tuple, [IRN,D,T,L,IC,IP,ND,IS,FN]) ←
    incident(IRN,D,T,L,IC,IP,ND,IS,FN)
info(event, tuple, [EI, D]) ← event(EI, D)
info(described_by, tuple, [IRN, EI]) ← described_by(IRN, EI)

5. Display Units

display(aircraft_page, [aircraft])
display(flight_page, [flight])
display(pilot_page, [pilot])
display(incident_page, [incident])
display(event_page, [event])
display(described_by_page, [described_by])
6. Operations
   \[ \text{op}(\text{search}_\text{incident}, \text{search}, [\text{incident}], [\text{incident}], \text{incident}) \]

7. Transitions
   \[ \begin{align*}
   \text{flight} & \Rightarrow \text{aircraft} \\
   \text{flight} & \Rightarrow \text{pilot} \\
   \text{flight} & \Rightarrow \text{incident} \\
   \text{incident} & \Rightarrow \text{flight} \\
   \text{incident} & \Rightarrow \text{described}_\text{by} \\
   \text{event} & \Rightarrow \text{described}_\text{by} \\
   \text{described}_\text{by} & \Rightarrow \text{incident} \\
   \text{described}_\text{by} & \Rightarrow \text{event}
   \end{align*} \]

8. Visualisation Specification
   \[ \begin{align*}
   \text{style}(\text{tuple}, \text{tuple}_\text{style}) \\
   \text{style}(\text{described}_\text{by}, \text{bullet}_\text{list})
   \end{align*} \]

9. Web Page Snapshots
Appendix B

SSP Web Site

The example presented here shows the specification of a research group Web site and the agents to maintain it. The site has a simple hierarchical organisation depicted by Figure B.1. People page presents a list of both current and previous members. Group

![Diagram of a Research Group Web Site Organisation]

Figure B.1: A Research Group Web Site Organisation

seminars page has the schedule of seminars and Publications page shows publication references. Research page has a list of project titles, each of which links to the corresponding project page.

We also present agents to maintain the following pieces of information:

- A seminar is presented weekly requiring updating the corresponding page, adding information such as author, title and abstract.
• The list of group members needs occasional maintenance as people join the group (current members) or when someone leaves the group becoming a previous member.

We simplify this example by presenting only the intermediate representation and the subsequent phases of the synthesis process.

This example is based on the actual Software Systems and Processes Group (SSP) Web site, at http://www.dai.ed.ac.uk/groups/ssp/.

1. Intermediate Representation

• Pieces of Information

\[
\text{info(intro, text, "We are interested in studying \ldots")}
\]

\[
\text{info(contact.address, tuple, [Organiser, Address, Phone, Fax, Email])} \leftarrow \\
\text{contact.address(Organiser, Address, Phone, Fax, Email)}
\]

\[
\text{info(current_members, list, CM)} \leftarrow \\
\text{CM} = \{\text{Name} \mid \text{member(Name, current_member)}\}
\]

\[
\text{info(previous_members, list, PM)} \leftarrow \\
\text{PM} = \{\text{Name} \mid \text{member(Name, previous_member)}\}
\]

\[
\text{info(project_titles, list, PT)} \leftarrow \text{PT} = \{\text{Title} \mid \text{project(Title, \ldots)}\}
\]

\[
\text{info(project, tuple, [Title, Abstract, People])} \leftarrow \\
\text{project(Title, Abstract, People)}
\]

\[
\text{info(publication, table, P)} \leftarrow \\
\text{P} = \{[\text{Author, Title, Reference, Abstract, Year}] \mid \\
\text{publication(Author, Title, Reference, Abstract, Year)}\}
\]

\[
\text{info(seminars, table, S)} \leftarrow \\
\text{S} = \{[\text{Date, Time, Place, Speaker, Title}] \mid \\
\text{seminar(Date, Time, Place, Speaker, Title)}\}
\]

\[
\text{info(seminar_abstract, tuple, [Date, Time, Place, Title, Speaker, Abstract])} \leftarrow \\
\text{seminar(Date, Time, Place, Speaker, Title, Abstract)}
\]
• Display Units
  display(home, [intro, contact_address])
  display(people, [current_members, previous_members])
  display(research, [project_titles])
  display(project_page, [project])
  display(publication_page, [publication])
  display(group_seminars, [seminars])
  display(abstract_page, [seminar_abstract])

• Transitions
  home ⇒ people
  home ⇒ research
  home ⇒ publication_page
  home ⇒ group_seminars
  project_titles ⇒ project_page
  seminars ⇒ seminar_abstract

P ⇒ home ←
  ∀ P display(P, _, _) ∧ ¬ P = home

2. Visualisation Specification

• Piece of information styles
  style(text, paragraph_style)
  style(list, bullet_list)
  style(tuple, tuple_style)
  style(table, table)
  style(contact_address, address_style)

• Page template

• Style sheet
3. Maintenance Agents

We have specified three agents:

(a) responsible for adding a new event to the schedule page.

(b) responsible for creating a new abstract page for the new event.

agent({seminars}, add, weekly)

agent({seminar.abstract}, add, weekly)

event(27, 3, '2002', 11, 'F13', 'João Cavalcanti',
     'Using hypergraphs for verifying properties of Web sites',
     'abstract/joao2002-2.html', 'group',
     ['Our approach ...']).

4. Web Page Snapshots
Appendix C

Synthesiser

The synthesiser is implemented in Sicstus Prolog [Intelligent Systems Laboratory, 2000]. We make use of some pre-defined library, such as lists in order to facilitate the manipulation of the list structures used along the code.

The Pillow library [Cabeza and Hermenegildo, 1997] provides a set of predicates to transform Prolog terms into HTML code. Hence, it facilitates our work so that we do not need to implement these procedures by ourselves.

The implementation primarily relies on the construction of an intermediate structure for the site containing all the information needed to produce the Web pages. This structure is expanded in subsequent steps adding more detailed terms until the final transformation to a target language.

The main steps are:

1. Build Site Graph
2. Check Constraints
3. Generating Page Structure
4. Synthesising Operations
5. Generating Visualisation in a target language

These steps are detailed below.
1. Generating Page Structure

2. Generating Visualisation in a target language

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

\[
\text{style(Style, Info, Additional\_parameters, \ldots) -> [T_1, \ldots, Info, \ldots T_m].}
\]

where Style is the style defined for the corresponding piece of information, Info is an instantiated information and depending on the style there are some additional parameters. T_i is a Pillow term corresponding to the visualisation of Style.

Some examples of DCGs for style generation:

\[
\text{style(itemize, InfoList) --> [itemize(InfoList)].}
\]

\[
\text{style(heading, N, Info) --> [heading(N, Info)].}
\]

\[
\text{style(text\_lines, []) --> [].}
\]

\[
\text{style(text\_lines, [TextLine|MoreText]) --> [TextLine, \\], style(text\_lines, MoreText).}
\]
Appendix D

An Implementation of a CGI-Prolog Interface

In order to implement operations we adopted the CGI approach. An interface to Prolog was developed [Vasconcelos et al., 2000]. We also make use of a small Perl script which serves as an interface between the browser and Prolog. Figure D.1 shows the architecture of the Prolog CGI interface adopted.

Figure D.1: CGI-Prolog Interface

1. The browser calls an operation via a form action.
2. The operation is started by a CGI script (written in Perl) that calls Sicstus Prolog giving the name of the corresponding saved program state.

3. The result is given back to the CGI script.

4. The CGI script forwards the result to the HTTP server to display it in the browser.

As an example of an application which involves operation, here is the shopping list application specification:

\[
\text{filter(available, in_stock).}
\]

\[
\text{d(home, [intro(Text)], []) :- intro(Text).}
\]

\[
\text{d(home, [form_text(Items)], available(Items, Result)) => d('shop list', [form_radio(Result)], [])}.
\]

The scripts/programs generated to implement operation available are:

- Perl Script: basically forwards the form input to Sicstus Prolog and print out the result in another Web page.

```perl
#!/usr/local/bin/perl
open(TMP, '/usr/local/bin/sicstus3p7 -r /hame/joaoc/progs/cgi/available.sv< -a $QUERY_STRING |');
while ( (($in = <TMP>)) )
{ $all = $all . $in; }
close(TMP);
```
print "Content-type: text/html
\n\n";
print "<<EOF;
$all
EOF

% start.pl

\nak\%-

(:- [parse_cgi]).
start_up :-
    prolog_flag(argv, [A]),
    ( tokenizeatom(A, B)
    ; write('Input '),
      write(A),
      write(' could not be tokenized'),
      ttyflush ),
    ( parse_cgi(B, C)
    ; write('Tokens '),
      write(B),
      write(' could not be parsed') ),
    main(C),
    halt.

% available.pl (my program)

main([key(query, [Query])]) :-
    output_html([start, title('Shopping List'), heading(2, 'Shopping Li:')},
Appendix D. An Implementation of a CGI-Prolog Interface

```prolog
response(Query),
output_html([end]).

response(Q) :-
    ( form_empty_value(A) ->
        output_html([\'You have to provide a list of items.\'])
   ).

response(Q) :-
    available(Q, R),
    output_html([\'Items in Stock\',\]),
    write_items(R).

write_items([]) :-
    output_html([\b(----- End of list -----\)])).

write_items([A|B]) :-
    output_html([\b(A),\]),
    write_items(B).

available([], []).
available([A|B], [A|C]) :-
    in_stock(A),
    available(B, C).

available([A|B], C) :-
    \+in_stock(A),
    available(B, C).

in_stock(cornflakes).
in_stock(beer).
in_stock(toothpaste).
```
available.svd contains:

available.pl
start.pl
parse_cgi.pl
pillow.pl

• start.pl: this is the first predicate called after the saved state resumes its execution. It processes ("tokenize" and parse) the HTML form input and then calls predicate main.

• shopping.pl: include predicate main and the implementation of the filter operation available.

The result of all this process is presented in Figure D.2.
This is an experimental Web site which is entirely automatically generated, including the CGI operations.

beer
water
wine
crips
juice
pizza
cornflakes

Figure D.2: Shopping List Application
Appendix E

TeLA - A Temporal Logic for Agent Specification and Execution

Details of TeLA syntax, semantics and a meta-interpreter are presented here. TeLA is used for executing the maintenance agents described in Chapter 10. Parts of this appendix are taken from [Vasconcelos, 2001].

E.1 TeLA Syntax

The basic construct are the temporal logic assertions of the form $\phi \Rightarrow \bigcirc(\varphi)$, where $\phi$ is a conjunction/disjunction of positive or negative literals, and $\varphi$ is a basic temporal formula.

Positive literals include the symbols $T, \bot$, representing truth and falsity and predicates, denoted by $p^n(t_1, \ldots, t_n)$ where $n$ is the arity of $p$ and $t_1, \ldots, t_n$ are terms.

Negative literals are all the positive literals preceded by exactly one occurrence of the $\neg$ unary operator, no nestings being allowed.

Terms are either constants, denoted by $a, b, c, d$, variables, denoted by $w, x, y, z$ or a function applied to them. Functions are denoted by $f^n(t_1, \ldots, t_n)$, where $n$ is the arity of the function and $t_1, \ldots, t_n$ are terms. Standard mathematical constants, like integers, reals, etc, and functions like addition, division, etc, may be used with their usual definition as their intended interpretation.
Basic temporal formulae, denoted by $\mathcal{R}$, include any positive literal and sequences of conjuncts and disjuncts, that is if $\varphi_1, \varphi_2 \in \mathcal{R}$ then $\varphi_1 \land \varphi_2 \in \mathcal{R}$ and $\varphi_1 \lor \varphi_2 \in \mathcal{R}$. Modal operators are also part of basic temporal formulae: if $\varphi \in \mathcal{R}$ then $\Box \varphi \in \mathcal{R}$, $\Diamond \varphi \in \mathcal{R}$ and $\square \varphi \in \mathcal{R}$. It is important to notice that negative literals are not allowed in $\mathcal{R}$.

Figure E.1 shows a context-free grammar for TeLA constructs.

$$
\begin{align*}
\text{Constant} & \rightarrow a \mid b \mid c \mid d \\
\text{Variable} & \rightarrow w \mid x \mid y \mid z \\
\text{Function} & \rightarrow f^n \mid g^n \mid h^n \\
\text{Term} & \rightarrow \text{Constant} \mid \text{Variable} \mid \text{Function} (\text{Term}, \ldots, \text{Term}) \\
\text{Predicate} & \rightarrow p^n \mid q^n \mid r^n \mid s^n \\
\text{PositiveLiteral} & \rightarrow T \mid \bot \mid \text{Predicate} (\text{Term}, \ldots, \text{Term}) \\
\text{NegativeLiteral} & \rightarrow \neg \text{PositiveLiteral} \\
\text{Literal} & \rightarrow \text{PositiveLiteral} \mid \text{NegativeLiteral} \\
\text{BasicTemporalFormula} & \rightarrow \text{PositiveLiteral} \mid \text{BasicTemporalFormula} \land \text{BasicTemporalFormula} \mid \\
& \quad \text{BasicTemporalFormula} \lor \text{BasicTemporalFormula} \mid \allowbreak \\
& \quad \Diamond \text{BasicTemporalFormula} \mid \Box \text{BasicTemporalFormula} \mid \\
& \quad \square \text{BasicTemporalFormula} \\
\text{NonTemporalFormula} & \rightarrow \text{Literal} \mid \allowbreak \\
& \quad \text{NonTemporalFormula} \land \text{NonTemporalFormula} \mid \\
& \quad \text{NonTemporalFormula} \lor \text{NonTemporalFormula} \mid \\
\text{TemporalAssertion} & \rightarrow \text{NonTemporalFormula} \Rightarrow \Diamond \text{BasicTemporalFormula}
\end{align*}
$$

Figure E.1: Context-Free Grammar for TeLA

### E.2 TeLA Semantics

A general framework to define the semantics of temporal logics is a Kripke structure [Manna, 1981] which is a collection of first-order interpretations. This approach is
employed to define the semantics of the language of temporal logic assertions.

A classic interpretation for first-order logic is initially defined as \( I = \langle D, F, P \rangle \), where \( D \) is the domain or universe of discourse, \( F \) is a mapping between the terms \( t \) of a well-formed formula and \( d \in D, F : t \mapsto d \in D \), and \( P \), a mapping between the atomic formulae \( p^n(F(t_1), \ldots, F(t_n)) \) (with its terms evaluated by \( F \)) and \{true, false\}, \( P : p^n(F(t_1), \ldots, F(t_n)) \mapsto \{true, false\} \). Given a well-formed formula in first-order logic, its distinct interpretations will form a collection \( I_1, \ldots, \) an infinite set if the domain is infinite. The important collections of interpretations are only those in which components \( D \) and \( F \) remain unchanged and the only varying part is \( P \).

A Kripke structure for modal (temporal) logics is a collection of classic interpretations as above plus some relation among them. In a Kripke structure each interpretation is more commonly called a possible world or possible state and is also denoted simply as \( S \). Let \( S = \{S_0, \ldots, S_n\} \) be the set of worlds or states of a Kripke structure. States are related via a reachability relationship \( \rho : S \times S \) such that \( (S, T) \in \rho, S, T \in S \), if, and only if, \( T \) is \( \rho \)-accessible from \( S \). \( \rho(S, T) \) is also employed to denote that \( (S, T) \in \rho \). A Kripke structure \( K \) is defined as the pair \( \langle S, \rho \rangle \).

The relation \( \rho \) should reflect the (generally accepted) intuitive notions about time, that is, \( \rho \) is:

1. **reflexive**, so that every state is \( \rho \)-reachable from itself, that is, \( \rho(S,S), S \in S \).

2. **transitive**, so that for any three states \( S, T, U \), if \( T \) is \( \rho \)-reachable from \( S \) and \( U \) is \( \rho \)-reachable from \( T \), then \( U \) is \( \rho \)-reachable from \( S \), that is, \( \rho(S,T) \land \rho(T,U) \rightarrow \rho(S,U), S, T, U \in S \).

The reflexive and transitive closure of a set \( \rho \) is defined as \( \rho^* : \rho^*(S,T) \) holds iff \( (S, T) \in \rho^* \). Clearly, \( \rho(S,T) \rightarrow \rho^*(S,T) \) but not the converse. It is also assumed that \( \rho \) is

3. **discrete**: it is always possible to find two distinct states \( S \neq T, \rho(S,T) \), so that there are no intermediate states \( U \), that is \( \exists S \exists T \forall U[\rho(S,T) \land \lnot(\rho^*(S,U) \land \rho^*(U,T))] \)

4. **infinite**: for any state \( S \), there is another distinct state \( T \) \( \rho \)-accessible, that is, \( \forall S \exists T(S \neq T \land \rho^*(S,T)) \)
and

5. has a smallest element \( S_0 \) from which every other state is \( \rho \)-accessible, that is,
   \( \exists S_0 \forall S_i \rho^*(S_0, S_i) \)

The evaluation function \( K(S, \varphi) \) is employed for mapping a temporal logic assertion \( \varphi \) onto the truth-values \{true, false\} using interpretation (or world or state) \( S \). \( K(S, \varphi) \) is defined as below:

1. \( K(S, T) = \text{true}, \ K(S, \bot) = \text{false} \), for any \( S \in S \).
2. \( K(S, p^n(t_1, \ldots, t_n)) = \text{true} \iff P(p^n(F(t_1), \ldots, F(t_n))) = \text{true}, S = I = \langle D, F, P \rangle. \)
3. \( K(S, \neg \varphi) = \text{true} \iff K(S, \varphi) = \text{false}. \)
4. \( K(S, \varphi_1 \land \varphi_2) = \text{true} \iff K(S, \varphi_1) = K(S, \varphi_2) = \text{true}. \)
5. \( K(S, \varphi_1 \lor \varphi_2) = \text{true} \iff K(S, \varphi_1) = \text{true} \) or \( K(S, \varphi_2) = \text{true}. \)
6. \( K(S, \varphi_1 \Rightarrow \varphi_2) = \text{true} \iff K(S, \varphi_1) = K(S, \varphi_2) = \text{true} \) or \( K(S, \varphi_1) = \text{false}. \)
7. \( K(S, \Diamond \varphi) = \text{true} \iff K(T, \varphi) = \text{true} \) for some \( T \in S, \rho(S, T) \).
8. \( K(S, \Box \varphi) = \text{true} \iff K(T, \varphi) = \text{true} \) for some \( T \in S, \rho^*(S, T) \).
9. \( K(S, \lozenge \varphi) = \text{true} \iff K(T, \varphi) = \text{true} \) for all \( T \in S, \rho^*(S, T) \).

All variables are considered implicitly universally quantified, that is \( F(x) = d \) for all \( d \in D \).

### E.3 A Meta-Interpreter for TeLA

In order to make maintenance agents executable, it is necessary to define a meta-interpreter for TeLA constructs. Given a specification \( \Sigma = (\Gamma, \Pi) \), where \( \Gamma \) is a set of temporal assertions and \( \Pi \) is an associated program, the execution of \( \Sigma \) proceeds as follows: given a state \( S_i \), for every \( \gamma \in \Gamma \) of the form \( \phi \Rightarrow \Box \varphi \) such that \( \phi \) (a first-order, non-temporal conjunction/disjunction) can be proved using \( S_i \) and \( \Pi \) then use all the
non-temporal literals in \( \varphi \) to build \( S_{t+1} \). If there are any temporal subformulæ in \( \varphi \) then they are to be carried over to future states \( S_{t+1}, S_{t+2}, \ldots \) and attempts will be made to satisfy them at every possible opportunity. Additionally all states \( S_t \) the execution goes through are kept in a circular list, which can be used for the re-enactment of a computation.

Based on the informal description above a meta-interpreter is implemented as follows:

- **execution**\((\Sigma, FC, H, S, \Delta)\) Given \( \Sigma = [\Gamma, \Pi] \) a temporal specification, this predicate executes \( \Sigma \) and constructs the history of the execution \( H \) as a list of states describing an execution of \( \Gamma \) using \( \Pi \). \( FC \) is a final condition used to stop the execution whenever the current state \( S \) of execution holds that condition.

\( \Delta \) is a set of basic temporal formulae to be satisfied at future states. \( \Delta \) is necessary because there might be subformulæ in \( \Gamma \) that refer to other future states in the execution, and thus cannot be satisfied simply in the current state.

Predicate **holds** checks if a given formula holds in the given state or in the associated program \( \Pi \). Predicates \( \rho \) and \( k \) are detailed below. The corresponding code is:

\[
\text{execution}(\Sigma, FC, H, S, \Delta) \leftarrow \\
\text{holds}(FC, S, \Pi)
\]

\[
\text{execution}(\Gamma, FC, H, S, \Delta) \leftarrow \\
\rho(\Gamma, FC, S, S_1, \Delta_1) \land \\
k(\Delta, S_2, \Delta_2) \land \\
\text{execution}(\Gamma, FC, H, S_1, S_2, \Delta_1, \Delta_2)
\]

- **\( \rho(\Gamma, FC, S, \text{NewS}, \Delta) \)**

Given \( \Gamma \), \( \Pi \) and present state \( S \) (a list of literals), this predicate obtains the next state \( \text{NewS} \) and \( \Delta \), a list of temporal constraints for the future state(s). The corresponding piece of code for the implementation of \( \rho \) is:
\[ p([r,n], S, NewS, NextState, \Delta) \leftarrow \]
\[ \forall A,B A \Rightarrow B, \text{holds}(A,S,\Pi) \wedge k(B,S,\Delta) \]

- \( k(F, S, \Delta) \)

Given formulae \( F \), a list of formulae or a single formula, this predicate builds a state \( S \), a list of literals in which \( F \) hold, and \( \Delta \), a list of temporal constraints that future states should fulfill. It obtains \( S \) by decomposing \( F \) into its basic subformulae and recursively analysing them.

For the implementation of \( k \), the evaluation function \( K(S,\varphi) \) defined in Section E.2 is extended to include \( \Delta \) as a set of temporal constraints which ought to hold in future state(s). The following expressions describes the cases that apply to this implementation, according to the format of the formulae \( F \). The complete specification of TeLA includes additional operators.

\[
\begin{align*}
 k(p(t_1,\ldots,t_n), \{p(t_1,\ldots,t_n)\}, \emptyset) &= \text{true}, \\
 \text{if } F \text{ is a positive literal, } S = F \text{ is the set that will make } k(F, S, \Delta) = \text{true}. \text{ In this case, } \Delta \text{ the set of temporal constraints for future states is the empty set.} \\
 k(F_1 \land F_2, S_1 \cup S_2, \Delta_1 \cup \Delta_2) &= \text{true}, \\
 \text{if } k(F_1, S_1, \Delta_1) = k(F_2, S_2, \Delta_2) = \text{true}, \text{ that is } F \text{ is a conjunction of the form } F_1 \land F_2, \text{ then its associated state } S_1 \cup S_2 \text{ is the union of the associated states of each conjunct. Similarly } \Delta_1 \cup \Delta_2 \text{ is the union of the sets of temporal constraints of each conjunct.} \\
 k(F_1 \lor F_2, S, \Delta) &= \text{true}, \\
 \text{if } k(F_1, S, \Delta) = \text{true} \text{ or } k(F_2, S, \Delta) = \text{true} \text{ or } k(F_1 \land F_2, S, \Delta) = \text{true}, \text{ that is if } F \text{ is a disjunction then its associated state } S \text{ and temporal constraints } \Delta \text{ are obtained by analysing each disjunct individually or by analysing their conjunction.} \\
 k(\bigcirc F, \emptyset, \{F\}) &= \text{true}, \\
 \text{that is, the empty state is associated with } \bigcirc F \text{ and } \Delta = \{F\} \text{ is the set of temporal constraints for the next state(s).}
\end{align*}
\]
E.3. Reenactment

Distributed systems are difficult to debug. The order in which events occur are not exactly the same even if the same agents are re-executed. However, TeLA makes possible a re-run of an equivalent execution of two or more communicating agents, by inspecting their history.

One of the parameters of the TeLA meta-interpreter is the *history* of the execution. The history comprises a list of all states that the computation has been through. This feature allows the reenactment of computations based on the list of states.

Communication between agents, via message passing, must be recorded as states in order to allow the reenactment of more than one communicating agent. This is the case, for instance, of a maintenance agent updating a piece of information and the publisher agent responsible for translating the intermediate representation into the target language and creating the Web pages.

Reenactment allows the Web site manager to verify what operations have been executed by the maintenance agents. This can be used for the purpose of verifying problems caused by wrong maintenances tasks carried out or auditing the agents behaviour. This feature is particularly useful if there is a large number of agents acting simultaneously on a Web site, allowing the detection of the agent responsible for the problems.

Reenactment is carried out by finding an order of state execution by interleaving the history of two or more agents. For example, consider the histories of agents A and B below:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
</tr>
<tr>
<td>a2</td>
<td>b2</td>
</tr>
<tr>
<td>send(p)</td>
<td>receive(p)</td>
</tr>
<tr>
<td>a3</td>
<td>b3</td>
</tr>
<tr>
<td>receive(q)</td>
<td>b4</td>
</tr>
<tr>
<td>a4</td>
<td>send(q)</td>
</tr>
<tr>
<td>a5</td>
<td>b5</td>
</tr>
</tbody>
</table>
We represent message exchange with predicates send(Msg) and receive(Msg). Predicate send/1 only sends a message which is placed in a common area shared by the processes. Execution of predicate receive(Msg) involves verifying if the message has been placed in the common area and retrieving it or suspending the execution of the process until the message arrives.

One possible execution of these two agents is X = \langle a_1, a_2, send(p), a_3, receive(q), b_1, b_2, receive(p), b_3, b_4, send(q), b_5, a_4, a_5 \rangle

In order to find valid execution paths, we follow sequentially the states of an agent until an operation receive is executed. Then the next agent's history is followed until the corresponding send operation is executed or the execution of the agent ends.

It is important to point out that it is unlikely that a re-run is going to follow the exact order of states of a previous execution. Nevertheless, the important issue is the synchronisation of the agents execution which can be controlled by corresponding send, receive pairs. States that are reached by an agent after the execution of a receive call cannot appear in an equivalent execution before the corresponding send call executed by another agent. This is how we define equivalent executions. In that sense, execution

X' = \langle a_1, b_1, a_2, b_2, send(p), receive(p), a_3, b_3, receive(q), b_4, send(q), a_4, b_5, a_5 \rangle

is equivalent to execution X, whereas execution

X'' = \langle a_1, b_1, a_2, b_2, send(p), receive(p), a_3, b_3, receive(q), b_4, send(q), a_4, b_5, a_5 \rangle

is not equivalent to either X or X', because state a_4 appears before the call send(q).

Re-enactment of executions is realised then by finding one equivalent execution given the history of communicating agents.
Bibliography


Bibliography


