Dissertation

A web application user interface specification language based on statecharts

by
Iwan Vosloo

September 2005

Submitted in partial fulfilment of the requirements for the degree Master of Science (Computer Science) in the Faculty of Engineering, Built Environment and Information Technology, University of Pretoria, Pretoria, South Africa

To Antoinette and Izak

(in recognition of a debt impossible to settle in the currencies of this world)

A web application user interface specification language based on statecharts

by Iwan Vosloo

Abstract

The Internet today has a phenomenal reach—right into the homes of a vast audience worldwide. Some organisations (and individuals) see this medium as a good opportunity for extending the reach of their computer systems.

One popular approach used for such endeavours is to run an application on a server, using web technology for displaying its User Interface (UI) remotely. Developing such a web-based UI can be quite tedious—it is a concurrent, distributed program which has to run in a hostile environment. Furthermore, the platform on which it is implemented (the web) was not originally intended for such usage.

A web framework is a collection of software components which provides its users with support for developing and executing web-based UIs. In part, web frameworks can be seen as being analogous to interpreters: given a specification of a UI using a specification technique dictated by the framework, server components of the framework can present the UI using web technology.

Topics related to web frameworks are scarce in the academic literature, but abound in industry and open discussion forums. Similarly, the designers of web frameworks seldom found their work on existing theory in the literature.

This study is an attempt to bridge this gap. It is focused on two aspects of web frameworks: the specification technique a framework mandates, and how such a specification can subsequently be used to present a UI via web technology.

As part of this study, a survey was conducted of 80 open source web frameworks. Based on the survey, a partial overview of the domain of web frameworks is given, covering what is seen as being typically required of a web framework and covering specification techniques that are used by existing frameworks. Two taxonomies are proposed of the strategies web frameworks use for specifying two aspects of web UIs.

Using the web as platform implies adherence to certain (intended) architectural constraints. Web framework designers often strain against these constraints. However, another point of view is to recognise that the success of the web platform is made possible precisely because of its intended architecture. (And the success of the web is surely the principal motivation for using it for remote UIs in the first place.)

With the bias of this viewpoint, a specification technique is proposed for web-based UIs. This technique is based on the well-known formalism of statecharts, with semantics explicitly defined in terms of the intended architectural components and constraints of the web.

The design of a web framework for presenting a UI so specified is also proposed (based on the theoretical background given, as well as two prototype implementations which have been developed).

Keywords: user interfaces, web applications, web frameworks, statecharts.

Degree: Magister Scientia
Supervisor: Prof. D. G. Kourie
Department of Computer Science

Acknowledgements

I would like to thank my supervisor, Prof. Derrick Kourie, who tends to bring not only knowledge, but much-needed wisdom to the endeavours of his students.

Thanks to Dr Andrew Boake for the carefully aimed, encouraging personal propaganda without which this study would never have started. I also thank Anton Malan who has had to weather the most horrible first drafts of this document and provided important advice and references, and Linda Weber for editing the final document.

Several programmers on informal forums and mailing lists have also provided valuable input. Thank you all.

Contents

List of Figures

1 Introduction

1.1 Background

The web was conceived as a hyper linked network of information which is embedded in documents using a simple document format that is easily editable and viewable on many different platforms [Berners-Lee, 1996].

Since then, the web has enjoyed overwhelming success and plays host to many unanticipated uses (and, it could be argued, abuses). The basic architecture supporting this massively distributed system has matured with experience, guided by the Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C). This architecture is enabled by three primary standards: Hypertext Transfer Protocol (HTTP), Uniform Resource Identifier (URI), and Hypertext Markup Language (HTML)[1].

Some uses of the web require more than what is provided by traditional web standards. Indeed, for many, the web has merely become the presentation tier of a multi-tiered conglomerate of systems [Ginige and Murugesan, 2001].

These applications are opportunistic attempts to exploit the large, standardised installed base of the web in our heterogeneous world for the purpose of delivering the UI of a system to a wide audience.

The specification of such "web applications" is core to this study. Two subsections now follow. One deals with web applications and the other with web frameworks. Web frameworks are executable architectural frameworks for implementing web applications—thus closely related to the structure and requirements of a web application.

1.1.1 Web applications

The term "web application" is now widely used to to distinguish a certain type of web site. Several definitions exist—in this study a web application is taken to be a possibly complex

[1]The latter is increasingly being replaced by the newer Extensible Markup Language (XML) and Extended Hypertext Markup Language (XHTML).

server application that utilises web technology in order to deliver its user interface. (See Conallen [1999] for an example of another definition.)

The end-to-end technical architecture of a web application needs to take into account a fairly impressive range of problems (some of which are catered for by the web standards). To name a few: since a web application is run in a hostile environment, it typically needs security—both for the server and its clients. Often scalability is of importance, since the potential audience is vast. Of course, the UI should be of acceptable quality and flexibility. Even in the most degenerate case a web application is a concurrent, distributed application, which, to make matters worse, depends on a completely stateless protocol (HTTP). Typically, web applications need to be integrated seamlessly with several back-end systems (or at least present a seemingly integrated front to them). Often database transactions and security realms need to span several such system boundaries.

While solutions for each one of these problems exist, the particular implementation of chosen solutions impact one another. Solving them simultaneously, elegantly, and in this particular context, would not seem to be a simple task at all.

1.1.2 Web frameworks

Web standards have to cater for a wide variety of uses. Thus, from the point of view of the web application architect, web standards standardise the specification of a least common denominator of generic functionality that can be used as a platform for implementing a web application.

To take this view of the standards is to have an agenda that fundamentally differs from the original agenda followed by the authors of the web standards.

This dissertation is motivated by the perception that, owing to the complexity of web applications, programmers need a combination of specification techniques and specialised architectural building blocks geared for building web applications. (The next paragraph presents some informal evidence for this perception.) These specification techniques need somehow to be mapped down to the common denominator represented by the web standards. In other words, execution environments need to be implemented that can present UIs as per such specifications.

Ideas and arguments regarding these topics (architectural building blocks, specification techniques and their implementation) abound in the form of thousands of programming libraries for web development (colloquially known as web frameworks), informal blogs, news groups and other unmoderated web-based discussion forums.

This proliferation of discussions and solutions is taken by some to be an indication that the problem (of finding the right abstractions with which to design web applications) is not satisfactorily solved in practice. It may also be that the problem just has a great number of variable parts—strung together in different ways, depending on the context. (It is valid to want the flexibility of utilising several small projects representing partial solutions of the problem and use them together to solve a complete problem.)

Whatever the case may be, there does not appear to be a clear overview in the literature of what is happening in this space. It would seem that the current development of the field happens mostly as a process of evolution in the world of technical discussion forums and projects.

The most tangible (and tested) form of ideas in this field is probably the architectural frameworks that comprise higher level architectural constructs with which programmers can specify an executable web application UI: web frameworks.

The next section briefly diverges from the main focus in order to define a closely related alternative to web applications (which will be referred to later).

1.2 An alternative

Developing a traditional Graphical User Interface (GUI) application (as opposed to a web-based application) that has to be installed on a number of client machines is cumbersome. Applications depend on other applications and libraries—installing an application properly requires that all its dependencies should be installed (at the correct versions) and configured correctly. Managing all the different distributed versions of such an application is a major job. If such an application is to run on many different operating system platforms, the problem tends to grow significantly more difficult.

To a large extent, the popularity of web applications has been advanced by difficulties associated with such traditional applications. Web applications adhere to standards that are implemented on many platforms. They also run on a server—there is no need to install them on clients—and upgrading a web application also happens once on the server, instead of at each installed client location.

On the other hand, web applications cannot provide the rich UI experience users have come to expect from traditional GUI applications. To overcome this barrier, another type of application has been evolving: the Rich Internet Application (RIA). An RIA is an application that runs on the client in a special environment. Users are required to access a web server once, from which the RIA will be downloaded automatically and executed (often in a browser or browser plugin). There is no need to install it (in the usual sense), or even to be connected constantly to its server of origin.

Prominent experiments in this direction include the MacroMedia Flash products, and Java Network Launching Protocol, also known as Java Web Start (JNLP).

1.3 Topics of particular interest

Before proceeding to outline this dissertation, it would be appropriate to briefly introduce a few topics that have an important contextual bearing on this work: namely the notions of Representational State Transfer (REST), Model-View-Controller (MVC) and page flow.

1.3.1 REST

There is a tension between what implementors of web applications need and what the rest of the web community is willing to allow the web standards to provide.

Web frameworks provide a high-level way in which to specify a web-based UI. They also provide facilities for executing such a specification. Web frameworks often evolve informally and seemingly without reference to academic literature.

The result is that this tension seems not to be consciously addressed and that web frameworks often violate the architectural constraints specifically intended for the web standards without taking the original intention of these standards into account. The result of this is that web application frameworks often attempt abstractions that are complex to implement using standard functionality and are difficult to scale.

However, this present research is inspired by the instinct that, in following one's agenda, to maximize synergy one ought to have the disposition of a guest who plays by the rules of the host. If one pressurises to change or add to those rules, then it is important—if only as a matter of courtesy—to understand them and to understand the reasoning behind them.

Web applications, representing a large portion of the web, do indeed exert such pressure on the web standards.

In Fielding [2000], an account is given of the motivation governing the web standards and their evolution. The "architecture of the web" is presented in the form of an architectural style called REST. REST defines some architectural components on a high level, listing and motivating the constraints they embody. (An overview of REST concepts is given in more detail in Section 3.2.) These constraints (like the stateless nature of the HTTP protocol) are often seen by web application implementors as limitations of HTTP for which they need to provide workarounds [Fraternali, 1999, Belapurkar, 2004], instead of acknowledging them as intended constraints and building on the intention motivating these constraints.

REST is important and prominent: formulating it grew out of the experience and lessons learnt during the process of extending the web standards over a period of several years [Fielding, 2000, p66, p74]. Also, according to Fielding, (one of the principal authors of the HTTP standard), REST is used to guide the design of ongoing improvements of the standards governing the modern web architecture.

The current web standards can be seen as an implementation of a REST architecture. These standards dictate the rules that enable the heterogeneous mix of web browsers, web servers and many intermediate components to all work together and present the web as we know it.

This work is a balancing act: an attempt to cater for web application needs while striving to adhere to the (often conflicting) intention put forward in the form of REST.

1.3.2 The MVC design pattern

The MVC architectural design pattern is mentioned throughout this dissertation. For a reader who is unfamiliar with it, a brief overview follows.

MVC is an old architectural design pattern used for structuring an interactive program [Gamma et al., 1995]. It is based upon the principle of separation of concerns. MVC enumerates three distinct concerns in a UI intensive application: that of the model, the view, and the controller. The model refers to the domain model embodied in a program. The view refers to how this model can be presented to a user. The controller is concerned with keeping the model and view(s) synchronised by relaying events and data between them, while keeping them decoupled.

A model is kept independent of the possible views of it. There may be several different views on the same model. Should an event occur in the model, the controller will update every view to reflect the change; similarly, events generated by the user via a view are translated into actions on the model.

In traditional GUI programs, quite complex interactions are possible between a domain model and the views presented of it. The environment within which web-based applications are deployed is more restrictive: web servers typically do not initiate updating what is being displayed by web browsers—all events are initiated by the browser by sending HTTP requests to the server. Hence, controller concerns are somewhat simplified to translating events into actions on the model and deciding what view to display after an event occurred. (See *Struts*, for example[2].

1.3.3 Page flow

In Chapter 2 an overview is given of a survey of web frameworks. Most of these recognise that a web application is more than a dynamically generated web site—it presents a view on a domain model and has to respond to dynamic UI events by interacting with this domain model. Many frameworks attempt to separate these three different concerns along the lines of the MVC design pattern.

However, most of them still focus on the problem of how a particular response is generated to an incoming HTTP request for a single page—a narrow focus for an architectural framework which aims to specify and run a complete UI for a web application. In most of the projects surveyed, a programmer cannot take a wider view and clearly specify the relationship between different pages in a web site—the possible paths a user could use to traverse them.

Pages are linked to each other by HTML links or buttons (or similarly scoped mechanisms) which are embedded into the specification of presentation of each individual, dynamically generated page. The resulting structure between pages is difficult to visualise and often quite chaotic.

Page flow is a colloquial term describing this relationship between the pages of a web site. Page flow is analogous to control flow in programming languages.

Indeed, from the perspective of this study, the low-level page flow tools (buttons and links) can be likened to the goto statement in programming languages: the Uniform Resource Locator (URL) of a page is really its address. An HTML link on another page to this address is to page flow what a goto statement is to control flow [Dijkstra, 1968]. In fact, it could be judged as being slightly worse, since the concern of specifying dynamic behaviour is dispersed between and embedded within the specification of several different web pages involved (the presentation).

A higher-level construct seems to be needed to specify the page flow of a web application. Few attempts at such specifications have been made by framework implementors, and only recently (see Section 2.3.11).

Page flow is a colloquial term. Closely related to it is the concept of the navigational model of a web application, which is widely discussed in the literature. (See, for example, Winckler and Palanque [2003].)

Here, as with REST, web frameworks typically appear ignorant of formal literature.

[2]Note that this style of reference is used throughout the text for referring to framework projects which form part of a survey (introduced later). Such surveyed projects are listed by name in a separate bibliography.

1.4 Overview and scope

This study focuses on the development of web applications.

The more ambitious RIAs have been mentioned, since these represent an honest and direct attempt at an architecture promoting the real agenda underlying many web applications. Although the RIA is an interesting topic in its own right, it is outside the scope of this study. The primary interest of this study is to investigate ways of complying with the demands and intention of web standards[3], and yet still allow web application implementors to follow their own more specialised agendas.

In the course of this dissertation, an overview will be given of how web applications can be specified and how such a specification can be executed in a standards compliant, REST-friendly way. This is done by focussing on web frameworks: what is required of them and their strategies for providing solutions.

As input, a sample of 80 open source web frameworks were studied. The Java 2, Enterprise Edition™ (J2EE) family of specifications was also referenced extensively, since these represent the thoughts of a fairly large and main-stream community of programmers, vendors and thinkers in this arena—boosted by the financial benefits of being main stream and developed according to a process involving public scrutiny [Sun Microsystems, 1005-2005].

Chapter 2 proposes a list of the perceived requirements for web frameworks, followed by two taxonomies: one for strategies regarding the specification of presentation concerns, and one for strategies concerning the specification of controller concerns (along the lines of the MVC design pattern).

Program specifications depend heavily on two cornerstones: the higher-level control flow specification methods introduced by structured programming and procedural abstraction [Dahl et al., 1972].

In Chapter 3 a page-flow-centric specification technique is proposed for web-delivered user interfaces based on these same programming principles, where semantics is related to the intention of the web architecture as presented by REST. Instead of using the usual control flow constructs of structured programming, a graphical notation is presented, based on statecharts [Harel, 1987] as standardised by Unified Modelling Language (UML) [OMG, 2003]. A graphical notation may appeal to a larger audience when it comes to UI design.

The design of an implementation of this specification technique is described in Chapter 4, touching on several implementation-related issues[4]—the intention being for it to serve as an

[3]The term "web standards" is used here to loosely refer to the main standards enabling the web as discussed in Section 1.1.

[4]By "an implementation of this specification", we mean a web framework that can execute the specification.

example which demonstrates the general implementation issues, still with reference to the intention presented by REST.

In Chapter 5 some related notations are discussed and a critical evaluation and conclusion are presented.

The reader is also made aware of the short glossary provided (Page 111) in which terms are defined which may be used with specific semantics in the context of this dissertation, or terms which are used in the text assuming reader familiarity.

1.5 Related work in brief

Web frameworks are architectural frameworks which provide an execution environment for web applications (often only for the UIs of such applications). Implicitly, these frameworks also dictate certain low-level specification methods for the UI of a web application. The topic of *web frameworks* is not well represented in the academic literature. (Examples of where frameworks are mentioned or of papers aimed at the same level of abstraction as web frameworks are Copeland et al. [2000], Chao et al. [2003], Hassan and Holt [2003], Draheim and Weber [2005].). In contrast, topics related to the *modelling of web applications* abound. Some of the more relevant categories are briefly exemplified below. (A more detailed discussion of related work can be found at relevant locations during the course of the dissertation and in Section 5.1.)

Reverse-engineering of web applications

Reverse engineering is a disciplined method of inferring the design models of the inner workings of an artifact. Legacy software systems are often reverse-engineered in order to facilitate their maintenance and further development. The same applies to web applications. Reverse-engineering of web applications and web sites is interesting in the context of this study, because it yields modelling techniques that are useful for visualising and modelling web applications. [Kienle and Müller, 2001, Huang, 2001, Lucca et al., 2002]

Web design tools

Some tools are presented with which web sites can be developed [Rode et al., 2004, Helman and Fertalj, 2003, Shimomura, 2005]. Many of these are aimed at empowering non-programmers to develop web sites. From such tool-based specifications, web sites are generated. (The target of such generation is a specification in terms of an existing web framework—such as JavaServer Pages™ (JSP), *Struts* or Active Server Pages (ASP)—which

This is explained in more detail in Chapter 4.

is responsible for executing such a specification). The generation of such lower-level specifications is discussed for a few tools in Helman and Fertalj [2003].

Model driven development of web applications

Model driven development centers on the concept of building abstract models of a system, and then generating the real system from the model in a particular implementation environment [Mellor et al., 2003, Thomas, 2004]. Usually models are constructed on a high level, and then transformed successively into lower-level models (with increasing detail) until a representation is reached which can be executed. Model driven development is thus usually very tied to an overall development process which covers a spectrum from requirements analysis up to implementation.

Since the problems associated with web application development are seen as being related to development process, many approaches are proposed for the model driven (or similar) development of web applications and web sites [Ginige and Murugesan, 2001, Ceri et al., 2000, Knapp et al., 2003, Güell et al., 2000, Winckler and Palanque, 2003, Schranz et al., 2000, Koch and Kraus, 2002]. Again, these approaches are relevant to the present study because of the specification techniques they employ.

1.6 This study contextualised

This study proposes a specification technique for web-based UIs, and a web framework which renders such a specification directly executable. As such, this study attempts to bridge the gap between the executable specifications of web application UIs (as expected by frameworks) and the higher-level models used in, for example, model driven web application development. An attempt is made to bring the thoughts and parlance of web framework authors into the body of academic literature.

In Chapter 2, detailed taxonomies are proposed of the specification techniques supported by existing web frameworks. A broad categorisation of tools for web development is given in Fraternali [1999] and a small number of tools are surveyed in detail in Copeland et al. [2000]. However, no detailed overview was found in the literature specifically regarding web frameworks.

The specification technique proposed in Chapter 3 is heavily based on statecharts [Harel, 1987]. Statecharts are mentioned by several authors in relation to specifying user interfaces [Horrocks, 1998], hypertext, and the navigational model of web sites and web applications [Zheng and Pong, 1992, Sauter et al., 2005, Leung et al., 2000, Gorshkova and Novikov, 2004, Winckler and Palanque, 2003, Turine et al., 1999, Winckler et al., 2001]. Navigational models of web applications are very closely related to the MVC controller concerns in web framework parlance, as well as the concept of page flow (Section 1.3.3) which is frequently mentioned on informal forums in relation to web frameworks.

On a high level, the specification technique proposed in this dissertation differs from other approaches in the literature in the following ways:

- The proposed specification technique uses a *subset* of statechart notation and semantics, whereas most others *extend* the semantics and notation of UML. (This is true for notations based on statechart diagrams and those based on class diagrams.) [Koch and Kraus, 2002, Conallen, 1999, Turine et al., 1999, Winckler and Palanque, 2003]

- The proposed specification technique constrains the end result of what can be specified. Many other approaches are more complicated, since they attempt to be able to model everything that can be built with low-level web technology (Leung et al. [2000], for example). The aim in this study is not to be able to model everything one could build, but to be able to model a UI and be able to implement it using a *subset* of the building blocks available.

- The proposed specification technique is meant to be directly executable by the framework implementing it, without the need for generating a lower-level specification (required by all other approaches in the literature)—thus raising the level of abstraction at which the framework operates[5].

- The focus in this study is very specifically on the UIs of web applications, with the effect that the specification technique need not cater for the more generic needs of general web sites or hypertext systems, nor need it model other aspects of web applications (such as the conceptual model, for example—Gómez et al. [2000]).

- Perhaps most importantly, the proposed specification is carefully mapped to (and constrained by) the architectural style of the web (REST) as presented in Fielding [2000]. To our knowledge this mapping is not done anywhere in the informal forums of web frameworks, nor in the literature. In fact, sometimes the constraints of REST are seen by authors as limitations of the HTTP protocol which need to be addressed (for example Fraternali [1999, p239 and footnote 3]).

Perhaps most relevant to this study is the work done by web framework implementors. However, this work is not well represented in the literature. Chapter 2 is an attempt to

[5]Some models in the literature are executable, but such executability is the seen only as useful for prototyping—see, for example, Draheim and Weber [2005].

provide an overview of this informal domain, providing a context for the notation presented in Chapter 3 and the design discussions of a framework in Chapter 4.

1.7 Summary

In this chapter the notions of a web application and a web framework are introduced (Section 1.1). Background to the present study is also given regarding three notable topics which influence the bias of this study in important ways: REST, MVC, and page flow (Section 1.3).

The dissertation is outlined and its scope is motivated in Section 1.4 (partly against the background of an example alternative to web applications introduced in Section 1.2).

The chapter is concluded with a high-level overview of related work in the literature and a contextualisation of the present study in terms of the work in the literature (Sections 1.5 and 1.6).

Chapter 2 gives an overview of the domain of web frameworks.

2 Web framework overview

A web framework is an architectural framework which provides an execution environment for web applications or the UIs of web applications. Each framework offers its user some or other specification mechanism so that the framework can present an application UI, once the framework has been provided with a specification of the required application UI. Different frameworks allow different strategies for specifying a UI[1].

In this chapter, an overview is given of the thinking of web framework designers, as reflected in their products. The overview is based on a survey of some 80 web frameworks (listed in a separate bibliography at the end of this dissertation[2]). These frameworks are typically expected to provide support for certain implementation-level functionalities—seen here as what is required by a web application of a web application framework.

Section 2.1 enumerates the *requirements* typically expected of web frameworks.

Sections 2.2 and 2.3 address the *strategies* employed by web frameworks for specifying an application's UI. Categorised according to the MVC pattern, such strategies are needed for two aspects of a UI: the view concerns (discussed in Section 2.2), and the control concerns (discussed in Section 2.3).

Although many frameworks include support related to model concerns, these are excluded. Strategies relating to model concerns tend towards solutions for the problems of transparently persisting data or providing mappings between programming language objects and the relational databases in which they are sometimes stored. (One example is the Enterprise Java Beans™ (EJB) specification—DeMichiel [2003].) This is quite a large field in its own right and peripheral to the focus of this study, hence the exclusion. View and controller concerns, in contrast, are cornerstones in the design of a UI.

[1]Note that during the course of this dissertation references may be made to "the implementation of a specification technique". This usage is shorthand for: the implementation of a framework which can present a UI specified using the said specification technique.

[2]In order to avoid confusion with the main bibliography of this dissertation, these frameworks are not cited in the usual way. The names of these projects are merely used in the text and indicated with a telling font as in the example: "the *Smile* framework".

2.1 Requirements

Although no comprehensive list of functional requirements has been found in the literature, the following list has been gleaned from the survey of 80 frameworks. Some of the requirements will be seen to be more fundamental to the problem of designing web applications than others. Their individual implementations tend to influence each another—the fundamental model used by a web framework is often dictated by its combined solution to all of these.

2.1.1 Presentation

Presentation is perhaps the simplest and best understood functional requirement for which a web framework should provide. It is the first requirement recognised—early web frameworks focussed exclusively on presentation.

"Presentation" entails everything a web application needs to do in order to render its user interface in a client's browser, using HTML or a similar markup language. Typically, in order to facilitate re-use, each page is not merely seen as a web page—it is viewed rather as a window in a GUI, composed of different UI components. The difference between the UI components in a web page and those used in a GUI is that the latter are usually available as re-usable programming language components with behaviour linked to their presentation; the former are mere low-level representations. The low-level representations on a web page not only lack behaviour, but more complicated compositions of them cannot be made for re-use.

2.1.2 Forms handling

User input on the web takes place through HTML forms that are submitted to the web server by a browser. Forms are submitted to a particular URL on a web server, as an HTTP request. "Forms handling" is a term often used to describe various tools which a programmer can use to deal with forms—and especially to deal with the user input that has been received as part of submitted forms.

For example: when user input is received via an HTTP request, it comes in the form of named text strings. These can be marshalled to more useful typed programming language objects. Errors can occur during this process which need to be reported to the end user.

Forms can also be submitted as a result of different buttons being clicked on the form. As part of forms handling, a program needs to determine which button caused the submission and needs to take appropriate action, depending on that information.

A web framework should provide tools to help a programmer deal with forms handling tasks.

2.1.3 Validation

Validation is also related to user input and is often seen as part of forms handling. Validation usually refers to checking user input against some constraints (a number may be required to be within a certain range, or a string representing an email address can be required to match a regular expression). However, validation might also be dependent on more sophisticated domain knowledge which might have to be checked in a back-end system on another tier in a multi-tier architecture (where domain-specific knowledge typically resides).

Provision needs to be made in a web framework for a programmer to easily specify what validations need to be checked, how validation errors will be reported to the user, and what influence such errors will have on the dynamic behaviour of the presented UI.

2.1.4 Event handling

Frameworks also need to react in response to events signalled from the browser (or the user).

An incoming request from a browser signifies that the user has triggered an event of some kind—either by submitting a form, or by having clicked on a link. Such a request can be interpreted in many ways. For example, it can be seen as notification that a UI event occurred, or a batch of UI events can be derived from it. The design of a web framework determines what constitutes an event, how events are triggered, how events relate to HTTP requests, and how custom program code will be invoked in response to events.

Sometimes a request is simply mapped to a method call, for example. Other frameworks have more complicated programming models for dealing with events.

2.1.5 Page flow

Page flow is of particular importance to this work and has been introduced in Section 1.3.3. In summary: page flow is a description of the possible ways in which a user can traverse the different logical locations (web pages) in a web based UI. It is to locations (or pages) in a web UI what control flow is to statements in a programming language.

The implementation of page flow has implications for the client browser: typically, a browser is based on the premise that a URL denotes such a location on a web site. The browser adds functionality, such as the ability to bookmark a location and to go backwards and forwards along the path traversed by the user, based on the identification of locations with URLs.

Frameworks should provide a means by which page flow can easily be specified.

2.1.6 Session state

HTTP is a stateless protocol. It provides little support for relating a particular request to others in a lengthy conversation. Web applications, though, need some way of relating all the requests from a particular user during a user session in order to be able to store information between requests for a single user session. The information stored in the scope of a user session is referred to as "session state".

Session state is used, for example, when a user has provided input for several input items on a form, has submitted the form and then expects that her input will subsequently still to be shown on the form, even in the event of a validation error requesting her to change some of the information. Such an interaction may span several requests to the web server. The information supplied at the beginning in the submission of the form is needed to render, for example, a half-completed form later on (possibly with an error message added).

Note that holding session state on the server is in direct conflict with REST.

2.1.7 Authentication

Authentication is related to session state. Web applications often need users to authenticate themselves, for example, by means of a user name and password. Once authenticated, the web application can modify its UI based on who the user is. Authentication, though, is something that most web application designers prefer to happen only once during a user session—to be stored by the application for the duration of the session. (The storing of such information again relates to storing session state which may be in direct conflict with REST, depending on how it is implemented.)

Of course, authentication needs to be implemented in a secure way—and in this case, neither the client nor the transport mechanism can necessarily be trusted.

The HTTP protocol provides more than one authentication mechanism, which can be further augmented with the use of HTTP over Secure Socket Layer (SSL). However, the problem is exacerbated by the fact that many web browsers implement the standard incorrectly—leading to complexities for a web server that has to cater for all browsers. (This is discussed further in Sections 4.1.4, 4.2.2.)

2.1.8 Concurrency

Web-based applications are, almost by their very nature, accessed concurrently by several users. Since servicing a single request may take a long time, a way is needed to service requests in parallel. This can be done by maintaining a pool of processes, or a pool of threads in a single process, or even a pool of machines with pools of processes on them.

Some servers also have an asynchronous model—they kick off processing (typically on a back-end system) for each request as it is received, without blocking to wait for the processing (which is often Input/Output (IO) intensive) to complete. The system can then alternatively poll for results from such processing (and quickly send a response) or start more processing for incoming requests (see Rushing [none], for example).

The particular concurrency model chosen has quite an impact on how many other requirements can be implemented. A naïve implementation of session state, for example, could keep some information for a user in memory. But, generally, memory cannot be shared between processes or machines, posing a problem in circumstances where subsequent requests from the same user could be routed to another machine, or another process.

When using multi-threaded approaches, care needs to be taken that the programs and libraries involved are all thread-safe.

2.1.9 Back-end integration

Usually, web applications are built using a multi-tier architecture, with their presentation tier being web based. Web frameworks are concerned with such web-based presentation tiers.

In a degenerate case, this web tier would need some way of accessing a local database. More often, it would need access to a remote database or application server. Such "back-end systems" often need to be accessed within the boundaries of a single database transaction or security realm.

A web framework should provide abstractions for a programmer that automatically deal with these low-level technical concerns in a sensible way. Apart from the fact that web programmers do not want to have to think about such low-level, complex, technical issues, most UI programmers are not well versed in such matters.

2.1.10 Resource usage

A common thread running through many implementation discussions for web frameworks has to do with the economical use of resources. Several scarce or expensive resources are used by web applications or web frameworks. Frameworks have to take care to use resources such as the following in ways that remain performance efficient when scale-up occurs:

File descriptors are in limited supply on a machine, and web serving software uses them for writing to log files, in writing to temporary files for persisting session state, and the like.

Sockets (or ports) are also limited on each machine—limiting the number of network connections that can be maintained at one time.

Connections to databases or other back-end systems are often only allowed in a limited number, and creating a connection typically involves a performance overheads too large to incur on every HTTP request. Thus, connections to back-end systems might not be created for each request. Instead, a pool of connections is usually maintained so that the total number of connections is centrally managed, and an already created connection can quickly be allocated to the servicing of one request.

Memory cannot remain allocated to each particular user for the duration of her session, given the number of concurrent users that has to be serviced.

Processes (and, to a lesser degree, threads) involve overheads to create and destroy. Thus, they also are typically created in advance in a pool from which existing processes or threads can be allocated to requests.

2.1.11 Miscellaneous

Miscellaneous requirements may be added to the important set explained thus far. For example, web applications often need special support to enable them to be presented in different languages, to work with different locales, or to use different character encodings used for web pages [Spolsky, 2003, Dürst, 2005].

2.2 Strategies for view concerns

A taxonomy of strategies employed for handling presentation concerns of web frameworks is proposed in Figure 2.1. The discussion of the taxonomy is structured as an ordered traversal of the nodes in the taxonomy tree, with a description of each node provided—roughly ranked in order of complexity or sophistication.

The basic problem is how web serving software can generate a response which a web browser can display as part of the UI of an application. In practice today, this means generating a textual document in a markup language such as HTML, XHTML or XML[3]. (We often loosely talk of HTML for brevity where another markup language can be substituted which browsers are able to display.)

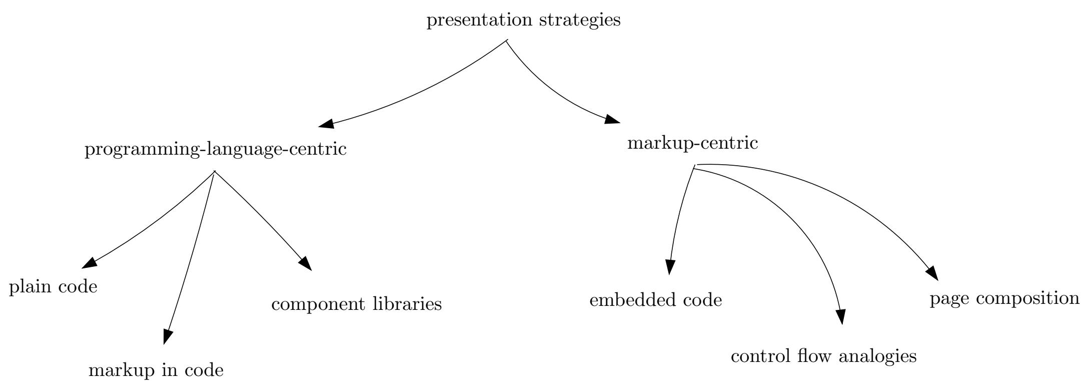

Figure 2.1: A taxonomy of strategies for presentation concerns

2.2.1 Programming-language-centric approaches

Programming-language-centric approaches to generating markup all consist of specifying a program in a general-purpose programming language which generates the markup file.

Sections 2.2.2, 2.2.3, and 2.2.4 describe further specialisations of this category.

2.2.2 Plain code

The "plain code" strategy is the very simplest of all strategies for generating markup. Code is invoked that either writes the markup as text to a file, or just returns it in a string. This is the strategy used by Java Servlets™ [Coward and Yoshida, 2003].

The Python [Python Software Foundation, 2005] programming language function provided below illustrates the point:

[3]The interested reader is referred to Raggett et al. [1999], Yergeau et al. [2004], W3C HTML Working Group [2002] in connection with the details of these standards.

```
def foo():
  print '<body>'
  for word in ['these', 'are', 'words']:
      print '<p>Word: %s</p>' % word

  print '</body>'
```

Note that the "print" statement in Python writes to standard output[4]. The function could also have written to a different file, or have returned a string value.

2.2.3 Markup in code

Some systems augment a programming language with special semantics that enable one to embed markup in the programming language. This can best be explained by an example.

Quixote is an extension to the Python programming language. An example of a *Quixote* function can be seen in the following code excerpt (which returns the body of an HTML document:

```
def foo [plain] ():
    '<body>'
    for word in ['these', 'words']:
        '<p>Word: %s</p>' % word

    '</body>'
```

A *Quixote* function always returns a string (even though no return value is explicitly specified). This return value is automatically computed as follows: at first the string is empty, but as each statement in the function is executed, its individual return value is converted to a string and appended to the return value of the function. *Quixote* is helped here by the fact that literal values in Python can be used as statements (with no side effects, but themselves as return value). Return values of `None` are ignored[5].

Hence, text can be embedded in a programming language with the programming language used to govern the final text that will be returned. The example above will yield:

[4]Standard output is often used by scripts for returning output to a browser.
[5]In Python, a return value of `None` denotes a void return value.

```
'<body><p>Word: these</p><p>Word: words</p></body>'
```

2.2.4 Component libraries

Some frameworks include a library of UI programming language classes similar to GUI framework libraries. With these, a web page can be composed using the programming language objects. Such a composition can then be rendered as markup.

As an example, here is a Java method returning a window (or page), using the *Echo* framework. (The example is a stripped-down adaptation of their own "hello world" example.)

```
public Window init() {

    Window window = new Window();

    ContentPane content = new ContentPane();
    window.setContent(content);

    Label label = new Label("Hello, World!");
    content.add(label);

    return window;
}
```

2.2.5 Markup-centric approaches

Markup-centric approaches represent a broad category of strategies comprising a syntax with which a template for an HTML document can be specified. Such a template can then be rendered (or executed) to yield different actual markup documents depending on the context and parameters with which the template was executed.

This category is further refined and exemplified by strategies in Sections 2.2.6, 2.2.7, and 2.2.8.

2.2.6 Embedded code

As far as templates go, a template with embedded code is probably the simplest model to understand. A special syntax is introduced with which programming language code can be embedded in an otherwise normal HTML document. Early JSP is an example [Roth and Pelegrí-Lopart, 2003][6]:

```
1 <body>
2   <% String[] words = "these", "are", "words";
3      for (int i = 0; i < words.length(); i+=1) {
4   %>
5   <p>Word: <% words[i] %></p>
6   <% } %>
7 </body>
```

Here, the brackets “<%” and “%>” are used to delimit Java code. Note that logically, line 5 is in the body of the for loop started in line 3. The semantics is that a copy of line 5 would be included in the output for each iteration of the for loop, yielding:

```
<body>
  <p>Word: these</p>
  <p>Word: are</p>
  <p>Word: words</p>
</body>
```

Some of these template languages employ the same semantics, but use their own, more lightweight syntax. Here is an example of *Cheetah*:

```
<body>
#for $word in $wordList
  <p>Word: $word</p>
#end for
</body>
```

[6]Note that specifications from the J2EE family are cited in this overview as if they were frameworks (notably, JSP and JSF). Many implementations of these specifications are available, but citing the specifications is preferred because of the prominent role of these specifications and because the strategies used by individual implementations do not differ from the specifications.

2.2.7 Control flow analogies

Some markup-centric approaches are attempts to introduce the same semantics expressed using the embedded code approaches (Section 2.2.6), but in a way (from the point of view of the target markup language) that is less intrusive than embedding the code.

They extend the markup language's own vocabulary with additional control flow statements. The semantics of these statements is the same as those used to control flow in conventional programming languages: if statements control the conditional inclusion of parts of the template, and looping statements allow one to include a chunk of the template several times.

The extension of the markup language is usually done either by introducing added tags to the markup language, or by adding attributes to existing tags. The motivation behind this seemingly cumbersome notation is that the source code of the template would then be valid according to its markup language (usually some form of XML, such as XHTML), and thus be editable in What You See Is What You Get (WYSIWYG) editors[7]. JSP restricted to its JSP Standard Tag Library™ (JSTL) is an example [Pierre Delisle, 2002, Roth and Pelegrí-Lopart, 2003]:

```
<body>
<c:foreach var="item" items="wordList">
 <p>Word: $item</p>
</c:foreach>
</body>
```

2.2.8 Page composition

Template languages based on control flow analogies alone (Section 2.2.7) suffer from the weakness that the low-level control flow constructs used often obscure the intention of the specification of a complex page. The intention is most often to compose a page from several UI components analogous to widgets in GUIs.

Current page composition approaches also extend the vocabulary of a markup language by adding to its tags or attributes. However, the aim is not to (only) introduce constructs on

[7]It should be noted that WYSIWYG editors will not display the page resulting from executing the template being edited. They can only show the logical structure of the source code of the template itself (which also happens to be valid XHTML). To get around this problem, some template languages are designed so that their source looks like an example of what may be rendered, given some execution of the template. This goal can of course not be wholly attained, since a template, by nature, does not evaluate to a static document.

a control flow level, but rather to allow a programmer to compose a page from a standard (often extensible) set of UI components. These languages also retain the more basic control flow level constructs, but their use is much rarer and the resulting template is more readable.

JSP serves as an example, but this time using the newer tag libraries provided by JavaServer Faces™ (JSF) [Roth and Pelegrí-Lopart, 2003, Pierre Delisle, 2002]:

```
 1  <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
 2  <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
 3
 4  <html>
 5    <head><title>Example</title></head>
 6    <body>
 7      <f:view>
 8        <h:form id='exampleForm'>
 9          <h:inputText
10                  id='message'
11                  value='#MessageBean.message'/>
12          <h:commandButton
13                  id='changeMessage'
14                  action='success'
15                  value='Change'>
16        </h:form>
17      </f:view>
18    </body>
19  </html>
```

Lines 1 and 2 merely state that certain libraries of XML tags will be used further on in the file (prefixed with "h" and "f", respectively). The "f:view" tag contains the specification of a whole window, composed of a form ("h.form") which in turn contains a text input box ("h:inputText") and a button ("h:commandButton").

Although the example uses very simple widget components, more advanced components can be used that encapsulate logic usually coded with conditional and looping logic.

Taxonomies can be used to discover previously unknown strategies. It is interesting to note that this taxonomy indicates a lack of examples of template languages with the same basic goal of these page composition variants, but with a syntax that is external to the host markup language.

2.2.9 Discussion

Sometimes it is difficult to draw hard and fast boundaries between presentation and other concerns. For example, one can express how pages will be rendered in a componentised way, using templates, while *also* specifying a mirror image of the same components with programming language counterparts. The template is used to generate pages, but each component in the template is somehow bound to its programming language counterpart. This way, events emanating from a rendered component on the template can automatically be mapped to event handlers on its programming language counterpart.

It is tempting to add another top-level category to the taxonomy called "hybrid models" for such solutions. But the strategy of duplicating the logical component composition structure of a display (specified with a template) by means of a programming language *in order to* be able to handle events in a structured way, is really a controller issue—hence not applicable to this taxonomy.

That said, some frameworks use such a programming language mirror for reasons other than merely handling events—data received from components can be type checked and converted to native data types, and validation can also be specified—the result is components that are not only rendered, but that also have server side state and server side behaviour. (JSF is an example of such a framework—see McClanahan et al. [2004].)

All of the strategies in the taxonomy are really some combination of two main choices:

Intention —the choice of whether the specification uses lower-level control flow-like constructs with which to specify templates, or whether it takes the higher-level approach of composing a page from several (often nested) UI components.

Language —the choice referring to whether the language used to specify the presentation is based on programming language code, or the target markup language.

The foregoing taxonomy lends the second choice more importance for historical reasons, but in reality both dimensions are equally important. Without further explanation, and mostly as an interesting aside, an alternative arrangement is shown in Figure 2.2.

2.3 Strategies for control concerns

Controller concerns deal with routing events between the presentation and the model, keeping these two in sync, while taking care that the model and its presentation stay decoupled. Controller concerns are thus core to the specification of the dynamic behaviour of a UI.

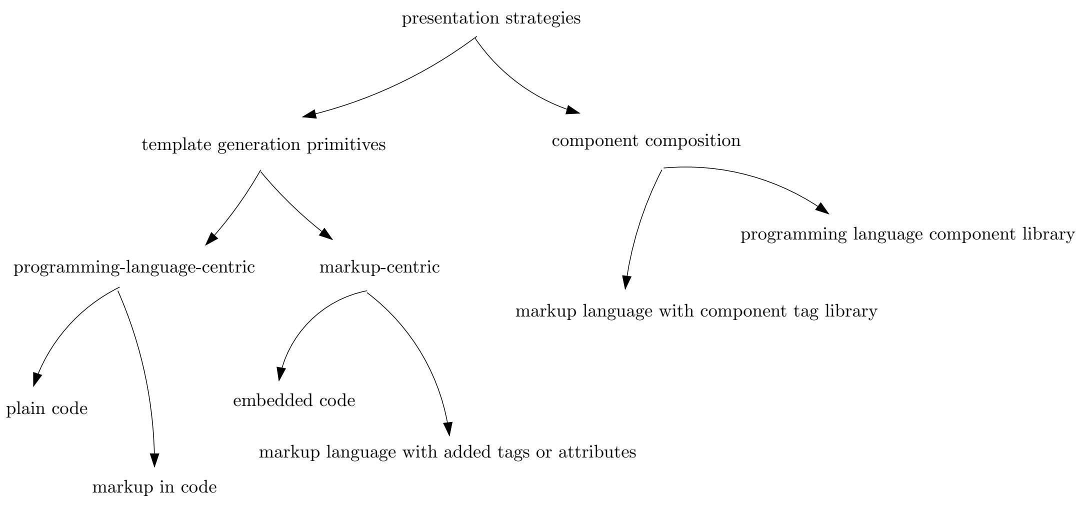
presentation strategies
template generation primitives
component composition
programming-language-centric
markup-centric
programming language component library
markup language with component tag library
plain code
embedded code
markup language with added tags or attributes
markup in code

Figure 2.2: An alternative arrangement

Traditionally, the MVC design pattern (as explained in Section 1.3.2) was especially useful when a GUI program presented a particular model in several different ways simultaneously. For example, a model could be specified without its specifications having to include information regarding the different ways used to present it. Or, once a running presentation of the model has indicated that it should be kept in synch with a model (via the controller), the display (or equivalent) of the presentation component can be updated in response to changes in the model (possibly initiated from other views, or from the model itself).

The client-server nature of a web-based UI is a more restrictive execution environment than what is available to a normal GUI application.

It is assumed that the display in a web browser will only be updated by the browser polling the model via the web server. Hence, controller concerns typically boil down to deciding how to relay events generated in the browser to the model, and how to generate a web page (presentation) in response. Page-flow-centric approaches (Section 2.3.11) expand the scope of controller concerns in an important way to include the relationship between pages in a UI.

Figure 2.3 shown a proposed taxonomy of the strategies, used by web frameworks, for controller concerns. The rest of this section is a discussion of the taxonomy, structured as a particular traversal of each node.

At the top level, the taxonomy distinguishes four broad categories: static files (Section 2.3.1), dynamic content (Section 2.3.2), page-flow-centric (Section 2.3.11), and AJAX (Section 2.3.15).

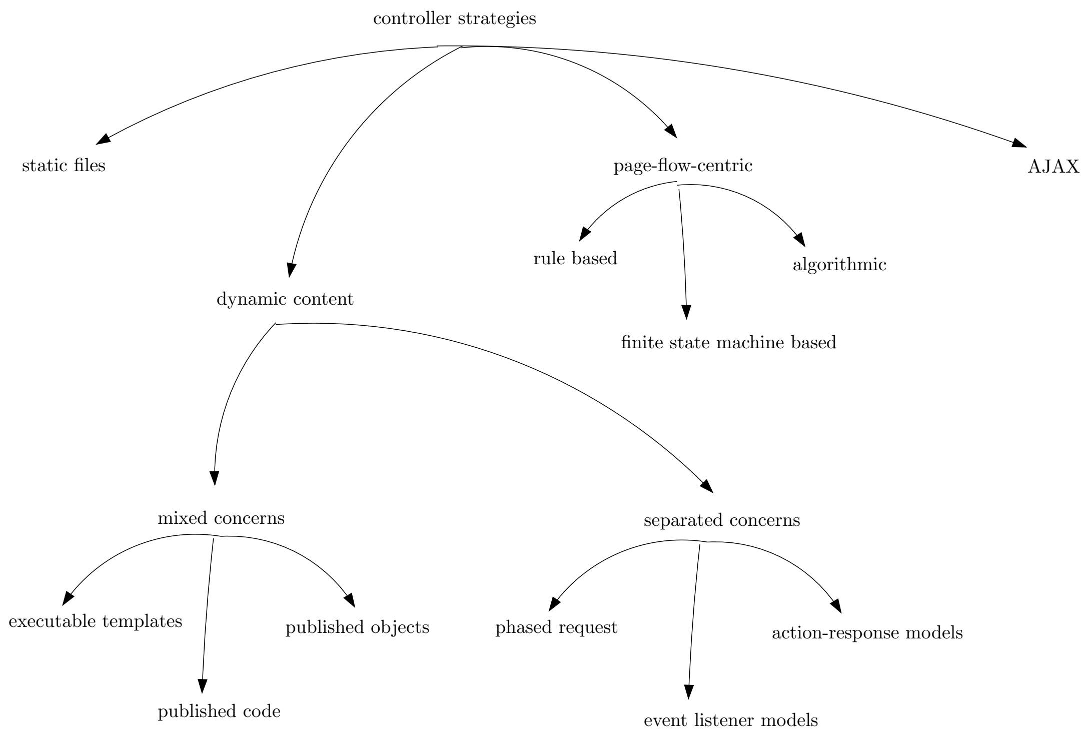

Figure 2.3: A taxonomy of strategies for control concerns

2.3.1 Static files

A traditional web server serving static files is the most elementary strategy for addressing control concerns—to have no dynamic behaviour at all. The use of static files is mentioned here for completeness.

A web server has access to a file system hierarchy containing HTML (and other) files. A URL directly maps to a path in this hierarchy, ultimately leading to a file. Upon an HTTP request, the server merely sends the file denoted by the request URL back to the client browser.

2.3.2 Dynamic content

Most current web frameworks fall into this category. Returning a document in response to an HTTP request is still the focal point. Here, however, the document sent back is generated on the fly, and various ways are used to invoke other code in the process.

The category of dynamic content is divided into approaches that mix MVC concerns (Section 2.3.3) and those that do not (Section 2.3.7).

2.3.3 Mixed concerns

Strategies in this category do not provide formal support for a programmer to separate logic to do with the generation of a dynamic page from logical actions and other code that have to be executed as a result of user-triggered events.

More detailed examples of strategies with mixed concerns are presented in Sections 2.3.4, 2.3.5, and 2.3.6.

2.3.4 Executable templates

Languages like *PHP* and the early versions of JSP [Roth and Pelegrí-Lopart, 2003] are often used in this way. Instead of having static files, a collection of templates can be organised in a file system (or similarly structured schema). Such templates are executable (some interpreted, some compiled), and yield (possibly different) documents as a result of their execution. Upon an HTTP request, the template file denoted by the URL is executed and the resulting document is sent back to the browser in response.

The programmer should specify any other code using facilities provided by the particular template language used, mingled with the presentation logic for which such templates are geared.

2.3.5 Published code

Published code is very closely related to templates that are executed. Here, the URL is mapped to a script or a function written in a programming language. Upon a request, the corresponding code is invoked. Optionally, the script or function can declare a signature specifying arguments with which it should be called. These named arguments would then be extracted automatically from values submitted with the request, and would be passed to the invoked code as normal parameters to the function. (There are several variations on this theme of how the URL, and possibly a form, can be mapped to named and/or positional arguments declared by a function or similar construct.) Such code will then either write text to be returned to a file or return the text as a return value which the web server then can send back to the client.

2.3.6 Published objects

A slightly more structured attempt is for a server to maintain a hierarchy of objects or classes (in the Object Orientation (OO) sense) instead of a file system hierarchy. The URL of an incoming request is then seen as a path along this object hierarchy leading to a particular object. Objects usually have an inheritance hierarchy, but they could also be said to have a containment hierarchy (via attributes or other method). Either of these hierarchies can be used for mapping a request to an object[8].

Depending on the system, a special method of the object (like render()) can then be called by the request.

The URL can also be taken to denote a method of an object in the hierarchy—so that each request results in a method call (as with published code approaches in Section 2.3.5). Again method signatures can aid in specifying and type-casting request parameters for use in the body of the method.

An object may have attributes other than methods and, although these are generally private and thus not directly addressable via a URL, a method so invoked can access these private attributes (some of which could be templates).

By using such an organised collection of methods and templates (and in fact many different kinds of objects), a programmer has access to many tools that are available for structuring the way in which a request is handled and for constructing a reply.

Note that objects have state (per user or application-wide). Thus, having to keep them on a server cause potential conflicts with REST and poses interesting implementation issues pertaining to physical deployment and concurrency of such systems.

2.3.7 Separated concerns

In contrast to these mostly unstructured approaches (Section 2.3.3), attempts have been made to provide for a more structured way to specify presentation and other code separately. Typically, "other code" is either vaguely defined by these approaches as being code that has to do with "logic", with "business logic", or the model or controller in MVC.

This class in our the taxonomy represents such attempts—those that focus primarily on the problem of generating a response to a single request, while keeping concerns separated.

Three sub-categories are discerned in this category of strategies: phased-request approaches (Section 2.3.8), action-response approaches (Section 2.3.9), and event-listener approaches (Section 2.3.10).

[8]For example: *Zope* uses a containment hierarchy, *Cherrypy* uses an inheritance hierarchy.

2.3.8 Phased-request

Some approaches have a well-defined "request processing cycle" during which specific pieces of code can be called at certain stages of handling the request. (For example, on entering the page, or on displaying it.) *Albatross* is an example.

2.3.9 Action-response

Action-response models separately define presentation pages (often called views), controller-specific code and other code. A URL is mapped to specific controller code (Actions in *Struts*, for example). Upon a request, the invoked controller code should in turn invoke model code and then decide, based on the outcome of such an invocation, which view to render in response to the request. Instead of immediately returning a response, such controller code can also instruct the browser to fetch a view from a different URL.

In the Java world, the basic models of "model 2 architectures" [Seshadri, 1999] roughly operate in this way. A model 2 architecture includes Java programming language code (some of which is used to handle controller concerns explicitly), and separate presentation code (typically templates, such as JSP) for rendering HTML pages.

2.3.10 Event-listener

Event-listener models are abundant in the Java world. These provide "listener classes"—classes which a programmer can subclass (or interfaces that can be implemented) to provide custom implementations of special call-back methods. An instance of such a class (or an instance of an implementor of such an interface) can register its interest in events of a particular kind with a UI component, such as a button or selection box. If the UI component detects the occurrence of that kind of event, it will call the appropriate call-back method of the relevant listener instance, and so invoke the custom-supplied event handling code.

In web-based UIs, this means that one needs a programming-language-based model of the components on a page on the server (such as used with component libraries—Section 2.2.4). These should also be maintained in session scope, so that they can maintain their state between different requests from a particular browser.

Incoming requests are then handled in one of two ways:

- The request is seen as the occurrence of a single event (usually a button having been clicked, or JavaScript being triggered by the browser). The event is related to the programming-language-based component to which it logically belongs, and that component is notified that the event took place. The component can then call any listeners for that event in response.

- A whole batch of UI events are inferred from a single request. An example best illustrates the point. Assume a page has two components: a button and a selection box. Upon its initial rendering, the components representing this logical layout can store their initial values. If the user selects a different value in the selection box, the browser does not generate any events to the server. But when the user eventually clicks on the button also provided, it results in the whole form being submitted. Upon receiving this request, the values held by each server-side programming language component can be compared to the corresponding values in the submitted form, and multiple events can be derived from the single request. In this example, the selection box can infer that a "selection changed" event took place, since it can detect that the value submitted as part of the form is different from the stored value previously selected. The button can infer a "button clicked" event. And both events are handled separately by their respective event listeners, but in one batch as a result of the submitted form.

2.3.11 Page-flow-centric

As mentioned in Section 1.3.3, page flow is a term used to describe how different logical locations (or pages) in a UI relate to one another and how a user could traverse them. All approaches discussed so far have left the specification of page flow up to the programmer, leading to the problems discussed in Section 1.3.3—the chaotic way in which one has to specify the relationship between pages, reminiscent of goto statements.

Page-flow-centric approaches recognise that controller concerns include the control of page flow. They have explicit notations for composing a UI from different pages, specifying how these can be traversed by a user.

Page-flow-centric approaches are significant because they allow a programmer to specify a UI in its entirety, as opposed to specifying a UI as a loose collection of individual pages (and other components) with interrelationships that are not made explicit.

Note that, although these approaches all specify page flow explicitly, they tend to deal with a page abstractly as "a logical location in the UI". Generally speaking, they do not map this conception of "logical location" strictly to the URL displayed by the browser.

There are three kinds of page-flow-centric strategies: rule-based, finite state machine-based, and algorithmic approaches. These are discussed in Sections 2.3.12, 2.3.13, and 2.3.14 respectively.

2.3.12 Rule-based

Rule-based approaches (for example JSF—see McClanahan et al. [2004]) have a central set of rules that state, for each location in the UI, all the paths to other locations.

In JSF, such locations are called views. Each view can be visited by a web browser, and upon such a visit, application code is invoked. Such application code should return an "outcome": a string denoting the result of the execution. The rules can then state which outcome leads to which successive view.

This sounds very similar to action-response models (Section 2.3.9). The difference is that the rules here span several requests—when a request is received, the server tracks which view the browser is at, and computes the next view based upon the combination of the current view and the outcome of executing some business code. Maintaining the knowledge which view is currently active is done for the programmer.

2.3.13 FSM-based

Some approaches model page flow as some form of Finite State Machine. The semantics and types of the Finite State Machines (FSMs) vary, but generally states represent either an action to be performed, a decision that has to be made on the server, or a view that has to be presented via the browser. Transitions specify the valid paths in the UI between states (the transition to be followed can be decided on the outcome of a state[9], or a named event that occurred).

Since this dissertation proposes a new approach in this category (Chapter 3), it is useful to give a more detailed example of a close relative: *SpringWebFlow* allows specification of site as a flow—a number of states that are interconnected with transitions. *SpringWebFlow* defines five different kinds of state (listed below). One state in the flow is designated as the starting point for the flow. Upon the application entering any of the states, an action of one kind or another occurs, mostly yielding some sort of outcome for that state. This outcome (or another method) is then used to determine what state to transition to next.

Action states. An action state is a state that, when visited, executes some application code. This code should return a string which is interpreted as an outcome of the execution of the action. This outcome is used to determine which transition should be followed to the next state in the flow.

[9]States representing code executed on the server may be said to have an outcome. When the code finishes executing, it returns a value which is interpreted as an outcome. This is also the time at which the transition for that outcome is triggered.

View states. A view state, when visited initially, merely renders a page to the client browser, causing the UI to pause in the view state, waiting for user input. Button clicks on the rendered page result in a form submittal. During such a form submission, the name of the actual button clicked is used as the outcome to use in determining which transition to take next.

Decision states. A decision state works slightly differently, in that it is not coupled with the concept of an outcome. It merely specifies which state should be transitioned to next, depending on certain conditions.

Sub-flow states. A sub-flow state is a way of nesting another flow inside the current one: transitioning to a sub-flow state means following the flow defined by it, starting from its start state (each flow specifies which of its states is a start state). Sub-flow states facilitate re-use of flows in a number of other "calling" flows.

End states. An end state itself denotes the outcome for a flow as a whole. Should a transition be followed to an end state, the flow is said to terminate with the end state's identifier as a outcome. When used as a sub flow state, this would be the outcome of that sub flow state, used by its parent flow to decide which transition to take next.

Another, different, example in this category is *Expresso*.

2.3.14 Algorithmic approaches

The same sort of specification expressed by an FSM or a set of rules can be expressed in an algorithm using a programming language. Tasks in the Ada language may be regarded as an analogy of what such algorithms will look like. The concept is best illustrated with an example (using the *Cocoon* framework, in the JavaScript programming language):

```
1  function example()
2  {
3    var result;
4
5    cocoon.sendPageAndWait("firstPage.html");
6    result = cocoon.request.get("someFormValue");
7
8    if (result == "a")
9      cocoon.sendPage("aPageForA.html")
10   else
11     cocoon.sendPage("aPageForOther.html")
12 }
```

Cocoon and similar approaches exemplify a programming language concept called "continuations". (The interested reader is referred to Strachey and Wadsworth [2000], Belapurkar [2004]). When the function in this example is executed and it reaches line 5 (sendPageAndWait), the state of the program is saved (in session scope), and the specified page is sent to a client browser. If the browser then submits a form in reply (the user having pushed a button, for example), then the state of the program is restored and processing continues at line 6.

All the powerful semantics of the host programming language constructs are thus available (exception handling, for example) for specifying page flow.

CherryFlow is another such example, using the Python programming language.

It is interesting to note that the fundamental requirement of implementing these approaches is to keep session state information on the server—which is in conflict with REST.

2.3.15 AJAX

Asynchronous JavaScript technology and XML (AJAX) is a technique that does not really represent a full solution (in comparison to the other strategies presented here). But it is a novel technique that will probably give rise to a host of other strategies, hence its inclusion here, and in such a prominent location in the taxonomy. (*DWR* is a framework for using AJAX specifically.)

AJAX refers to the use of JavaScript together with the ability of modern browsers to change an HTML document while it is already being displayed in the browser [Murray, 2005].

Using plain HTML (without JavaScript), events are only sent to the web server when a form is submitted via a button click, or when a link is followed. JavaScript can attach code to other UI elements on a form.

AJAX refers to the technique of using JavaScript to intercept such UI events on the browser, and then making a request to a web server without disturbing the original document being displayed (or the current state of the browser). Upon receipt of the response of this background request, JavaScript is again used to change parts of the contents of the original document—all while it is being displayed.

Such a background request is a request for more information from the web server, not for a complete rendered page (since the page is already rendered, and just needs to be modified). For this purpose, such requests usually request XML documents, which are nothing more than containers for generic information which the AJAX code in the browser is able to make use of. This information can be used to populate a selection box, for example, depending on the current state of another radio button.

2.3.16 Discussion

At a first glance it may be difficult to see the distinction between some page-flow-centric approaches (2.3.11) and those with separated concerns (2.3.7), particularly the action response models (2.3.9). Many of these allow a programmer to specify which "view" should be shown depending on, say, the result (or outcome) of some action. The difference, though sometimes subtle, is important (as indicated by their respective positions in our taxonomy).

To a separated concerns (2.3.7) strategy (particularly the action-response models of 2.3.9), the notion of "view" is basically "the template to be used to generate the resulting document for the current request". At base, these approaches attempt to generate a response for a request, using a template—they just allow the programmer to specify code separately from the template, and provide a means for that code to influence which template will be used (so there is not a strict one-to-one mapping between the template used and the request URL). These approaches are the most sophisticated, youngest descendents in the lineage of: static files (2.3.1), executable templates (2.3.4), separated concerns (2.3.7).

A page-flow-centric strategy (2.3.11) sees a view more strongly as a logical location in the UI and keeps track of where the user currently is: when a request comes in, the server will have some way of knowing where the user is (in terms of its conception of logical location in the UI). Its specification of page flow specifies not which template should be shown in response to a request—it has a wider, higher-level scope: it states which logical location can be transitioned to from others, depending on certain conditions.

While strategies with separated concerns are essentially presented as techniques for specifying web sites with dynamic pages, page-flow-centric approaches are presented from the important perspective of aiming to specify UIs which happen to be delivered via the web.

It is also interesting to note that the strategies presented here as distinct are often combined in a web framework. For example, an implementation of a phased request (2.3.8) can be built by having published objects (2.3.6), each of which is mandated to have methods named for particular phases in the request. The appropriate method of the object denoted by the request URL will then be called during the appropriate phase of the request. When categorising such a combination of strategies, the differentiation is based on the basic intention of the whole combination, not how that intention is achieved.

2.3.17 Notes on implementation

In the taxonomy, the focus is on strategies for *specifying* controller concerns and (by definition of controller in this context) the relationship of controller concerns to other concerns in MVC.

While not discussed, the choices used in various framework projects for executing these specifications are also significant. One issue is, for example, whether the notion of location (or view) in the web UI corresponds with that of the browser. Browsers are built with the assumption that a particular URL denotes a particular location. Standard browser functionality is built on this assumption: book marks can be placed for locations (by storing an URL) and a user can go backwards and forwards along a traversal path of such locations. If an framework implementation is ignorant of the notions of a browser in this respect, it renders such basic browser functionality useless or confusing.

Many frameworks do not accommodate the perspective of the browser (and thus the concepts in REST) very well in their basic models—thus they often have problems resulting from such discrepancies. For instance, the very popular and well-known *Struts* as well as the high-profile JSF do not correlate the “view” shown with the URL displayed by a browser.

The implementation of some strategies also depend on functionality in a client browser that is not well standardised, if at all. This is an indication of the pressure that the use of these strategies place on web standards—an important reason for web framework designers to consider the motivations for constraints embodied by REST. Some strategies need JavaScript[10], some JavaScript with special extensions. AJAX needs support for the XMLHttpRequest object in JavaScript which is not a part of the ECMAScript specification, but

[10]It should be noted that although the W3C does not officially have a standard for JavaScript, it is standardised in the form of ECMAScript, standardised by European Computer Manufacturers Association (ECMA).

supported by many modern mainstream browsers. Even in cases where such functionality is standardised and widely available (like plain vanilla JavaScript), these are always optional additions which can be disabled by users.

It is widely advocated on informal forums that such optional functionalities should be used with care [Tobias, 2004, Korpela, 2002]. In particular, a web site should still be able to function in the absence of such functionalities—albeit with a less rich UI. Implementations of the basic controller strategy of a web UI that are dependent on such optional functionality cannot degrade gracefully in the absence of such functionality—they just cease to be able to operate.

2.4 Summary

In this chapter, an overview is given of the field of web frameworks.

First, requirements which are normally expected of a web framework are enumerated (Section 2.1).

Apart from being responsible for presenting UIs, web frameworks also dictate how such UIs are specified. The rest of the chapter provides an overview of present specification strategies used, in the form of two taxonomies. A taxonomy is proposed for strategies regarding view concerns of MVC (Section 2.2), and another taxonomy is proposed for strategies regarding controller concerns of MVC (Section 2.3).

While an in-depth overview of how frameworks implement the execution of UIs so specified is beyond the scope of this work, a few notable observations in this regard (roughly related to controller concerns) are discussed in Section 2.3.17.

In Chapter 3, a simple language is proposed for specifying the UI of a web application. It focuses on controller concerns and uses a page-flow-centric approach for these. The language is designed so that it is extensible and can be used in conjunction with other approaches to specify presentation concerns, among other things.

3 Harel

In this chapter a small, special-purpose language called Harel is proposed for the specification of web-based UIs.

Harel is used to specify the protocol that dictates the way in which a user and a system will interact. A Harel specification can be extended using other techniques to add more detail regarding a UI—resulting in a specification from which an executable web-based UI can be derived.

Harel concepts and notation naturally map to the concept of page flow (Section 1.3.3). It thus addresses the controller concerns of a UI (in terms of the MVC pattern).

However, before describing Harel in detail, a brief overview is given of the concept of statecharts (on which Harel is based) in their incarnation as UML statechart diagrams. An overview is also presented of the architectural constraints of the web, as set forth by Fielding [2000]. These are contextualised in terms of web applications.

3.1 UML statechart diagrams

Process algebras can be used to describe a dynamic process such as a protocol. FSMs are of particular interest, since they usually have convenient graphical notations that are useful for the particular application proposed in this chapter.

In Harel [1987], David Harel introduced statecharts—which added (among other embellishments) the ability to nest states in a FSM. This allows for breaking up a large, flat specification into a structured specification composed of smaller, manageable parts.

The Object Management Group (OMG) standardised what is mostly an adaptation of Harel's statecharts as part of the UML specification [OMG, 2003] in the form of statechart diagrams. The standardised UML statechart diagrams include constructs whose semantics relate to their use with objects. They are, for example, used to specify the order in which methods of an object can be used—i.e. the protocol external entities should adhere to when calling methods of an object. Traditional statecharts are concerned with a dynamic process reacting to events as signals in global scope, not with the state of individual objects and their parameterised methods. (A detailed discussion of the differences can be found in OMG [2003, part 2, page 169].)

This dissertation is based on the statechart diagrams of UML [OMG, 2003] (thus indirectly on Harel [1987]). But these are applied in an environment which needs to describe an overall dynamic process in the spirit of traditional statecharts.

A detailed understanding of statechart diagrams is not necessary for this work. (The interested reader is referred to OMG [2003] and Harel [1987].) The overview presented here is simplified and of the relevant concepts only.

3.1.1 Basic model

A statechart diagram is a graph with nodes connected by directed arcs. The nodes denote state vertexes, the arcs transitions between them. Usually, state vertexes represent states. (There are also pseudo-states and other constructs which would clutter our brief overview.) A state is defined in OMG [2003, part 3, page 137] as being "a condition during the life of an object or an interaction during which it satisfies some condition, performs some action, or waits for some event".

Three kinds of states are defined:

Composite state A composite state is a state that is refined further by containing nested states in its own, localised diagram. Every statechart diagram is a composite state.

Simple state A simple state is a state that does not contain further sub-states.

Submachine state A submachine state does not add any new semantics—it represents "the *invocation* of a state machine defined elsewhere" [OMG, 2003, part 3, page 125]. As such, it allows one to define several state machines in separate modules, so to speak, and reference them from one another as submachine states.

An event is a significant occurrence, such as a condition that becomes true, or a method being called. Events are atomic and treated as instantaneous. Events may lead to changes in state.

Arcs represent transitions between states. Transitions are usually labelled. A basic label indicates the name of an event. Should an event occur in the current state, and there is a transition labelled for it leaving from the current state, the transition will "fire": the state will be changed to the target state as indicated by the transition.

Figure 3.1 shows a simplified statechart diagram for a traffic light.

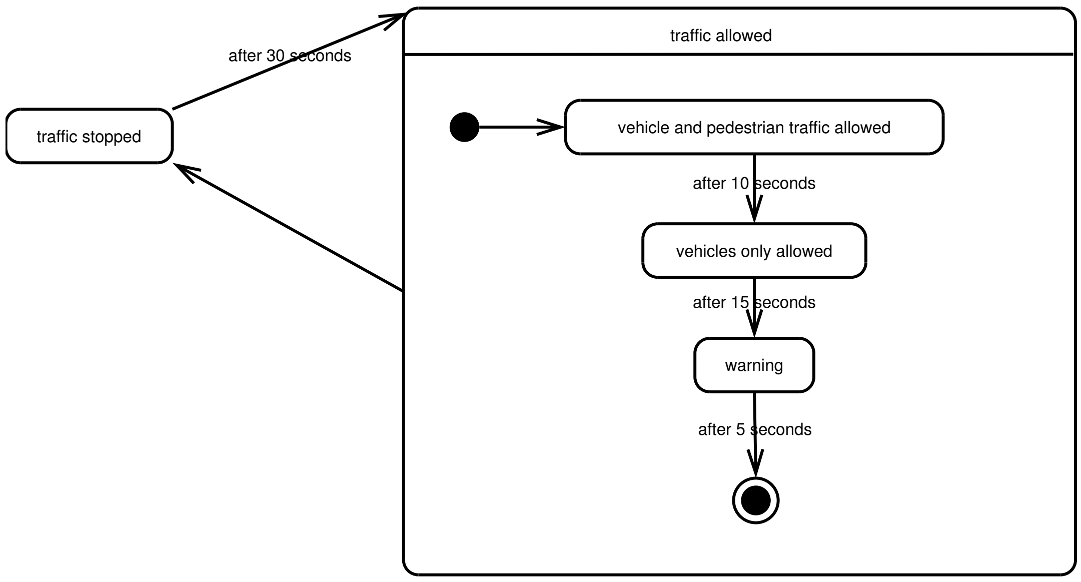

Figure 3.1: A simple statechart for a traffic light

At the top level in the example, there are two states: “traffic stopped” and “traffic allowed”. The former is a simple state, the latter a composite state. The transition going from “traffic stopped” to “traffic allowed” will fire when the event “after 30 seconds” takes place. This is a special event indicating that the a certain time has elapsed since entering the state. Other events could have been specified here, such as “pedestrian button pressed”.

The composite state “traffic allowed” has an initial “pseudo-state”, indicated with a filled black circle. If entered by the transition from “traffic stopped”, the transition from the initial state will immediately be followed. Similarly, the composite state has a final state, indicated with a filled black circle which has another circle around it (a bull’s eye). When the final state is entered, “traffic allowed” terminates and the transition leading from its bounding box back to “traffic stopped” will fire.

The detail of how a composite state is decomposed into sub-states need not be shown as depicted in Figure 3.1. Its decomposition can be hidden. A composite state with hidden decomposition is depicted by a single rounded rectangle with a special icon in the bottom right corner of its second compartment (as shown in Figure 3.2).

3.1.2 Events, guards and actions

According to OMG [2003, part 3, page 142], “An event is a noteworthy occurrence. For practical purposes in state diagrams, it is an occurrence that may trigger a state transition”.

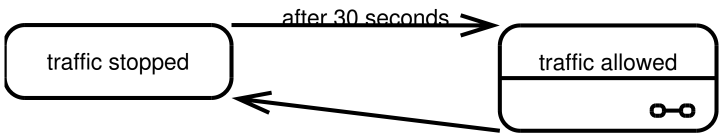

Figure 3.2: Traffic light example with hidden decomposition of the composite state

Statechart diagrams are used to specify dynamic behaviour in response to the occurrence of events, governed by the state of the system. The reaction for a particular event can be specified in different contexts, using the same textual notation: the name of an event, followed by an optional parameter list, followed by an optional guard condition, followed by an action expression. For example:

```
buttonClicked(location) [time < 12:00] / showWindow(location)
```

In the example, the event name is "buttonClicked" and it has a single parameter: "location". UML statechart diagrams closely relate events to method calls, hence the similarity of this syntax to method calls. The guard condition is indicated by its being enclosed in square brackets. An "/" indicates that an action expression follows. The semantics of such a specification varies slightly depending on the context (explained below). Parameters declared with the name of the event are available in the action expression (analogous to a method signature and body).

A textual specification as described can be attached to a transition. Its semantics is then as follows: if the named event occurs and the guard condition evaluates to true, the action expression is executed and a transition is made to the target state of the transition.

A list of "internal transitions" can also be added to states (see Figure 3.3 for the notation). Each internal transition is specified using the textual notation as described above. The semantics in this context is that if the named event occurs in the state, the action expression will be executed if and only if the guard condition evaluates to true. No state transitions are made.

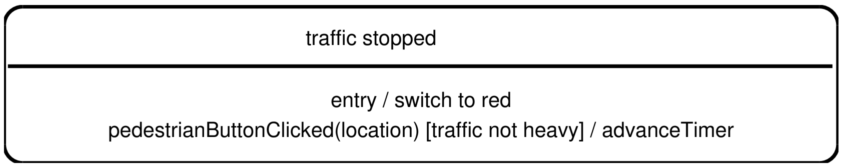
traffic stopped
entry / switch to red
pedestrianButtonClicked(location) [traffic not heavy] / advanceTimer

Figure 3.3: A state with internal transitions

The textual specification of an internal transition is said to consist of an action label and an action. The action label specifies the circumstances under which the action is invoked (the event name with its parameters and the guard condition). Several event names are reserved and have special meaning in this context.

For example, "entry" actions are executed each time the state is entered via a transition (including transitions to self, but excluding internal transitions). Similarly, "exit" actions are executed when the state is exited via a transition.

3.1.3 Further details

UML embroiders this basic model extensively. For example, it adds the concept of concurrent sub-states: composite states can be partitioned into several concurrent parts so that each part is a composite state, but one state in each concurrent part is active at the same time. Events are also more richly defined than portrayed in our overview—there are different kinds of events (only one of which is shown here). There are a number of finer details with regard to semantics which are carefully defined. For instance, a particular sub-state's being active is really a refinement of the fact that its parent state is active—implying, for example, that while in the sub-state, events may trigger transitions specified on the parent.

Most of these richer semantics are not needed in the context of web UIs. Although some of them can definitely be argued as relevant, most of them are excluded in this overview—because the aim is to present and implement the most basic model which draws on these ideas, but which solves a problem in a very different context. The more complete statechart diagram semantics and notation can be drawn on in future, though, based on experience gained in practice.

3.2 REST

As stated previously, the web is intended to be a hypermedia network of information [Berners-Lee, 1996]. Web application authors have a different agenda: they aim to utilise the widely deployed and standardised infrastructure of the web in order to deliver the UIs of systems.

As we all know, the web has come a long way since its inception. It is important to acknowledge the architectural properties that aided this spectacular success story when building on top of it. In order to understand the intended nature of the host of web applications, a short overview is given of the architecture of the web (and the motivation for its being the way it is).

A good account of this is given in Fielding [2000] where this knowledge is formalised as an "architectural style" called REST (introduced in Section 1.3.1). REST is also used to defend the web standards and guide their further development.

3.2.1 Basic model

A system adhering to the model put forth by REST has several communicating components. These communicate with each other via connectors, using a simple, uniform interface. Chief among the components are the origin server and the user agent.

The interface between components follows a simple, stateless request-response protocol. A user agent initiates a request which is sent to the origin server. From the point of view of a single request, the request flows though a pipeline of components (connected to each other by means of connectors). Upon receipt of the request, the origin server sends a message in response.

Central to REST is the notion of a resource, its representation, and its identifier. Naïvely put, a resource is a named piece of information—or a named concept. (Its value may change over time, but the concept it expresses does not change.) Requests indicate operations on resources (the same small set of operations being supported for all resources) and, in response to a request, a representation of the resource in question is returned. The intended effect of sending a representation of a resource in response is that the onus of maintaining the current state of a conversation is removed from the origin server and instead becomes the responsibility of the user agent. From an architectural point of view, each request-response interaction is independent of previous ones (and of the order of requests) and each message contains all the information necessary to understand it.

These features are geared towards making components insensitive to contextual and other information that may be difficult to provide on a large, distributed scale. Components thus have a lot of freedom when it comes to how they can improve scalability, performance, provide tunnels through firewalls, and the like.

3.2.2 REST constraints (or architectural decisions)

Any architectural decision is a trade-off decision, made for a reason. Many of the constraints of REST are geared to allow properties of the web architecture such as: scalability, caching, and tunneling through firewalls. In this section, the architectural constraints of REST are briefly paraphrased from Fielding [2000], together with their trade-offs. (The reader interested in clear definitions of the properties mentioned is referred to Fielding [2000, p28, Section 2.3].)

Client-server

By being a client-server system, the web separates the concerns of a UI from other concerns, giving the client the responsibility for the UI and thus also improving portability of the UI. Note that UI here means presenting the representation of a single resource to a user—not presenting the UI of a web application to a user.

Statelessness

Communication is stateless—each request contains all the information necessary to understand it and is not allowed to take advantage of any stored context on the server. The state of the client's session with the server is kept entirely on the client. The statelessness of REST improves visibility[1], reliability and scalability. It is payed for, however, in increased overheads (state that is not maintained on the server is sent to and fro from the client). The application also becomes dependent on the correct implementation of multiple user agents.

Ability to cache

Responses are labelled as being cacheable or not. This allows an intermediary component or a user agent to re-use a response for similar requests. Cacheable responses improve efficiency, scalability and user-percieved performance at the cost of decreased reliability in the event of stale cache data.

Uniform interface

All components in REST have a uniform interface: they use a standard for identifying resources, define a small, uniform set of operations allowing the manipulation of resources via their representations, use self-descriptive messages, and use hypermedia as the engine of application state. This uniform interface simplifies the overall architecture and improves visibility and independent evolveability. Efficiency is decreased, since information is transferred in a standardised form, instead of a form tailored to a specific application. It is important to note that the interface is designed to be efficient for large-grain hypermedia transfer, thus optimising for the common case of the web.

A layered system

Components do not have access beyond the immediate systems with which they interact. This layering brings all the benefits of encapsulation and allows for the introduction of intermediaries that can provide a wide range of functionality (for example, load balancing or

[1]According to Fielding [2000, p36], "Visibility in this case refers to the ability of a component to monitor or mediate the interaction between two other components".

enforcement of security constraints). These benefits are traded for increased latency and thus decreased user-percieved performance.

Code on demand

Clients can extend their standardised functionality in non-standard ways by downloading and executing code from the server. This increases extensibility, but at the cost of visibility—a cost considerable enough to make this constraint optional. The intention of such an optional constraint is that a smaller subset of the system (such as within the boundaries of a single corporate realm) could allow code on demand and so gain its advantages locally by paying its cost locally.

3.2.3 Notes relating to web applications

It is interesting to relate each of the constraints of REST to current practices in the building of web applications.

Most importantly, it is precisely the success of the separated UI of the web and its wide availability in the form of standardised user agents that makes it so enticing as a host for web applications.

The stateless nature of the web, and its intention of keeping application (or conversation) state off the server and in the user agent has generally been ignored by web applications. This is mostly seen as a hindrance and every web application framework provides one way or another for keeping session state on the server. In fact, we have listed this as one of the perceived requirements for a web framework (Section 2.1.6). Web application frameworks have to provide complicated infrastructure in order to be able to scale when providing this functionality which goes against the grain of the web architecture.

Most often it does not make sense to cache a response in request to a web application, since the request is not independent of the context held on the server and because of the high change velocity of resources published by web applications.

Layering allows for components like proxies and gateways which are instrumental in allowing a web application to be hosted in a protected realm behind one or more firewalls.

Many properties of REST conspire to result in a relatively high latency of messages sent between user agent and origin server. Web applications relate to this in terms of the user-perceived responsiveness of their UIs. When considering the topic of user-perceived responsiveness, it is wise to bear in mind that the interface between REST components has been specifically optimised for the transfer of large-grain hypertext messages. And several options are provided in line with this decision to improve latency and user-percieved performance.

Retrofitting a conflicting model on top of the basic web protocols often results in unnecessary complexity.

The code-on-demand constraint is actually a strange member of the REST family. It is an optional constraint because it undermines one of the very basic properties that so much of REST strives to protect: visibility. This raises a number of questions: has it perhaps been retrofitted and incorporated precisely because of pressure from the camp of the web application to provide a richer user experience? And how should a web framework handle its presence, seeing that it is optional?

3.3 Harel specified

A UI relays an application-specific protocol governing the conversation between a user and a system: a GUI window, for instance, is a representation of the current state of such a conversation: by means of what it displays and the (visually depicted) state of the widgets on it, it lays down the rules of what the user can do next to change the state of the current conversation (i.e. valid methods in which the user can continue the conversation from its current state). In like manner, a page that is being rendered by a web browser is also a representation of the current state of the conversation—intentionally so, according to Fielding [2000, page 103]

Users often talk of “being in a specific location” in a UI, particularly with regard to the web. It is also common to talk of moving around from location to location in a UI. In fact, so natural is the metaphor, that it is common and valuable to draw a map of it. This metaphor has a natural mapping to a state-intensive protocol.

A graphical language, Harel, is proposed, with which a UI can be specified in terms of this common metaphor. Harel’s semantics is defined in terms of the protocol that should be followed during a conversation between a user and a system.

The notation used by Harel is a vast simplification of UML statechart diagrams. Harel’s semantics will be seen below to be a subset of the intersection of statechart and REST semantics.

The initial aim with Harel is to provide the simplest language based on REST and statechart diagrams, together with a framework implementation which can present a UI via the web according to a Harel specification. (The design of a framework is proposed in Chapter 4.) Such a framework can thus be used to test the specification technique in practice. A great deal of the richer notation and semantics defined as part of the UML statechart diagram specification is thus excluded.

Implementing rich semantics in the distributed environment of the web also poses interesting challenges, especially if the intention is to stick to the principles set forth by REST. Thus, implementation constraints also preclude adding unnecessary rich notation and semantics.

The richer semantics excluded now can be drawn upon later as needed, guided by practical experience and applicability.

The abstract syntax and semantics of Harel are discussed next (Section 3.3.1), followed by a comparison to UML statechart diagram semantics in Section 3.3.2. A concrete syntax is proposed in Section 3.3.3, followed by an example specification (Section 3.3.4).

3.3.1 Abstract syntax and semantics

The example of OMG [2003, page 2-142, Figure 2-25] is now followed by depicting the abstract syntax of Harel graphically in Figure 3.4 as a UML class diagram, followed by a more detailed discussion.

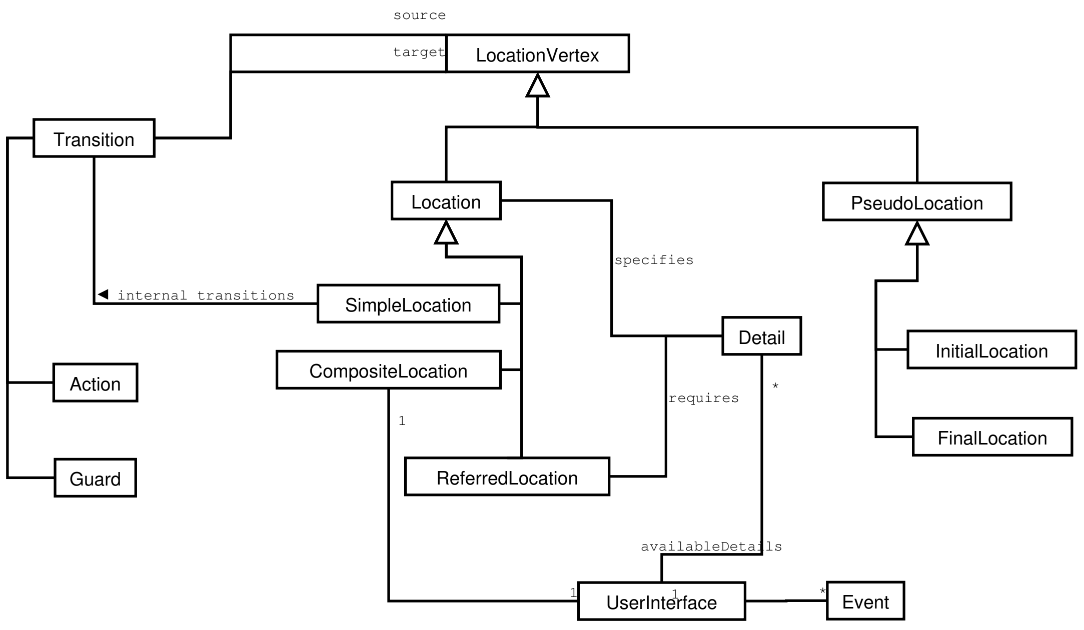

Figure 3.4: Harel abstract syntax

A Harel specification of a UI is structured around the specification of the protocol followed during conversation between a user and a system. This implies that it takes a controller-concern-centric (in terms of MVC) approach to defining a UI as a whole. The specification technique will also be shown to be page-flow-centric (Section 2.3.11).

There is a limit to the fineness of granularity at which Harel allows specification. Details have to be provided by other tools. For example, many solutions are available for the presentation concerns (view). Harel is also assumed to be used in conjunction with a general-purpose programming language (referred to further on as the host programming language).

3.3.1.1 Basic model

UserInterface
A UserInterface is the specification of a state-intensive protocol to be followed during a conversation between a user and a system. A UserInterface is specified by a single top-level CompositeLocation.

Event
Events are signals triggered by the user. A UserInterface should allow the user to trigger Events and should itself respond to such Events meaningfully (i.e. as per its specification). An Event is triggered by an operation on a resource in REST terms.

Location
A Location in a UserInterface represents the current state of the protocol between user and system.

SimpleLocation
The finest granularity at which Harel specifies state explicitly is a SimpleLocation. A SimpleLocation represents a state that does not contain any further sub-states (a simple state).

CompositeLocation
A CompositeLocation is a coarse-grained state of a conversation, specified further in terms of other Locations. It is a structured collection of other locations together with a specification of how the occurrence of Events will cause transitions between sub-Locations. A CompositeLocation can contain any other kind of Location. In other words, a CompositeLocation maps to a composite state in a statechart diagram.

Transition
Transitions specify the reactions to Events that occur in a particular Location. A Transition (except when it is an internal Transition, discussed later) has a source LocationVertex, and a target LocationVertex and usually is labelled for a specific Event name. When that Event occurs while the UserInterface is in the source Location of the Transition, the Transition may be triggered. A Transition is said to "fire" when successfully triggered. Firing a transition means to change the state of the conversation to the target location (or state) indicated by the Transition.

Guard

Guards provide finer-grained control over when transitions may fire. A Guard is an expression in the host programming language which can be associated with a Transition. The result of the expression is interpreted as a boolean, hence amounting to a conditional test. When the occurrence of an Event triggers a Transition, its Guard (if any) is first evaluated. If it evaluates to true, the Transition will fire, otherwise not. Should more than one Transition be triggered, and more than one of their Guards allow them to be fired, one is chosen nondeterministically to fire.

Action

Actions provide a means by which model code (in terms of MVC) is invoked. An Action is a statement in the host programming language. An Action can also be associated with a Transition. When (and only if) a Transition fires, its Action (if any) is executed.

Internal Transitions

A SimpleLocation may have Internal Transitions. An Internal Transition is a Transition (optionally with its Guard and Action) which does not lead to a change of state. Some event names are reserved for use in this context to indicate the occurrence of special conditions related to the SimpleLocation. At this stage, one name is reserved:

render In REST terms, a resource can be asked for its current default representation, without specifying any actions on the resource. Should this happen, the render event is said to occur.

PseudoLocation

Both InitialLocation and FinalLocation are PseudoLocations. PseudoLocations have a special meaning in a CompositeLocation. Apart from their special semantics, PseudoLocations are mostly used in a specification as normal Locations are: Transitions may use them as source or target.

A CompositeLocation always contains a single InitialLocation and a single FinalLocation.

InitialLocation

Only one Transition is allowed from an InitialLocation. This Transition is a wildcard Transition denoting any Transition to the containing CompositeLocation. Any Transition to the containing CompositeLocation thus implicitly targets the sub-Location indicated by this "initial transition".

No Transitions are allowed with an InitialLocation as target.

FinalLocation

Any number of Transitions are allowed to target the FinalLocation of a CompositeLocation, but no Transitions are allowed to leave from it. A Transition to the FinalLocation of a CompositeLocation indicates that the CompositeLocation is completed. Upon completion of a CompositeLocation, a Transition on the enclosing CompositeLocation, leading from the completed CompositeLocation should fire.

ReferredLocation

A ReferredLocation maps to a submachine state in UML statechart diagrams. It serves as an invocation of the top level CompositeLocation of a UserInterface which is defined elsewhere. ReferredLocations facilitate re-use, whereas CompositeLocations cannot be re-used—they are only used to improve the structure of a large diagram.

Analogy with structured programming

An analogy with structured programming is apt to explain the basic model presented so far.

A SimpleLocation is analogous to a statement, whereas a CompositeLocation is akin to a subroutine. Transitioning to a CompositeLocation (and by implication a ReferredLocation) then corresponds to the calling of a subroutine. Such a call is independent of the details in the subroutine body, and the subroutine body is independent of the details of the context within which it may be called. Calling a subroutine amounts to starting execution at the first statement of the subroutine body. A CompositeLocation is similarly entered without reference to its internal details (its body). The initial Transition of a CompositeLocation indicates its starting Location without reference to the context from which it may be entered—the starting Location so indicated is akin to the first statement in the body of a subroutine. The FinalLocation of a CompositeLocation is analogous to a return statement in a subroutine. When a subroutine returns to its caller, execution is resumed at the statement following its invocation. When a CompositeLocation exits, the protocol resumes at the Location following the exited CompositeLocation as indicated by a Transition leading from it.

Mapping to REST

A Location maps to a resource in REST terms (which implies that each Location has a REST identifier that uniquely denotes it in its UserInterface).

A SimpleLocation, in particular, maps to a REST resource whose representation is a hypertext document which, when rendered by a web browser (for example) would present the current state of the user's conversation with the system accurately and enable the user to continue the conversation according to the specification. In order for the user to continue the conversation, the representation of the SimpleLocation should allow the user to initiate

Events for which responses are defined in the current SimpleLocation. Note that over time, different representations of the same SimpleLocation could vary (because of fine-grained variations in state), but the conceptual location (or coarse-grained state) which a particular SimpleLocation denotes should not vary.

PseudoLocations are not mapped to REST resources like Locations are.

3.3.1.2 Notes

Exceptions to Transition semantics

As mentioned, Transition semantics is influenced at the boundary to a CompositeLocation where FinalLocations and InitialLocations come into play. This has an impact on how much of a Transition needs to be specified, depending on the context in which it is used.

Labelling an initial Transition (the Transition leading from an InitialLocation) for example, is not meaningful, since it denotes any Transition to the containing CompositeLocation. Labelling a Transition leading from a CompositeLocation is also not meaningful, since it is triggered upon completion of its source (no matter what Event triggered the Transition causing its completion).

In these special circumstances where Transitions should not be labelled, they are also not allowed Guards or Actions.

Should more than one Transition lead from a CompositeLocation, one of them will be chosen nondeterministically upon completion of the CompositeLocation.

The scope of Events

The scope of Events upon which the UserInterface will react in any given state is restricted to those Events for which Transitions are labelled, leaving from the current SimpleLocation. (This is a significant restriction of statechart diagram semantics, discussed further in Section 3.3.2.)

REST identifiers

Locations map to REST resources—each of which needs to have an identifier. A particular Location has an identifier that denotes it uniquely in its defining UserInterface. Since each message in a protocol adhering to REST is addressed to a particular REST resource, each message indicates the current state of the conversation between the user and the system—without the need for storing session state on the server.

ReferredLocations have to be handled with care, though. A UserInterface can be composed from several smaller units of UserInterface by the use of ReferredLocations. If the same

Location is referred to by means of more than one ReferredLocation in the same UserInterface, each such referenced instance of that Location is seen as mapping to a distinct REST resource in its own right, with its own unique identifier—because each referenced instance of the Location represents a distinct state in the conversation. (The discussion as part of the proposed framework in Section 4.3.2.3 serves as a motivating example.)

Context

From the moment the occurrence of an Event is detected, a context is created that is shared by all Actions and Guards. This context is temporary and ceases to exist just before a new Location is entered (but after all Actions on the firing Transition have completed). The context is a dictionary which maps variable names to their values. Variable names in the context are available for use in Guards and Actions. An Action can introduce a new variable into the context simply by assigning to the (possibly new) name[2].

Exceptions

Generally, if an exception is raised in the host programming language during execution of an Action or Guard, it is taken as a serious error. A special type of exception (a UIException) is handled differently.

A UIException can be raised inside an Action to indicate an invalid condition regarding the event that was signalled by the user. Raising a UIException halts execution of the Action where it is raised. The UserInterface does not transition to a new Location. Instead, it stays at the Location where the fated event occurred and includes the exception instance as part of the fine-grained detail of the current state of the UserInterface. This information can be used by a presentation system to let the user know of the exception, and allow the user to signal a different event (or supply more information).

3.3.1.3 Extending the basic model

The language discussed so far does not include sufficient detail to make it executable. For this purpose it is necessary to add more detail to a Harel specification than allowed by the model introduced thus far. These missing details more often than not fall outside the main concerns of a Harel specification (controller concerns), but need to latch on to the framework provided by Harel.

A generic, extensible way is provided by which more details can be added to a Harel specification.

[2]This is analogous to how variables are declared by assignment in the Python programming language.

A DetailAttribute is defined as a type in Harel. Values of this type represent a chunk of detailed specification—using a notation and semantics undefined by Harel. It is assumed that the necessary information about a specific DetailAttribute type will be provided by a module (possibly provided by an external tool). Such a module should provide the syntax, semantics and implementation that go with that specific DetailAttribute type.

The concept of a type of foreign specification has been introduced and the suggestion made that it can have a value. In order to incorporate such a type into the model introduced so far, two concepts from OO are borrowed (with slight modifications): class attributes and inheritance.

Statecharts are closely related to OO. States and sub-states form a hierarchy (of containment). Sub-states are refinements of their super-states, and they inherit behaviour from their super-states.

Thus far, Locations have behaviour, but no data—and the inheritance of behaviour has been specifically prohibited, contrary to the richer semantics of UML statechart diagrams.

A DetailAttribute is attached to a Location analogously to how a class attribute is attached to a class in OO. A named, typed variable can be defined for a Location. The type of these variables specifies the type of DetailAttribute, the values are chunks of additional specification (which are opaque to Harel but defined by the type).

As with inheritance in OO, the DetailAttributes defined in a Location are inherited by sub-Locations lower down in the containment hierarchy. A DetailAttribute so specified may also be overridden lower down in the hierarchy by specifying a new value for a name already used higher up in the hierarchy.

Note that ReferredLocations need special handling.

The hierarchy specified by SimpleLocations and CompositeLocations alone is a static specification analogous to static inheritance hierarchies in OO. A ReferredLocation, though, allows such a static hierarchy (defined elsewhere) to be grafted onto many other different hierarchies. A ReferredLocation contains information about the context from which it refers, but the UserInterface to which it refers is independent of the (possibly many) referring context(s).

Hence, a ReferredLocation cannot blindly inherit DetailAttributes like a CompositeLocation.

Each UserInterface includes the names and types of DetailAttributes it expects to inherit from a possible referring context, similar to a subroutine defining a formal parameter list. These are made available to be inherited by the top level CompositeLocation of the UserInterface.

A ReferredLocation includes a specification of how the names present in its containing

Location map to the expected names as defined by the UserInterface it refers to—similar to a call to a subroutine which provides actual parameters to a formal argument list.

3.3.2 Comparison with UML statechart diagrams

As mentioned before, the initial aim with Harel is to provide the simplest language based on REST and statechart diagrams, together with a framework implementation that can be tested in practice. A lot of the richer notation and semantics defined as part of the UML statechart diagram specification are therefore excluded.

In order to provide a more complete picture of the relationship between Harel and statecharts, some of the more pertinent differences between the two are now highlighted.

According to statechart diagrams, if a sub-state is active, the state it is contained in is implicitly also active. The sub-state's being active is in effect a further refinement of the parent state's being active. This means that transitions leading from a composite state are relevant for all of its sub-states too—should an event happen while in a sub-state of the composite, and a transition departing from its containing composite state is labelled for that event, that transition will fire. (This is how behaviour of super-states is inherited by sub-states.)

In Harel, some of the same semantics applies for Locations: being in a sub-Location of a CompositeLocation amounts to being in the CompositeLocation. However, the scope of events that have defined reactions is restricted to Transitions (internal and external) related to the current SimpleLocation only. As a consequence of this, SimpleLocations are the only Locations for which internal Transitions can be specified.

Events in Harel lean more towards traditional Harel statechart events in that they are not parameterised. An Event, being triggered as an operation on a resource in REST terms, is accompanied by the representation of a resource (such as a form submit). Although not specified anywhere, the information accompanying the occurrence of an event is required to be available to Guards and Actions in Harel, just as event parameters are available to the action expressions of statechart diagrams. Exactly how this information is made available to Actions and Guards is left to depend on a specific implementation.

According to UML the handling of some events may be deferred. In Harel, this is excluded, since the event loop cannot be easily controlled in a distributed, REST environment—any execution happens as a direct result of some user action on a user agent such as a web browser. The origin server cannot initiate events and cannot keep track of the state of the current conversation.

In statechart diagrams, the name of a state is optional; in Harel the name of a Location is required. In statechart diagrams, it is undesirable to specify the same state twice on a single diagram; in Harel it is not allowed at all. (It also follows that two distinct states with the same name in the same CompositeLocation are not allowed either.)

With UML statecharts, the transition from an initial pseudostate can be labelled for the event that creates the object (whose statechart the state depicts). In Harel such a label (or Guards and Actions) will be ignored if specified. The same is true for Transitions leaving CompositeLocations.

Whereas UML statechart diagrams allow a Transition's target and source state to be the same state, this is not allowed in Harel for the simple reason that accommodating such Transitions in a GUI editor for the graphical notation (defined later) is more difficult. Much the same effect can be obtained by using Internal Transitions.

Harel only defines one type of Event; UML statechart diagrams discern between several different event types (such as conditions becoming true).

The basic structure of the abstract syntax of Harel differs from that of UML statechart diagrams in subtle ways (see OMG [2003, page 2-142, Figure 2-25]):

- UML statechart diagrams define a richer hierarchy of StateVertixes, especially PseudoStates.
- In UML statechart diagrams, SubmachineState is defined as being a kind of CompositeState, in Harel ReferredLocation is completely distinct from CompositeLocation.
- In Harel, both FinalLocation and InitialLocation are seen as PseudoLocations; in UML a FinalState is a State, and an InitialState is one of several types of PseudoState.

UML also defines some composite states to be concurrent—a concept excluded in Harel.

Whereas UML statechart diagrams specify behaviour only for a state, Harel allows both behaviour as well as data—the latter in the form of DetailAttributes.

In UML statechart diagrams, a submachine state is seen as a kind of a macro expansion [OMG, 2003, part 2, page 148]. This implies that behaviour is inherited across this boundary uninhibited, leading to the strange side-effect that a state machine can inherit different behaviour in each different state it is referred to (or, more correctly, included and expanded in). In Harel, the inheritance of DetailAttributes is controlled across this boundary in the same manner as parameter passing is controlled between a subroutine and a call to it.

UML also defines the concept of chained transitions. A chained transition can, for example, be constructed by chaining together the transition leading to the final state of a composite state and the transition leading from that composite state on its containing diagram. The individual transitions in a chain could have, for example, actions specified which would all be executed in order if the chain fires. UML needs this for other sophisticated elements in its abstract syntax related to its numerous pseudo-states. This is not necessary in Harel, and thus excluded—resulting in the prohibition of Guards, Actions and event labels on Transitions leading from InitialLocations or CompositeLocations.

In the next section, a concrete syntax for Harel is proposed.

3.3.3 A concrete syntax based on UML

UML statechart diagram notation can be used as basis for a concrete syntax for Harel. Such a graphical notation is useful in UI design.

The details of a CompositeLocation is shown as a graph, with nodes that are connected by directed arcs. The nodes represent various Locations and LocationVertexes, the arcs represent Transitions between them.

On such a graph, all Locations are shown as rounded rectangles with one or more compartments. The compartments are separated by horizontal lines. The following compartments are available:

Name compartment The name compartment simply contains the name of the Location.

Internal transitions compartment A compartment for specifying a list of internal Transitions (textually) for a SimpleState.

DetailAttributes compartment Should a Location have DetailAttributes attached, they are specified here, similar to internal Transitions.

Sub-structure compartment A compartment for indicating when a Location has further internal structure.

Referred compartment A compartment used by ReferredLocations for stating which UI it refers to, and how actual DetailAttributes map to formal DetailAttributes.

A SimpleLocation is shown as a rounded rectangle with the following compartments: a name compartment, a DetailsAttribute compartment, and an internal transitions compartment.

Figure 3.5 shows an example.

The list of internal Transitions is a number of text lines (one per Transition) in the internal transition compartment. Each line has the form:

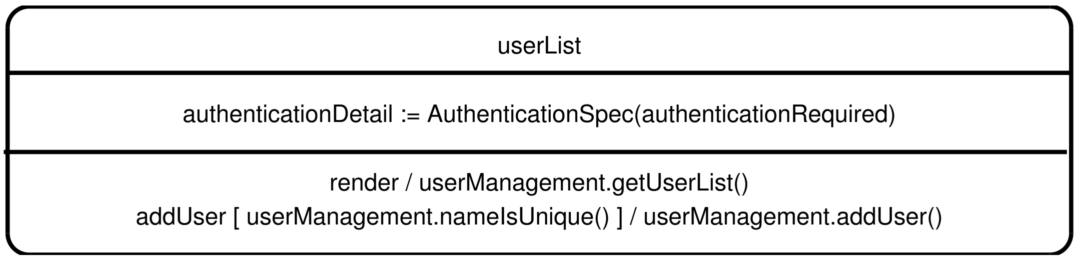
userList
authenticationDetail := AuthenticationSpec(authenticationRequired)
render / userManagement.getUserList()
addUser [userManagement.nameIsUnique()] / userManagement.addUser()

Figure 3.5: A SimpleLocation

```
<event name> [ <guard> ] / <action>
```

Where:

<event name> is a text string denoting the name of the event for which the Transition is labelled;

<guard> is an optional boolean expression in the host programming language, delimited by "[" and "]" if it is present; and

<action> is an optional statement in the host programming language, preceded by "/" if present.

DetailAttributes are also shown as lines of text, one per line, in the DetailAttributes compartment. A DetailAttribute line has the form:

```
<variable name> := <detail attribute type>(<specification summary>)
```

Where:

<variable name> is the name of the DetailAttribute instance attached here;

<detail attribute type> is the type of DetailAttribute; and

<specification summary> is some textual summary of the value of the DetailAttribute. As Harel is a graphical language, it is assumed that a graphical tool will provide a way for a user to edit the foreign specification somewhere else—this short summary is merely a convenient visual summary of it.

A CompositeLocation is represented by a rounded rectangle with a name compartment, a DetailAttributes compartment, and a sub-structure compartment. The DetailAttributes compartment looks similar to that for a SimpleLocation; the sub-structure compartment is an empty compartment with a small symbol in its bottom-right corner—the symbol used with UML composite states to show that their decomposition is hidden.

Figure 3.6 is an example (note that there are no DetailAttributes specified in it).

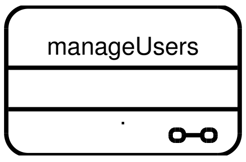

Figure 3.6: A CompositeLocation

The figure for a ReferredLocation looks like one for a CompositeLocation (including the decomposition icon), but it has a referred compartment. The referred compartment starts with a line of text indicating which UI the ReferredLocation refers to.

The text line has the form:

```
include / <included UI>
```

Where:

include is a special keyword; and

<**included UI**> is the file name (relative to the current UI) of the UI referenced.

Subsequent lines in the referred compartment indicate how DetailAttributes in the scope in which the ReferredLocation is placed (actual DetailAttributes) map to the DetailAttributes expected by the UI referred to (formal DetailAttributes). These lines have the form:

```
<formal detail attribute> := <actual detail attribute>
```

Where:

<**formal detail attribute**> is the name of a formal DetailAttribute expected by the UI referred to; and

<**actual detail attribute**> is the name of a DetailAttribute in the current scope which will be inherited by the UI referred to as indicated by the formal DetailAttribute.

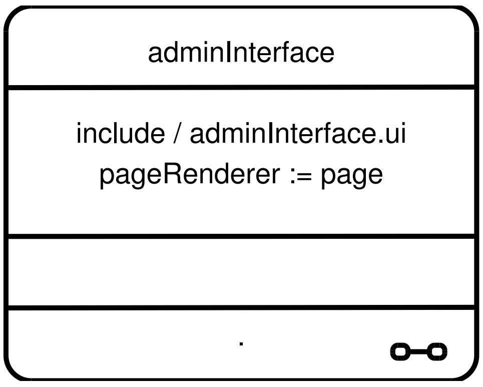

Figure 3.7: A ReferredLocation

Figure 3.7 shows an example (again with an empty DetailAttribute compartment).

An InitialLocation is denoted by solid black circle, and a FinalLocation by a circle surrounding a solid black circle (a bull's eye).

Transitions between Locations are denoted by directed arcs with arrows pointing to the target Location. The arcs can have textual labels using exactly the same syntax as internal Transitions in a internal Transition compartment, as discussed above.

3.3.4 Example

In the official J2EE tutorials [Armstrong et al., 2005] from Sun MicroSystems™ (Sun), a simple and well-known example application is used, called "Duke's Bookstore".

The Duke's Bookstore application is a simple online bookstore that allows a user to browse a catalogue of books, and add books to a shopping cart. At some stage during this process, the user can proceed to a checkout counter where the accumulated books in the shopping cart can be paid for.

As an illustration of Harel notation, a version of the Duke's Bookstore example follows. The example is not complete—it is merely sufficient to illustrate the notation introduced in this chapter.

Figure 3.8 shows the top level composite location representing the Duke's Bookstore UI. It contains a composite location called "checkout", which is shown in Figure 3.9.

If a user visits the bookstore, she will enter at "bookstore", since it is indicated as the starting location for the top level composite location of the entire UI. Upon entering here, "catalogue.getPromoDetails" is called, which fetches all the necessary information for dynamically generating a web page. This page displays some information about a book which is on promotion. The page provides a link to an URL where the book's details may be viewed.

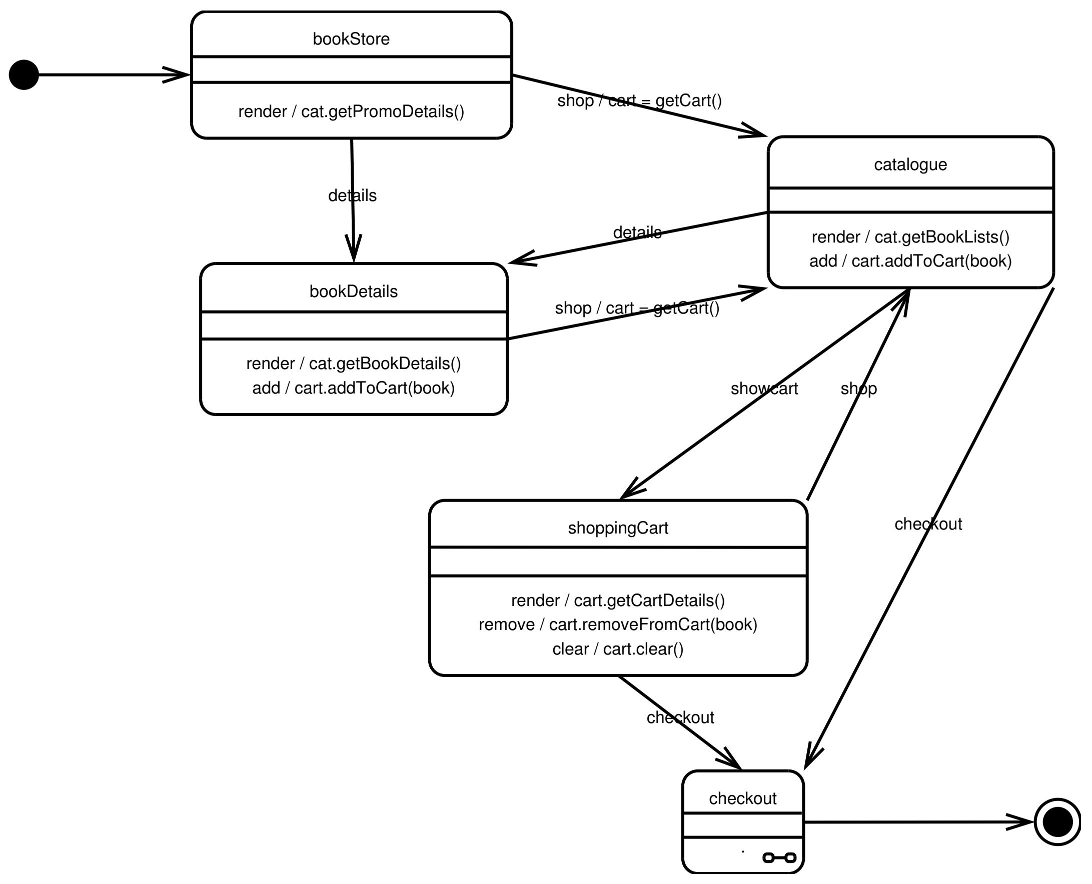
bookStore
render / cat.getPromoDetails()
shop / cart = getCart()
catalogue
render / cat.getBookLists()
add / cart.addToCart(book)
details
details
bookDetails
render / cat.getBookDetails()
add / cart.addToCart(book)
shop / cart = getCart()
showcart
shop
shoppingCart
render / cart.getCartDetails()
remove / cart.removeFromCart(book)
clear / cart.clear()
checkout
checkout
checkout

Figure 3.8: Duke's Bookstore—top level CompositeLocation

It also provides a link to where the entire catalogue can be browsed.

Supposing the user now clicks on the link leading to the catalogue, she will see a new web page which has a list of books on it. To be able to show this page, "cat.getBookList" is first called upon visiting the page. Each book is listed with a button or link next to it, labelled "add to cart". If the user now clicks on "add to cart" next to a particular book, she stays on the same page, but the book is added to her shopping cart (as indicated on the new rendition of the same logical page). This is implemented by the internal transition which handles the "add" event by calling "cart.addToCart".

Assume the user then clicks on another link labelled "proceed to checkout". The catalogue location responds by following the transition to the checkout composite location. As specified in Figure 3.9, this lands our user in the cashier location, where a web page is shown where she can enter a creditcard number and details to finalise the order. Upon completion, she can click a "submit" button, which will take her to a location showing a receipt (also creating the order when the transition fires).

If, for example, the user neglected to specify all the necessary details for payment at the

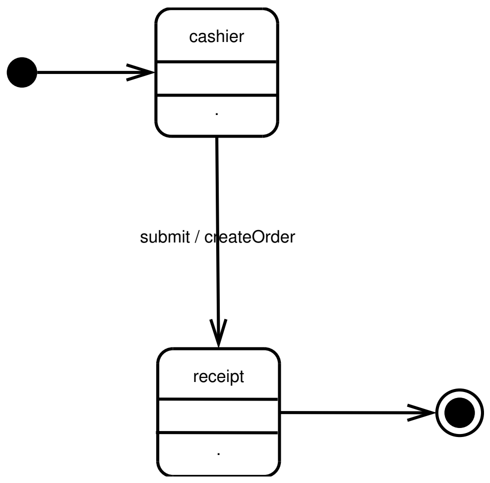

Figure 3.9: Duke's Bookstore—the "checkout" CompositeLocation

cashier, the code invoked by "createOrder" could have thrown a UIException. This would have had the effect of re-rendering the cashier page, but with an error message on it indicating that the user should specify additional details.

3.4 Summary

In this chapter, Harel is introduced as a language for specifying web-based UIs. Harel is heavily based on the well-known formalism of statecharts, in its incarnation as statechart diagrams in UML. In an effort to adhere to the intentional architecture of the web (which is seen as being critical to the success of the web), the semantics of Harel is defined in terms of the architectural concepts guiding the present-day web standards—as defined by the REST architectural style [Fielding, 2000].

In Harel, the notion of "location" is substituted for that of a state in statecharts. Such a location is defined as a coarse-grained state in the conversation between a user and a system. A Harel location is also mapped to a REST resource. Transitions between locations represent the possible paths a user could take between locations. Guard conditions allow fine-grained control over when transitions may fire. Actions allow the UI to invoke operations on the system to which a UI is presented, or to query such a system.

A concrete syntax is proposed for Harel, based on the graphical notation of UML statechart diagrams.

Having defined Harel, a system of sorts is needed on which to run it. Such a system can be seen as a framework for executing web UIs that are specified in Harel. Chapter 4 presents a discussion of issues to be addressed in designing and implementing such a framework.

4 The design and implementation of a web framework for Harel

In this chapter, topics related to the design and implementation of a framework are discussed. The main responsibility of this framework is to execute Harel specifications. Executing a Harel specification in this context means to present a UI via a web server as per the specification.

First an overview is given of related standards in terms of which the framework is implemented (Section 4.1), followed by a discussion of some relevant current practices (Section 4.2).

The design of a framework for Harel is proposed in Section 4.3, including some implementation considerations.

Some important observations and further related topics are also mentioned regarding the role of configuration (Section 4.4) and the impact of typical deployment options (Section 4.5).

4.1 Overview of related standards

To be able to implement a framework for Harel, it is necessary to understand the underlying REST implementation serving as the platform for executing a Harel specification. For that reason, this section presents a brief overview of important web standards that implement REST. This overview is not intended to be complete—it merely highlights the essence of what is needed with regard to the implementation of a framework for Harel[1].

4.1.1 Resource identifiers

Several standards exist for identifying resources (in REST terms) on the web. Reports, recommendations and standards regarding these can be found in Mealling and Denenberg Eds. [2002], Berners-Lee et al. [1998], T Berners-Lee [1994], Fielding [1995], Moats [1997] and Daigle et al. [2002].

[1]Note that standards more related to presentation—*HTML, XML, XHTML, CSS and XSLT* are seen as outside the scope of this work.

Our interest is mostly with Berners-Lee et al. [1998] and the older T Berners-Lee [1994], which discuss Uniform Resource Identifiers (URIs) and Uniform Resource Locators (URLs), respectively.

Generically speaking, resources on the web are identified by URIs. URLs are a subset of URIs which identify a resource by its primary access mechanism (i.e. its location on the network) [Berners-Lee et al., 1998, p3].

URIs (and thus URLs) are very visible in that each web browser displays (and allows a user to type in and change) a URI in some text area at the top. (Interestingly enough, this bar is called the "location bar" by some browsers.)

What follows in this section is an overview of URLs, and implicitly some concepts that are more generically defined for URIs.

4.1.1.1 URL basics

A URL is a string of text of the form:

```
<scheme>://<authority><path>?<query>
```

Where:

<**scheme**> identifies the URI scheme according to which the rest of the identifier should be parsed[2];

<**authority**> specifies the naming authority responsible for allocating the resource its identifier;

<**path**> is a hierarchical identifier assigned to the resource; and

<**query**> is a string containing information to be interpreted by the resource.

The path consists of a list of segments concatenated by a slash character ("/"), representing a path from the root of a possible hierarchical name space, such as commonly found in file systems.

The path can be empty, in which case it is a single "/". The query can be omitted, in which case the literal question mark ("?") delimiting it is omitted.

URLs using the "http" scheme further define the query string to be a collection of (name, value) pairs. Each pair is a text string of the form:

[2]Several different schemes are defined; only one is covered here—http.

```
<name>=<value>
```

The pairs are separated from each other by ampersand ("&") characters. Names, here, are not unique: more than one instance of a name can each map to a different value. Here is an example:

```
searchString=swim&allowedLanguage=en&allowedLanguage=fr
```

For "http" (and other schemes using an Internet Protocol (IP) based protocol to a server in the Internet), the authority part has the form:

```
//<userinfo>@<host>:<port>
```

Where:

<**userinfo**> is a string specifying user authentication information;

<**host**> is an IP, version 4 (IPv4) address, or a Domain Name Service (DNS) host name; and

<**port**> is the IP port to connect to.

The userinfo part and the port are optional. When omitting the userinfo part, the literal "@" delimiting it is also omitted; similarly the literal ":" delimiting the port, when the port is omitted.

An example should look familiar:

```
http://www.google.co.za/search?q=goggle
```

Here:

http is the scheme;

www.google.co.za is the authority, in the form of a DNS host name;

/search is the path to the search function on that particular Google host; and

q=goggle is a query string the search function understands. (In this case, it is asked to search for the word "goggle".)

A distinction is also made between a URI and a URI reference. A URI reference is how URI are commonly referred to (what you type into a browser, write on a serviette, or use as a hypertext link). A URI reference may include extra information apart from the URI itself, in the form of a fragment identifier. This, however, is not useful in the present discussion (the interested reader is referred to Berners-Lee et al. [1998, p15]).

4.1.1.2 Relative and absolute URI

URIs are sometimes used in a shared context. A collection of web pages arranged in a common hierarchy, for example, may refer to each other. In such a case it is useful to specify partial URIs that are relative to the common root of the hierarchy—making it possible to move the collection of pages, etc.

For this reason, some URIs are absolute, and some relative. In Section 4.1.1.1 absolute URLs are described. An absolute URL identifies a resource, independently of context.

A relative URL is the subset of a full URL starting with the hierarchical path part (or the tail end of that path), towards the end.

A relative URL, being a subset of an absolute URL, needs a base URL from which the missing information can be derived.

How this base is determined is fully specified in [Berners-Lee et al., 1998, p18]. For our discussion, it is sufficient to know that in a typical web page retrieved by URL, if nothing else is specified, the base URL for relative URLs used in referencing capacity in the document (such as a relative link in an HTML document) would be the URL used to retrieve the document.

An empty relative URL reference is seen as a reference to the document itself.

The absolute forms of relative-path-references (the commonest form of references to relative URLs—see [Berners-Lee et al., 1998]) are computed by appending the relative URL to the base URL from which the last path segment (and everything after it) has been removed. For example, given the base URL `http://a/b/c.html?as=df`, the relative reference `x.html` is resolved to `http://a/b/x.html`.

A dot (".") and double dot ("..") can be used as segments in the path of a relative URL. These have an analogous meaning to their use in common file system paths: ".." means one level up in the hierarchy; "." means the current level in the hierarchy.

4.1.2 HTTP

HTTP is an application-level protocol used by the Word Wide Web (WWW) global information initiative since 1990. HTTP is a generic protocol, though, which can be used for other purposes than by the WWW [Fielding et al., 1999, p1].

4.1.2.1 Background

The success of the WWW can lead to a subtle confusion of the WWW with its host, the Internet. The Internet consists of physical components (hardware) and supporting software (protocol suites) that enable basic communication between systems.

The Internet is thus what gives computers globally the ability to talk to one another. HTTP is an application-layer protocol for such communication, on top of basic facilities provided by the Internet. Web applications are applications that utilise HTTP specifically in order to present their user interfaces to the wide audience of the WWW.

There is a tension between the needs of web applications and what HTTP provides for them in order to implement those needs. The rise of RIA spearheads the agenda of web applications architects, for example. A very valid question at this point is: why exactly do web applications use HTTP, if they could just use their own protocol built on top of IP and Transmission Control Protocol (TCP)?

The answer probably has a lot to do with how well each protocol scales. Fielding [2000, p69] mentions "anarchic scalability" as part of the requirements of the WWW. Briefly, "anarchic scalability" is concerned with the fact that the system is not under control of one entity or organisation: the WWW (as embodied by HTTP) extends across technical, political and security trust boundaries. The web needs to operate flawlessly under unanticipated load, or when given malformed or maliciously constructed data.

The components participating in an HTTP conversation include proxies, caches and gateways. A lot of improvements since early deployments of HTTP have been towards enabling HTTP to work well with these components—and they are all geared towards enabling the WWW to scale in its anarchic environment. (See Fielding [2000, p71] regarding such problems with early HTTP.)

Of these components, gateways and proxies are probably the most visible in everyday organisations, since these enable HTTP to penetrate fire walls (inward bound and outward bound respectively)—an ability the lack of which has severely limited the applicability of other protocols that could be competitors to HTTP for some applications (such as Internet Inter-ORB Protocol (IIOP) used by Common Object Request Broker Architecture (CORBA) and EJB).

Importantly, HTTP is also able to do content negotiation: a server can negotiate with its client as to the format in which it should return data. This allows a wide variety of clients to evolve independently of the server, in a heterogeneous environment.

4.1.2.2 Overview

Again, the overview of HTTP presented here is not intended to be complete. In fact, it does not cover mature elements of HTTP which enable important properties such those needed for anarchic scalability. The presentation is limited to an understanding of the essentials, from the point of view of implementors of web applications. The interested reader is referred to Fielding et al. [1999].

HTTP is a standard for a protocol adhering (mostly) to REST [Fielding, 2000]. It thus is a simple request-response protocol: a user agent sends a request message to an origin server (possibly passing through intermediaries such as proxies or gateways). The origin server sends a response message back. No conversation state is kept by the server.

Each HTTP message has a message body containing its payload (some kind of representation of a resource), and is accompanied by headers used for communicating meta-information about the message. The headers of a message comprise a list of name to value mappings; HTTP standardises the particular names and values used and their meaning.

An HTTP request message is a request to perform one of a few defined "request methods" on a particular resource as identified by a URI (or part of a URI) included with the request, and other meta-information.

The following request methods are defined:

options is a request for information about the communication options available along the chain of components traversed by a normal request to the resource indicated;

get requests the retrieval of a representation of the targeted resource;

head is similar to `get`, but it only retrieves the meta-information that would have accompanied a complete response to a `get` request;

post requests include the representation of a resource with them, which can be used by the server to perform some kind of action, this action being parameterised by the resource representation that was included in the request;

put requests ask the server to store the enclosed resource representation under the URI supplied in the request;

delete requests the deletion of the resource identified by the request URI;

trace is used to trigger a remote application layer loop-back of the request message for testing purposes; and

connect is a special method used by some proxies.

Important to note is that in the HTTP specification, implementors are explicitly reminded of the fact that the software involved represents users in their interactions over the Internet. Thus, "[implementors] should be careful to allow the user to be aware of any actions they might take which may have an unexpected significance to themselves or others" [Fielding et al., 1999, p51].

For this reason, some methods are said to be safe, others idempotent. Safe methods should have no meaning other than retrieval (and thus no side effects). Idempotent methods have the property that the side effects of $N > 0$ identical requests is the same as that of a single request.

Although the HTTP protocol cannot enforce these properties, implementors should take note of the distinctions made and use the methods accordingly.

For the purposes of implementing Harel, only the `get` and `post` methods are fundamental. The `get` method is designated as safe (and therefore implicitly idempotent too); the `post` method is neither safe nor idempotent.

The response sent as a result of a particular request includes an (optional) representation of a resource and a status code (as well as meta-data as with all messages). The status code either reports the meaning of the server's response to the request (which can be a number of different kinds of successful results, warnings, error conditions) or it directs the client elsewhere.

Status codes may be, for example:

2xx Successful Several codes in the numeric range 200-206 are used to report success, depending on the request method;

3xx Redirection Codes in the range 300-306 direct the client in some way or another to issue a new request, possibly using a different method, different URI, or both. Code 303 (See Other) is of particular importance to our implementation. According to Fielding et al. [1999, p63] (defining its semantics): "The response to the request can be found under a different URI and SHOULD be retrieved using a GET method on that resource. This method exists primarily to allow the output of a POST-activated script to redirect the user agent to a selected resource."

4xx Client errors Codes in the range 400-417 indicate error conditions apparently caused by the client.

5xx Server errors Codes in the range 500-505 indicate error conditions arising in the server during its attempt to service the request.

Caching in HTTP is seen as a significant way of improving performance and of softening the blow of high loads. Integral to, and inextricable from, the HTTP protocol are a number of elements that specify and improve the effectiveness of caching. While it is unnecessary to delve into the details, the following should at least be noted:

- Responses to a `get` request may be cached, depending on some conditions;
- Responses to a `post` request are not allowed to be cached; and
- Any response with status code 303 (See Other) is not allowed to be cached.

Separate extensions to HTTP have been proposed for dealing with keeping session state, and for dealing with security. Each of these are important topics, discussed next in Sections 4.1.3 and 4.1.4.

4.1.3 State management in HTTP

In Kristol and Montulli [2000], an optional addition to HTTP is proposed, known as the “Cookie”. This idea was first introduced in a proprietary way by Netscape [Netscape, 1999], but lately most participants implement the proposed standard [Kristol and Montulli, 2000].

Cookies are an attempt at establishing a larger context within which requests can be sent to a server: that of a session. A Cookie is a named piece of data, opaque to the protocol and user agent, but meaningful to the server which set it. A server can start a session by asking that the user agent store a Cookie on its behalf. This is done by including a special header containing the Cookie and some meta-information about it as part of a response. From then on, a client should include the Cookie with each subsequent request to the site whose server set the Cookie. This is done again by sending the Cookie in a special header as part of the meta-data of the request message. Thus, since each client request includes the Cookie, the server will always have access to the session state stored in it.

According to Fielding [2000, p130], the Cookie extension is inappropriate and violates the architectural constraints of REST. This is mainly because it imposes an implicit ordering of requests—requests sent before the Cookie was sent are handled differently from requests sent after the Cookie was set. So, for example, if a user clicked on the back button of the browser after the Cookie was set to go to a location visited before the Cookie was set, the request could be interpreted differently than the first time it was issued, since the second time it was issued was within the context of the state held in the Cookie.

Cookies are also a security concern, since they have been used inappropriately, for example, to track the browsing habits of a user covertly [Whalen, 1996, Perkins, 2000].

Nevertheless, Cookies are widely in use for storing session state in spite of their problematic nature. User agents usually allow one to disable Cookies, or modify how they manage the Cookies that they receive.

4.1.4 Authentication and security

Most serious web applications have several security-related requirements. Security in itself is a large topic which falls outside the scope of this dissertation. Suffice it to say that user agents and origin servers often want to authenticate themselves to one another and ensure that sensitive data cannot be seen or modified by eavesdroppers. Often, servers also need to authenticate the user at a particular user agent and grant different access to resources based on the identity of a user.

These security considerations are not very well supported by HTTP at the moment.

HTTP includes two mechanisms for authenticating a user with the aim of controlling access to resources based on the identity of the user. These are called basic and digest access authentication [Franks et al., 1999].

Both the basic and digest authentication schemes assume that a request is made to a server as usual, but that the server can refuse to grant access and instead challenge the client to authenticate its user. The server does this by returning a special status code in response. The user agent can then resubmit its request, including authentication information in the headers of the request (after presumably having asked its user for a user name and password, for example).

Basic authentication was first introduced and is widely implemented by clients. It has a serious flaw, though, in that it sends the user name and password in clear text over the network.

Digest authentication was introduced as a fix to this problem. While digest authentication does its job well, many older browsers do not implement it.

Although the standard includes a way for the server to state its preferences as to which authentication method should be used by a user agent, many user agents do not implement the specification accurately. The result is that in practice today, a server cannot be sure that it circumvents the flaws of the basic authentication scheme when challenging a client.

Apart from these failings of the native authentication mechanisms of HTTP, they do not cover other important security requirements, such as authenticating the end points of the conversation or providing protection against eavesdroppers.

Netscape, being an early advocate of many web-related technologies, introduced several solutions to such problems (which may be judged by some to have been rash, in hindsight). One such solution to the security problem was the SSL standard [Freier et al., 1996], and the use of HTTP over SSL. The modern incarnation of this solution is to use HTTP over Transport Layer Security (TLS) (the successor to SSL). TLS is documented in Dierks and Allen [1999] and Blake-Wilson et al. [2003], and the use of HTTP over TLS in Rescorla [2000].

TLS is a transport layer protocol, similar to (but built on top of) TCP. Whereas TCP ensures that data is sent accurately, TLS adds security. It allows clients and servers of its connections to authenticate each other, and can sign and encrypt data sent—thus potentially providing complete protection from problems such as eavesdroppers and man-in-the-middle attacks.

For all other purposes, TLS appears to HTTP as its usual transport layer partner, TCP. Thus it is relatively easy to use HTTP over TLS instead of TCP.

HTTP over TLS (HTTPS) is widely in use today, and seems to be accepted colloquially as the de facto solution to providing for security requirements on the web.

However, work is in progress to incorporate security features into HTTP, in the form of Secure HTTP (S-HTTP) [Rescorla and Schiffman, 1999]. This is still experimental, though, and thus irrelevant for current practical application.

A brief explanation of some common and recommended implementation practices using these standards follows in Section 4.2, before a framework for Harel itself can be discussed in Section 4.3.

4.2 Current and recommended practices

Many informal information sources on the web document so-called best practices for web site design. These are very important when designing web sites, since testing what a web site actually looks like is practically impossible—there are just too many permutations of browsers, browser versions and user preferences that play together in order to create the look (and partially, the behaviour) of a website in the context of a particular user. The best practices for web design really are heuristics for handling this diverse environment. It seems appropriate to draw from this experience, rather than to blindly trust that a site would look acceptable in all environments because the site looks acceptable in the few environments with which a programmer is familiar. Most of these fall outside the scope of this work, being largely related to presentation concerns. Some, however, concern our implementation, and are thus discussed here.

Similarly, some requirements of web applications are generally implemented in typical ways. These are not best practices (indeed, some of them are discouraged), but are current practices that are important to note, and are also discussed under the current heading.

4.2.1 Session state

Many web applications need to store information between multiple requests from the same user. A famous example of this is the "shopping cart" that online stores often provide. The cart is analogous to a real-life shopping cart into which items can be placed while browsing—for later perusal or so that they can be bought in batch.

Session state is also kept for the purpose of controlling page flow (Section 1.3.3)—a flag can be set by one page, to be used for deciding the behaviour of later pages. This kind of session state is perhaps best illustrated by implementations of the algorithmic approaches (Section 2.3.14).

Considerable session state is also kept by some web frameworks in an attempt to implement more powerful ways of UI specification. For example, JSF [McClanahan et al., 2004] is a page-flow-centric approach (Section 2.3.11) that also derives a number of more traditional GUI events from each incoming request (Section 2.1.4). A JSF framework has to maintain a relatively large amount of data as part of session state: for each view it stores (in JSF terms) the UI components on that view, and the last-specified user input to each view. Having access to this information means that the framework can infer a number of events from a single request, allowing for the handling of UI events in the style of event-listener models. (This technique is explained in Section 2.3.10.)

Web frameworks can easily tend towards favouring a programmer's perspective when it comes to session state. Programmers would like to be able to specify a UI using the full power of variables that can be set in so-called "session scope", not always consciously considering the implications of this specification feature.

Seeing that REST expressly prohibits the storage of session state on the server, it is perhaps wise to be careful about *how, when, and how much* session state is kept. Moreover, when proposing a specification technique, care should be taken to consider its demands in terms of session state.

Web UIs cannot be realistically specified without keeping session state. However, note that REST does not prohibit the notion of session state—it merely mandates that session state should be held on the client as opposed to the server in order (chiefly) to enable scalability.

In line with these facts, it would seem wise to limit the amount of session state information: small amounts of session state information can easily be kept on the client entirely, by using Cookies (Section 4.1.3), by simply adding the state as parameters to URLs embedded in a particular page, or by other similar methods. These methods keep session state information off the server (as originally intended), even though there are some problems with Cookies, per se. (The interested reader is referred to Fielding [2000, p130].)

Cookies and request parameters cannot handle large amounts of data. Frameworks that keep a lot of session state (despite the recommendations of REST) and that use one of these techniques, therefore usually do no more on the client side than set a unique session id at the browser.

Upon a request, the server can then use the session id thus stored to identify the required session, and then, based on that information, query a database of sessions kept by the server itself. (That is, such frameworks keep just enough session state information on the client to be able to keep heavy-weight session state information on the server.)

Note, however, the complexity involved when a server maintains its own database of sessions. (An example of this is the implementation of clustering of the Apache Tomcat Servlet/JSP container—explained in Penchikala [2004].) One strategy for doing this is to keep a list of sessions in memory. However, apart from the fact that this would use valuable memory for each client concurrently in communication with the server, it also presents problems when more than one machine is used to serve requests (often the case for large sites). For example, a client's first request may be routed by load-balancing software to machine A where its session is stored in memory. The next request may be routed to machine B which, being a separate machine, does not share memory with A, and will have no session state for the user. Of course, this problem can also be solved—one solution is to implement a way for servers to broadcast requests for session state to each other.

While it is possible to implement such functionality in conflict with REST, it may be unnecessarily complex to do so. The reason is that REST constrains an architecture in specific ways, one of the reasons being precisely to make the solution easily scalable in the Internet environment. Hence, it is wise for framework designers to critically evaluate the functionality provided by a framework—possibly choosing to exclude functionality whose implementation would be in conflict with REST. If certain functionality is deemed absolutely necessary, a framework designer should investigate the option of supporting it in a weakened form. Thus, if storage of session state information is deemed desirable, one should look to storing the smallest quantity of session state information possible.

4.2.2 Authentication and security

Often, web applications require their users to be authenticated. The common procedure is to use a user name and a password. The problems with basic and digest authentication (Section 4.1.4) mean in practice that when these are used, they have to be used in conjunction with HTTPS in order to ensure proper security.

Many designers also do not use the built-in authentication mechanisms at all, preferring rather to perform a user login via a normal page with an HTML form. Reasons for this decision are, for example, that with a custom-built page, one has more control over what the login page looks like (whereas with the built-in mechanisms of HTTP one is left to the disparate devices of a wide range of web browsers). This method also allows for ways of authentication other than specifically using a user name and password prompt. For example, some sites show a graphic with distorted letters on it that are difficult to parse by computer, but easily recognisable by humans. The idea is then for a human user to type the letters seen in the graphic in a separate box, in addition to a user name and password. So doing, access is restricted to human users only.

Whatever the case may be, the name of the currently authenticated user is then usually held as part of session state.

Note, however, that this is not strictly speaking necessary when using the built-in mechanisms of HTTP: user agents can store the user name and password supplied by a user upon a challenge from a server. When a new request is made to the server, a user agent may subsequently include this authentication information or supply it automatically upon a challenge from the same server without disturbing the user again.

Hence, saving the name of the authenticated user as part of session state (when using the built-in mechanisms of HTTP) is an optimisation: the server does not need to verify the same user name and password combination upon each request in a user session, and the extra round-trip implied by a challenge upon each request is avoided. (Note that these issues are being addressed by Rescorla and Schiffman [1999].)

4.2.3 The PRG pattern

When user input is needed on a web page, it is sent to the server as an HTML form which is submitted as part of a `post` request. And true to the nature of a `post` request, such user input usually results in the server performing an action that may have side effects and is not idempotent or safe (Section 4.1.2.2).

So far, this practice does not pose any problems. It is common, however, to return a new page for display by the user agent in response to such a request. This is problematic because of the semantics of a `post` request.

The HTTP protocol states that a `get` request should be a simple retrieval operation without side effects. A user agent can thus remember a trail of URI so visited and, for example, allow a user to go back along this trail to places previously visited. Visiting such an "old" location is implemented by the user agent issuing the original request for that URI again. Doing so is not problematic for well-behaved `get` requests, since they should be safe. `Post` requests, on the other hand, cannot be repeated automatically, because they may have side effects each time they are executed by the server. For example, a user who was previously on a page where she clicked on a button in order to make a payment, would not want the request that was sent by clicking the button inadvertently to be repeated.

In order to deal with this issue, web browsers typically do not repeat a `post` request automatically—they will notify the user that a `post` request is about to be repeated and ask for confirmation before proceeding. This, however, is confusing to an end-user, and the risk is high that the user would make the mistake of allowing another request to be made. Web browsers can ensure that the user is accountable, but that does not solve the core problem.

In response to this problem, some have been promoting a design pattern, called the Post-redirect-get, also known as Redirect after post (PRG) pattern [Jouravlev, 2004]. The PRG pattern states that the server should never send a response which is intended to be rendered back to a client upon a `post` request. Instead, it should always respond by redirecting the client to a new URI where it can use the `get` method again in order to retrieve the results of the action effected by the `post`. It follows that the results may be retrieved repeatedly, without side effects.

Using the PRG pattern has the result that the browser only ever displays (and therefore keeps track of) `get` requests. And a user can safely navigate recent browsing history with the back and forward buttons of the browser (since the browser only paused to display `get` requests).

Browsers also allow users to bookmark the locations they visit (the pages displayed by a browser). Bookmarks are similarly implemented—the browser stores the URI of a request and issues that request again in order to visit a marked place. The PRG pattern ensures that these basic functionalities of browsers work in an intuitive manner without confusing the user—by sensibly dealing with the semantics of the various request methods as defined by HTTP.

4.2.4 Dealing with optional extensions to the standards

A system is said to degrade gracefully if it continues to operate in the face of failures of some of its parts, losing functionality in proportion to the quantity of failures.

Some of the extensions to standards mentioned so far (and others not mentioned) are features that are "optional". It may be that in certain environments they are not available, or it may be that a user can explicitly disable them. Well known examples of these are Cookies and JavaScript.

Web site design pundits advocate that when such features are used by frameworks, the absence of an optional feature should not make the web site unusable [Tobias, 2004, Korpela, 2002]. In other words, a web framework which takes advantage of such optional extensions should degrade gracefully in situations where they are not available. For example, the lack of JavaScript should merely make a site less responsive, or less aesthetically pleasing, or disable a few non-core features instead of causing the site to cease operation entirely.

A web framework should thus deal with the absence of optional extensions by degrading gracefully—its basic model should not be based on such an optional feature.

In the next section (Section 4.3) a design is proposed for a framework for Harel, based on the standards and (some of the) practices explained.

4.3 A framework for Harel

This section outlines a proposal for how a web application framework based on Harel could be implemented. It is based upon the theoretical considerations discussed up to this point, as well as on experience drawn from two prototype implementations that have already been developed: one in Java and one in Python. Both prototypes are functional, but some of their details differ from what is presented in this section[3]. The proposal given below, once fully developed, will be an improvement of the prototypes—both in terms of adhering more strictly to the Harel specification as described in this work, and in the structure of the software. What is presented here represents the design of the next versions of the prototypes. Their

[3]The Java prototype is incomplete—it does not include support for actions.

development is part of an ongoing project, undertaken by the author, but is seen as beyond the scope of a single-year research project, which is what this dissertation represents.

Although the presented design is different from that of the prototypes, most functionality covered is implemented using the same techniques. The odd instance where presented functionality is not implemented in prototype form is pointed out in the text.

The prototype implementations differ from what is presented in this section in the following ways:

- The notation used in the prototypes is incomplete and not based on the UML standard, but has the same semantics as presented here.
- In the prototypes, actions and guards are not statements and expressions (respectively) as defined in this work. These are specified in the prototypes as methods to be called on named instances of DomainAdapters. The methods specified for guards should return a boolean value. In line with this, the prototypes implement a slightly different concept of the context which is used by actions and guards during a request (but which fulfills the same function).
- The prototypes do not allow the declaration of formal detail attributes expected by a UI, nor do they allow the specification of how actual detail attributes in a hierarchy map to these expected formal detail attributes (Section 3.3.1.3). The equivalent of DetailAttributes in the prototypes are inherited across the boundary of a composite location without these controls.
- The prototypes are not modularised—all functionality presented here as part of extension modules is intertwined with the core framework.
- Instead of making use of *Spyce* for handling presentation, the prototypes support *Quixote* (the Python version) and *Velocity* (the Java version). The particular use of these products in the prototypes also differs from how the use of *Spyce* is presented in Section 4.3.4: the prototype implementation is an experiment to provide a new method of specifying presentation by composing pages using inherited detail attributes. This experiment is excluded from the work presented here, but cited in Section 5.3 as a possible direction for future work.
- The prototypes use the term "state" instead of "location"—the use of location was seen as being more UI-designer-friendly, but has been defined with exactly the same semantics. Also, the term "attribute" is used in the prototypes instead of "detail attribute".

4.3.1 Components and the scope of the discussion

The proposed framework for Harel would comprise[4]:

- A GUI tool with which diagrams can be drawn adhering to the Harel notation. This tool has a built-in web server which allows a programmer to surf to the application UI under development. (In the following discussion this tool is referred to as "the development environment".)
- The core framework itself, which is an object-oriented programming language library roughly correlating with the abstract syntax of Harel. It can be thought of analogously as an interpreter for Harel (henceforth referred to as "the core framework").
- A number of extension modules. The extension modules provide, for example, tools to use for the actual generation of pages or to keep track of session state.

The core framework and the extension modules are intended to be deployed separately from the development environment in a production environment served by a heavyweight web server such as Apache [The Apache Software Foundation, 2005]. (This is also implemented as such in the prototypes.)

The details of the development environment are not important for the purposes of this dissertation. Neither is the precise detailed design of the core framework. Instead, the framework is presented on a higher, conceptual level—focusing on how a Harel model is executed—in terms of the important design decisions (and conventions used) and explanations of how some important extension modules would be implemented[5].

4.3.2 Core framework

The core framework can be reduced to a server component which is invoked upon an HTTP request to which it should generate a response. The core framework is now discussed in terms of how incoming requests are interpreted and responded to. The discussion is applicable to the prototype implementation (or the improved design thereof where the latter is different).

4.3.2.1 State of the UI

Each request pertains to a particular resource—mapping to a particular location as defined by Harel. Since each location has an identifier (its URL), and this identifier accompanies each

[4]In the prototype implementations, the last two items listed are merged into one.

[5]The reader is reminded that most of the functionality of extension modules is implemented using the same techniques, but using a less clear, monolithic design.

request, the server implicitly has access to the state of the conversation (at the granularity of locations), without having to keep any state information. Note that this is a direct result of Harel semantics that maps locations to REST resources, and the result is in line with the intention of REST architectures.

4.3.2.2 The interpretation of requests

An HTTP `get` request (which is supposed to be safe) is always interpreted by the framework as a request to render the contents of the particular location that it targets. An HTTP `post` request signals the occurrence of an event in the location it targets.

Thus, should the framework receive a `get` request for a simple location, it returns a rendition of that location in a format which the user agent can use to display a page which represents that location to the user.

When the user initiates an event from a page (such as clicking on a button), a `post` request is sent to the location which that current page represents. Upon receipt of a `post` request, then, the framework can check what transitions are available at the location which received the `post` request, and fire one (whose guard evaluates to true). (In order for a `post` request to be generated *to the location represented by the currently displayed page*, the generated page should follow the convention that its HTML form be submitted to the same URL where the page was fetched from.)

Firing a transition consists of first executing its action, and then replying with an HTTP status code 303 (Use Other) which directs the user agent to issue a `get` request for the target location of the firing transition. Internal transitions are treated as though they target the location in which they are defined. This response to `post` requests is in adherence of the PRG pattern (Section 4.2.3).

Note that, since events are only handled by simple locations, `post` requests only have defined responses for simple locations. `Get` requests for a composite or referred location are implemented by redirecting the user agent to the first location of the implied composite location (as indicated by its initial transition).

4.3.2.3 Representing and indexing the Harel model

The framework maintains a model of the UI specification. This internal model can mirror the abstract syntax of a Harel specification of the UI—thus an abstract syntax tree. A URL-based naming scheme is needed to identify locations, and the model should be organised in such a way that it is indexed with the aim of quickly being able to find a location, given its

URL.

Seen as a whole, a UI specified in Harel consists of location nodes, related by means of containment and transitions.

For implementation purposes, the containment relationship can be taken as the main point of view: locations form a hierarchy of containment. The leaves of this graph are simple locations; intermediary nodes are composite states.

Figure 4.1 shows the containment hierarchy of the example Harel specification in Section 3.3.4.

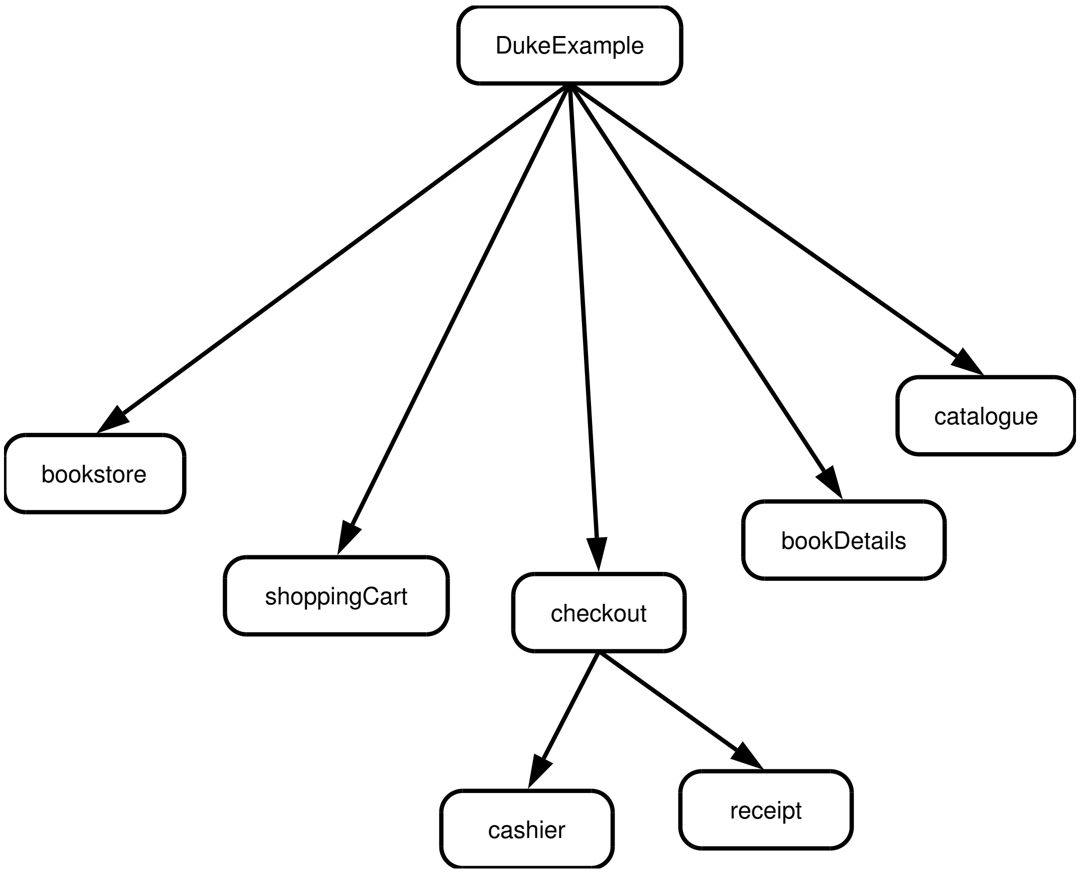

Figure 4.1: A containment hierarchy

The node named "DukeExample" is the top-level composite location of the entire bookstore. The "checkout" node is the only other composite location in the simple example.

The use of referred locations transform such a containment hierarchy into a containment graph. Figure 4.2 shows a more complicated containment graph with referred locations.

In Figure 4.2, nodes are labelled for their types: CL denotes a composite location, SL a simple location, and RL a referred location. Some nodes also include names in brackets so

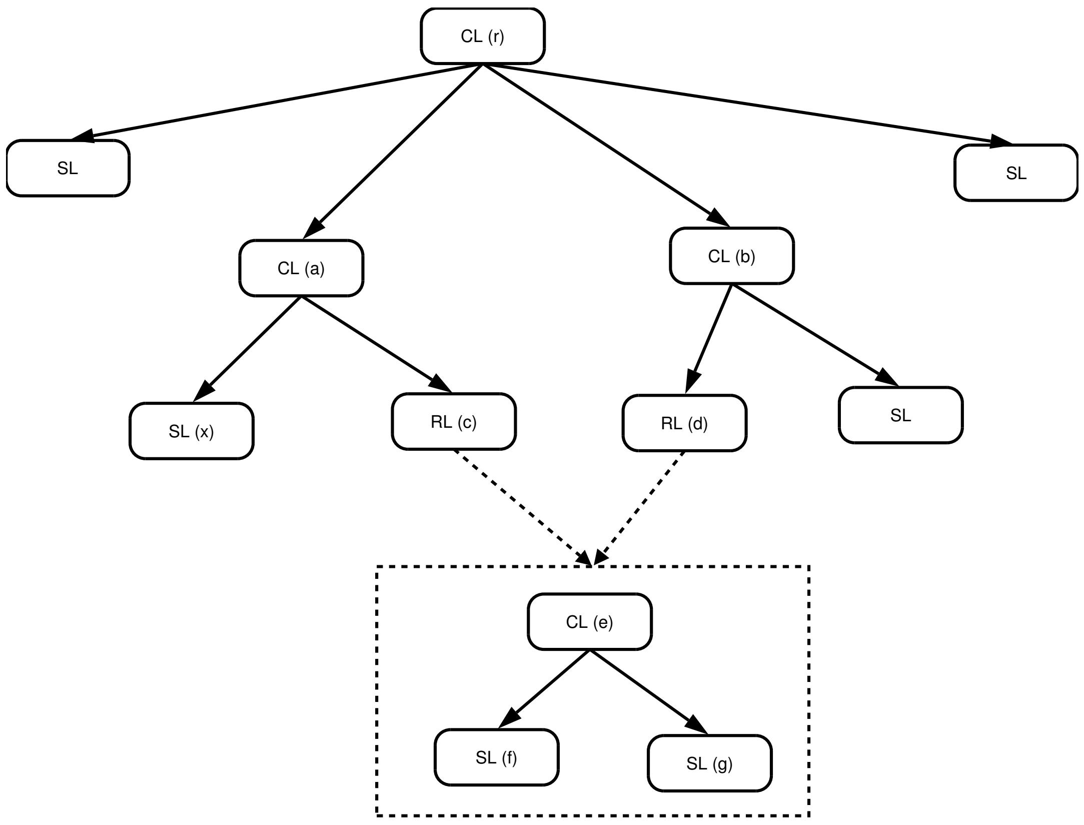

Figure 4.2: A more complicated containment graph

they can be referred to later on in the text. The hierarchy in the dashed box is a UI defined separately. Note that it is being referred to by two different referred locations (c and d).

URLs include a hierarchical path element, and define semantics for relative URLs in terms of this.

The containment graph of locations is a hierarchical name space: each node in the graph is uniquely identified by the path to it from the root of the graph. Since each node has a name, a node can be identified by the sequence of names of nodes along the path to it in the containment graph, starting at the root.

For example, the sequence of names: [r, a, x] identifies the simple location named x. Similarly, [r, a, c, e, f] identifies the location f in its capacity as referred to by c, whereas [r, b, d, e, f] identifies f in its capacity as referred to by d.

The path segment of a URL can be constructed from such a sequence of names by concatenating all the names with a slash ("/") character, "/" on its own indicating the root of the graph: "/r/a/x" or "/r/b/d/e/f" are examples.

Upon receipt of a request, the containment graph can thus be traversed quickly along the path specified in the URL to find the intended location.

Note that the state of the conversation in f, in its capacity as being reached via c, is different from f in its capacity as being reached via d. Hence, these are seen as mapping to two distinct resources, each with a unique identifier. (This example may clarify the abstract explanation of REST identifiers in Section 3.3.1.2.)

4.3.2.4 Instances of locations and the call stack

A URL, with a path as defined in the preceding discussion, also doubles in our implementation to fulfil the function of a call stack in the runtime system of a conventional programming language[6]. The function of a call stack is to keep track of where a specific call to a subroutine was made *from* in order for the runtime system to be able to return to the correct caller—all the while keeping the subroutine itself independent of such information. A framework also needs to be able to compute the "caller" of a composite location (or referred location), so that the correct transition can be fired upon its completion. This information could be inferred from the fact that the implementation has access to the complete model, and the current location is indicated by a request. However, the same information is conveniently stored in the URL: it encodes a call stack—the bottom of the stack being the start of the list represented by the URL. For example, the name of the "caller" of "/a/b/c" is "/a/b" (the last element is popped from the stack).

A distinction is made by the framework between a location and the instance of a location. A location is the definition of a state of the UI—an instance of it is created by the framework when a request is received and destroyed after the request has been replied to. Such an instance keeps track of the location it represents and the context in which it is currently used: the current request, the context, and the current call stack.

The framework keeps a call stack of location instances which it constructs for each request, according to the path in the URL: upon a request, the path in its URL is traversed in order to find the targeted location. During this traversal, an instance of each location along the path is created and placed on the stack.

4.3.2.5 Extension module framework

An implementation of Harel must be extensible by external modules which contribute detail attribute types.

Although the current prototype implementations are not extensible by modules, the planned design for such modular extensibility is presented here.

[6]Remember that composite locations (and similarly, referring locations) are analogous to invocations of subroutines (Section 3.3.1.1).

Each module can be represented by a programming language class implementing a standard interface (for extension modules) which the framework can use to query the module and invoke its functionality. Some methods so provided can be used to query the module for the detail attribute types it provides. Other methods can be called at certain phases during the request-handling process in order to invoke the functionality provided by the extension module. (These methods are named for particular phases during the request cycle—see Section 4.3.2.6 for their names and when they are to be invoked.) The core framework would keep a list of modules as part of its configuration.

Modules would operate by populating the context with special objects before guards and actions are executed. Since the context is available to actions and guards, elements so introduced would be available to them.

A module could also have module-specific configuration associated with it—settings and options that are applied site-wide.

Some extension modules will be provided as part of the standard framework, but a user could write modules as custom extensions. Example modules that should be provided by the standard framework are discussed in Sections 4.3.4–4.3.7 (most of this functionality is implemented in the prototype implementations).

4.3.2.6 The request cycle and exceptions

The request cycle is the sequence of steps followed upon each request.

The request cycle described here is different from that used in the current prototype implementations. The request cycle used by the prototypes makes no reference to extension modules, since this concept is not implemented in the prototypes. The particular functionality implemented by the proposed modules (Sections 4.3.4–4.3.7) is, however, invoked in a non-generic fashion during the request cycle of the prototypes. The request cycle presented here is based on the prototype version—it is essentially a cleaned-up version of the prototype version, with provision for generic invocation of extension module functionality.

During the request cycle, access will be provided to special objects. These are used during processing of the request cycle to influence the eventual response in some way, or to query the contents of the request or the request method.

The request cycle should have the following phases:

1. *Instantiation:* Instantiate the requested location (constructing the call stack of instances in the process).

2. *Context:* Create the context.

3. *Begin modules:* On each extension module, `request_begin` is called.
4. *Determine event:* From the request method and other data in the request, determine which event it signals.
5. *Determine transition:* Determine which transition to take (by evaluating guards and comparing the event to transition labels).
6. *Fire transition:* Fire the transition (this should execute its action and note what the response should be).
7. *(optional) Render a page:* In case of a `get` request the representation of a resource should be generated to send back to the user agent. So, only in the case of a `get` request should this phase be executed. Since the specification of this detailed task is outside the scope of a Harel specification, this phase should consist of calling `request_render` on all the extension modules. One of them should add a variable "rendered" into the context which should be returned to the user agent at the end of the request cycle.
8. *End modules:* On each extension module, `request_end` is called.
9. *Finalise modules:* On each extension module, `request_finalise` is called.
10. *Send response:* Return a response.

Note that events (as signalled by `post` requests) are triggered by, for example, a user clicking on an HTML button. This results in a `post` request. A `post` request is accompanied by data which is organised as a list of name to value mappings. The button clicked in our example would show up as part of the data of the `post` request—its name would be one of the names in the data of the `post` request.

In order to infer the event signalled by the `post` request, the names of its data are compared to the event names on the labels of all transitions that can legally be triggered from the current location. There should be one (and only one) match—the event.

Note also that firing the transition results in executing the transition action and that the framework then should "note what the response should be". In the case of `post` requests, the response is to redirect the user agent to the target location of the transition (assuming no exceptions occurred during execution of its action). This can be "noted" by setting appropriate headers in the response to indicate the new location, and by setting the status code that would be sent back to 303 (See Other).

In the case of a `get` request, redirection is not done, so the transition should not do anything. Rendering should occur in the (optional) next step, discussed below.

Exceptions may happen during the request cycle. These may be domain exceptions (as per their definition by Harel). Domain exceptions can occur during the action of a firing transition. Thus, when a domain exception occurs, the firing transition would take note of the exception and ensure that the user agent does not leave the current location by redirecting the user agent to the source location of the transition, instead of the target location. The exception data is added as an extra query parameter—thus the exception becomes part of the fine-grained state of the location, without the need to be stored on the server.

Any other exception would cause the request cycle to be interrupted, only to resume at the "finalise modules" phase in order to allow the modules to gracefully clean up in spite of the exception. And a response (error or otherwise) is needed regardless of whether an exception occurred or not.

Modules could prevent normal request processing by raising a PreemptException[7]. A PreemptException should halt the request cycle where it is raised and should specify what the response should be—pre-empting the response that would have been computed by following the normal request cycle. PreemptExceptions should be used by modules that need some control over the user agent or that implement access control on a resource.

4.3.2.7 Detail attributes and their inheritance

In the prototype implementations, detail attributes are inherited uncontrolled by name from higher-up in the containment graph. The result of this is that the same UI referred to from two different referred locations may inherit different detail attributes, depending on the referring context.

The specification of expected "formal detail attributes" and the mapping of actual detail attributes to formal detail attributes in Harel (Section 3.3.1.3) are meant to provide a cleaner solution to the inheritance of detail attributes. The design for executing this cleaned-up notation is now presented.

To a location, a detail attribute is just a typed, named piece of data. The framework would need to ensure that detail attributes are inherited correctly (especially across referred location boundaries). Further handling of detail attributes should be done by the modules themselves: the intention is that at each stage of the request cycle, when module code is invoked, the module can check what detail attributes pertaining to it are specified for the

[7]The prototype implementations do not include the concept of a PreemptException, which accompanies the concept of extension modules.

current location and activate the functionality it provides accordingly.

Modules (or other elements) should be able to query a location for detail attributes by name. The detail attributes defined locally to the location would be found immediately, but inherited detail attributes should be searched for at locations upwards in the containment hierarchy. Each location should also be able to be queried for detail attributes that *can be inherited from it*. For the sake of discussion, these are called exported detail attributes. In the event of a location being queried for the name of a detail attribute, and that name not being found locally, it should query its parent location for an exported detail attribute by that name.

An exported detail attribute name should be searched for differently, depending on which kind of location the query is made on. Composite locations should merely query themselves for detail attributes by name as per usual. Referred locations should behave differently: a referred location specifies by name which detail attribute in its containing location may be inherited, and under which name it is inherited (the actual detail attribute and formal detail attribute, respectively). Thus, when a referred location is queried for an exported detail attribute by name, it should first check to see whether such a name is part of its formal inherited detail attributes. If not, the name is assumed not to be found. Otherwise (i.e. if the name is indeed part of its formal inherited detail attributes) the referred location should query itself for a detail attribute with the actual name which maps to the specified formal detail attribute name.

In the remainder of Section 4.3, implementations are explained of specific modules that provide functionality necessary by most real-world solutions.

4.3.3 Options

Before each extension module is discussed, a brief aside: some of the modules in the framework export very simple detail attribute types. The framework includes a syntax (and editor for it) shared by many modules for such simple detail attributes, called options.

Options are collections of named data values. A module can state that the specification of a detail attribute is a set of options. In this case, the module also has to name the options that are elements of the set, and it has to provide default values for these elements, where appropriate.

An editor is also provided for detail attributes that are specified using options. The editor allows the programmer to change the default values of an option (or set of options) and to save the detail attributes as part of the specification[8].

4.3.4 Page renderer

In the prototype implementations, a more complex experimental solution is implemented for rendering pages. This experimental solution is an attempt at a novel technique for specifying presentation which leverages the particular semantics of Harel. A simpler, more conventional solution to the problem of page rendering is proposed in this section. The prototyped technique could also be re-implemented in the form of an extension module, but the topic of presentation is outside the scope of this study, hence the simpler solution presented here.

A Harel specification itself does not provide a mechanism for specifying how a page should be rendered in order to represent a particular simple location upon a `get` request. The intention is that an extension module should provide this functionality, using one of the many solutions available for dynamically generating a page (Section 2.2).

One such existing solution is the *Spyce* project. *Spyce* is a Python-based framework which has strong page rendering capabilities. The page rendering module discussed here would depend on the mature *Spyce* for generating web pages.

Spyce stores its executable page templates in files with a ".spy" extension. *Spyce* can execute these templates—executing a template amounts to interpreting the template (including embedded code), resulting in a dynamically generated page. (*Spyce* templates fall into the "page composition" category, as discussed in Section 2.2.8.)

The page renderer module would provide a detail attribute type called "SpycePage". A SpycePage detail attribute is defined in terms of options (as discussed in Section 4.3.3), but it comprises an empty set of options. It is meant to modify the containing Harel specification by its presence or absence.

The page renderer module should provide code invoked during the optional page rendering step of the request cycle. At this point, the module should check for the presence of a detail attribute named "page" in the current location. Only when such a detail attribute is found, should the module assume responsibility for generating a rendition of the current location and adding the rendition as the value of a variable (named "rendered") to the context.

This proposed page renderer module is extensible itself. Depending on the type of detail

[8]Note that, although options are introduced here as being used by modules, the concept of options (and the generic option editor) is implemented in the prototypes—where options are used for exactly the same purpose, albeit not in a modularised way.

attribute used, it may use a different method of generating the rendered page[9]. SpycePage in particular should act as follows: it should compute a file name for a ".spy" file, based on the name of the current location and then would invoke *Spyce* to execute the template stored in that file. The resulting dynamically generated page should then be added to the context as the value of the "rendered" variable.

The file name of the *Spyce* template can be computed as follows: each Harel UI is stored in a file (the UI file, for the sake of this discussion). The example page renderer module follows the convention that a directory should exist in the same directory as the UI file with the same name as the UI file, but with a ".templates" extension added. This is taken to be the directory in which all ".spy" files for that specific UI should be placed (called the templates directory for the UI).

The templates directory for a UI should then contain a directory hierarchy corresponding to the containment hierarchy in the UI—each directory corresponding to (and named for) a composite location. Referred locations are ignored. Simple locations should be represented by *Spyce* templates in ".spy" files. The name of the template for a particular simple location is defined as the name of the location with a ".spy" extension appended. The ".spy" file for a simple location should be located in the directory of its containing composite location.

The SpyceFile detail attribute could thus compute the name of a *Spyce* template file for each simple location, relative to the UI templates directory.

4.3.5 Session tracker

In Harel, session state is not part of the language, but can be implemented in an extension module. This design leaves the language independent of specific session state requirements, and a project could choose at what level of sophistication session state should be held and how that is implemented. The session tracker module discussed in this section is not implemented in prototype form, but serves as an example of how session tracking could be implemented as an extension module. (The functionality of the authentication module presented in Section 4.3.6 is implemented in prototype form, however, and it follows the same implementation strategy.)

The example session tracker module would simply assign a unique identifier to a particular connected user agent—allowing an implementation to have access to the user session which pertains to the current request.

This can be achieved by making use of Cookies. In the request cycle, just after the

[9]Remember that a DetailAttribute in Harel is like a type: is has both data and behaviour (Section 3.3.1.3). SpycePage has no data (since it has an empty set of options), but it does have behaviour.

context has been created, `begin_request` would be called on the session tracker module. The module then would inspect the current request. If the user agent did not include a Cookie identifying its session with the server, the module would assume that a new session should be started. So, it could compute a unique identifier for the session and set a Cookie containing the identifier (which will accompany the response when it is sent). However, if the request already included a session Cookie, the module would merely read the unique session identifier from the Cookie.

In either case, the module would add a “sessionid” variable to the context with a variable called “sessionid”, which contains the unique session identifier that will be available to actions and guards during the handling of the request.

This module need not export any detail types.

The session tracker module can specify as part of its configuration the name that is to be used at user agents for its Cookie.

4.3.6 Authentication

Some applications need a user to be authenticated. Typically, this is done by asking the user for a user name and password which the system can verify. If the user name and password combination is successfully verified, the user name of the authenticated user is stored as part of session state.

An implementation of such functionality has been done in the prototype form. The techniques used in the proposed authentication module below have thus been successfully implemented.

Authentication can be done using HTTP basic authentication to challenge a user for a password, and an encrypted Cookie to store the name of the authenticated user on the user agent for the duration of a session.

The assumption is that this will only ever be used over HTTPS, hence the pitfalls of the basic authentication scheme can be safely ignored.

Not all locations require that visitors be authenticated. Some way is needed to specify which locations need authentication. This can be done by using a detail attribute. The authentication module will export a detail attribute type, “AuthConfig”, which contains a boolean option, “needsAuthentication”. An AuthConfig detail attribute can be attached to a location using the name “authConfig”. The detail attribute then applies to all locations where it is specified and inherited.

The authentication module would detect whether authentication is needed for a location by searching for an AuthConfig detail attribute named “authConfig” at the location. It would assume authentication is only needed if such a detail attribute is found and its "needsAuthentication" flag is set. Only when authentication is needed should the module do any further processing.

An incoming request can fall into one of three categories:

- The request is new and the user is not authenticated (a new request);
- The request is a replay of a previous request which now includes a response to a basic authentication challenge from the server (a challenge-response request); or
- The request is part of a session for which the user has already been authenticated (an authenticated request).

If authentication is needed, then the module should process requests based on which of these categories they fall into.

New requests should not be allowed to continue normal request handling—the server immediately should respond by returning a basic authentication challenge to the user agent (interrupting the request cycle with a PreemptException). The user agent would react to such a challenge by prompting the user for a user name and password, and then re-issuing the same request—this time accompanied by the user name and password in special headers.

This results in a challenge-response request. Upon a challenge-response request, the module should extract the supplied user name and password from the basic authentication headers included in the request and should verify that the password is correct. If not, the challenge should again be issued, else the user should be "logged in" (an authorised session should be successfully established).

Logging a user in should be done by the module by setting an authentication Cookie on the user agent. The authentication Cookie contains the user name of the authenticated user and a time stamp indicating when the Cookie was set. The contents of the Cookie should be encrypted using an encryption key which is a secret of the server, so that malicious user agents cannot fake a login without a password.

Authenticated requests can be detected by the presence of an authentication Cookie in the request. If such a Cookie is found, its time stamp should be checked. If the Cookie is too old, its presence should be ignored and the request is further handled as if it were a new request. But fresh Cookies are meant to be used to determine the name of the currently logged-in user (and indicate a successful continuation of an authorised session).

The authentication module should set a new authentication Cookie upon each successful continuation or login. In the case of a successful login, this is needed in order to start keeping track of the authenticated session. In the case of a successful continuation, it should be done in order to refresh the time stamp on the authentication Cookie held for the current authenticated session by the user agent. Hence, a user who is active enough (i.e. generates requests to the server frequently enough) will stay logged in (since the time stamp on the Cookie is continually refreshed), whereas inactive users will automatically be logged out.

Upon a successful login or a successful continuation of a logged-in session, the authentication module should insert a variable ("authenticatedUser") into the context, containing the user name that has been authenticated.

A UI that requires its user to be authenticated could thus use the authentication module and be developed without concern for how the authentication is actually accomplished. Its actions and guards will always have access to a variable ("authenticatedUser") which is guaranteed to contain the user name of the user who issued the current request.

The authentication module would need to be configured by two parameters. The secret key (private to the authentication module) to use for encrypting and decrypting the Cookie, and the maximum age of an authentication Cookie.

4.3.7 Back-end integration

Harel specifies the UI of a system only. A Harel specification can communicate with the system to which it provides the UI via guards and actions on transitions: actions (and guards) can invoke methods of objects that provide an interface to the system[10]. However, the context used by actions and guards needs to be populated with such interface objects.

This example back-end integration module would comprise a framework of Connectors and DomainAdapters (again, such functionality is implemented in prototype form, albeit not as a generic extension module). A Connector is a singleton class which implements the functionality of establishing connections to systems or databases[11]. Since establishing a connection is usually time-consuming, a Connector instance manages a pool of connections. A number of connections are established at system startup. Upon a request, an existing connection can be allocated from the pool and released again afterwards.

DomainAdapters are objects that provide domain-specific methods that can be called in order to manipulate or poll a system. (Such methods are sometimes called mutator and accessor methods respectively.) These methods are what guards and actions are expected to call. Each DomainAdapter is instantiated with a connection allocated for its use from the

[10]Of course, these could call any method on any arbitrary available object, but used in this way, they provide a means to manipulate and query the system to which a UI is presented.

[11]A singleton class can only ever have one instance [Gamma et al., 1995].

connection pool held by a Connector.

Upon a request, the back-end integration module would first allocate a connection from the connection pool held by the configured connector. It would then proceed to instantiate a number of DomainAdapters (as per detail attribute specification), all using that same connection, and then it would proceed to populate the context with these instances.

A programmer is thus freed from the technical programming concerns usually involved in connecting and transaction handling—guards and actions can merely call methods on connected instances of DomainAdapters.

After handling a request, the module would destroy the DomainAdapters it instantiated and the module would release the connection back to the connection pool held by the Connector.

The back-end integration module would implement several kinds of Connector classes for different back-end systems. Extending it to be able to communicate with a new database or middleware system amounts to writing a new type of connector class. A lot of functionality for accomplishing this task would already be available for re-use in the framework provided in the back-end integration module.

The current prototype implementation of the functionality provided by the back-end integration module described here only allows a single Connector to be configured per site, but it is conceivable to implement the module in such a way that more than one Connector (to different back-end systems) could be used concurrently.

Support for starting, committing or aborting transactions associated with connections can also be built into the module[12]. This ability is not critical at the moment, since only a single action is ever executed per request (many guards may be executed, but they are not supposed to have side effects). Without such transaction handling functionality, the assumption is that an action would trigger code in a back-end system which automatically handles transaction semantics.

Note that when a back-end system provides defined interfaces such as done by EJB or CORBA, generic DomainAdapters can be written that merely translate a local method call into an EJB or CORBA call. Other systems will need DomainAdapters to be written which can translate every method call on them to, for example, database queries or calls to stored procedures, etc. An entire application could also be run in the address space provided by the UI server by running all of its code from DomainAdapters.

The configuration of the back-end module should specify which Connector should be used and which options it needs to be able to connect to its specific back-end system. The configuration should also specify a name to type mapping of DomainAdapters. The module

[12]In fact, such functionality is part of the prototype implementations.

would use these mappings to determine which type of DomainAdapter it should instantiate and what the name of this instance should be.

It would be wasteful, however, to instantiate every configured DomainAdapter for every incoming request (this what is done in the prototype implementations). Hence, the back-end integration module could export a detail attribute type, "DomainAdapters", which can be used to specify which of the configured DomainAdapters should be instantiated at a particular location. A DomainAdapters detail attribute should include a list of names of DomainAdapters which should be a subset of the names of configured DomainAdapters. Upon a request, the back-end integration module can check for a detail attribute named "domainAdapters" at the current location. Such a detail attribute should be of type DomainAdapters and the module need only instantiate DomainAdapters whose names are listed in the detail attribute found. In the absence of such a detail attribute, it is assumed that the list is empty.

4.3.8 Other possibilities

The extension module implementations discussed so far exclude some features that are typical of web frameworks[13]. These exclusions were made based on the fact that they are not strictly speaking necessary and are in conflict with REST.

However, should it be deemed necessary, these could be added as further extension modules.

For example, one can easily implement a module which maintains a dictionary for general variables in session scope. Such a module can create a dictionary object (named, "session", say) in the context upon each request—populating the dictionary by querying a database at the start of the request cycle and saving its contents to a database at the end of the request cycle. Since such data is private to a particular user session, the module would need to keep track of sessions—this can be implemented in a similar fashion to the session tracker module implementation presented earlier (Section 4.3.5).

It is possible to also implement more interesting types of session state. For example, one may wish to have variables that are visible in the scope of a particular composite location only (analogous to local variables in a subroutine). If a sub-location is entered, those variables would not appear in the session dictionary, but all requests in the same session in the same composite location would have access to the variables set in this scope. This can be implemented as explained in the previous example, with added semantics as to which variable

[13]The prototype implementations also exclude such functionality.

is stored in the session, depending on the current location.

4.4 Configuration vs specification

Care should be taken when adding another extension module. Modules usually need some configuration data, and allow one to extend a Harel specification in a module-specific way. In such a case it is worthwhile to carefully distinguish the added information as being either configuration data or specification data. Specification data should be seen as part of the UI specification in the form of detail attributes. Configuration data is not part of the specification and is stored externally to the specification.

For example, the authentication module needs a specification of:

- how to name its authentication Cookie on a user agent;
- what secret key to use for encryption of the Cookie;
- how old Cookies can become before they expire; and
- which locations need authentication.

Of these bits of information, only the last it taken to be part of the specification of a UI. The other information is not really interesting from the level of abstraction of the specification—these information items concern details regarding how the authentication functionality is implemented. From a specification point of view, it is more interesting to state where authentication is needed. Furthermore, the information in the first three items on this list may change depending on the environment where the UI is deployed. Not every site where a particular system is deployed should use the same secret key, for example.

In the prototype implementation of back-end integration functionality, the careful distinction between configuration and specification items has a useful side effect. Since the mapping between names and types of DomainAdapters is specified as part of configuration, it is possible to use a special configuration in a testing environment which supplies stub DomainAdapters for a system. This allows unit testing of the UI of a system without the need for a real system backing it. The ability to substitute DomainAdapters for stub DomainAdapters (and vice versa) is an extremely useful feature in a large development team, or a development team with diversely skilled developers.

4.5 Deployment

The core framework discussed so far can be used in different environments. The development environment provided by the current prototype is one such an environment (it also acts as a web server for the UI under development); a production environment running a heavyweight web server is another.

The deployment environment and how it is configured can have a drastic impact on the UI. This is a topic in its own right, which is outside the scope of this work. This section provides a quick glance at the topic of deployment environments, in order to show its relevance.

Typically, web server software is geared towards serving high volumes of mostly static data. Web servers usually allow powerful extensions that, for example, let a site use TLS or extensions that allow re-mapping of requested URL by pattern-matching. They typically also provide for plug-ins that would, instead of serving a static file, treat a file as a template which is executed by an interpreter, yielding a dynamically generated page instead.

Typically, web application frameworks have complicated processing requirements and are developed separately from web servers. But they still need some of the functionalities provided by traditional web servers. So, generally a web server is run and configured to forward requests to a web application framework.

There are different ways of doing this, and there are different ways of configuring the web server itself.

For example, the Apache web server [The Apache Software Foundation, 2005] can be configured to either:

- run a single process, handling concurrent requests in concurrent threads (which it keeps in a pool);
- run a pool of single-threaded processes that concurrent requests are farmed out to; or
- run a pool of multi-threaded processes—a combination of the previous options.

Added to that, such an installation can be duplicated on several machines.

One of the prototyped web frameworks, providing the run time implementation of Harel, is implemented in Python. Python is an interpreted language. Using an Apache plugin module[Trubetskoy, 2005], one can run a Python interpreter inside an Apache process.

In such a scenario, the Apache configuration impacts the design of the UI—at least with regard to how some modules are implemented.

The back-end integration functionality, for example, contains implementations of Connectors that open connections to databases. If a Connector implementation needs to open a file exclusively, but several instances of it are being run concurrently in several distinct processes on the same machine, only one will be able to lock the opened file.

Also, it would be wasteful to let each Connector keep a pool of database connections open if it is deployed in a number of single-threaded Apache processes—one connection per process would be sufficient, since only one request will be handled at a time per process. In effect, the pool of processes (each with a Connector which keeps only one connection) play the role of a pool of connections.

Code in multi-threaded processes need to be thread-safe. While the presented implementation is meant to be thread-safe, code supplied in DomainAdapters should also be thread-safe if they are to be deployed in a multi-threaded environment.

A module implementing session state needs to carefully evaluate its possible deployment environment. It could not keep session state in memory when deployed in several processes (or in local files when deployed across several machines), since a new request in the same session can be routed by load-balancing software to another process, or another machine entirely which may not share memory, or a file system with the process that handled the first request.

Another deployment option in Apache is to start a multi-threaded process (which runs a framework) separately to Apache. Apache can then route requests to it from its various processes. Several standards for this kind of communication also exist, the most well known of these probably being the Fast CGI (FCGI) [Brown, 1996] protocol.

The deployment options far exceed the few mentioned here, but these should give the reader the general idea. The topic is broad and the concurrency model best used by a scalable web server is widely debated. For example, how much sense does it make to run an Apache web server as several processes, but funnel all request handling to a single multi-threaded process running a web application framework? Or, which is the best concurrency model: many single-threaded processes, or a single multi-threaded process? A small number of web servers also claim superior performance using an asynchronous polling model, mentioned in Section 2.1.8.

Performance and scalability are directly influenced by the particular system under scrutiny—its own design as well as the deployment options of its web-based UI. It is difficult (and dangerous) to argue generally about these topics, outside the context of a specific system. Suffice it to say that a particular implementation should be very flexible when it comes to supporting several different deployment options.

The proposed implementation (and the implemented prototypes) can be deployed using mod_apache, FCGI, and a proprietary web server built into the GUI tool.

4.6 Summary

In this chapter, it is shown how a framework could be implemented which supplies an execution environment for Harel specifications. In order to do this, some background is first provided: relevant web standards are explained which provide the REST-compliant platform of the web (Section 4.1); and several current or recommended practices for implementing common functionality provided by web frameworks are also explained (Section 4.2).

With that important information as background (as well as the defined semantics of Harel from Section 3.3), the design of a framework is proposed for executing Harel specifications (Section 4.3). The designs of some extension modules which would provide functionality expected of a realistic framework are discussed in Sections 4.3.3–4.3.8.

Sections 4.4 and 4.5 briefly provide overviews of related topics outside the scope of this work—the role of configuration and specific deployment environments, respectively.

The design presented in this chapter is heavily based on two prototype implementations that have been developed (one in Java, and one in Python). Many insights presented have been gained via the iterative development process of the prototype implementations (and experiments in using the prototypes).

In Chapter 5, a more detailed overview is given of related work in the literature. A critical discussion of the present study is given within this context, and a conclusion is presented.

5 Discussion and conclusion

The purpose of this chapter is to contextualise and assess the work discussed in previous chapters. It begins (Section 5.1) with an overview of related work that has not been introduced thus far. This is followed by a discussion of Harel within the context of such related work in Section 5.2. In Section 5.3, problem areas and opportunities which warrant more work are highlighted. Finally, a conclusion is presented in Section 5.4.

5.1 Related work

Perhaps the work most closely related to this study is the work pertaining to web frameworks and their implementation. Since descriptions of these frameworks are not well represented in the formal academic literature, a detailed overview of web frameworks was given in Chapter 2—the result of a survey of some 80 of these frameworks.

Other closely related work has been introduced at the relevant times during the course of this dissertation.

In this section, a brief overview is given of some work related to the modelling of web applications, with a focus on work that employs statecharts (or similar notations).

5.1.1 ArgoUWE and related approaches

UML-based Web Engineering (UWE) is an example of a methodology for the systematic development of web applications, supported by a Computer Aided Software Engineering (CASE) tool, ArgoUWE in this case. An overview of the former is given in Koch and Kraus [2002], the latter is presented in Knapp et al. [2003]. UWE is a model driven approach to web application development. UWE defines the notation and specific models that are specified during development, as well as the process and relationship between the models. ArgoUWE provides the tool support with which those models can be built and transformed. ArgoUWE eventually can generate a web application semi-automatically.

For the navigational model of a web application, UWE follows roughly the same strategy as Conallen [1999]. This discussion of UWE serves as an example of such a strategy.

UML can be extended by the addition of stereotypes, tagged values and constraints. A stereotype in UML is a named classification that can be attached to an existing UML modelling element (such as a class). A stereotype adorns the element to which it is attached and provides that element with additional semantics. A stereotype thus provides a means to further refine the semantics of an existing element in the UML meta-model. Tagged values are key-value pairs that can be attached to a modelling element. Constraints are rules that can be used to further specify semantics and well-formedness of a stereotyped model. (These methods of extending UML are discussed in the present context in Conallen [1999] and in more detail in Koch and Kraus [2002]. The interested reader is also referred to OMG [2003].)

For the purposes of this study, it is adequate to understand that using these intended extension mechanisms of UML, UWE provides an extension to UML class diagrams in order to be able to model the navigation of a web application. (In fact, two navigational models are proposed.) Several stereotyped classes are defined, representing (amongst others) the logical locations in a web site between which a user could navigate. Stereotyped relationships between such classes indicate possible navigation paths a user could take. Such a stereotyped class diagram results in a directed graph.

The navigational models of UWE are not intended to model dynamic aspects of the web application. In Koch and Kraus [2002], statecharts are also explored for specifying more dynamic aspects of a web application. (The authors also explore the use of activity diagrams and collaboration diagrams for this purpose.) Statecharts are used to model "web scenarios", not entire web sites. In such a model, states are also named after presentation classes that would be displayed in a web browser. From Koch and Kraus [2002], it is unclear how this model would contribute to the eventual generation of the web application.

Although not an extension of UML, Ceri et al. [2000] propose a similar type of navigational model as part of WebML. Here also, a number of different nodes are defined representing a number of entities. Some of these entities are web pages displayed to a user, some related to certain processing steps, for example. Directed edges between nodes represent possible navigational paths.

5.1.2 OOHDM

Object-Oriented Hypermedia Design Model (OOHDM) is also a model-based approach for developing web applications.

In Güell et al. [2000], derivations of statecharts are used in "Abstract Data View (ADV)-charts" and "navigational charts".

ADV-charts specify more details regarding the appearance and behaviour of, for example, a particular node in the navigational model (i.e. a logical location in the web site).

Navigational charts add to the navigational schema by allowing sets of navigational nodes to be active at the same time. (For example, a user may visit two logical locations simultaneously by viewing them in separate frames in a browser. Statechart notation can also be used to specify how such concurrently displayed locations should be synchronised.)

Both ADV-charts and navigational charts extend statechart semantics extensively for their very specialised domains.

5.1.3 HySCharts and HMBS

Hyperdocument Model based on Statecharts (HMBS) is a "statechart-based, navigation-oriented model for hyperdocument specification" [Turine et al., 1999]. With the tool, Hyperdocument System based on StateCharts (HySCharts), HMBS models can be created and a user can also use HySCharts to browse the hyperdocument represented by such a model.

HMBS models use a simplification of statechart notation, with states mapped explicitly to pages, and transitions to navigational links between states. HMBS is not geared for web applications per se, but for hyperdocuments that may be viewed using HySCharts. Hence, the notation does not include support for triggering actions while traversing such a hyperdocument. It has no notion of a server or an application that needs to be executed, the aim being that of specifying a hyperdocument which can be browsed and navigated. An HMBS model can be used to generate static HTML web pages with navigation implemented as HTML anchors.

5.1.4 StateWebCharts

StateWebCharts (SWC) is a statechart-based formalism for specifying the navigational model of web applications [Winckler and Palanque, 2003]. SWC is presented as an abstract navigational model of web applications without reference to how such a model is to be executed.

SWC is an extension of traditional statecharts. Different kinds of states are introduced—some, for example, representing a page displayed in a browser and some representing an operation that is performed by the application. A distinction is also made between different kinds of transitions—depending on the agent (user or system) that triggered the event which caused a transition.

This strategy of extending statechart notation and semantics is similar to the approach followed by an existing framework—*SpringWebFlow* (as discussed in some detail in Section 2.3.11).

5.1.5 Leung et al. [2000]

In Leung et al. [2000], the authors indicate how statecharts can be used to specify the navigational model of web applications in the same vein as Turine et al. [1999].

The web technology available today provides web application builders with a large number of building blocks. For example, Java applets can be used in a client browser, a browser can display more than one frame at a time, or platform-dependent technology such as ActiveX can be embedded into web pages.

In Leung et al. [2000] an attempt is presented to model the navigational aspects of a web application that may potentially use any such technologies and/or methods. States are mapped to the pages that are displayed by a web browser, and concurrent sub-states depict multiple pages or parts of a page that are concurrently displayed by a browser. The emphasis is on a navigational model, so transitions are directly related to hyperlinks (or similar mechanism). No mention is made of triggering actions on the server.

5.1.6 Gorshkova and Novikov [2004]

Another statechart-based approach to the navigational modelling of web applications is given in Gorshkova and Novikov [2004]. Statechart semantics is extended by introducing different kinds of states, but all of them are related to what can be displayed at a client browser. (Other approaches that extend statechart notation and semantics usually also include types of states depicting server-side operations and concepts not related to what is being displayed at a client browser.) A distinction is also made between events based on which agent (the user or the system) initiated the event.

In contrast to other statechart-based approaches, in Gorshkova and Novikov [2004] the UML notation for statechart diagrams is followed strictly, and stereotypes are used for the extension of UML as is done with class diagrams in Koch and Kraus [2002].

Although Gorshkova and Novikov [2004] recognise the availability and, to some extent, usefulness of actions in statecharts, actions are explicitly excluded because of the focus being navigational modelling and not dynamic behaviour.

5.1.7 Models for form-oriented analysis

Form-oriented analysis is a "holistic approach for engineering form-based, submit/response style systems" [Draheim and Weber, 2005]. As part of this approach, several modelling techniques are proposed. In Draheim and Weber [2005], a proposal is made for modelling "form-based UI", using screen diagrams, form storyboards and formcharts.

Screen diagrams are very similar to Harel, in that they depict pages shown to a user as nodes, with transitions between them indicating how a user could move between such pages. Screen diagrams do not use statechart notation—they include no concept of composite states, guards or actions. What is interesting, though, is that they include the notion of "conditional transitions". Conditional transitions are transitions that branch—based on a condition they would lead to one of several target locations (analogous to an `if` statement).

Form storyboards have two kinds of state: a state representing a page presented to a user, and a state representing a server-side operation. Form storyboards also do not make use of statechart concepts, such as guards and actions. In order to be able to decompose a large model into manageable, smaller parts, the notion of sub-storyboards is introduced. Sub-storyboards are storyboards that contain a subset of the states and transitions of the final UI. Such a sub-storyboard is a named entity which is meant to describe a conceptual part of the whole UI. A final UI is composed by the unions of the node and edge sets of all the sub-storyboards it comprises. A shorthand notation is also introduced for showing several states as one node, in order to be able to specify one transition leaving or targeting all of the states in the set represented by the "state set".

Both form storyboards and screen diagrams are proposed as informal notations used in facilitating various stages during design discussions and the like. Given this background, a set of related models are proposed which are meant to provide a rigorous specification of a software system: formcharts, an information model of data, and the user message model. Explaining the details of these models would not be productive here—it is enough to know that formcharts are basically form storyboards with some information removed. This information is then specified more completely in the user message model.

What is interesting, though, is that although form storyboards are presented as an informal notation, the authors provide a textual language, called Angie, with which form storyboards can be expressed. Using a compiler-like tool (part of Angie), an executable web UI can be generated from an Angie specification in terms of JSP pages and Java Servlets.

5.2 Discussion

In Section 1.6 the most important high-level differences are pointed out between the present work and related work in the literature. Although very relevant at this stage, it will not be repeated here—the interested reader is referred back to Section 1.6. However, some more detailed differences can now be pointed out.

Several authors distinguish between more than one kind of event based on how an event is initiated—events initiated by the user, or by the system. In this work, no such distinction was made. The execution environment presented by the web is quite specialised. Execution can occur on a client in a browser (when using JavaScript), or on the server. The latter can only take place upon a request from the client browser. Hence, a server cannot initiate events; only a user can, by clicking on buttons or links rendered by a browser—these result in requests to the server upon which processing can take place. The implication is that only one type of event is necessary: those initiated by a user.

To allow for more than one type of event seems unnecessarily complex. This complexity is necessitated in many approaches by their policy of representing server side operations as states (or components of states). (See the following paragraph for a motivation of this statement.) In the approach proposed in this study, states always represent pages presented to a client and actions correspond to server-side operations. Events are always initiated by a client and triggered by the server receiving a resulting request.

To see how representing server-side operations as states cause the necessity for an additional complication in the form of a second type of event, consider the following scenario. Assume a specification with three states: A, B and C. A and C are pages to be displayed to the user, B is a server side operation. Assume further that there is a transition from A to B for the event "submit", and a transition from B to C for the event "success". The scenario starts with page A being displayed to the user. The user now clicks on a button which results in the "submit" event being triggered. The web browser thus has to make a request to the server indicating that this event has taken place. In response to such a request, the web server can do some processing, but ultimately *has* to produce the next page that should be displayed. So, in our scenario, the server should execute the operation B and then respond with instructions to the browser to display C (assuming the execution of B was successful). This is done by interpreting the result of executing B (often called an outcome by framework designers) as an event—the event that B has completed, with a particular result. Depending on the result of the operation, then, the server can infer which page to display in response to the request. A model needs to include a specification of which page to display after having done an operation—with operations as states, it seems logical to model this information using another transition (from the operation to the next page to be displayed). But, according to the semantics of statecharts, a transition is triggered upon the occurrence of an event. This is why the completion of an operation is modelled as an event, and this is the source of the “server initiated” events. (Examples of modelling operations as states and the completion of operations as events can be seen in Gorshkova and Novikov [2004, Figure 3, p48] and Winckler and Palanque [2003, Figure 8].)

Approaches that model server-side operations as states provide a flexibility to modellers that is not currently catered for in Harel. In the latter, the result of an operation that was triggered in response to an event initiated by the user does not determine the page to which a transition is to be made—this decision is already made when a transition is chosen to fire (i.e. before the an action is executed). By depicting a server-side operation as a state, the decision as to which page should be shown next can be deferred and made based on the outcome of the operation.

What is provided by Harel in its present form is intentionally restricted by a conservative policy: the first aim is to finalise a complete framework for the simplest notation based on statecharts and REST. Further complexities can be added later, based on experience gained. The flexibility mentioned here can also be catered for in Harel using notation and semantics from UML statechart diagrams. For example, “dynamic choice points” can be used together with guards to specify transitions that branch depending on certain conditions [OMG, 2003, part3, p151]. (Dynamic choice points are an example of “choice pseudostates”, defined in OMG [2003, part2, p146].)

Apart from states representing displayed pages and server-side operations, authors often define other kinds of states (for example, Winckler and Palanque [2003]). The proposal in this study strictly maps states to REST resources (Section 3.3.1.1) (informally, thus, logical locations in a web UI). The preceding discussion motivates this decision to some degree. Another reason why this was done, however, was that it has been the specific aim of the work presented in this study to remain aligned with the intention of the web standards as put forth by Fielding [2000].

Several authors also use more detailed elements of statechart notation and semantics for specifying the contents of a page, as opposed to merely indicating the relationship between pages. A prime example of such usage is when concurrent sub-states (in UML parlance) are used to show components of a particular page or pages shown concurrently in different frames [Leung et al., 2000, Gorshkova and Novikov, 2004, Turine et al., 1999]. The present study is only concerned with the relationship between pages and the dynamic behaviour that can be specified at this level. It does not attempt to model the contents of a page—deferring those details to other well-developed specification techniques (such as JSP). Statecharts would appear to be about modelling dynamic behaviour, not presentation. Without an extension like JavaScript, HTML (the essence of a page) does not have any dynamic behaviour. Hence, the practice of using, for example, concurrent sub-states for modelling the contents of a page is slightly suspect (except where these use Javascript and influence behaviour). It is, however, tempting to suggest a means of modelling the contents of a page, using another technique which can take advantage of the statechart model. This, as well as taking a closer look at when statechart notation is applicable and when not, are directions for future work.

The focus of most related proposals in the literature is on the navigational modelling of a web application (or hyperdocument). Two effects of this focus are that: dynamic behaviour of the web application is excluded; and transitions are strongly correlated with hypertext links. The proposal in this study is specifically aimed at specifying dynamic behaviour—hence the liberal use of actions and guard conditions. Also, transitions are not as strongly related to hypertext links. The specification of the URL to which a form will be submitted, for example, is required by Harel to be the location (state) which contains the form.

When submitted, an event is inferred and the transition then chosen to fire indicates which next page to display to a user and which action to execute. The transition is thus effected by means of server-side execution semantics, not by HTML links. Links can also be used to provide a user with a means to cause a transition to another page. Such a transition, however, does not need server intervention in order to fire—the browser merely requests the new page. The effect of this is that the server cannot govern the firing of the transition using guard conditions, and no actions can be executed. This suggests that Harel could be extended with a new type of transition which explicitly embodies such constrained semantics.

The explicit mapping in this study to concepts of REST has several desirable effects. For example, the built-in operations of browsers which are based on the semantics of HTTP and the URI specification stay functional, whereas with other frameworks these are often a problem[1]. The properties induced in the web by the constraints of REST (like scalability) are extended to the web application UI. For example, the fact that little information—if any at all—needs to be held in "session state", delegates issues relating to the scaling of a web application to an application or database server. By adhering to REST, Harel's basic model is standards-based, meaning that it will work with the most basic set of standards available. Designers can thus use optional technologies such as JavaScript, but within the

[1]Related to this, some may criticise Harel's adherence to the PRG pattern for the extra round-trip that it implies. However, the PRG pattern was leveraged precisley in order to implement REST semantics. Its use is the price one pays in order to adhere to REST.

framework provided by Harel, resulting in a UI that degrades gracefully in the absence of optional technologies.

5.3 Future work

As is evident from the discussion in Section 5.2, the specification technique presented in this study is a novel (and simple) application of statechart notation and semantics towards specifying a dynamic, web-based UI. The discussion in Chapter 4 shows that such a specification can be rendered executable in the environment of the web.

There are many possibilities for improvement and further work, though. Many of these possibilities can be explored more effectively once Harel has been tested on real-world applications.

- Transitions are not mapped to hypertext links. Instead, they represent more generic ways of navigating between pages, such as a decision made by a server as to which page to display next upon receipt of a UI event. However, some transitions may be implemented as links—implying that their semantics should be weaker, since such transitions cannot have guard conditions or actions. A possibility is that a second type of transition can be added to explicitly model these transitions.

- No mention is made of how the UI would be different for different kinds of users (users with different roles). This is a requirement of many web applications (often not explicitly modelled). Several possibilities exist for adding this dimension to the specification. For example, the Harel specification as described in this work can be seen to reflect a complete site map. Separate masks for each user role could be defined which would restrict this map depending on the roles of different users. Depending on the technology used for generating individual pages, pages can also be rendered differently for different users. (For instance, what one user sees as a text input box, another may see as static text.)

- In this study the specification of how each dynamic page would be generated is explicitly excluded. Although many techniques are available for this purpose, novel ways are possible within the framework and semantics provided by statecharts. It could be interesting to venture into this domain and explore such options. (In fact, the prototype frameworks include such experiments already.)

- The Harel notation does not include a means to specify the "formal details" expected by a UI (Section 3.3.1.3). The specification of this aspect of a UI is currently done (i.e. in the prototype implementations) by means of a window presented by the editing tool. The incorporation of this aspect in the notation is not a mere oversight—incorporating it is difficult to do elegantly, since the notation does not include syntax for UIs. Further work is needed to solve this problem.

- Modularisation is currently done by packaging a composite location as a UI. Users of that UI can only use the single top-level composite location of such a UI. It may be productive to allow more entry points to such a UI—for example to let users call sub-locations directly. (UML statechart diagrams include notation for this kind of usage.) Another possibility is to package more than one composite location together on the same level in a module.

- As explained in the preceding discussion (Section 5.2), practical experience with Harel may show that more flexibility is needed in the form of transitions that can branch to different states, depending on certain conditions. It may then be necessary to investigate the addition of UML statechart diagram notions, such as the choice pseudostate with which dynamic conditional branches can be specified for transitions [OMG, 2003, part2, p146].

- Connections to back-end systems can be explored further. It may be useful to allow several connectors concurrently to different back-end systems, thus allowing the UI to be an integrated UI to many back-end systems. In line with this, transaction handling may be controlled implicitly from the web UI.

- More support can be added for development activities, such as automated testing or refactoring of UIs.

- The use of AJAX techniques can decrease the number of necessary web pages in an application and result in a more responsive UI. It is worthwhile to investigate the incorporation of such techniques into the current Harel proposal.

5.4 Conclusion

The widespread availability of the Internet has made it appealing as a medium by which organisations (or individuals) can extend the reach of their systems to a vast audience at remote locations—often into the homes of users. Many architectures are possible for such systems, ranging from fat clients and peer-to-peer systems to thin clients with responsibilities restricted to presenting a user interface for a client at a remote location.

Web applications leverage the widely deployed and standardised infrastructure of the web to deliver their UIs. These applications have enjoyed widespread success in recent years.

However, using web technology as a UI for a general application has its drawbacks. The presented UI is constrained by what can be presented via HTML (or newer, related standards) and by the somewhat peculiar characteristics of the HTTP protocol. This results in UIs that are not as rich as traditional GUIs and are often seen as unresponsive. Moreover, the web was not originally intended to be used as a UI delivery mechanism for applications—the effect is that such web development tends to be complex.

These difficulties may prompt one to ask why designers concern themselves with the awkward platform (from the perspective of a UI designer) provided by the web standards. Other approaches are available with which a UI can be delivered at a remote location. For example, Rich Internet Applications (RIAs) is an emerging form of application, the evolution of which is driven by the need to be able to execute part (or more) of an application at a client without the tedium related to installing an application at many clients, while presenting a more traditional GUI. The X Window system™ has been remotely delivering full-featured GUIs on UNIX™ systems for years. Why, then, do so many choose to solve this problem using the web despite its peculiar UI capabilities and different intention?

The answer may well lie in the fact that the web has proven to be operating well (and scaling well) in the environment represented by the current Internet. Participants in the web span not only several different platforms, but also different organisational boundaries. The widespread availability of web browsers implies a vast audience. The web has triumphed in this hostile, anarchic environment with its particular solutions to issues such as security, congestion, scalability and reliability.

Web frameworks dictate the low-level specification of web-based UIs. They are responsible for executing such UIs using the web as platform. However, it would seem that such frameworks are not well represented in the literature. Moreover, application designers wishing to exploit the web for delivering the UIs of their applications tend to be more concerned with *what they need in order to be able to present good UIs*, and less concerned with *why they are using the web standards* (which are "awkward" from a UI point of view) in the first place.

Many web framework designers appear to be ignorant of the philosophical underpinnings of the web standards and often strain against the implicit constraints. In this study a different view is explored—that the constraints of the web standards should be embraced and built upon, since they are the concrete result shaped by the intention behind the web standards. And this intention is the driving force behind the success of the web—the very reason for the choice of delivering a UI via the web in the first place.

With this particular bias, this study makes the following contributions:

- In Chapter 2, topics related to web frameworks—that have escaped the academic literature—are discussed. In Section 2.1, an enumeration of what is typically seen as being required of a web framework is presented. The rest of Chapter 2 is devoted to an overview of specification techniques employed by web frameworks in the form of two taxonomies of the strategies used.

- In Chapter 3, a proposal is made to apply the well-known formal notation and semantics of statecharts towards specifying the dynamic behaviour of web-based UIs. Although not yet empirically tested in the field, the *prima facie* case has been made that the resulting graphical language will simplify the development of web-based UIs and that it will greatly improve the visualisation of a web-based UI design.

- Contrary to other approaches, the semantics of this application of statecharts is carefully mapped to the architectural concepts put forth as REST, resulting in a specification with semantics that is consciously informed by the motivating intention behind the architecture of the web.

- In Chapter 4, an explanation is given of how such a statechart-based specification can be executed on the platform provided by the web, serving as *a priori* evidence that this approach is realistically implementable.

The particular architectural solutions used by the web have been proven in the challenging environment of the present-day Internet. Web application designers have the benefit of being able to build on this foundation. The aim of this study has been to derive a realistically implementable system that is consistent with the sound philosophical underpinnings guiding the architecture of the web.

Building on prior experience gained during the development of prototype systems, as well as on information gleaned during this study, the short term goals for this work are to provide a fully operational framework with the aforementioned philosophical bias, to test it on a number of real-life applications, and then to release it as an open source product.

Glossary

Note that the glossary is constrained to terms that are used in the text with the (possibly erroneous) assumption of reader familiarity, or terms that are used with specialised meaning in the context of this dissertation. Terms introduced during the course of this dissertation are not listed in the glossary, since they can be explained far better in the text.

blog The term blog is a shortened form of web log: an online publication on which short articles can be posted. Blogs are often used by individuals in the computer industry to publish their own opinions and knowledge.

call back method Programming libraries, such as web frameworks, sometimes need to invoke user-supplied code at predefined locations in their execution. One approach for doing this is to supply the library with a method (supplied by an application programmer) which library code would call, for example, upon the occurrence of a particular event. Such supplied methods with user code are called call back methods.

control flow Control flow refers to how the order of execution of individual statements in a program is controlled. Statements can typically follow one another sequentially, be repeated, or a program can execute different statements depending on the value of one or more expressions.

dynamic content Dynamic content is a term often used by web developers to refer to web pages (or parts of them) that are generated upon request by a web server.

form The term form, as used in this dissertation, refers to an HTML form (or equivalent). Forms are presented by browsers as web pages on which a user can supply input using a variety of methods. When such input has been supplied, a form can be submitted to a web server for processing.

graceful degradation A system is said to degrade gracefully if it continues to operate in the face of failures of some of its parts, losing functionality in proportion to the quantity of failures. In the web context, graceful degradation is often used to describe the ability of a web site to continue operation even though it depends on optional extended components (such as JavaScript). In the absence of such optional components, a site which degrades gracefully would merely lose some of its characteristics and not cease working entirely.

locale A locale is a set of user preferences, such as language, currency and time zone, which can be used by UIs to provide a presentation tailored to a particular user.

name space A name space is an abstract container for identifiers which represent entities of some kind. In this dissertation, the term is used in connection with identifiers for REST resources. A hierarchical name space is a name space that allows structuring identifiers in a tree.

navigation model The navigation (or navigational) model of a web site is a model representing how a user can navigate between the hypertext pages of the site using hypertext links and similar methods.

page flow Page flow is a colloquial term used to refer to the relationship (often navigational in nature) between different pages of a web-based UI. The term is analogous to control flow (which concerns the relationship between different programming language statements).

regular expression A regular expression is a pattern which describes or matches a set of strings. Regular expressions are often used to search bodies of text for certain patterns.

security realm The term security realm refers to an area for which certain security restrictions apply. For example, a web site could be a security realm—a user can log into the site as a whole with a single user name and password and be granted access to all pages in the site based on that initial authorisation. Security realms sometimes span collections of systems.

session scope A user session with a web server usually spans several HTTP requests to the server. Session scope refers to a server side context, which persists across multiple HTTP requests, and in which variables private to a particular user are stored.

session state Session state refers to all the information stored in session scope for a user.

socket The term socket, as used in this dissertation, refers to a programming abstraction introduced by the BSD operating system and widely in use in network programming. A Berkeley socket is a file-like abstraction denoting one endpoint in a TCP connection (or the address of sender or receiver of UDP packets).

standard output Standard output is a communications channel wrapped in a file-like abstraction for programming language usage. Many operating systems supply a standard output (opened for writing) to processes. Output written to the standard output of a process would then be handled in a predefined way: for example, the standard output of a program run on a terminal in Linux would be written to the terminal as output for the user.

statechart The formalism of statecharts can be used to describe the dynamic behaviour of a system, based on a FSM. State charts were introduced by Harel [1987].

stateless In this dissertation, the HTTP protocol is often described as being stateless. It may seem strange to call any protocol stateless: what is meant is that state is not maintained between different requests to a server. Every new request is independent of others.

(The alternative would be to require one kind of request—such as a handshake—to happen before another kind of request is allowed in a conversation.)

transaction The term transaction is used in this text to mean an atomic transaction: a number of actions that either occurs completely, or not at all.

user agent A user agent refers to a web browser or similar program which acts on behalf of the user as the one endpoint in an HTTP-based conversation.

web application A web application is defined in this dissertation as an application that presents its UI via the web.

web framework A web framework is an architectural framework which provides an execution environment for web applications or the UIs of web applications. Such an architectural framework usually comprises server components as well as supporting programming language libraries (which also aid users of the framework in their development task).

web standards The term web standards is used in the text to refer to the main standards enabling web technology: HTTP, URI and HTML (or similar).

web UI The term web UI is used to mean a UI which is presented via the web.

Abbreviations and acronyms

BSD	Berkeley System/Software Distribution
AJAX	Asynchronous JavaScript technology and XML
ECMA	European Computer Manufacturers Association
FSM	Finite State Machine
Sun	Sun MicroSystems™
JSP	JavaServer Pages™
JSF	JavaServer Faces™
JSTL	JSP Standard Tag Library™
J2EE	Java 2, Enterprise Edition™
JNLP	Java Network Launching Protocol, also known as Java Web Start
EJB	Enterprise Java Beans™
PHP	PHP Hypertext Preprocessor
PRG	Post-redirect-get, also known as Redirect after post
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPv4	IP, version 4
DNS	Domain Name Service
URL	Uniform Resource Locator
URI	Uniform Resource Identifier
URN	Uniform Resource Name
HTTP	Hypertext Transfer Protocol
S-HTTP	Secure HTTP

HTTPS	HTTP over TLS
HTML	Hypertext Markup Language
XHTML	Extended Hypertext Markup Language
XML	Extensible Markup Language
CSS	Cascading Style Sheets
XSLT	eXtensible Stylesheet Language Transformations
UI	User Interface
GUI	Graphical User Interface
MVC	Model-View-Controller
RIA	Rich Internet Application
REST	Representational State Transfer
SSL	Secure Socket Layer
TLS	Transport Layer Security
IO	Input/Output
WYSIWYG	What You See Is What You Get
W3C	World Wide Web Consortium
WWW	Word Wide Web
OMG	Object Management Group
OO	Object Orientation
UML	Unified Modelling Language
TCP/IP	TCP over IP
TCP	Transmission Control Protocol
IP	Internet Protocol
UDP	User Datagram Protocol
IIOP	Internet Inter-ORB Protocol
ORB	Object Request Broker

CORBA	Common Object Request Broker Architecture
FCGI	Fast CGI
CGI	Common Gateway Interface
WSGI	Python Web Server Gateway Interface???
ASP	Active Server Pages
UWE	UML-based Web Engineering
ADV	Abstract Data View
OOHDM	Object-Oriented Hypermedia Design Model
HySCharts	Hyperdocument System based on StateCharts
HMBS	Hyperdocument Model based on Statecharts
SWC	StateWebCharts
CASE	Computer Aided Software Engineering

Surveyed frameworks and related projects

Note that the projects surveyed are to a large extent open source projects (precisely for the reason that the designs of these are easily accessible). The effect is that authoritative information on these projects is web based. In this bibliography of surveyed projects, the project web site (or equivalent) of each project is listed by project name. Each entry in the list contains the title of the referenced web site. This is followed by one of: the copyright holder or author of the project; or, the copyright holder of the web page (if such information is available). Each entry ends with the URL of the project web site. All these URLs have been referenced within the period of January to August 2005.

ActionServlet ActionServlet. Petr Toman and Mark D. Anderson.
`http://www.actionframework.org`

Albatross Albatross—a Toolkit for Stateful Web Applications. Object Craft P/L.
`http://www.object-craft.com.au/projects/albatross`

AppServer Faces AppServer Faces. Dataosoft.
`http://www.dataosoft.com/asf`

Aquarium Aquarium.
`http://aquarium.sf.net`

ASPy ASPy—Active Server Python. Bradley Schatz.
`http://archive.dstc.edu.au/aspy`

Barracuda Barracuda Presentation Framework.
`http://barracudamvc.org`

Beehive Beehive. The Apache Software Foundation.
`http://incubator.apache.org/beehive`

Bento Bentodev.org—Home of the Bento Language. bentodev.org.
`http://www.bentodev.org`

Bishop Bishop. Johan Redestig.
`http://bishop.sourceforge.net`

Castalian Castalian. Stuart Langridge.
`http://www.kryogenix.org/code/castalian`

Cheetah Cheetah—The Python-Powered Template Engine. Tavis Rudd.
`http://www.cheetahtemplate.org`

CherryFlow CherryFlow.
`http://subway.python-hosting.com/wiki/CherryFlow`

CherryPy CherryPy—a pythonic, object-oriented web development framework. CherryPy team.
`http://cherrypy.org`

Chrysalis Chrysalis. Paul Strack.
`http://chrysalis.sourceforge.net`

Cocoon The Apache Cocoon Project. The Apache Software Foundation.
`http://cocoon.apache.org`

DWR DWR—Easy AJAX for Java. Joe Walker and Getahead.
`http://www.getahead.ltd.uk/dwr`

Echo NextApp . Echo. NextApp Incorporated.
`http://www.nextapp.com/products/echo`

ECS Jakarta ECS—Element Construction Set. The Apache Software Foundation.
`http://jakarta.apache.org/ecs`

EmPy EmPy. Erik Max Francis.
`http://www.alcyone.com/software/empy`

Expresso Expresso Framework Project. Jcorporate Limited.
`http://jcorporate.com/expresso.html`

FormKit dAlchemy — FormKit : A webware form library. dAlchemy, Incorporated.
`http://dalchemy.com/opensource/formkit`

FreeMarker FreeMarker. The FreeMarker Project.
`http://freemarker.sourceforge.net`

HTMLgen HTMLgen. Robin Friedrich.
`http://starship.python.net/crew/friedrich/HTMLgen/html/main.html`

Jaguar Jaguar for Python: lean, fast and mean. Terrel Shumway.
`http://jaguar.sourceforge.net`

JBanana JBanana. JBanana.
`www.jbanana.org`

JonPy Jon's Python modules. Jon Ribbens.
`http://jonpy.sf.net`

JPublish JPublish. Anthony Eden.
`http:///www.jpublish.org`

JSPWidget SPWidget Open Source Project.
http://edu.uuu.com.tw/jspwidget/default.jsp

Jucas Jucas object oriented pull MVC Web-Framework. Jucas.
http://jucas.sourceforge.net

Karrigell Karrigell. Pierre Quentel.
http://karrigell.sourceforge.net

Keel Keel Framework. Keel Groups Limited.
http://www.keelframework.org

Maverick Maverick. Infohazard.org.
http://mav.sourceforge.net

Melati Melati—Java SQL Website Development Engine. PanEris.
http://www.melati.org

Millstone Millstone—Free Open Source Web UI Library and Reusable Components for J2EE and Java. IT Mill Limited.
http://millstone.org

Nevow Nevow: A Web Application Construction Kit. Donovan Preston.
http://www.divmod.org/projects/nevow

Niggle Niggle Web Application Framework. Jonathan Revusky.
http://niggle.sourceforge.net

Myghty Myghty—High Performance Python Templating Framework. Michael Bayer.
http://www.myghty.org/index.myt

PEAK.web PEAK—The Python Enterprise Application Kit. Phillip J. Eby.
http://peak.telecommunity.com

PHP PHP: Hypertext Processor. The PHP Group.
www.php.net

PMZ PMZ—Poor Man's Zope. Andreas Jung.
http://pmz.sourceforge.net

PSP Python Server Pages (PSP). Angell Enterprises, Inc.
http://www.ciobriefings.com/psp

PyHP Python Hypertext Preprocessor. Lethalman and Christopher A. Craig.
http://freshmeat.net/projects/pyhp

PyML PyML—Python HTML Pre-Processor. David Snopek.
https://gna.org/projects/pyml

PyServ PyServ—a servlet-like engine for Python. Steve Purcell.
`http://pyserv.sourceforge.net`

Python Pages Ramu's Python Page. Ramu Chenna.
`http://www.embl-heidelberg.de/~chenna/pythonpages`

pyWeb pyWeb HOME. David McNab.
`http://www.freenet.org.nz/python/pyweb`

PyWX PyWX: Python for AOLserver.
`http://pywx.idyll.org`

Quixote Quixote. Corporation for National Research Initiatives.
`http://www.mems-exchange.org/software/quixote`

RIFE RIFE: About. The RIFE team.
`http://rifers.org`

Roadkill Roadkill—embedded Python.
`http://roadkill.sourceforge.net`

RubyOnRails Ruby on Rails. David Heinemeier Hansson.
`http://rubyonrails.com`

Seaside Seaside. Avi Bryant.
`http://seaside.st`

Sitemesh SiteMesh. OpenSymphony.
`http://www.opensymphony.com/sitemesh`

SkunkWEB SkunkWEB — News. Drew Csillag et al.
`http://skunkweb.sf.net`

Smarty Smarty : Template Engine. New Digital Group, Inc.
`http://smarty.php.net`

Smile Smile, the open source JavaServer Faces implementation. Dimitry D'hondt, Edwin Mol and Steve van den Buys.
`http://smile.sourceforge.net`

Snakelets SNAKELETS—Python Web Application Server. Irmen de Jong.
`http://snakelets.sourceforge.net`

Sofia Sofia—JSP GUI Java Development Framework. Salmon LLC.
`http://www.salmonlcc.com/sofia`

Spring Spring Framework.
`http://www.springframework.org`

SpringWebFlow Spring Web Flow. Ervacon.
http://www.ervacon.com/products/springwebflow

Spyce Spyce—Python Server Pages (PSP). Rimon Barr.
http://spyce.sf.net

Struts Struts. The Apache Software Foundation.
http://jakarta.apache.org/struts

Subway Subway.
http://subway.python-hosting.com

Swinglets Javelinsoft. Javelin Software.
http://www.javelinsoft.com/swinglets

Tapestry Jakarta Tapestry. The Apache Software Foundation.
http://jakarta.apache.org/tapestry

Teaservlet Tea Trove Project. Walt Disney Internet Group.
http://teatrove.sourceforge.net

Tiles Tiles. Cedric Dumoulin and The Apache Software Foundation.
http://www.lifl.fr/~dumoulin/tiles

Turbine Jakarta Turbine. The Apache Software Foundation.
http://jakarta.apache.org/turbine

TwistedMatrix Twisted Matrix Laboratories.
http://twistedmatrix.com

TwistedWeb Overview of Twisted Web.
http://twistedmatrix.com/projects/web/documentation/howto/web-overview.html

Velocity Velocity. The Apache Software Foundation.
http://jakarta.apache.org/velocity

VRaptor VRaptor—Simple Web MVC Framework. Arca.
http://www.vraptor.org

Wasp Wasp Documentation. Robin Parmar.
http://www.execulink.com/~robin1/wasp/readme.html

WebMacro Web Macro. Semiotek Inc.
http://www.webmacro.org

WebOnSwing WebOnSwing—Multi environment application framework. Frebes.
http://webonswing.sourceforge.net

Webware Webware For Python Wiki. .
`http://wiki.w4py.org/`

Webwork Webwork. OpenSymphony.
`http://www.opensymphony.com/webwork`

Wicket Wicket. Wicket developers.
`http://wicket.sourceforge.net`

wingS wingS—LGPL Open Source.
`http://wings.mercatis.de`

Woven Woven.
`http://twisted.sourceforge.net/TwistedDocs-1.2.0rc3/howto/woven.html`

Zope Zope.org. Zope Corporation.
`http://www.zope.org`

Bibliography

OMG. *Unified Modelling Language v1.5*. OMG, March 2003.

E. Armstrong, J. Ball, S. Bodoff, D. B. Carson, I. Evans, D. Green, K. Haase, and E. Jendrock. The J2EE™1.4 tuturial. `http://java.sun.com/j2ee/1.4/docs/tutorial/doc`, June 2005. (last accessed August 2005).

A. Belapurkar. Use continuations to develop complex web applications: A programming paradigm to simplify MVC for the web. `http://www-128.ibm.com/developerworks/library/j-contin.html`, December 2004. (last accessed August 2005).

T. Berners-Lee. WWW: Past, present, and future. *Computer*, 29(10):69–77, 1996. ISSN 0018-9162.

T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifiers (URI): Generic syntax. Internet Engineering Task Force, August 1998. RFC2396.

S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. Transport layer security (TLS) extensions. Internet Engineering Task Force, June 2003. RFC3546.

M. R. Brown. FastCGI specification. `http://www.fastcgi.com/devkit/doc/fcgi-spec.html`, April 1996. © Open Market, Inc. (last accessed August 2005).

S. Ceri, P. Fraternali, and A. Bongio. Web modeling language (WebML): a modeling language for designing web sites. In *Proceedings of the 9th international World Wide Web conference on Computer networks : the international journal of computer and telecommunications networking*, pages 137–157, Amsterdam, The Netherlands, The Netherlands, 2000. North-Holland Publishing Co.

L. Chao, H. Keqing, L. Jie, and Y. Shi. Some domain patterns in web application framework. In *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*, pages 674–677, November 2003.

J. Conallen. Modeling web application architectures with UML. *Communications of the ACM*, 42(10):63–70, 1999. ISSN 0001-0782.

D. Copeland, R. Corbo, S. Falkenthal, J. Fisher, and M. Sandler. Which web development tool is right for you? *IT Professional*, 2(2):20–27, March/April 2000.

D. Coward and Y. Yoshida. *Java™ Servlet 2.4 Specification*. Sun Microsystems, Inc., November 2003.

O. Dahl, E. Dijkstra, and C. Hoare. *Structured programming*. Academic Press, Inc, Orlando, FL, USA, 1972.

L. Daigle, D. van Gulik, R. Iannella, and P. Faltstrom. Uniform resource names (URN) namespace definition mechanisms. Internet Engineering Task Force, October 2002. RFC3406.

L. G. DeMichiel. *Enterprise JavaBeans™ Specification, Version 1.2*. Sun Microsystems, Inc., November 2003.

T. Dierks and C. Allen. The TLS protocol version 1.0. Internet Engineering Task Force, January 1999. RFC2246.

E. W. Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968. ISSN 0001-0782.

D. Draheim and G. Weber. Modelling form-based interfaces with bipartite state machines. *Interacting with Computers*, 17(Issue 2):207–228, March 2005.

M. J. Dürst. The HTTP charset parameter. `http://www.w3.org/International/O-HTTP-charset.html`, February 2005. (last accessed August 2005).

R. Fielding. Relative uniform resource locators. Internet Engineering Task Force, June 1995. RFC1808.

R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. Internet Engineering Task Force, 1999. RFC2616.

R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP authentication: Basic and digest access authentication. Internet Engineering Task Force, June 1999. RFC2617.

P. Fraternali. Tools and approaches for developing data-intensive web applications: A survey. *ACM Computing Surveys*, 31(3):227–263, September 1999.

A. O. Freier, P. L. Karlton, and P. C. Kochner. The SSL protocol, version 3.0. `http://wp.netscape.com/eng/ssl3`, November 1996. (last accessed August 2005).

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, Reading, Massachusetts, 1995.

A. Ginige and S. Murugesan. Web engineering: an introduction. *IEEE Multimedia*, 8(1): 14–18, January-March 2001.

J. Gómez, C. Cachero, and O. Pastor. Extending a conceptual modelling approach to web application design. In *CAiSE '00: Proceedings of the 12th International Conference on Advanced Information Systems Engineering*, pages 79–93, London, UK, 2000. Springer-Verlag. ISBN 3-540-67630-9.

E. A. Gorshkova and B. A. Novikov. Use of statechart diagrams for modeling of hypertext. *Program. Comput. Softw.*, 30(1):47–51, 2004. ISSN 0361-7688.

N. Güell, D. Schwabe, and P. Vilain. Modeling interactions and navigation in web applications. In *ER '00: Proceedings of the Workshops on Conceptual Modeling Approaches for E-Business and The World Wide Web and Conceptual Modeling*, pages 115–127, London, UK, 2000. Springer-Verlag. ISBN 3-540-41073-2.

D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8 (3):231–274, 1987. ISSN 0167-6423.

A. Hassan and R. Holt. Migrating web frameworks using water transformations. In *27th Annual International Computer Software and Applications Conference, 2003. COMPSAC 2003*, pages 296–303, Washington, DC, USA, November 2003. IEEE Computer Society.

T. Helman and K. Fertalj. A critique of web application generators. In *Proceeding of the 25th International Conference on Information Technology Interfaces (ITI), 2003. ITI 2003.*, pages 639–644, June 2003.

I. Horrocks. *Constructing the User Interface with Statecharts*. Addison Wesley, 1998.

S. Huang. Evaluating the reverse engineering capabilities of web tools for understanding site contents and structure. Master's thesis, University of California Riverside, March 2001.

M. Jouravlev. Redirect after post. `http://www.theserverside.com/articles/article.tss?l=RedirectAfterPost`, August 2004. (last accessed August 2005).

H. M. Kienle and H. A. Müller. Leveraging program analysis for web site reverse engineering. In *WSE '01: Proceedings of the 3rd International Workshop on Web Site Evolution (WSE'01)*, page 117, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1399-9.

A. Knapp, N. Koch, F. Moser, and G. Zhang. ArgoUWE: A CASE tool for web applications. In *Proceedings of the First International Workshop on Engineering Methods to Support Information Systems Evolution (EMSISE03)*, September 2003.

N. Koch and A. Kraus. The expressive power of UML-based web engineering. In D. Schwabe, O. Pastor, G. Rossi, and L. Olsina, editors, *Second International Workshop on Web-oriented Software Technology (IWWOST02)*, pages 105–119, June 2002.

J. Korpela. Augmentative authoring—a different look at "graceful degradation" in web authoring. `http://www.cs.tut.fi/~jkorpela/html/augm.html`, August 2002. (last accessed August 2005).

D. Kristol and L. Montulli. HTTP state management mechanism. Internet Engineering Task Force, October 2000. RFC2965.

K. R. P. H. Leung, L. C. K. Hui, S. M. Yiu, and R. W. M. Tang. Modeling web navigation by statechart. In *COMPSAC '00: 24th International Computer Software and Applications Conference*, pages 41–47, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0792-1.

G. A. D. Lucca, A. R. Fasolino, and P. Tramontana. Towards a better comprehensibility of web applications: Lessons learned from reverse engineering experiments. In *WSE '02: Proceedings of the Fourth International Workshop on Web Site Evolution (WSE'02)*, page 33, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1804-4.

C. McClanahan, E. Burns, and e. Roger Kitain. *JavaServer™ Faces Specification, v1.1*. Sun Microsystems, Inc., February 2004.

M. Mealling and R. Denenberg Eds. Report from the joint W3C/IETF URI planning interest group: Uniform resource identifiers (URIs), URLs, and uniform resource names (URNs): Clarifications and recommendations. Internet Engineering Task Force, August 2002. RFC3305.

S. Mellor, A. Clark, and T. Futagami. Model-driven development - guest editor's introduction. *Software, IEEE*, 20(5):14–18, September-October 2003.

R. Moats. URN syntax. Internet Engineering Task Force, May 1997. RFC2141.

G. Murray. Asynchronous JavaScript technology and XML (AJAX) with Java 2 platform, enterprise edition. `http://java.sun.com/developer/technicalArticles/J2EE/AJAX`, June 2005. (last accessed August 2005).

Netscape. Persistent client state: HTTP cookies. `http://wp.netscape.com/newsref/std/cookie_spec.html`, 1999. (last accessed August 2005).

S. Penchikala. Session replication in tomcat 5 clusters. `http://www.onjava.com/pub/a/onjava/2004/11/24/replication1.html`, November 2004. (last accessed August 2004).

S. Perkins. Internet cookies: Security implications. `http://citeseer.ist.psu.edu/cachedpage/307950/1`, May 2000.

e. Pierre Delisle. *JavaServer Pages™ Standard Tag Library*. Sun Microsystems, Inc., June 2002.

Python Software Foundation. Python programming language. `http://www.python.org`, 2005.

D. Raggett, A. L. Hors, and I. Jacobs. *HTML 4.01 Specification*. W3C, December 1999. W3C Recommendation.

E. Rescorla. HTTP over TLS. Internet Engineering Task Force, May 2000. RFC2818.

E. Rescorla and A. Schiffman. The secure hypertext transfer protocol. Internet Engineering Task Force, August 1999. RFC2660.

J. Rode, M. B. Rosson, and M. A. Perez-Quinones. End-users' mental models of concepts critical to web application development. In *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC'04)*, pages 215–222, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7803-8696-5.

M. Roth and E. Pelegrí-Lopart. *JavaServer Pages™ 2.0 Specification*. Sun Microsystems, Inc., November 2003.

S. Rushing. Medusa: A high-performance internet server architecture. http://www.nightmare.com/medusa/medusa.html, none. (last accessed August 2005).

P. Sauter, G. Vögler, G. Specht, and T. Flor. A model-view-controller extension for pervasive multi-client user interfaces. *Personal and Ubiquitous Computing*, 9(2):100–107, 2005. ISSN 1617-4909.

M. W. Schranz, J. Weidl, K. M. Gschka, and S. Zechmeister. Engineering complex world wide web services with JESSICA and UML. In *HICSS '00: Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 6*, page 6068, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0493-0.

G. Seshadri. Understanding JavaServer Pages model 2 architecture: Exploring the MVC design pattern. http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html, November 1999. (last accessed August 2005).

T. Shimomura. Visual design and programming for web applications. *Journal of Visual Languages and Computing*, 16(3):213–230, June 2005.

J. Spolsky. The absolute minimum every software developer absolutely, positively must know about unicode and character sets (no excuses!). http://www.joelonsoftware.com/articles/Unicode.html, October 2003. (last accessed August 2005).

C. Strachey and C. P. Wadsworth. Continuations: A mathematical semantics for handling fulljumps. *Higher Order Symbol. Comput.*, 13(1-2):135–152, 2000. ISSN 1388-3690.

Sun Microsystems. Community development of java technology specifications. http://www.jcp.org/en/introduction/overview, 1005-2005. (last accessed August 2005).

M. M. T Berners-Lee, L Masinter. Uniform resource locators (URL). Internet Engineering Task Force, December 1994. RFC1738.

The Apache Software Foundation. Apache. http://www.apache.org, 2005.

D. Thomas. MDA: revenge of the modelers or UML utopia? *Software, IEEE*, 21(3):15–17, May-June 2004.

D. R. Tobias. Dan's web tips: Graceful degradation. `http://webtips.dan.info/graceful.html`, April 2004. (last accessed August 2005).

G. Trubetskoy. Mod_python: Apache/python integration. `http://www.modpython.org`, 2005.

M. A. S. Turine, M. C. F. D. Oliveira, and P. C. Masiero. HySCharts: A statechart-based environment for hyperdocument authoring and browsing. *Multimedia Tools and Applications*, 8(3):309–324, 1999. ISSN 1380-7501.

W3C HTML Working Group. *XHTML™ 1.0 The Extensible HyperText Markup Language: A reformulation of HTML in XML 1.0*. W3C, second edition edition, August 2002. W3C Recommendation.

D. Whalen. Cookies FAQ. `http://www.cookiecentral.com/faq`, 1996. (last accessed August 2005).

M. Winckler, C. Farenc, P. Palanque, and R. Bastide. Designing navigation for web interfaces. In *Joint d'Interaction Homme-Machine and Human-Computer Interaction Conferences (IHM-HCI)*, September 2001.

M. Winckler and P. Palanque. StateWebCharts: A formal description technique dedicated to navigation modelling of web applications. In *Interactive Systems. Design, Specification, and Verification: 10th International Workshop, DSV-IS 2003, Funchal, Madeira Island, Portugal, June 11-13, 2003. Revised Papers*, volume 2844/2003 of *Lecture Notes in Computer Science*, pages 61–76, GmbH, December 2003. Springer-Verlag.

F. Yergeau, J. Cowan, T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. *XML 1.1*. W3C, February 2004. W3C Recommendation.

Y. Zheng and M.-C. Pong. Using statecharts to model hypertext. In *ECHT '92: Proceedings of the ACM conference on Hypertext*, pages 242–250, New York, NY, USA, 1992. ACM Press. ISBN 0-89791-547-X.