Designing of Portlet-based Web Portals

Dissertation
presented to
the Department of Languages and Information Systems of
the University of the Basque Country
in Partial Fulfillment of
the Requirements
for the Degree of

Doctor of Philosophy

M^a Aránzazu Irastorza Goñi

Advisor: Óscar Díaz García

Donostia-San Sebastián, Spain, July 2008

Nere gurasoei

Summary

The main aim of this dissertation has been improving the design and use of portlet-centric Web portals. About the first concern, our proposal is a design method *with reuse*, i.e., reusing available Web components, that is, portlets. The approach takes statecharts as the main conduit for describing how portlets are gathered together. The outcome of the design process of a new Web portal is an *annotated statechart* which collects the main design decisions, i.e., tasks, structural and workflow organization, and aesthetic concerns. The rendering model takes advantage of hierarchical construction of the orchestration model, i.e., statechart, to specify the presentation through an inheritance-like mechanism based on the state hierarchy. However, this approach would remain short, if the portal master had to create the portal code by hand, while he interprets the annotated statechart sketch. This process would be error-prone. Thus, the proposal in this dissertation is a model-driven approach to move automatically from annotated statechart models to implementation concepts like Web page, CSS class, and the like. Two metamodels and the transformation between them are described. With the ultimate objective of improving the use of Web portals, another of the concerns of this dissertation has been the *portlet interoperability*, that is, the ability of portlets to exchange information. The proposal in this dissertation is based on Semantic Web, specifically it is a front-end approach, where presentation fragments of portlets are annotated with information about the rendering process. Then, OWL rules are which make inference of data, i.e. carry out the dataflow.

Contents

Chapter 1

Introduction

1.1 Overview

Although there is no common agreement as to what a Web portal is, two main points of view can be distinguished. On one hand, Web portals as Web sites which act as a starting point or 'gateway' to a wide variety of resources (like search engines, news, discussion groups, references and so on). Hence, the idea of a Web portal is to collect information from different sources and create an information hub. From this perspective, Web portals are evolved content managers. On the other hand, Web portals can also play the role of frameworks for integrating applications and processes across organizational boundaries. From this viewpoint, Web portals are somehow related with Enterprise Application Integration (EAI) approaches. This thesis addresses the second perspective.

Application integration is a long lasting endeavour in Software Engineering. Most of the approaches so far mainly address back-end integration where integration takes place behind the scenes. Web portals consider front-end integration. Besides all the intricacies that go with back-end integration, front-end integration has to tackle also the GUI issue. In this scenario, GUI usability matters. Indeed, according to Jafari and Sheehan [47] usability is one of the keys to portal success and *usability is the extent to which a system supports its users in completing their tasks efficiently, effectively and satisfactorily*. This makes interoperability even a more important issue than in back-end integration. Interoperability is the ability of software and hardware on multiple machines from multiple vendors to communicate, and now interoperability extends beyond protocols and parameters to also take into account the GUI. That is, portal-based integration needs to face not only parameter heterogeneity or platform discrepancies, but also look-and-feel mismatches between the GUIs of the applications to be integrated.

As a result of the broad range of aspects to be considered, portal design is far from being a simple task. Jafari and Sheehan [47] state that *"... 80% of the success of a portal project depends on the quality of its design work and the forward thinking put into the conceptual and technical architecture of the system. The other 20% of the project's success is due to the quality of the portal software ..."*.

In light of the previous considerations, this doctoral thesis addresses portlet-based design of Web portals. Portlets are presentation-oriented Web Services which are packed to be delivered through third-party Web applications (e.g., a Web portal). Portlets are user-facing (i.e., return markup fragments rather than data-oriented XML) and multi-step (i.e., they encapsulate a chain of steps rather than a one-shot delivering). So far, portlets are mainly used as a modularization technique to structure portal content. However, their ability to be delivered through other Web applications, makes portlets be the enablers of service-oriented architectures (SOAs) but now at the front-end.

From this perspective, portlets strive to play at the front-end the same role that Web services currently enjoy at the back-end, namely, enablers of application assembly through reusable services. On the portlet case, the difference stems from what is being reused (i.e., which includes the presentation layer) and where the integration is achieved (i.e., at the front-end). From this thesis perspective, a main feature of portlets is that its "container model" has been standardized: Web Services for Remote Portlets (WSRP) 2.0 [74] and Java Portlet Specification (JSR-286) [49]. This is a main achievement that ensures that portlets developed by IBM's WebSphere can be rendered through an Oracle-engineered portal, and vice versa. This ensures "renderization" interoperability, but other design problems are still open. Specifically, this dissertation is centered around three main areas, that is, Web portal design, Web portal code generation and application interoperability. Next sections outline each of these issues. For completeness sake, Section 1.5 outlines related standards. These standards provide the baseline on which the rest of the work builds up.

1.2 Portlet-centric Portals: design

Current portal applications tend to be rather monolithic in both conception and support. This can be due to the fact that many first generation portal development environments evolved from content management origins. To take hold portal applications, customers need to look at solutions that support a programming model centered around service interfaces. Ideally, these service interfaces would be provided by loosely coupled components that are unshackled from process and relatively free of dependency on underlying infrastructure technologies. This is a distinct move away from monolithic portals to the idea of portals as entry points into a combination of services. The result is a collection of re-usable, and more importantly, easily upgradeable services as opposed to those assets being locked into rigid monolithic portals. As stated in [103], *"needed is a development environment designed with an application orientation, rather than a content-centric perspective, which offers the ability to manage portals at component-level across the entire application life-cycle"*.

The work presented in this dissertation focuses on Web portals as aggregators of third-party applications, i.e., portlets. Therefore the portal designer will not have to design their inner functionality. According with her requirements, she will have to decide the tasks the new portal will offer, to choose the appropriate portlets (i.e., portlets whose functionalities fit the tasks), and then, to design the orchestration among the tasks. She will have to decide which task must be executed first, if there is any precedence-rule among tasks, and the like.

However, portal IDEs (Integrated Development Environments) have not yet exploited the full potential of portlets. The content-management role is still prevalent in most of current IDEs. The Web portal is still conceived as a conglomerate of pages where portlets tend to be considered as a modular mechanism during page implementation. The page is still the main notion, and the portlet is subordinated to the page. This has important consequences since *portlet* navigation (i.e., browsing along the portlet fragments) is completely detached from *portal* navigation (i.e., browsing along the portal pages). And all portlets are readily rendered when entering in the container page.

The new perspective of taking portlets as enablers of service-oriented architectures (SOAs) requires a departure from how current Web portals are envisaged. The Web portal is no longer perceived as a set of pages, but as an integrated set of Web components (i.e., portlets) that are now delivered through the Web portal. From this perspective, the portal page now acts as a mere conduit for portlets. Page and *page* *navigation* dilute in favor of portlet and *portlet orchestration*. This thesis promotes this service-oriented view of Web portals.

Our contribution is to describe portlet orchestration using statecharts [39]. This approach was selected because the Unified Modeling Language (UML) uses statecharts as a means for modelling behaviour and UML is a standard. Moreover, although we have not searched this field, the statechart model offers rigorous semantics for formal analysis on various aspects of portal functionality orchestration, e.g., not reached states (i.e., tasks), invalid transitions, and the like. Having an analysis and design model helps the designer to abstract: she works about tasks and their relations and she does not have to study a new language, probably nearer to the implementation, and to start specifying namespaces, port descriptions, partner links, and so on, as it is in the case of Web service orchestration and BPEL language [73].

In addition to the orchestration model, this dissertation also proposes the use of another two models during portal design: *the task model* and *the rendering model*. The first one describes the set of tasks the new Web portal will offer, while the rendering model specifies the parameters for the aesthetic presentation of the portal. This approach departs from traditional page-centric design approaches towards a design where the functionality rather than content rendering plays the pivotal role.

1.3 Portlet-centric Portals: implementation

As said, a main effort of this thesis is for Web portal design to be abstracted away from pages and be conceived in terms of portlets and portlet orchestration. However, the mapping from portlet orchestration (design time) to page navigation (implementation time) is too tedious and error-prone. The fact that the same portlet can be placed in distinct pages produces code clones that are repeated along the pages that contain this portlet. This redundancy substantiates in the first place the effort to move to model-driven development (MDD). The combined use of models and transformations made MDD an excellent reuse technique even if diversity in the implementation platform is not an issue.

Therefore, this thesis concentrates on an MDD approach to portlet-centric portal development. It follows the MDA (Model-Driven Architecture) proposal [76], and defines a Platform Independent Model (PIM), a Platform Specific Model (PSM) and the mapping transformations between them.

The PIM abstracts *portal* and *portlets* into *workspace* and *tasks,* respectively. In this way, portal development is moved away from the technology domain towards the concepts of the problem domain. However, one of the big concerns in software development, in general, is the gap between the design model and the implementation code. Taking a portal platform, as *eXo* [27] for instance, developers can use its assistants to define and configure several components of the new Web portal, e.g., portal pages, navigation among them, and so on.

However, they will have to do this process interpreting the model-based design, previously carried out by them or a designer. This stage may be hazardous and error-prone depending on the conceptual gap between the design and implementation models. Thus, our approach specifies a PIM metamodel agglutinating the three viewpoints of Web portal design, i.e., tasks, orchestration and rendering.

Otherwise, a widely extended open source portal platform, *eXo* [27], has been selected and with some of its features a PSM metamodel has been defined.

Finally, the transformation from PIM to PSM has been described and fully implemented, specially the transformation from conceptual concepts such as states, transitions, rendering descriptors, and the like, to implementation concepts, such as pages, portlets, CSS classes, and so on. The transformation is completely automatic, so the design and implementation models are concordant, and if a change was necessary in the design model, the corresponding code could be generated.

1.4 Portlet-centric Portals: portlet interoperability

A key factor for a Web portal to succeed is the support it offers to its users in completing their tasks satisfactorily. One of these enablers is parameter reuse. In the context of this thesis, this refers to portlet interoperation, i.e., the exchange of information between portlets. Aggregating portlets into a Web portal is more than merely invoking these services, or arranging their fragments together in the same portal page (i.e., the so-called *"side-by-side"* aggregation). Information contained in one portlet will surely be required in another, and forcing the individual end-user to manually copy and key in data from source to target portlets leads to frustration, lost productivity, and inevitable mistakes.

Current approaches to portlet interoperation rely on the existence of a common data structure that supports data exchange. However, such approaches defeat the view of portlets as SOA-enablers. That is, interoperation should be achieved between portlets with different origins (i.e., different providers), and this means that no common data structure will be available (in the same way that traditional Web services do not rely on the existence of such data structure to exchange data). The new release of

WSRP 2.0 [74] addresses this issue by providing an event-based mechanism. Using a publish/subscribe pattern, portlets can publish events which other portlets can subscribe to, and the Web portal plays the mediator role. By contrast, this thesis proposes a front-end approach to portlet interoperation.

By "front-end" is meant that the visual part of a portlet, the fragments, are supplemented with information about what these fragments render. Rather than resorting to another mechanism, "the event" is described as metadata of the fragment using an *annotation* approach [35]. Specifically, we consider the so-called *deep annotation* as particularly valid for portlet interoperation due to the controlled and cooperative environment that characterizes the portal setting.

Therefore, the contribution of this thesis rests on two ontologies. First the concept of *portlet ontology* is defined. It will be used by the portlet producer to annotate portlet fragments with data about the processes whose rendering each fragment supports. Due to annotation will not be about presentation concerns, but about semantic information (which data is rendered, which ontology they fit, and so on), we call it deep annotation, as Handschuh *et al.* did also in [36]. Moreover, the concept of *portal ontology* is also defined, which will be used to weld somehow portlet annotations. During enactment of portlet fragments, and using their annotations, the Web portal (i.e., the portlet consumer) will produce portal annotations that will be used to infer new portal annotations. These inferred annotations will be used to feed other portlet fragments automatically (i.e., some input widgets will be rendered filled up). The inference is carried out by a rule set the portal designer will have to define.

1.5 The use of standards

Standards constitute a way to gather the best-proved approaches. Moreover, standards are a means of communication among developers. Because a workproduct may go through several developer teams during its lifetime, it is important that those people are able to comprehend the design and code and to modify it easily. Standards also offer a set of rules that every developer can follow, understand, and become familiar with, in other words, standards offer a framework for tested good practice.

Otherwise, looking at developed software products, the use of standards supports software maintainability and protects software products against obsolescence, because most of standards are generally designed taking into account version compatibility. Moreover, the use of standards assures the interoperability among applications.

Hence, the adoption of standards is of paramount importance and this work has tried to use them as much as possible.

- **UML statecharts** [78] constitute the foundation for our proposal to design the task orchestration. The designer of Web portals does not need to learn a new language and there are lots of tools to edit and check statecharts.

- **Web Services for Remote Portlets** (**WSRP**) specification [71]. It defines a Web service interface for accessing and interacting with "interactive presentation-oriented Web services" (i.e., portlets). Thus, applications can consume those Web components without having to write unique code for interacting with each component. Our proposal for Web portal design and portal code generation assumes the reuse of WSRP components and it defines a metamodel for WSRP.

- **Java Portlet Specification** (**JSR-168**) [50]. It defines a standard for the Java portlet API designed to enable interoperability between portlets and Web portals. One of its most important gaps is interportlet communication. Taking advantage of the extensibility mechanisms available in the standard, we have proposed to extend the portlet description with an additional *ontology* property, thus, facilitating portlet annotation and then portlet interoperability. The new standard, Java Portlet Specification (JSR-286), published in April 2008 [49], aligns with WSRP 2.0 [74] and *it enables portlets to communicate with each other through sending and receiving events* [74].

- **OWL-S** [110] is an ontology built on top of Web Ontology Language (OWL) for describing Semantic Web Services. OWL-S facilitates, amongst others, composition and interoperation of Web services through their semantic description. It has been used by an important number of research efforts [25]. Our proposal for portlet interoperability also takes OWL-S as the baseline ontology to add annotations in portlet fragments. Besides facilitating portlet interoperability, all the benefits of using explicit ontologies (e.g., better documentation, search, knowledge acquisition [48]) are brought to the portlet realm.

- **Query/View/Transformation (QVT) Specification** [80] defines a standard to write transformations from source models into target models. These models conform to MOF metamodels. Our proposal for model transformation has used *RubyTL* [99], a pattern-based transformation language which also takes EMOF metamodels but it does not implement the standard. At the time of the implementation, transformation languages conformant to QVT proposal [77] were not expressive enough to specify the required transformations.

1.6 Document Organization

This dissertation is composed of six chapters, including this one, namely:

- Chapter 2 outlines Web portals and portlets.
- Chapter 3 introduces the software development method known as Model-Driven Engineering, and two of its most well-known approaches Model-Driven Architecture (MDA) and Domain-Specific Modelling (DSM). The chapter also introduces RubyTL, a transformation language developed by the GTS research group at the University of Murcia, and extensively used in this dissertation.
- Chapter 4 presents three metamodels: (1) the SOP metamodel, which agglutinates three viewpoints in the design of a portlet-centric portal i.e., tasks, orchestration and rendering; (2) the WSRP metamodel, which describes the portlets to be reused in the portal design, and (3) the EXO metamodel, which describes the artefacts needed to implement an *eXo* portal. These metamodels are the base for the *SOP-to-EXO* transformation rules, which show the applicability of the MDA approach for portal code generation.
- Chapter 5 describes how a deep annotation approach can be used for portlets to share data. First, it defines what an annotation is, and describes portlet and portal ontologies. Both will be used to produce annotations, the former to annotate portlet fragments, and the latter to get and infer annotations that will be useful to flow data from one portlet fragment to another.
- Chapter 6 presents the conclusion of this thesis. It also proposes future research topics following this work.

Finally, Annexe A describes thoroughly the SOP-to-EXO transformation rules, implementation of our MDD approach.

Chapter 2

Introduction to Web Portals and Portlets

2.1 Introduction

First Web sites consisted of static pages, with a limited interaction with the user. Then, Web applications dedicated to e-commerce, content publication, and management, focused on enabling users to perform simple operations, like searches, data uploads, and browsing of large volumes of data structured in hypertexts. Later, the Web became a platform for B2B applications, whose goals are intra- and inter-organization business processes.

Web portals are one of such applications, and they provide integration with third-party applications at the user interface level, whereas other integration technologies support business process, functional or data integration. The portal technology is based on the notion of a portal container that provides the basic infrastructure to host several applications wrapped up as *portlets*. Apart from application integration, features as adaptation and personalization also characterize Web portals.

This chapter presents a summary for Web portal definitions in Section 2.2 and a description of portlet features in Section 2.3. Moreover, Section 2.4 describes briefly the *Web Services for Remote Portlets 1.0 (WSRP)* specification. WSRP defines the means for using portlets as Web components and for portlet-portal interoperation.

2.2 Web portals

There is no common agreement about what a portal is [113]. Many references point out that portal is a Web site which acts as a starting point or 'gateway' and provides a wide variety of resources, services, tasks and links to other websites. Among those resources there are search engines, news, e-mail, discussion groups, online shopping, references and so on. This type of portals, sometimes called *horizontal portals* [101], is generally offered by Internet Service Providers or search engines. Yahoo! is an example, with an index to a lot of services, that is, the first screen that a user will see when going online, a place to go to find an organized view of the online information space. More specialized portals, sometimes called *vertical portals* [101], are those addressed to a specific interest or field, for example portals with the aim at medical information. There, users can get information about clinical trials, professional directories, patient forums, support groups, health articles, health care associations, and so on. Even more specialized portals, *enterprise portals* [30] deliver organization-wide information in a user centric manner, based on user authentication they offer customized services to specific users, employees, customers, and the like. They offer support for tasks, workflow, groupware, and the creation and integration of knowledge. In this last category, we can find, for example, the employee portal of an university. There, employees, in general, can access their salaries, information about their medical insurances, and the like, and, more specifically, research staff can access a service to complete their curriculum vitaes, forms to request financial support for research, and so on. *Personal portals* are also distinguished. They are customized by the user and typically are associated with a search engine and display selected information such as news, weather, dictionaries and so on. iGoogle and My Yahoo! are examples of this type of portals.

Next, we summarize the fundamental features of portals:

- *Single sign on.* A portal is a doorway for a wide range of applications. Rather than expecting an end-user to remember and maintain a password for each application hosted by the portal, the portal offers a strong authentication scheme, where the end-user only has to remember one password. Once authenticated, the end-user has unrestricted access to all applications to which she is entitled. For applications external to the portal, a mapping is needed between authentication parameters of the portal, and the authentication parameters of the external application.

- *Personalization.* The end-user can change the interface and behaviour of the portal according with the way she works or with her needs and preferences. She can subscribe and unsubscribe to channels and alerts, add and remove specific links, set application parameter defaults, or format portal page (i.e., colours, fonts, columns, and the like).

- *Adaptation.* The portal is able to save common tasks the end-user does, her schedule and workflow, and then, it is able to change services it offers her or to make new recommendations, depending on the stored information. Therefore, the portal changes its behaviour depending on context.

- *Integration.* Companies use portals to help disseminate information to their employees in a timely and efficient manner. From this perspective, portals can be seen as the natural evolution of *Content Management Systems* (CMSs), but now portals strive to integrate legacy applications. This feature is seen as paramount. Indeed, some authors define portals *"a framework for integrating applications and processes across organisational boundaries"* [9]. Portal system features can also be viewed as "managing content", but what differentiates them from a CMS is they facilitate the access (integration) to information from various applications, data sources and structures, and back-end systems. Users select from a list of pre-defined site components (sometimes called *"portlets"*, see Section 2.3) and manage the layout and presentation of this information in a page location of their choice. They can add selected application interfaces, real-time data dashboards, reporting functions, and personalize how their page looks.

This latter feature is the one we would like to highlight. Portals as hubs that offer a consolidate view of content and services. Content/services can be provided locally or being offered by third parties or applications. In this scenario is when the notion of *portlet* shines up.

2.3 Portlets

Portlets are presentation-oriented Web Services which are packed to be delivered through third-party Web applications (e.g., a portal). Portlets are user-facing (i.e., return markup fragments rather than data-oriented XML) and multi-step (i.e., they encapsulate a chain of steps rather than a one-shot delivering). So far, portlets are mainly used as a modularization technique to structure portal content. However, their ability to be delivered through other Web applications makes portlets be the enablers of service-oriented architectures (SOAs) but now at the front-end.

From this perspective, portlets strive to play at the front-end the same role that Web services enjoy at the back-end, namely, enablers of application assembly through reusable services. On the portlet case, the difference stems from what is being reused (i.e., which includes the presentation layer) and where the integration is achieved (i.e., at the front-end).

To better asses the notion of portlet, next subsections compare portlets with Web services and traditional Web applications.

Java Specification Request 168 Portlet Specification (JSR-168) [50] standardizes how Web components, i.e., portlets, are to be developed. It addresses the portlet life cycle management, the portlet modes, the packaging and deployment, and so on. The work presented in this dissertation did not aim at the development of portlets, it has used them as Web components, as necessary pieces of Web portal development, therefore this chapter does not include a deeper description of that specification.

2.3.1 Portlets vs. Web services

Web Services provide enabling technology to deliver on the promise of Internet-based business-to-business connectivity. Web service standards facilitate the sharing of the business logic, but suggest that Web service consumers should write a new presentation layer on top of the business logic. As an example, consider a Web service that offers two operations, namely, *searchFlight* and *bookFlight*. The former retrieves flights that match some input parameters (e.g., *departureAirport, flightDates* and so on), while *bookFlight* takes the selected flight and payment data, and books a seat on this flight.

This WSDL-based API can then be used by a consumer application. First, the application would collect the *departureAirport, flightDates* and other parameters via an input form. Within the form, an *http* request might support a call to *searchFlight* which, in turn, returns a set of flights whose presentation is left to the calling application. Next the user selects one of the flights and, through another form, the Web application collects the user's information and payment data. This interaction will in turn invoke *bookFlight*. This example illustrates the traditional approach where Web services provide the business logic, and both presentation and control layers are left to the calling application.

This scenario illustrates the traditional use of Web services as a *function-integration technology* whereby one application programmatically invokes code that lies in another application. However, such an approach underscores the presentation layer [88]. This layer not only addresses aesthetic aspects, but a whole range of concerns like usability issues, state management, error handling, client-side scripting, navigation logic etc. Indeed, most of the aspects that characterize a good Web site are related to interactive issues [60]. Re-creating this presentation logic in each consumer application has potentially two main limitations, increases time-to-market and can jeopardize the company's image [88].

Increasing time to market. The presentation logic is one of the most critical but also, cumbersome and time-consuming software to be created. As the previous example highlights, the reconstruction of the screenshots not only involves aesthetic aspects but also lefts to the consumer application the recomposition of the workflow among the API's operations. Current practices imply custom programming to create a user interface tier for each new Web service. This results in set-up and maintenance efforts that hinders portal initiatives as the number of Web services increases. Therefore, an API-based approach as the one provided by traditional Web services, falls short for complex interactive applications whose flow spans several Web pages.

Jeopardizing the company's image. The presentation logic realises brand and customer experience strategies that are becoming critical business factors for a company to be ahead of its competitors. Letting the consumer application decide both how parameters are requested or results rendered back, can jeopardize the image of the service provider. The company might be interested in maintaining this experience in the case of its services being offered through third-party portals. As a Gartner analysis pointed out, *"Successful software vendors and Web services providers will find innovations in usability and user interface to be a source of competitive advantage. Better-than-average usability is one reason why Yahoo, Amazon, AOL, Google, and Palm came to dominate their respective markets.*"

What is required is to leverage Web Service technology as an *application-integration* enabler. True application integration results from making one application available within the context of another, and this can also include the user interface [114]. Microsoft OLE objects are a case in point. For example, this technology allows embedding an Excel spreadsheet directly into a Word document through simply dragging. Once embedded, you can work on the spreadsheet *from* the Word document as if you were within Excel. This is the scenario that portlet proponents aim for Web applications.

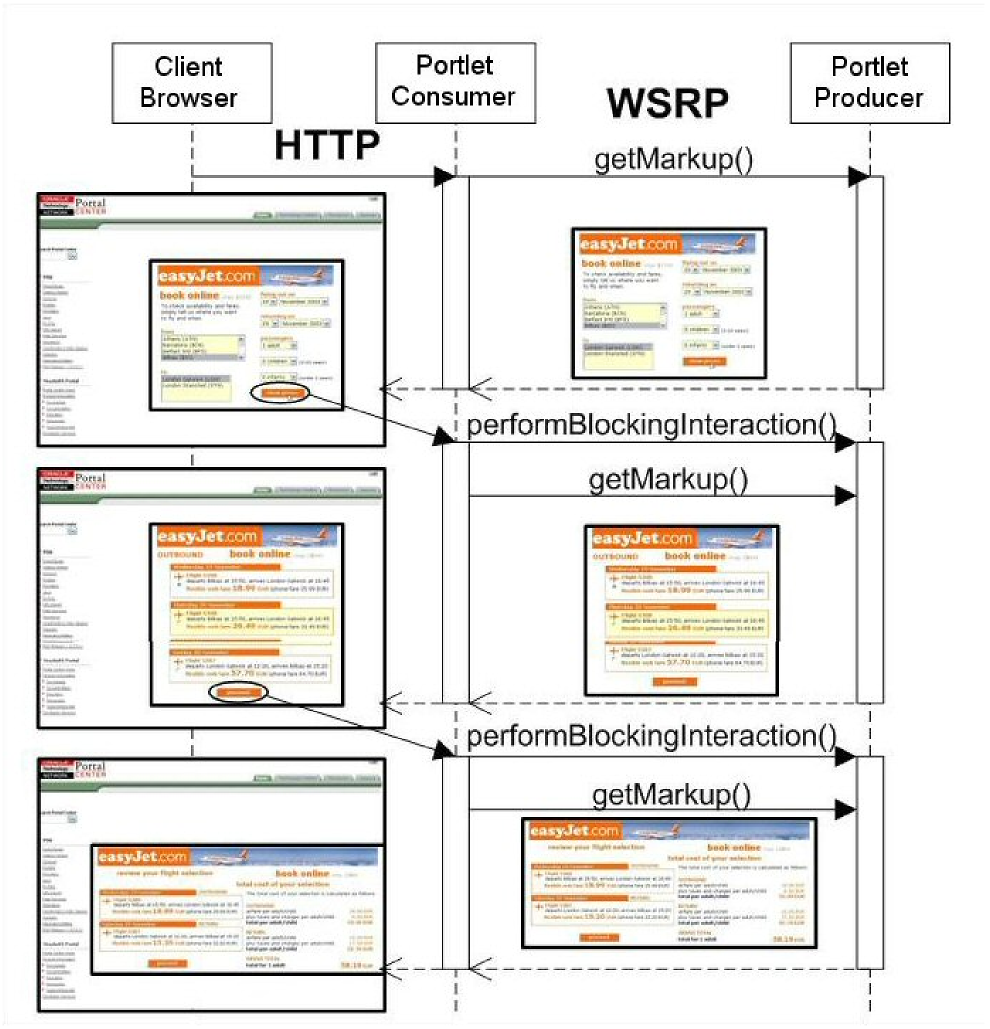


Figure 2.1: Interaction diagram for a portlet inside a portal.

Let's go back to our flight-booking application, but now delivered as a portlet. A *flightSearch* portlet is defined that encapsulates the previous sequence of operations (*"multi-step"*) and the XHTML fragments (*"user-facing"*). This portlet can then be used as a Web component to be plugged into third-party applications (e.g., a portal). Figure 2.1 shows the three actors involved, namely, the *End-user*, the *Portlet Consumer* and the *Portlet Producer*.

What the consumer is now re-using is a whole application. First, portlet operations might not only return raw data but fully rendered markup such as XHTML (known as "fragments" in the Portlet parlance) that is to be included within the portal page, with very few changes to be made by the consumer. Second, all interactions with a given portlet (see Figure 2.1) belong to the very same session, and hence, session and state maintenance should be preserved along these interactions. Although it depends on the approach, this can be the duty of the portlet producer. While in the Portlet realm, the consumer is relieved of the burden of complex and intricate session maintenance and control flow.

2.3.2 Portlets vs. Web applications

The previous comparison stresses the notion of a portlet as a full-fledged application. However, and unlike Web applications, portlets have an additional requirement: they can *"be subject to composition by third parties"*. This has two important implications: clear interfaces and configurability.

Clear interfaces. This implies the existence of well-defined and programmatic interfaces for the portlet to be plugged into the consumer application. Moreover, interoperability advises this interface to be generic so that the invoker can interact with portlets in a standard way. This is precisely the endeavour of the WSRP standard (see Section 2.4), which offers two methods in the interface: *getMarkup()* and *performBlockingInteraction()*.

Broadly speaking, the lifecycle of a portlet session begins when the first *getMarkup()* request is issued (see Figure 2.1). Once the first markup is rendered, a *two-phase protocol* is initiated [71]. The *getMarkup()* operation retrieves the markup that corresponds to the current state of the portlet. The consumer next invokes *performBlockingInteraction()* on the portlet whose markup the end-user has interacted with. This is a synchronous operation that routes the user-enacted interaction to the producer. The consumer has to wait for the response from *performBlockingInteraction()* before invoking *getMarkup()* on the portlet it is aggregating, in order to get the markup of its new state. The portlet will receive only one invocation of *performBlockingInteraction()* per client interaction, excepting for retries. If this operation ends successfully, the consumer can then retrieve the next markup by invoking *getMarkup()* on *all* the portlets within the portal page.

Configurability. It is well-known in the component community that, the larger the component, the more reduced the reuse. Portlets tend to be stateful, coarse-grained components since they encapsulate the presentation layer and all the navigation that goes with it. Consequently, mechanisms should be in place to configure the portlet to the environment where the portlet is going to be "hooked on". This context includes the window state, the user profiles, aesthetic guidelines and additional portlet-specific data collected, as WSRP-compliant portlet preferences. Next paragraphs outline some of these context properties.

Window state. This property sets the amount of page space that the portal will assign to the fragment generated by the portlet. Options contemplated by WSRP include: *normal*, indicates the portlet is likely sharing the aggregated page with other portlets; *minimized*, instructs the portlet not to render visible markup, but lets it free to include non-visible data such as JavaScript or hidden forms; *maximized*, specifies that the portlet is likely the only portlet being rendered in the aggregated page, or that the portlet has more space compared to other portlets in the aggregated page; and *solo*, denotes the portlet is the only portlet being rendered in the aggregated page. This property is set by the portal among the values supported by the portlet producer.

User profile. The user profile is used to personalize content to the idiosyncrasies of end-users. Now, this content is offered via portlets. Thus, parameters are defined in WSRP to pass these data from the portal to the portlet producer. User information attributes are derived from the *Platform for Privacy Preferences 1.0* (P3P 1.0) by OASIS where attributes are described such as *user.name.given*, *user.business-info.telecom.telephone.intcode* and the like.

Aesthetic guidelines. Now a portal page is produced as "portlet quilt". Hence, it is most important to ensure a common look-and-feel across the distinct portlet markups to be rendered in the same portal page (i.e., similar background, fonts, titles and the like). To this end, the portlet markup should use *Cascade StyleSheets* (CSSs) [108]. CSSs permit HTML fragments to parametrize some of their aesthetic aspects. The portlet returns CSS-parametrized fragments which are then processed by the portlet consumer. This process includes providing the actual values for the CSS parameters. Interoperability requires these parameters to be standardized so that the portal can expect always the same terms regardless of how the portlet producer is. This has also been achieved by the WSRP endowment.

Portlet preferences. A portlet preference is a named piece of string data that serve to personalize the portlet. As an example, go back to the *flightSearch* portlet. Its preferences can include *arrivalAirport* with values "San Sebastián", "London" or "New York", and *departureAirport* with value "Madrid". These preferences offer a parametrization-based mechanism to adapt the portlet (in this case, the input forms). These preferences can be changed at configuration time (by the portal administrator) or at enactment time. In this latter case, values can be set by the portlet itself -based on the user profile- or prompting the current user.

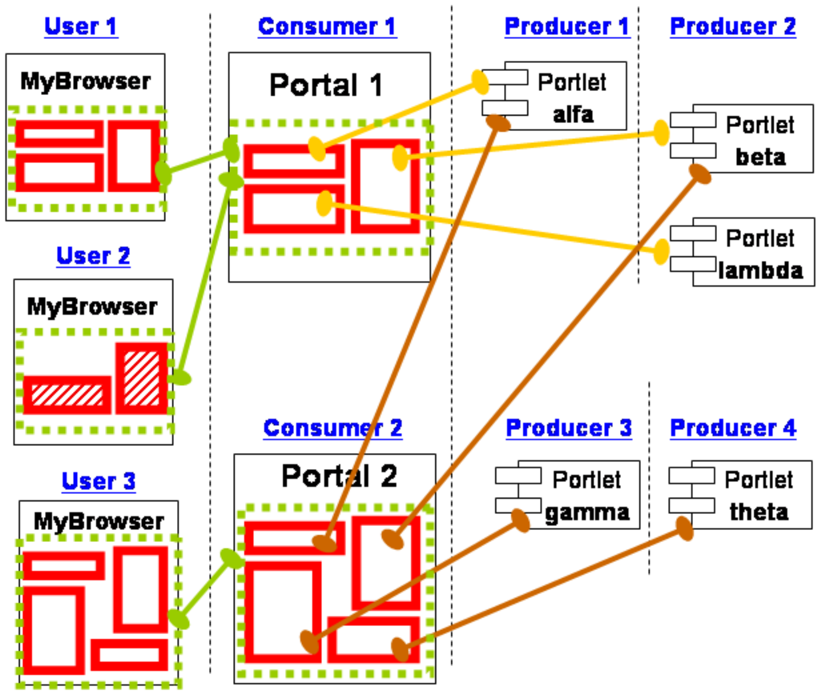


Figure 2.2: Portlet Producers and their Consumers.

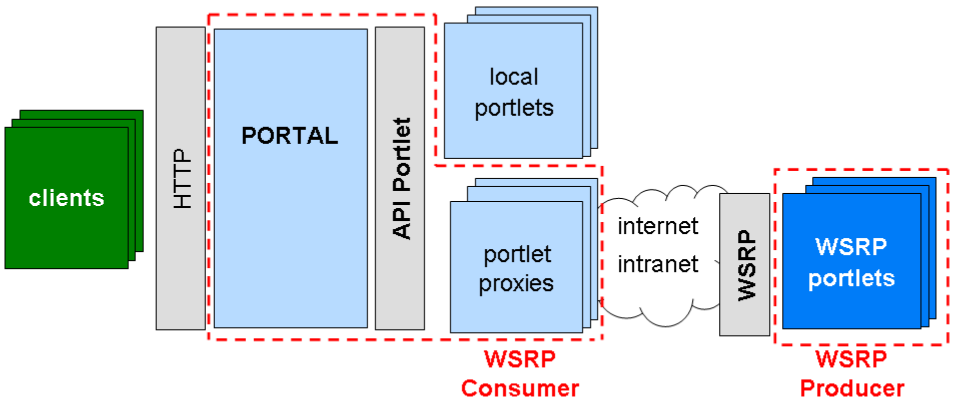


Figure 2.3: Portal architecture and WSRP.

2.4 Portlet interoperability: the WSRP standard

Previous subsections have highlighted the role of portlets as SOA enablers. However, this SOA scenario requires portlet interoperability, whereby portlets developed in, lets say, Oracle Portal, can be deployed at a Liferay portal, and vice versa.

The *Web Services for Remote Portlets* (WSRP) specification [71] brings this interoperability by providing a protocol that decouples portlet providers from portlet consumers. This provides the infrastructure to make feasible a portlet market *à la COTS*[1] so that portals can deliver portlets being provided by third parties[2] (see Figure 2.2). *"WSRP allows portals to display remotely-running portlets inside their pages without requiring any additional programming by the portal developers. To the end-users, it appears that the portlet is running locally within their portal, but in reality the portlet resides in a remotely-running portlet container, and interaction occurs through the exchange of SOAP messages"* [16]. Figure 2.3 depicts the portal architecture with their components.

WSRP is a joint effort of two OASIS technical committees, namely, the *Web Services for Interactive Applications* (WSIA) and the *Web Service for Remote Portals* (WSRP) [71]. WSRP layers on top of the existing Web Service stack, utilizing WSDL for defining a set of interfaces. It standardizes the application programming interfaces (API) between Consumers and Producers of Portlets (see Figure 2.3), the communication protocol, and some aspects of the component model (e.g., modes, personalization descriptions, CSS terms and the like). WSRP interfaces provide the same core feature set as locally deployed portlets, but JSR-168 is Java only, while WSRP is platform independent. WSRP includes extension mechanisms.

The specification 1.0 contemplates four interfaces, namely [71]:

- *Service Description*. This interface allows consumers to ascertain both the capabilities of the producer and the portlets it hosts. The latter includes the metadata necessary for a consumer to properly interact with each portlet. It defines the operation *getServiceDescription()* for acquiring the metadata of the producer.

- *Markup*. This interface allows consumers to request and interact with markup fragments. This includes the *getMarkup()* operation that returns the presentation markup which corresponds to the current portlet state, as well as the *performBlockingInteraction()* operation which is the means used by the consumer to route the chosen interaction to the producer. As the WSRP does not

[1]COTS stands for *Commercial off-the-shelf.*

[2]Indeed, the *Open Source Portlet Repository Project* was launched in 2006 to foster the free and open exchange of portlets. The *Portlet Repository* is "*a library of ready-to-run applications that you can download and deploy directly into your portal with, in most cases, no additional setups or configurations*" [7]. Other similar initiatives include *Portlet Swap* (jboss.org) and *Portlet Exchange* (portletexchange.com).

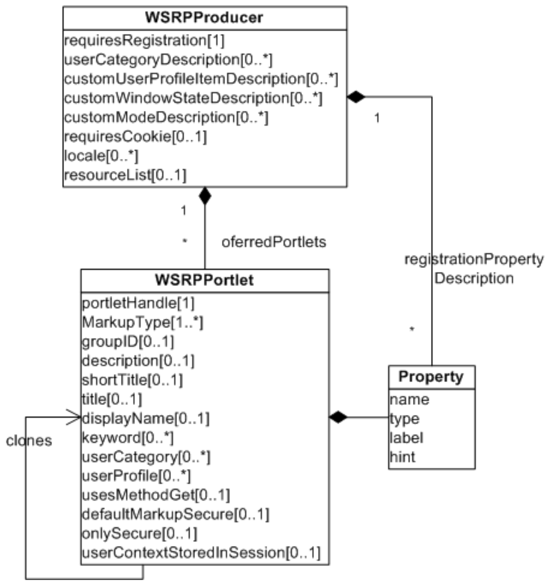


Figure 2.4: WSRP description concepts.

force that neither the producer nor the consumer is stateful, these operations carry the state necessary for the portlet to render the current markup to be returned to the consumer. If the producer utilizes local state then, it will return a *sessionID* to the consumer for using during the lifetime of the session.

- *Registration*. A registration reflects a particular relationship between a consumer and a producer. It can include how the service is going to be charged or book-keeping modalities. This optional interface permits consumers to register, deregister and modify this relationship information. The portlet functionality can be dependent on whether the consumer is registered or not.

- *PortletManagement*. This optional interface gives consumers access to portlet state and property information. Specifically, it includes operations for getting portlet metadata (i.e., *getPortletDescription()*), cloning portlets for further customization, and setting/getting portlet properties.

According to the specification, producers *"are presentation-oriented Web Services that host Portlets"*. Hence, we model a WSRP **Producer** as a compound of **Portlets**. Figure 2.4 shows this situation. Both Producer and Portlet class attributes correspond to the metadata as retrieved by the *getServiceDescription* and *getPortletDescription* methods, respectively.

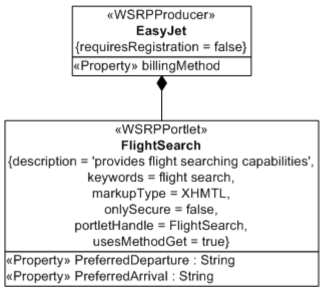

«WSRPProducer»
EasyJet
{requiresRegistration = false}
«Property» billingMethod
«WSRPPortlet»
FlightSearch
{description = 'provides flight searching capabilities',
keywords = flight search,
markupType = XHMTL,
onlySecure = false,
portletHandle = FlightSearch,
usesMethodGet = true}
«Property» PreferredDeparture : String
«Property» PreferredArrival : String

Figure 2.5: *flightSearch* portlet with its producer.

As an example of producer metadata, consider *"requiresRegistration"*. This field is a boolean which indicates whether the producer requires or does not require the consumer to be previously registered. On the other hand, *"portletHandle"* is a Portlet attribute. A handle serves to uniquely refer to the portlet at hand. Finally, a *"clones"* association is introduced to indicate the association between a portlet and its clones.

Both producers and portlets can have properties. A **Property** carries typed information between the consumer and the producer. Properties of the producer (e.g., *billingMethod*) are set at registration time, and they affect all the hosting portlets. By contrast, properties of the portlet (e.g., *PreferredDeparture*) are set when a portlet clone is created.

A property is described by a *name*, a *type*, a *label* (i.e., a short, human-readable name which is used to display in any consumer-generated user interface for administering purposes), and a *hint* (i.e., a short description of the property to be displayed as a tooltip when the property is edited).

Figure 2.5 depicts an example model with a producer (i.e., *EasyJet*) that offers the portlet *FlightSearch*. Attributes of the metamodel are mapped as tagged values on the model, and properties are reflected as attributes. The example model describes that prior registration for the use of that portlet is not required (due to *{requiresRegistration = false}*).

This portlet has a set of tagged values that indicates metadata about the returned markup. Some examples follow: the supported mime types (e.g., text/xhtml); the window states supported for each returned mime type (*{markupType =... }*); a brief description of the portlet functionality (*{description=... }*); a set of keywords, which can be used for search (*{keywords... }*); a flag which indicates that the generated markup includes the method *get* in an HTML form (*{usesMethodGet=... }*); whether the portlet requires secure communication on its default markup (*{onlySecure=... }*)

and so on.

Next, we will summarize the conversations struck up between portlet producers and consumers during the configuration and during the interaction with the portal end-user in the execution.

Conversation between the Portlet Consumer and the Portlet Producer for configuration: an example

1.- The consumer finds out about the producer. This basically implies the consumer getting the producer's metadata, with its description of the registration requirements, and, possibly, the list of the "Producer-offered Portlets". A snippet of the returned *ServiceDescription* structure follows:

```
<ServiceDescription xmlns="urn:oasis:names:tc:wsrp:v1:types">
   <requiresRegistration>true</requiresRegistration>
   <requiresInitCookie>none</requiresInitCookie>
   <offeredPortlets>
      <portletHandle>FlightSearch</portletHandle>
      <markupTypes>
         <mimeType>text/html</mimeType>
         <modes>wsrp:view</modes>
         <modes>wsrp:edit</modes>
         <windowStates>wsrp:normal</windowStates>
         <windowStates>wsrp:solo</windowStates>
      </markupTypes>
      <title lang="us">
         <value>FlightSearch</value>
      </title>
      <description lang="us">
         <value>...</value>
      </description>
   </offeredPortlets>
</ServiceDescription>
```

The *getServiceDescription()* operation provides a discovery means for a consumer to ascertain the producer's capabilities. In this example, these characteristics include: registration is required, no cookies are used, a *FlightSearch* portlet is available that uses *text/html* as the mime type, etc.

2.- The consumer registers with the producer. If registration is permitted, the consumer can state some of its specificities at this time. For instance, the consumer can restrict the *modes* or *window states* it is willing to manage by issuing the *register()* method with the following parameter:

```
<RegistrationData xmlns="urn:oasis:names:tc:wsrp:v1:types">
    <consumerName>myCompany_Portal</consumerName>
    <consumerAgent>...</consumerAgent>
    <methodGetSupported>true</methodGetSupported>
    <consumerWindowStates>wsrp:normal</consumerWindowStates>
</RegistrationData>
```

In the example, the consumer *myCompany_Portal* indicates that *"normal"* is the only *window state* supported.

3.- The consumer finds out about the portlets being offered by the producer it registered with. Through the *getServiceDescription()* method, the consumer knows the set of portlets offered by the producer (see step 1). Once the consumer is registered, the *getPortletDescription()* method can be used to obtain detailed information about those portlets, along with the preference set during registration. The answer is a *PortletDescriptionResponse* document as the one that follows:

```
<PortletDescriptionResponse>
   <portletDescription>
     <portletHandle>FlightSearch</portletHandle>
       <markupTypes>
         <mimeType>text/html</mimeType>
         <modes>wsrp:view</modes>
         <windowStates>wsrp:normal</windowStates>
       </markupTypes>
       <title lang="us">
         <value>FlightSearch</value>
       </title>
   </portletDescription>
</PortletDescriptionResponse>
```

4.- The consumer finds out about the properties available to configure the portlet. By this time, the consumer knows the configuration option available. Now, it discovers the properties through which it can set the corresponding amendments. To this end, the *getPortletPropertyDescription()* returns the following document:

```
<PortletPropertyDescriptionResponse ...>
   <modelDescription>
     <propertyDescriptions name="PreferredDeparture" type="types:AirportsType"/>
     <propertyDescriptions name="PreferredArrival" type="types:AirportsType"/>
```

```
        ...
      </modelDescription>
      <modelTypes>
        <!-- XMLSchema Type definitions for the properties -->
          <xsd:schema ... >
            <xsd:simpleType name="AirportsType">
              <xsd:restriction base="xs:NMTOKENS">
                <xsd:enumeration value="BIO"/>
                <xsd:enumeration value="MAD"/>
              </xsd:restriction>
            </xsd:simpleType>
            ...
          </xsd:schema>
      </modelTypes>
    </PortletPropertyDescriptionResponse>
```

This document states that the *FlightSearch* portlet supports a set of properties (e.g., *PreferredDeparture, PreferredArrival*) whose types indicate the range of values available.

Conversation between the Portlet Consumer and the Portlet Producer for interaction: an example

1.- The consumer receives a markup from the producer. This is achieved through the *getMarkup()* function. A snippet of the returned parameter follows

```
<MarkupResponse ...>
    <markupContext>
        <mimeType>text/xhtml</mimeType>
        <useCachedMarkup>false</useCachedMarkup>
        <markupString><html><tr>..</tr>...</table></markupString>
    </markupContext>
    ...
</MarkupResponse>
```

2.- The end-user interacts with the portlet fragment, and the consumer sends the interaction to the producer. This is achieved using the *performBlockingInteraction()* function, and one of its parameters contains name-value pairs to be processed. In the following example, *flightNumber* and *date* are two parameters with values filled by the end-user in a form. Portlet logic will process those values and will decide the new state of the portlet. To get the markup related to that state the consumer will use the *getMarkup()* function again.

```
<InteractionParams>
   ...
   <formParameters>
      <NamedString>
         <name>flightNumber</name>
         <value>LF9834</value>
      </NamedString>
      <NamedString>
         <name>date</name>
         <value>2008-08-25</value>
      </NamedString>
      ...
   </formParameters>
</InteractionParams>
```

2.5 Conclusion

Nowadays, there are many portal platforms which make hard the implementation of a Web portal or migrating it from one platform to another. Even using standards like JSR-168 and WSRP the deployment procedure varies from container to container, so a means to facilitate this procedure is needed. This is the aim of the work presented in Chapter 4.

Web Services for Remote Portlets 1.0 (WSRP) specification established the basis for interoperability between portlets and portals, however interoperability among portlets remained open. Distinct portlets, within the scope of the same producer, could share a common piece of information but portlets which pertain to distinct producers remained isolated. Although, a later WSRP version has proposed an event-based mechanism for inter-portlet communication, in the meantime, our work was concentrated on a semantic Web-based approach which is presented in Chapter 5.

Chapter 3

Model Driven Engineering

3.1 Introduction

Software engineering main efforts concentrate on developing new software systems and maintaining existing ones [107]. The aim is to get high-quality systems in a cost-effective way. Different approaches and technologies have been applied along the years, but with the increasing complexity of problem space many of them fall short of getting a good solution. Among these complexities we can mention the heterogeneity of hardware architectures, inherent complexities in network-centric, dynamic and large-scale systems, strict simultaneous quality of service (QoS) demands, integration of autonomous application domains, and so on. Moreover, we should not forget complexities related to the solution space, i.e., the use of third-generation languages to write and maintain manually most application and platform code, the growth of platform complexity (which has evolved faster than those general-purpose languages), highly heterogeneous platform, language and tool environments, the need to integrate legacy applications, the inherent abstraction of reusable components and frameworks (which makes it hard to engineer their quality and to manage their production) [94, 93].

Facing some of those challenges is the rationale behind *Model-Driven Engineering* (MDE). Model-Driven Engineering is in the origin of different projects or approaches, OMG's Model Driven Architecture (MDA), Domain-Specific Modelling

(DSM), Generative Programming, Software Factories, and so on. While there are differences between these approaches, the common goal is to achieve a higher-level of abstraction in software development. This chapter outlines MDE. The aim is not to provide an exhaustive description of the MDE field but provide the grounds to understand how MDE principles have been used throughout our work.

This chapter is structured as follows. First, MDE is introduced. Next, two main approaches, Model-Driven Architecture (MDA) and Domain-Specific Modelling (DSM) are addressed in Sections 3.3 and 3.4, respectively. Finally, Section 3.5 focuses on model transformation, and introduces the transformation language used in the work presented in this dissertation: RubyTL.

3.2 Model-Driven Engineering

Model Driven Engineering (MDE) can be defined as a software development method where all the relevant information in the project is stored in some kind of abstract model. Software design and validation is then carried out as a set of model transformations. Bézivin [13] compares the evolution and achievements of object technology with MDE. He claims that the motto *"Everything is an object"* of object-oriented engineering, has a counterpart in MDE through the principle *"Everything is a model"*. Two basic relations give support to that principle: *representedBy* and *conformsTo*. A real system can be captured by a model, i.e., it is represented by a model, and each model is written in the language of its metamodel (formal specification of an abstraction), i.e., it conforms to a metamodel.

Kent [52] sets a framework for MDE with following features,

- *Dimensions* of the modelling space, i.e., a way to structure the modelling space categorizing perspectives. Kent [52] mentions the followings: the dimension of the platform (specific vs. independent), the dimension of subject area (e.g., the area of the system dealing with customers and the area for processing of orders), aspect dimension (e.g., concurrency control and distribution), and the dimensions concerned with the managerial and societal aspects (e.g., authorship, version control, and location). The model built in the development process will be at the intersection of the different dimensions, i.e., it can take on different perspectives.

- *Modelling languages* for the models. Domain models are described using a language, and thinking about language definitions as a type of models (called metamodels), another language for them is needed. So there are languages for the models and languages for the metamodels. The latter includes Extensible Markup Language (XML) and Meta Object Facility (MOF) [52].

- *Translations* between models. Kent [52] distinguishes between model translation (as the mappings between models in the same language) and language translation (as the mappings between models in different languages). Other authors refer to *endogenous* transformation for model translation and *exogenous* transformations for language translations. Model translations should be writable in the same language (or an extension of it) as the models are expressed, however language translation must be expressed in terms of the definitions of the languages themselves (that is, language translation is meta-model translation).

- *Processes*. MDE has both models and translations as its main artefacts. And the process of software construction is conceived as a successive translation of models till code is obtained. Kent [52] distinguishes between macro processes, which concern the order in which models are produced and how they are coordinated, and micro processes, which guide for producing a particular model. The definition of the models developed by a particular process and the definition of that process are dependent on each other.

- *Tools*. The value of models is greatly enhanced if appropriate tooling is available for model simulation, validation, verification and transformation. This point should not be underestimated. Models have been used since the beginning of programming. Statecharts, entity-relationship diagrams or dataflows have been used for a long time. The difference now is that models are not just documentation but full-fledged software artefacts, liable to be processed. But what is meant by "model processing"? Code processing implies compiling or linking. Likewise, model processing implies transforming, validating or verifying models. The availability of tools to achieve these operations is inherent in MDE. MDE does not really exist if model tooling is not in place: tools to check/enforce well formedness constraints on models, to support mappings between models, to support model driven testing, for version control and distributed working, for managing the software process, for working with instances of models, and so on.

Other common names for this discipline are Model-Driven Software Development

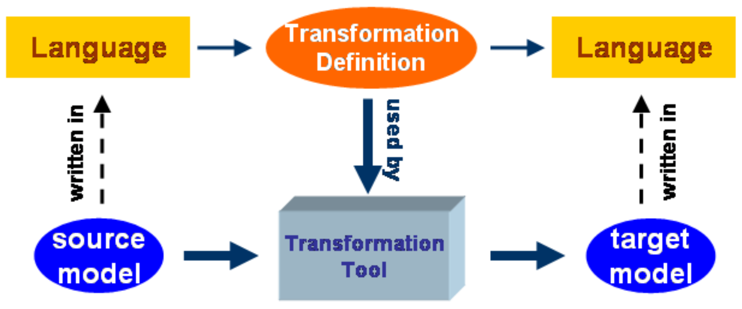


Figure 3.1: The MDA basic framework.

(MDSD) and Model Driven Development (MDD), and among the best known realizations of MDE are Model-Driven Architecture (MDA) and Domain-Specific Modelling (DSM). OMG's MDA focuses on the platform dimension and proposes the use of the MOF language to describe metamodels, and specifically, UML and its profiles as metamodel language for modelling of models. DSM is centered in the problem domain, it proposes starting with the definition of a Domain-Specific Language (DSL) to describe the models. MDA and DSM are the topics of the next two sections.

3.3 Model-Driven Architecture

The OMG's Model-Driven Architecture (MDA) framework [76] is made up of different elements: models, transformations and transformation tools. In [53] all these elements are defined as:

- *"A model is a description of (part of) a system written in a well-defined language". "Platform Independent Model (PIM) is a model with a high level of abstraction that describes a system without any knowledge of the final implementation platform". "Platform Specific Model (PSM) is tailored to specify the system in terms of the implementation constructs that are available in one specific technology". "Computation Independent Model (CIM) describes the requirements for the system and the situation in which the system will be used. It shows the system in the environment in which it will operate".*

- *"A well-defined language is a language with well-defined form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer".*

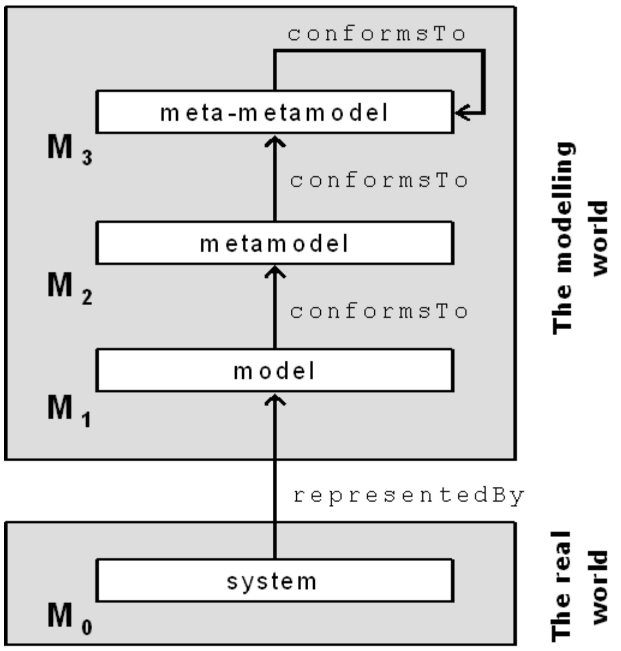


Figure 3.2: MDA's four-layer organization[13].

- *"A transformation is the automatic generation of a target model from a source model, according to a set of transformation rules. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language"*.
- *"A transformation tool performs a transformation for a specific source model according to a transformation definition"*.

The relationship among these elements is shown in Figure 3.1. Moreover, MDA proposes the metamodelling mechanism to define well-defined languages. A model defines what elements can exist in the system and these elements come from the language used in the modelling. In turn, considering a language as a system to be defined, a model is needed to describe the language: this model will be a metamodel. Therefore, a metamodel completely defines a language that will be used to write a model, and MDA framework does not make a distinction between metamodel and language. Moreover the OMG uses a four-layered framework [13] with two relationships *conformsTo* and *representedBy* (see Figure 3.2).

- *Layer M0*: *the instances*. They are the items in the business itself, the software representations of the real world items, and the like [53].

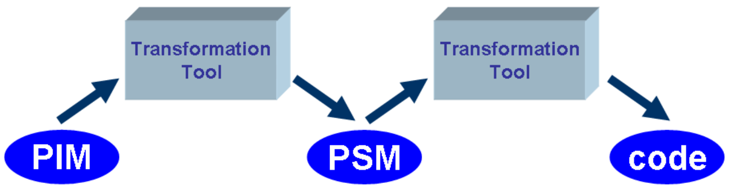


Figure 3.3: The MDA development process [53].

- *Layer M1*: *the model of the system*. The concepts at the M1 layer are all classifications of instances at the M0 layer. Each element at the M0 layer is always an instance of an element at the M1 layer. Therefore the real system *is represented* by the M1 layer model. An M1 model provides abstractions of a physical system that allow engineers to reason about that system by ignoring extraneous details while focusing on relevant ones [13].

- *Layer M2*: *the metamodel of the model*. The elements that exist at the M1 layer conform to concepts at M2. The same relationship that is present between elements of the M0 and M1 layers exists between elements of M1 and M2: the concepts at the M2 layer are all classifications of instances at the M1 layer. The model at the M2 layer is called a metamodel. A metamodel is a formal specification of an abstraction, usually consensual and normative [13], in other words, a metamodel defines a modelling language in which to write models. UML is an example of such languages, but MDA is not restricted to UML and any well-defined language is accepted.

- *Layer M3*: *the metamodel of M2*. The elements at the M2 layer conform to elements at M3, and as before, the concepts at the M3 layer are all classifications of instances at the M2 layer. Within the OMG, the MOF (Meta Object Facility) is the standard M3 language, and UML conforms to MOF.

Instead of defining an M4 layer (for the model of M3) and so on, the OMG established that all elements of the M3 layer must be defined as instances of concepts of the M3 layer itself. The M3 layer allows building coordination between models, based on different metamodels. One example of such coordination is model transformation [13].

The main difference of MDA process (see Figure 3.3) from the traditional development is that the transformations from model to model and from model to code are automated, i.e., a transformation tool is used. A transformation generates a tar-

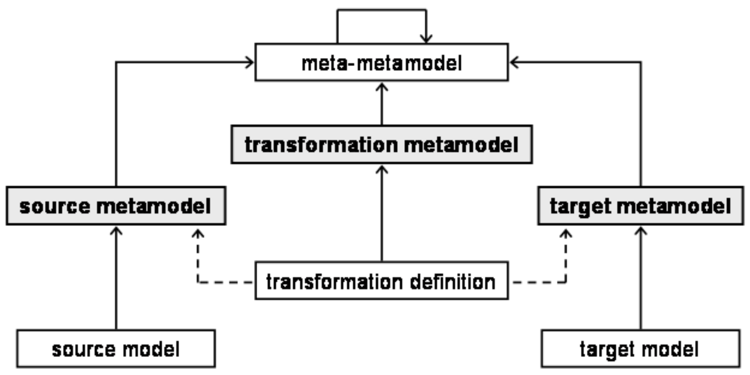


Figure 3.4: Transformation as model [13].

get model from a source model (see Figure 3.1). Applying *"Everything is a model"* principle, the transformation itself should be a model. And as a model conforms to a metamodel, the transformation model (i.e., the transformation definition) conforms to a metamodel, which defines the common model transformation language (see Figure 3.4). One of these transformation languages is described in Section 3.5.

As for tooling, some examples follow: ArcStyler of Interactive Objects [45], OptimalJ of Compuware [18], and androMDA [3], to name just a few (for an updated account refer to *http://www.omg.org/mda/committed-products.htm*)

The use of model (and abstractions), and the reuse that goes with transformations are reckoned to achieve the following goals [100]:

- Increase of software development efficiency through automation and avoidance of redundancy.
- Increase of software quality through propagation of quality attributes from production (factory) to product.
- Better changeability and maintainability of software.

3.4 Domain-Specific Modelling

Domain-Specific Modelling narrows down the design space or domain. In other words, it raises the level of abstraction by specifying the solution directly using domain concepts. It uses a Domain-Specific Language (DSL) to represent the various facets of a system. A Domain-Specific Language is a custom language that targets a specific problem domain, it represents the domain knowledge. It is designed so that it can more directly represent the problem domain which is being addressed. Domain-Specific Languages usually support higher-level abstractions than general-purpose modelling languages (e.g., Unified Modelling Language (UML)) and they can only be used to describe a specific problem. Examples of Domain-Specific Languages in the industry are the Nokia's language for SmartPhone applications, Systems Modelling Language (SysML) specified as a UML 2.x Profile and suited for systems engineering applications, PostScript, its domain is page rendering, to describe the appearance of text, graphical shapes, and sampled images on printed or displayed pages, SQL that targets the querying and updating of databases, a very specific domain. Domain-Specific Languages are known as *modelling languages* too, because they are used to build models of the domains they address.

To define a language another language is needed. The language of a model is often called a metamodel, so the language for defining a modelling language is a metametamodel. Usually metametamodels or modelling languages are derived from or customizations of existing languages, for instance, EBNF, XML Schema, and MOF. The strengths of those languages tend to be in the standardization of the original language.

Of key importance is the ability to define and maintain the Domain-Specific Language. Company experts encapsulate their expert knowledge in the modelling language, this reflects the problem domain in which developers are working. The process for the definition can be divided into three phases [105]: (1) Identifying abstractions of the problem domain and how they work together; (2) Specifying the language concepts and their rules (i.e., metamodel) and mapping the major domain concepts to modelling language objects; (3) Creating the visual representation of the language (i.e., notation). The first phase demands a thorough domain analysis: identifying which abstractions are the same for all products and which are different, identifying their properties and connections, and so on. The constructed metamodel should allow to define the concepts and properties of the language, hierarchy structures, and correctness rules.

A Domain-Specific Language is *"a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain"* [106]. Traditional modelling languages like UML are on the same level of abstraction as the programming languages supported, and as Tolvanen [105] says *"changing the representation of a construct without increasing the abstraction level doesn't*

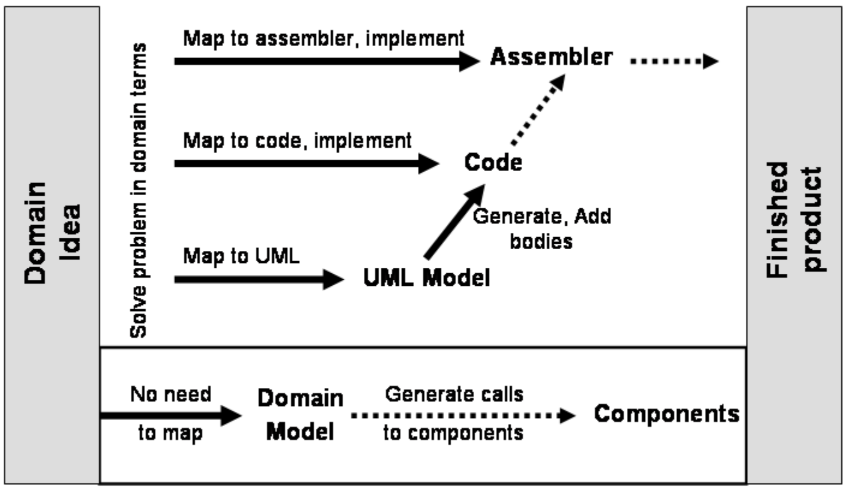


Figure 3.5: Domain-Specific Modelling vs. Traditional development [24].

improve productivity. In UML, using a rectangle symbol to illustrate a class in a diagram and later creating the equivalent code representation in a programming language does not provide a higher level of abstraction!”. As Kelly [51] states the modelling language can be used with customers for discussing and thinking about applications, and therefore, *“the tight fit with the problem domain, high level of abstraction, precision, and ability to use with customers are clear areas where DSLs excel over generic languages like UML. ... The greatest productivity increases come from in-house domains, because the language can be made to precisely fit the needs of just one company”*.

Once the Domain-Specific Language is defined, Domain-Specific Modelling also includes the idea of code generation. Building a generator is about defining how model concepts are mapped to code, with the aim for automating the creation of executable source code in a chosen programming language from the DSL models, from the high-level specifications. The application is automatically generated from the high-level specifications with domain-specific code generators in a similar way as programs in a general-purpose programing language are transformed in code of assembler language. Figure 3.5 illustrates the difference between the UML-based development and the domain-specific development [24]. Within the former the developer has to solve the problem using domain concepts and after he has to map into UML concepts first and then to map them into code. These steps have to be processed by the developer, perhaps with the aid of a tool, but ’by hand’. Getting the assembly code is the only step that is automatic, i.e., processed by a compiler. As for the latter, as been explained in this section, the problem is solved using the domain concepts, i.e., the language specific for the domain, and the generator creates the code automatically.

The expert developers also condense their experience into the definition of a code generator. In Tolvanen's opinion [105], a good generator should produce complete code, i.e., the developer should not need modifying the generated product, in the same way as the machine code created by a C compiler is not modified. Moreover, model-based generators should target code directly, instead of producing intermediate models that need to be extended during the development process, i.e., the source model and the generated code must be completely synchronized. General-purpose modelling languages such as UML are well suited for documentation, but not as well suited for generation. UML models are at sketching level and code generation requires that details are correct too.

Within Domain-Specific Modelling methodology an expert developer, who masters the problem domain and has experience in developing software in that domain, defines a Domain-Specific Language containing the chosen domain's concepts and rules, and specifies the mapping from that to code in a domain-specific code generator [105]. The other developers then make models with the modelling language, guided by the rules. Of note, only the concepts that exist within the Domain-Specific Language are valid candidates for discussion during the design phase of the new software and the model avoids delving into technological details. Once the customers and developers agree with the models, the code is automatically generated. Another important question pointed out in [51] is that the company controls the language and generators, so they can evolve as the problem domain evolves.

Moreover, changes in the technological platform can be added by manipulating the code generator, leaving the model unchanged.

As for tooling, some examples follow: DOME (the DOmain Modelling Environment) of Honeywell [42], Software Factories of Microsoft [34], MetaEdit of Metacase [62] and GME (the Generic Modelling Environment) of ISIS institute in Vanderbilt University [44]. These development environments offer tool sets to define new Domain-Specific Languages, and then, taken the language, to edit a model and generate the corresponding code.

3.5 Transformation Languages

Transformations lie at the heart of MDA. Model transformation is the process of converting one or more models, called source models, to one output model, the target model, of the same system (see Figure 3.1). This section first outlines a classification of transformations along the lines of the one given in [76], [19] and [55]. Next, we focus on a specific transformation language, RubyTL. Following Chapter 4 uses this classification and language to support an MDD approach to portal design.

3.5.1 Transformation classification

Koch *et al.* [55] and MDA Guide [76] distinguish the following aspects of transformations: type, complexity, use of marks, execution and implementation types.

- *Transformation Type*. Model transformations can be of type CIM to PIM, PIM to PSM and PSM to code. Moreover, a model in an abstraction level can be refined in another one of the same level, i.e., a CIM can be mapped to another CIM, and the like. Although there is not a complete agreement [55] transformations like PSM to PIM are also included. This transformation type is required for abstracting models of existing implementations in a particular technology into a platform-independent model. This procedure often resembles a "mining" process that is hard to be fully automated [63].

- *Transformation Complexity*. [55] A transformation is simple or a merge, depending on the number of source models involved. A merge transformation combines elements of different source models in order to build a target model.

- *Use of Marks*. Transformation rules rely on certain marks (types, patterns, templates or UML profile elements) in order to select the elements to which a rule applies. These marks can be part of the elements of the source model, such as stereotypes of UML profiles and patterns, or be included in the transformation definition, like templates, which are parameterized models that may include much more specific specifications to guide the transformation.

- *Execution Type*. Transformations are classified in automatic, semi-automatic and manual based on the decisions the designer takes on the source and target models. A transformation is automatic if the source model can provide all the information needed for implementation, and there is no need to add marks in order to be able to generate the target model. The transformation is semi-automatic if the source model is prepared using a platform independent UML profile or patterns are used in the specification of a mapping. It is manual if the designer makes design decisions and is who produces the results.

- *Implementation Technique.* Czarnecki and Helsen [19] classify transformation languages into 8 categories:
  - *Model-to-Code.* Code is generated simply as text. Two types are distinguished:
    - *Visitor-based approach.* It consists in providing some visitor mechanism to traverse the internal representation of a model and write code to a text stream.
    - *Template-based approach.* The template consists of the target text containing slices of metacode to access information from the source and to perform iterative expansion.
  - *Model-to-Model.*
    - *Direct-manipulation approach.* It offers an internal model representation plus some API to manipulate it. Users have to implement transformation rules using a programming language such as Java.
    - *Relational approach.* The relation between the source and the target elements is specified declaratively, using constraints.
    - *Graph-transformation-based approach.* It operates on typed, attributed, labelled graphs, specifically designed to represent UML-like models.
    - *Structure-driven approach.* It has two distinct phases: the first phase is concerned with creating the hierarchical structure of the target model, whereas the second phase sets the attributes and references in the target.
    - *Hybrid approach.* It combines different techniques from the previous categories. QVT specification [77] has a hybrid declarative/imperative nature, with the declarative part being split into a two-level architecture.
    - *Other.* The transformation framework defined in the OMG's Common Warehouse Metamodel (CWM) Specification [75], and XSLT [109] to transform models serialized using XMI.

3.5.2 A transformation language: RubyTL

In order to describe the transformations, MDA proposes the standard *Query/Views/-Transformations Specification* (QVT) [77]. We resort to RubyTL [99] because near the spring of 2006 it could fulfill the transformation requirements posed by our MDD-approach to portal development. Requirements such as need for recursion, local-to-global transformations, and in general, "transforming-in-the-large", which we did not see clear in QVT. Such characteristics will be better comprehended when providing the specific cases in Chapter 4.

RubyTL is the main outcome of the current PhD work of Jesus Sánchez Cuadrado, led by Jesus García Molina at the University of Murcia (GTS Research Group). It should be said that the use of a beta-program such as RubyTL is always a risky business, where bug-amendment should be included as part of the challenges!. We much appreciate the involvement of the GTS group in helping us to understand (and debug) RubyTL.

RubyTL supports both model-to-code and model-to-model transformations, the former in a template-based approach, and the latter, in a hybrid approach. Imperative constructs described in Ruby could be used to write some kinds of complex transformations, not suitable for a declarative style. Ruby [104] is an object-oriented and dynamically typed programming language. It is very suitable to embed a Domain-Specific Language in it and hence RubyTL has been defined as a Ruby internal DSL. That means that parsing and evaluating the RubyTL transformation definition is on hands of the Ruby interpreter.

- Source metamodel, target metamodel and source model are XMI files. The parser reads these input files and a set of Ruby classes are generated and loaded in the Ruby interpreter. These classes correspond to the classes defined in the source and target metamodels.

- Once metamodels have been loaded, the transformation definition is read by the Ruby interpreter itself, which leads to the creation of a set of rule objects. These rules will be used by the transformation engine to perform the transformation.

- Applying a rule is simply executing the code block of its mapping part. Just before a rule is applied, new target elements are created - one element for each metaclass specified. While the first parameter of the mapping code block receives the source element, the rest of parameters receive the target elements

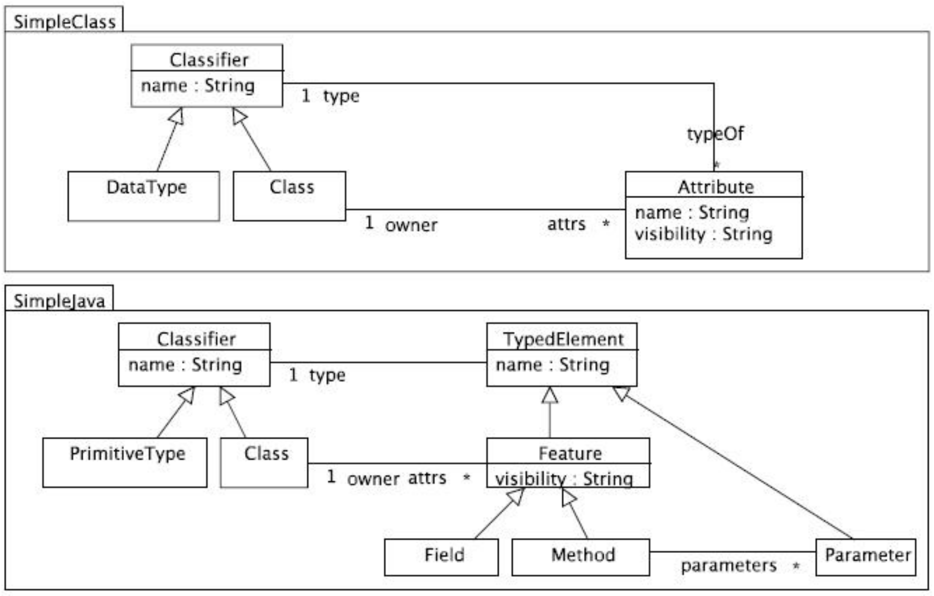


Figure 3.6: *SimpleClass* and *SimpleJava* metamodels [99].

created as a result of the rule execution. A source element is never transformed twice by the same rule.

- Since RubyTL is an embedded DSL, checking if a transformation definition is well-formed must be achieved at runtime. A transformation definition is well-formed if for each binding involving two non-primitive types there exist one or more conforming rules but there is one and only one applicable rule. As for the *"conforming rule"* concept, a rule conforms to a binding if the source and target types in the rule conforms to the types in the left and right parts of the binding assignment.

- The output of the transformation process is an XMI file containing a target model conforming to the target metamodel.

A transformation definition in RubyTL is a set of transformation rules packaged in a transformation module, and each rule has a name and four parts [99]:

- A *from* part, where the source element metaclass is specified.

- A *to* part, where the target element metaclass (or metaclasses) is specified.

```
module Transformation
  rule 'klass2javaclass' do
    from SimpleClass::Class
    to SimpleJava::Class
    mapping do |klass, javaclass|
      javaclass.name = klass.name
      javaclass.attrs = klass.attrs
    end
  end
  rule 'attribute2features' do
    from SimpleClass::Attribute
    to SimpleJava::Field, SimpleJava::Method, SimpleJava::Method
    filter do |attr| attr.visibility == 'public' end
    mapping do |attr, field, get, set|
      field.name = attr.name
      field.type = attr.type
      field.visibility = 'private'
      get.name = 'get' + attr.name
      get.type = attr.type
      get.visibility = 'public'
      set.name = 'set' + attr.name
      set.visibility = 'public'
      set.parameters = attr.type
    end
  end
  ...
end
```

Figure 3.7: Excerpt of a transformation module [99].

- A *filter* part, where a condition over the source element is specified, such that the rule will only be triggered if the condition is satisfied; this part is optional and if a rule has no filter it will always be triggered.

- The *mapping* specifies relationships between source and target metamodel elements. These relationships can be expressed either in a declarative style, through a set of bindings, or in an imperative style using Ruby constructs. However, the declarative style is recommended.

As a sample case consider two metamodels*: SimpleClass* and *SimpleJava* metamodels. Figure 3.6, taken from [99], provides the details[1]. Figure 3.7 shows an excerpt of a transformation definition with two rules. It uses *SimpleClass* as source metamodel

[1]According to the *SimpleClass* metamodel, a class is composed of attributes; an attribute has a name and a visibility and the type of an attribute can be a class or a primitive type. According to *SimpleJava* metamodel a Java class is composed of features which can be fields or methods; a method can have zero or more parameters; both features and parameters are typed, therefore they inherit from *TypedElement*, which gives them a type and a name.

and *SimpleJava* as target metamodel. More specifically, it is a transformation from a class model to a Java model, such that i) each class is transformed to a Java class, ii) each public attribute of a class is transformed to a pair of get/set methods plus a private field in the Java class, and iii) each private attribute of a class is transformed to a private field in the Java class. Next, using the sample transformation we explain some language features:

- Metamodels are described in packages and to use the metamodel classes (metaclasses) in the *from* and *to* parts the name of the metaclass is prefixed by the name of the package in which that metaclass is enclosed. Thus, the *SimpleClass::Class* expression stands for the *Class* metaclass of the *SimpleClass* metamodel.
- The first parameter of the mapping code block receives the source element, the rest of parameters receive the target elements created as a result of the rule execution. In the example, the mapping code block of the *attribute2features* rule has four parameters: *attr* whose type is *SimpleClass::Attribute*, *field* whose type is *SimpleJava::Field*, and *get* and *set* whose type is *SimpleJava::Method*.
- The binding *javaclass.attrs = klass.attrs* in the first rule establishes a mapping from class attributes to Java attributes (features) and yields to the execution of a rule that specifies such mapping (i.e., the *attribute2features* rule). Instead of this type of rule invocation, an explicit invocation is also available, i.e., to call rules by their name; in the example the binding might have been like this: *javaclass.attrs = attribute2features(klass.attrs)*.
- The bindings established between primitive types (e.g., *field.name = attr.name*) do not involve any rule invocation.
- In the *attribute2features* rule the *attr.visibility == 'public'* filter expression checks if the attribute visibility is public. The evaluation of the filter decides if the mapping block is executed or not.
- RubyTL does not use any OCL-like query language, instead it uses a Ruby library for managing collections. For example, the *klass.attrs.select {|attr| attr.visibility == 'public'}* expression collects all the public attributes of a class.

So far the basics of RubyTL are introduced. More advanced features include: the *phasing* mechanism (a transformation definition is organized in an ordered set of phases and each phase consists of a set of rules; the order in which phases are executed is decided by the user), *top rules* (a top rule is always applied to all instances of the type specified in its *from* part, thus a transformation definition could have more than one entry point) or *creator rules* (rules evaluated more than once for a source element). The description of these features falls outside the scope of this dissertation (refer to [99] and [98] for further details).

3.6 Conclusion

This chapter just provides background on MDD needed to understand the work presented in Chapter 4. Specifically, the artefacts, process and advantages of MDD are briefly introduced. An introduction to transformation is also presented together with an outline on RubyTL, the transformation language used in this work.

Chapter 4 defines PIM and PSM metamodels for Web portal development and then, it defines a RubyTL transformation rule-set.

Chapter 4

Portlet-based portal construction: an MDD approach

4.1 Introduction

The significance of portal applications stems not only from being a handy way to access data but also from being the means of facilitating the *integration* with third-party applications. This has led to the so-called *portal imperative*: the emergence of portal software as a universal integration mechanism [103].

Key to this view is the notion of *portlet* (see Chapter 2). Portlets are the enablers of service-oriented architectures (SOAs) but now at the front-end. This perspective requires a departure from how current Web portals are envisaged. The Web portal is no longer perceived as a set of pages but as an integrated set of Web components (i.e., portlets) that are now delivered through the portal. From this perspective, the portal page now acts as a mere conduit for portlets. Page and page navigation dilute in favor of portlet and portlet orchestration.

However, Integrated Development Environments (IDE) for portal construction have not yet exploited the full potential of portlets (e.g., Plumtree, IBM WebSphere, Oracle Portals, Liferay, eXo). The content-management role is still prevalent in most of current IDEs. The Web portal is still conceived as a conglomerate of pages where portlets tend to be considered as a modular mechanism during page implementation.

The page is still the main notion, and the portlet is subordinated to the page. All portlets are readily rendered when entering in the containing page. This has important consequences since *portlet* navigation (i.e., browsing along the portlet fragments) is completely detached from *portal* navigation (i.e., browsing along the portal pages).

The work presented in this dissertation promotes the service-oriented view of Web portals and portal design abstracted from pages. Portal design is now conceived in terms of portlets and portlet orchestration. More specifically, it abstracts *portal* and *portlets* into *workspace* and *tasks,* respectively, and it proposes an extension to statecharts as a formalism to describe the flow among tasks, i.e., to specify both the structural organization and the browsing semantics of portlet aggregation [29].

On the other hand, the mapping from portlet orchestration (design time) to page navigation (implementation time) is too tedious and error-prone. The fact that the same portlets can be placed in distinct pages produces code clones that are repeated along the pages that contain these portlets. This redundancy encourages us to look for a solution around the model-driven development (MDD). The work proposes a MDD approach to service-oriented portal development. It follows the MDA (Model-Driven Architecture) proposal [76], and defines a Platform Independent Model (PIM), a Platform Specific Model (PSM) and the mapping transformations from the former to the latter.

The rest of the chapter is structured as follows. Section 4.2 provides an overview of portal development as an MDD development process. Section 4.3 introduces a metamodel for WSRP interfaces. The Platform Specific Model (i.e., the *eXo* platform) and the Platform Independent Model (i.e., annotated statecharts) are the topics of Sections 4.5 and 4.4, respectively. The transformation between the PIM and the PSM is addressed in Section 4.6. Next, Section 4.7 outlines the benefits drawn from the approach. Related work is presented in Section 4.8. Finally, Section 4.9 sums up conclusions from this approach.

4.2 Outline of an instance of the MDD approach

The MDD development approach strives to separate platform independent design from platform specific implementation and, in so doing, to delay as much as possible the dependence on specific technologies. The idea is to create distinct (meta) models of a system at different levels of abstraction. Then, transformations are applied that eventually produce code. Hence, code programming is substituted by modelling and transforming. Consequently, MDD focuses on the construction of models, specifica-

	Type	**Complexity**	**Approach**	**Execution**	**Techniques**
Identify portlets	–	–	–	manual	–
Get WSRP model	code to PSM	simple	metamodel	automatic	RubyTL
Transf. from portlets to tasks	PSM to PIM	simple	metamodel	automatic	RubyTL
Get orchestration skeleton	PIM to PIM	simple	metamodel	automatic	RubyTL
Complete orchestration model	PIM to PIM	simple	–	manual	–
Get rendering skeleton	PIM to PIM	simple	metamodel	automatic	RubyTL
Complete rendering model	PIM to PIM	simple	–	manual	–
Model merging	PIM to PIM	simple	metamodel	automatic	RubyTL
Transformat. into *eXo* model	PIM to PSM PSM to code	merge	metamodel	automatic	RubyTL

Table 4.1: Characteristics of model transformations in the SOP MDD process.

tion of transformation patterns, and automatic generation of code. And, the software development process is regarded as a pipeline of model transformations that eventually leads to a complete application.

Models. The best-known MDD realization is the OMG's Model-Driven Architecture (MDA) [76]. MDA suggests building computational independent models (CIMs), platform independent models (PIMs), and platform specific models (PSMs) corresponding to different levels of abstraction or viewpoints (see Chapter 4 for more details). This work focuses on PIMs and PSMs for portlet-based portals. To this end, three metamodels are introduced, namely,

- *WSRP* metamodel, which is a PSM metamodel of the interfaces defined by the WSRP standard,
- *SOP* metamodel, which is a PIM metamodel that promotes a service-oriented architecture also for Web portals. The notion of perspective is used to separate the specification of the portal along three distinct concerns, namely: the TASK model, the ORCHESTRATION model and the RENDERING model,
- *EXO* metamodel, which is a PSM metamodel for the *eXo* platform, better said, a view on the implementation details of relevance for this work.

These metamodels are described in more detail in Sections 4.3, 4.4 and 4.5, respectively.

Transformations. Model transformation is the process of converting one model (a.k.a. source model) to another output model (a.k.a. the target model) of the same

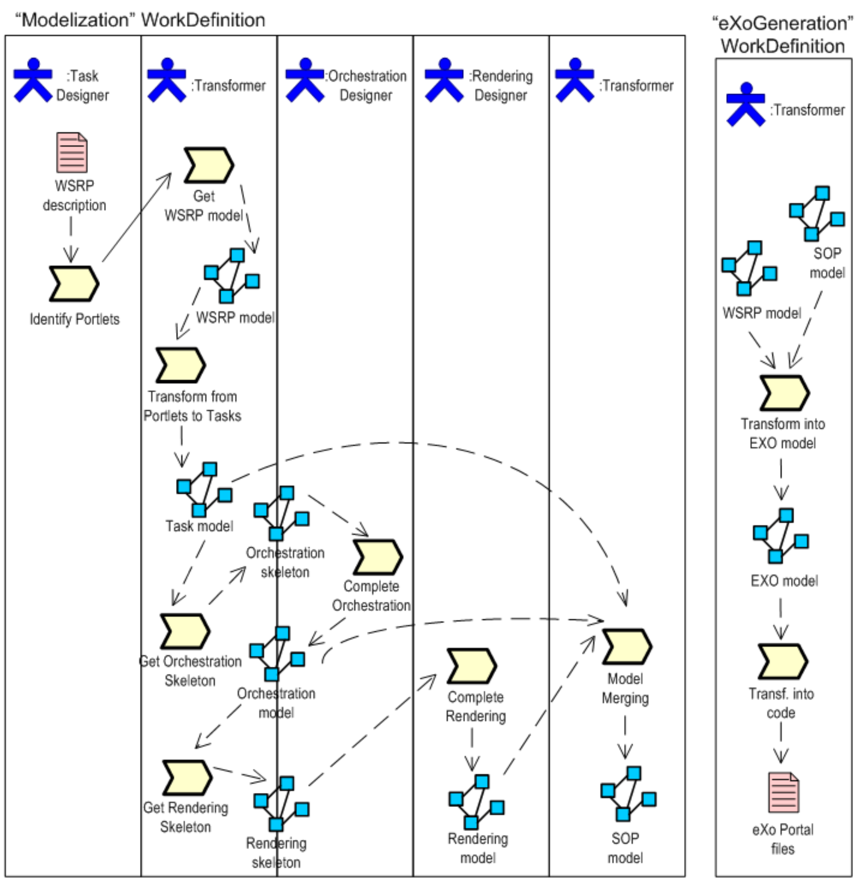


Figure 4.1: The SOP process.

application [76]. Table 4.1 outlines the distinct transformations used in this work along the characteristics introduced in [55, 76]. Model transformations can be of type *CIM to PIM, PIM to PIM, PIM to PSM, PSM to PIM* and *PSM to code*. Our work uses extensively the *PIM to PSM* transformation. However, it is also possible a bottom-up approach where existing components or legacy systems are wrapped, and abstracted into higher models to be then integrated into more complex applications [64]. In our opinion, SOA also requires this *PSM to PIM* transformations to abstract from service descriptions into design models that can be later integrated to achieve broader functionalities through service orchestration. Portlets as front-end Web services, use this approach to abstract away the WSRP interface specification. This allows to integrate also non-WSRP compliant portlets. Additionally, transformations can be also classified as *simple* or *merge* based on the number of source models involved in the mapping process. *'Transformation into eXo model'* is an example of a merge transformation with two source models: SOP and WSRP.

The MDD process. MDD conceives development as transformation chains where the artefacts that result from each phase must be models. SPEM (*Software Process Engineering Metamodel*) is a notation for defining processes and their components whose constructs are described in UML notation [81]. In MDD terms, SPEM is a metamodel for process modelling. Hereafter, SPEM terminology is used to specify the milestone, roles and dataflow that go with producing an *eXo* portal from a set of WSRP-compliant portlets through a chain of model transformations (see Figure 4.1).

First, four distinct *ProcessRoles* are introduced: the *transformer* as such (i.e., a set of rules for model mapping); the *task designer*, which is responsible for determining the portlets to be integrated; the *orchestration designer*, which defines how tasks are intertwined; and the *rendering designer*, which focuses on the look-and-feel of the Web portal.

These roles collaborate along two main *WorkDefinitions*: *modelization* and *eXoGeneration*. This work views Web portals as integration platforms built upon existing portlets i.e., portlets are already there when the portal is being designed. Hence, *modelization* starts by taking a textual description of the WSRP interfaces of the available portlets, and producing the TASK model that extracts only those features that are relevant during design.

Next, this TASK model is used to obtain a first skeleton of the ORCHESTRATION model with the basic milestone of the orchestration. This template is subsequently enriched with flow dependencies by the orchestration designer. This ORCHESTRATION model serves in turn to produce a first template of the distinct decorators to be faced during the specification of the RENDERING model. This template is then filled up by the rendering designer with appropriated aesthetic parameters.

Once the three perspectives are completed (i.e., TASK, ORCHESTRATION and RENDERING), they are integrated into the SOP model which constitutes the main *WorkProduct* of the *modelization* process. This SOP model is used to validate the interaction of the distinct perspectives as well as to check the correctness of distinct portal constraints (see next sections for details).

Finally, during the *eXoGeneration* process, the SOP model is automatically transformed into an EXO model which is later used to generate the *eXo* code. Interestingly enough, this final transformation also takes as input the WSRP model which contains implementation details about how portlets can be deployed.

Next sections introduce the details of the models and transformations.

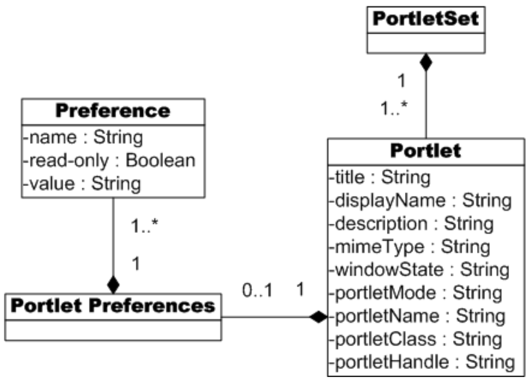


Figure 4.2: The WSRP metamodel.

4.3 The WSRP model: a PSM for the WSRP interfaces

WSRP [71] standardizes the programmatic interfaces for portlets (see Chapter 2). Besides method signatures, WSRP 1.0 standardizes window states, CSS classes for portlet rendering, *P3P*-based user profile description, etc. For the purpose of the work in this chapter it is important to note that WSRP 1.0 treats portlets as isolated entities where interoperation between portlets occurs beneath the GUI interface (e.g., sharing some data common to two portlets by using the so-called “portlet application”) [21]. However, a new version, WSRP 2.0, introduces an event-based mechanisms where portlets can subscribe to events being generated by other portlets. This enhancement is not considered in this work where portlets are treated as isolated components.

Moreover, another specification more has been taken into account, i.e., the OASIS *ebXML Registry Services Specification* [70]. ebXML Registry defines a framework for global electronic business that will allow service producers to find each other and conduct business based on well-defined XML messages. A key ingredient of this framework is the ebXML Registry Information Model where organizations can describe their profile (the so-called Collaboration Protocol Profile), and offerings can be canonically described. WSRP portlets have been added as one of these offerings. To this end, metadata about the Portlet descriptions as well as about the Service Implementation WSDL [112] for the Producer Service need to be published. Specifically, ebXML Registry [72] proposes a way to publish the following major WSRP artefacts: (1) WSDL description for a WSRP Producer service, (2) metadata describing a Producer service and the Organization which provides it (optional), and (3) metadata describing one or more WSRP Portlets hosted within Producer services (optional). For our purpose, this work just focuses on the latter.

Ours is a bottom-up approach for Web portal design, which will start gathering available WSRP portlets. The designer can use ebXML Registry Services and get WSRP portlet metadata. Thus, the proposed metamodel, shown in Figure 4.2, represents a simplified description of WSRP metadata.

4.3.1 A WSRP model for the sample case

As an example, the following portlets are available[1]:

- the *IEEESearch* portlet, which comprises a subset of the functionality of the IEEE portal;
- the *ACMSearch* portlet, which covers part of the offering of the *portal.acm.org* for the ACM organization;
- the *CiteSeerSearch* portlet, which includes the functionality for author searching at *citeseer*;
- the *DBLPSearch* portlet, which embodies the functionality for author searching at Ley's site;
- the *DeliciousStore* portlet, which provides the functionality available at *del.icio.us* for keeping track of references found in the Web;
- the *ISIWoK* portlet, which permits to obtain distinct quality parameters of a journal (e.g., impact factor) or paper through the *ISI Web of Knowledge* portal.

The portlets could have been developed in house, bought from third-party providers or generated from existing Web application using wrapping techniques [22], as it is the case for this sample problem.

4.4 The SOP model: a PIM for portlet-based portals

Portal development platforms conceive Web portals as a compound of different types of artefacts (e.g., containers, content pages and portlets) where implementation depends on the vendor at hand. We depart from this vision and conceive Web portals as "universal integration mechanisms" where an increasing amount of their content

[1]Disclaimer: the portlets used in this chapter were implemented as wrappers of third-party sites. They are used only for illustrative purposes, and not for commercial advantage.

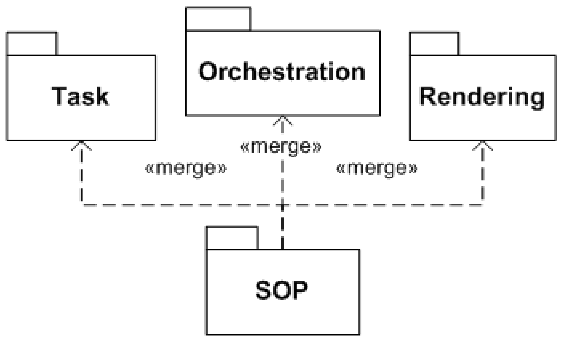


Figure 4.3: The SOP metamodel.

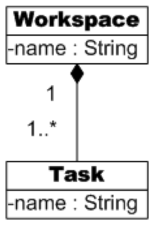


Figure 4.4: The TASK metamodel.

comes from outside the Web portal itself. This brings a service-oriented perspective to Web portal conception where the Web portal is no longer perceived as a *set of pages* but as an integrated *set of services*.

This vision is realized through a PIM where the notion of *portal* and *portlet* are abstracted into the notion of *workspace* and *task*, respectively. A Web portal defines a workspace, i.e., a compendium of **front-end** tasks which are achieved as portlets. For the purpose of the work presented in this chapter, Web portal modelling implies three viewpoints, that is, the SOP[2] model to describe the Web portal is composed of three submodels (see Figure 4.3):

- the *TASK model*, which describes the functionality, i.e., the set of tasks, that the Web portal will offer.
- the *RENDERING* of the portal, i.e., layout and aesthetic considerations which include addressing how tasks are distributed both among pages and within a page.
- the *ORCHESTRATION* of the portal, i.e., the order in which tasks are being

[2]SOP stands for *Service-Oriented Portal*.

made available to the end-user.

The section ends describing how prior metamodels can be completed in order to take into account the personalization matter.

4.4.1 The TASK metamodel

This metamodel captures the Web portal as a workspace that aggregates the set of tasks that the Web portal will offer to the users (see Figure 4.4).

A TASK model for the sample case

The components in the WSRP model are abstracted as tasks. Using a bottom-up design approach, first WSRP portlets are chosen and then WSRP descriptions are used to return the corresponding TASK model.

As an example, let us build a Web portal for helping researchers in their work, for searching and storing articles, journals, authors, impact indexes, and the like. Once analyzed the available WSRP portlets and the functionality the new Web portal should offer, the designer will decide the tasks the Web portal must include. In this case, the TASK model is composed of six tasks whose names are *IEEESearch*, *ACM-Search*, *CiteSeerSearch*, *DBLPSearch*, *DeliciousStore*, and *ISIWoK*, derived from the names of prior portlets. This sets the pieces for the *Browsing* portal.

4.4.2 The ORCHESTRATION metamodel

For the purpose of this work, orchestration describes how tasks are seamlessly aggregated into the workspace. Broadly speaking, aggregation can be defined as the purposeful combination of a set of artefacts to achieve a common goal. The peculiarities of the artefact (i.e., text, Web services, portlets) influence the aggregation model. For instance, Web Services have input/output parameters. Hence, Web service aggregation needs to address the role of parameters during aggregation (e.g., using semantic approaches [85, 97]). By contrast, portlets do not have input/output parameters. Instead of delivering a data-based XML document, portlets deliver markup fragments (e.g., XHTML) ready to be rendered by the Web portal. Moreover, portlets tend to be stateful, and the interaction lasts for a whole session, rather than the simple request that characterizes the one-shot interaction of Web Services.

Based on the peculiarities of Web portals, we identify two main requirements for the portlet aggregation model, namely:

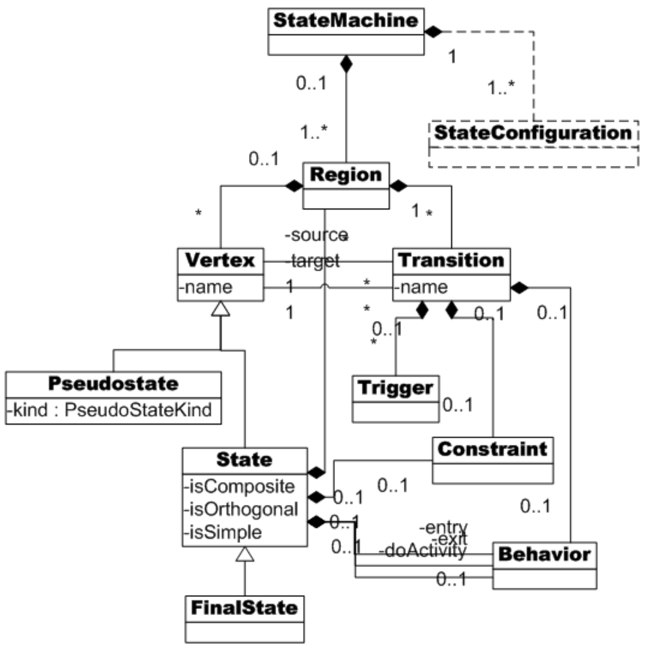


Figure 4.5: The ORCHESTRATION metamodel.

- *hypertext-like navigation style.* Portlet aggregation should permit users to explore the Web portal content freely, by skipping among portlets if required. This does not preclude that a more conducted process *à la workflow* could need to be enforced in some cases, but the free-surfing style should be predominant.

- *front-end aggregation.* Portlets do not return data structures but markup (i.e., semi-structured documents where content and presentation are mixed together). Therefore, aggregation should be achieved in how markups from distinct portlets are arranged and sequentially presented. As an example, consider three portlets: *IEEESearch*, *ACMSearch* and *DBLPSearch*. The Web portal designer wants to enforce a precedence rule so that a *DBLPSearch* can not be requested till some browsing has been conducted through either *IEEESearch* or *ACMSearch*. This rule can be enforced in the back-end through some pre-conditions attached to the *DBLPSearch* service or through a some workflow engine enforcing the flow dependency. By contrast, a "front-end" approach relies on GUI widgets, e.g., disabling the *"DBLPSearch"* button until either *IEEESearch* or *ACMSearch* are enacted. Note that integration is not achieved at the back-end but via presentation widgets.

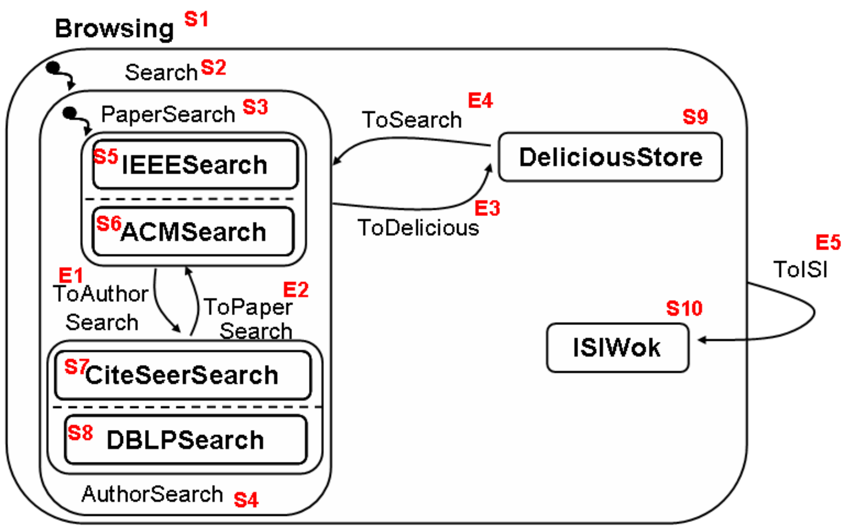


Figure 4.6: Statechart of the *Browsing* portal.

To accomplish these requirements, the *Hypermedia Model Based on Statecharts (HMBS)* [29] is adapted for our purposes. The use of statecharts [39, 38] or their predecessors, state-transition diagrams, is common for modelling hypermedia applications [29], Web service composition [5, 15] and reactive systems. A hypermedia system, and hence, a Web portal, may be considered as a reactive system, since it must interactively attend to external events given in the form of user requests during browsing. Statecharts provide a concise and intuitive visual notation as well as being rigorously defined with a formal syntax and operational semantics.

Therefore, statecharts are used to define the orchestration. The statechart metamodel proposed in the UML specification [78] is used as a base for the ORCHESTRATION metamodel (see in Figure 4.5 a simplified version, which has been extended with the notion of *"state configuration"*). Statecharts extend the classical formalism of state transition diagrams by incorporating the notions of hierarchy, orthogonality (concurrency), a broadcast mechanism for communication between concurrent components, composition, and refinement of states. Specifically, an OR-type decomposition is used when a state is to be decomposed into a set of exclusive substates, whereas an AND-type decomposition is used to decompose a state into parallel, or orthogonal substates. Each concurrent region in an AND state is delimited by a dashed row (for a gentle introduction see [82]).

The statechart constructs are used to model the task flow. Simple states stand for atomic tasks (those defined in the TASK model). Tasks available simultaneously conform more abstract AND states, whereas alternative tasks are enclosed into OR states. Both AND and OR states can be successively nested until the statemachine is complete, and it is the counterpart of the workspace (i.e., portal) itself.

Transitions permit to move among states when an event arises, provided the associated condition is met. Conditions permit to personalize orchestration based on the user profile (e.g., whether the user is a student or a lecturer) or the navigation trace (whether a given state has already been visited or not). Personalization subject will be further analyzed in Subsection 4.4.4.

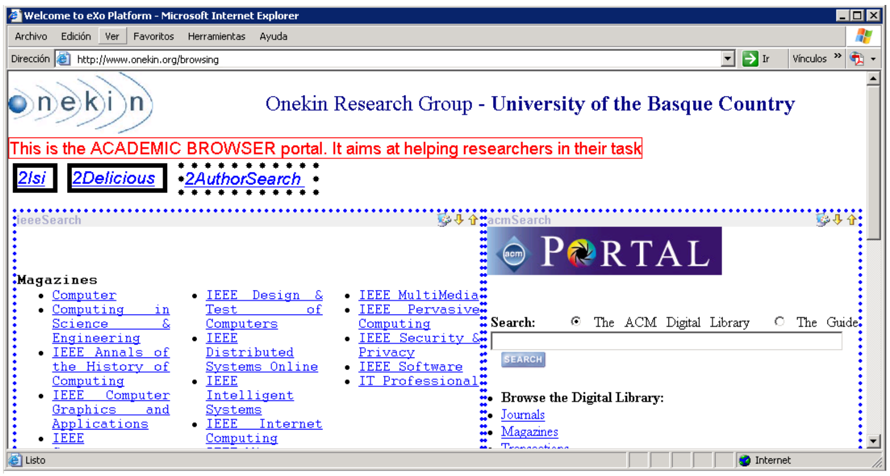
An ORCHESTRATION model for the sample case

From a workspace perspective, tasks are simple states: you are either visualizing the task or you are not. These tasks can be arranged along a flow as illustrated in Figure 4.6 for our sample tasks. The portal designer initially considers two states: you are either searching for something on the Web, or you are storing your finding in *del.icio.us*. At any time, you can move to the *ISIWoK* task to consult the impact of a specific paper.

Search is an abstract state which contemplates two situations: searching for a paper or searching for an author. Papers in turn can be located at either *IEEE* or *ACM*. These options are simultaneously available as denoted by the AND state (i.e., dotted line). On the other hand, author information can be obtained through either the *CiteSeerSearch* task or the *DBLPSearch* task. Both tasks are also simultaneously available. At any time, an end-user can move between both searching modes.

For the purpose of this work, it is important to recall the notion of *state configuration*, i.e., the set of currently active states of the statechart [39]. Basically, a state configuration comprises one simple state, or more, and its container states, on the understanding that OR substates can not be simultaneously active, and AND substates can be simultaneously active, as long as their container states remain also active. For example, the state configurations for the statechart in Figure 4.6 are:

- *configuration1*: *{Browsing, ISIWoK}*
- *configuration2*: *{Browsing, DeliciousStore}*
- *configuration3*: *{Browsing, Search, PaperSearch, IEEESearch, ACMSearch}*
- *configuration4*: *{Browsing, Search, AuthorSearch, CiteSeerSearch, DBLPSearch}*



http://www.onekin.org/browsing

Onekin Research Group - University of the Basque Country

This is the ACADEMIC BROWSER portal. It aims at helping researchers in their task

2Isi 2Delicious 2AuthorSearch

Magazines
- Computer
- Computing in Science & Engineering
- IEEE Annals of the History of Computing
- IEEE Computer Graphics and Applications
- IEEE
- IEEE Design & Test of Computers
- IEEE Distributed Systems Online
- IEEE Intelligent Systems
- IEEE Internet Computing
- IEEE MultiMedia
- IEEE Pervasive Computing
- IEEE Security & Privacy
- IEEE Software
- IT Professional

Search: [✓ The ACM Digital Library] The Guide

- Browse the Digital Library:
- Journals
- Magazines

Figure 4.7: Presentation counterpart of the state configuration *{Browsing, Search, PaperSearch, IEEESearch, ACMSearch}*.

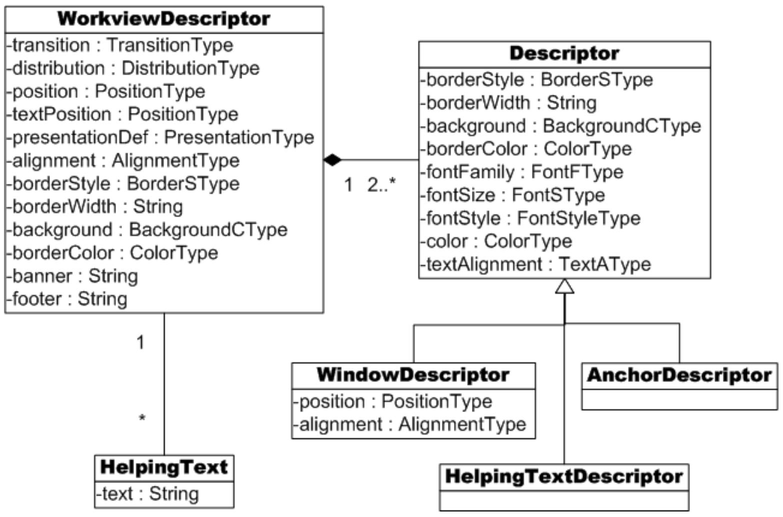


Figure 4.8: The RENDERING metamodel.

The importance of state configurations rests on being the PIM counterparts of portal pages (transformations will make this explicit). For our sample problem, Figure 4.7 shows the presentation counterpart of *configuration3*. The details of this transformation are given in Section 4.6.

4.4.3 The RENDERING metamodel

The orchestration viewpoint is complemented by the rendering viewpoint. The rendering counterparts of the orchestration primitives, i.e., state machine, simple states and transitions, are ***workview***, ***windows*** and ***anchors***, respectively (see Figure 4.8). Additionally, the RENDERING metamodel contributes with the ***helpingText*** construct, a construct to complete the information shown in the Web portal in order to help users using it. The RENDERING model will indicate how those constructs are distributed and decorated along the display area. To this end, layout and style parameters are used.

Style parameters are those of CSS [108]. CSS files externalize presentation parameters such as *font family*, *font size*, *background*, *border style*, *border color*, etc. Among style parameters we include ***background***, ***border-style*** (values include *none*, *dotted*, *dashed*, and so on), ***border-color***, ***border-width***, ***font-family***, ***color*** (often referred to as the foreground color), ***font-size***, ***font-style*** (values include *normal*, *italic* and *oblique*), ***text-alignment*** (i.e., how text is aligned within the element) and ***transition***. The latter does not come from CSS and it indicates how anchors are realized. The value options include *button* or *helping text* where the transition is achieved by clicking on the underlined text. Except *transition* parameter, the rest of parameters, with distinct flavors, can be found in most IDEs for portal development such as *eXo* [27], Oracle Portal [83] or IBM's WebSphere [43].

As for the **layout**, it indicates how windows/anchors are arranged along a table-like structure using the following parameters: ***distribution***, indicates how to locate anchors along the portal page, and options include *together* (i.e., anchors are all located together, regardless of their orchestration counterparts, i.e., transitions) and *detached* (i.e., anchor A is located beside window W if A stands for a transition that leaves from the state whose counterpart is W); ***position***, indicates whether anchors are placed at the *top*, *bottom*, *left* or *right* of either the page (together) or the associated window (detached); ***alignment***, indicates how windows are rendered together (values are *column*, i.e one below the other, and *row*, i.e., one by the other); ***banner/footer***, it holds a banner/footer which is kept constant along the workview.

Layout parameters are related to the overall portal, hence in the RENDERING metamodel they are described as part of the *WorkviewDescriptor*, and style parameters are attached to the distinct artefacts through *WindowDescriptor*, *AnchorDescriptor* and *HelpingTextDescriptor* (see Figure 4.8).

The simplicity of this metamodel comes from the fact that portlets free the Web portal designer from most of the burden related with rendering concerns. The reason is two-fold. First, a portal's primary function is to provide an entry to content already available elsewhere, not acting as a separate source of information itself. And second, these external sources of information are portlets which already convey how this information is presented. Unlike traditional Web Services, portlets deliver markup ready to be co-located into the portal's page. Hence, the RENDERING model needs just to focus on the presentation of the relationship among states and transitions, leaving outside what happens when in a given state.

According to the metamodel in Figure 4.8, the rendering of a workview is generated from an aggregate of descriptors. But this does not mean that the designer has to set a descriptor *for each* of the orchestration constructs (states and transitions). Indeed, ensuring a common look-and-feel through the Web portal pages is one of the main concerns for the portal designer to improve usability and the user experience. To this end, *skins* are used, i.e., templates defined at the portal level that set some presentation properties that are then "inherited" by all the portal pages. Besides ensuring presentation homogeneity, this mechanism accounts for maintainability as (some) changes in the presentation uniquely involve updating the skin rather than modifying the distinct pages.

However, the use of skins in current platforms can be too coarse grained, i.e., skins are defined at either the portal or the page level. There is nothing in between. For large portals where pages can be grouped into clusters based on content or functional grounds, finer grained skins could be most convenient. To this end, we introduce the notion of **state skin** as a *descriptor* associated to a *state*.

The idea is to use the containment hierarchy provided by statecharts (i.e., the relationship between a *state* and its *substates*) to specify the rendering in a stepwise manner. Rather than attaching *descriptors* to simple *states*, this work proposes the **rendering-descriptor inheritance**: now a *descriptor* can also be associated with any AND or OR state, and its scope includes all contained substates. That is, *a descriptor for state S affects any widget associated with any substate directly or transitively below S*. Moreover, a rendering parameter value (e.g., *fontSize =12pt*) is overridden if newly defined in the descriptor of a substate. These descriptors are called *state skins*. State diagrams that incorporate state skins are referred to as *annotated statecharts*.

This approach offers the generality required to provide a common look-and-feel along the Web portal, while at the same time addressing specificities of certain tasks or task clusters where presentation singularity is sought. Traditional skins correspond to rendering parameters defined for the upper state (e.g., *Browsing*) whereas

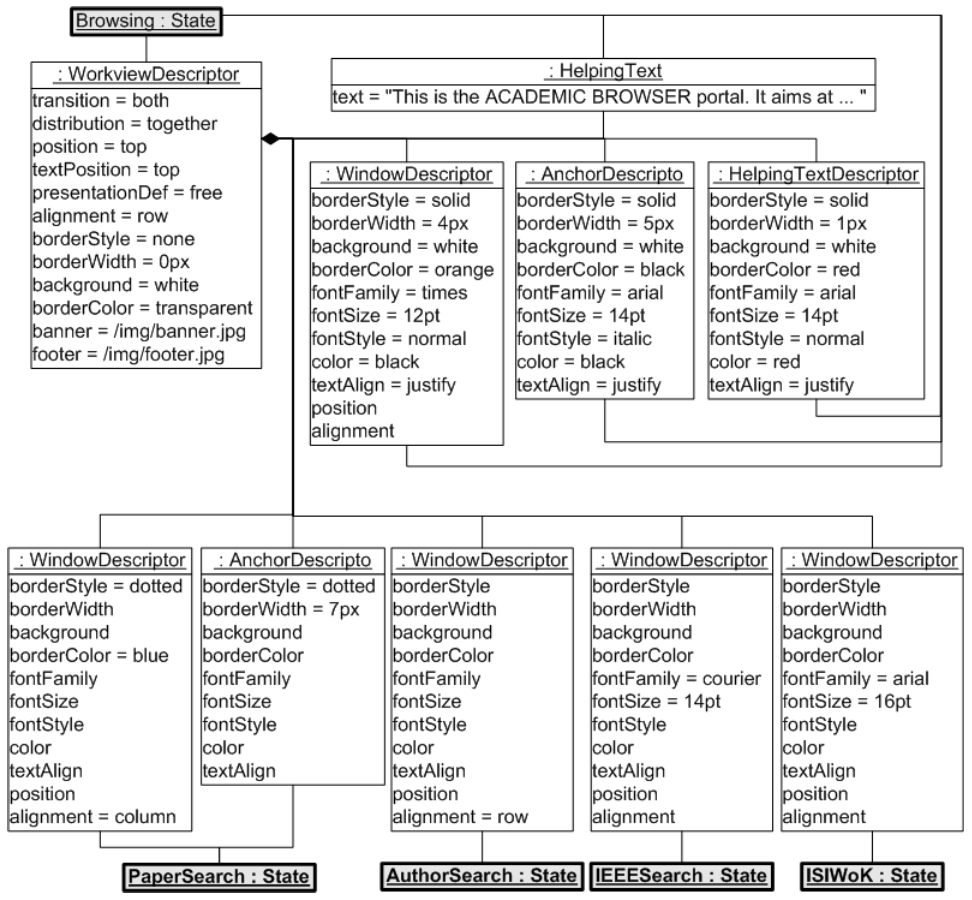


Figure 4.9: States and their rendering counterpart.

it is still possible to completely override this skin for a particular simple state (e.g., *IEEESearch*). Next subsection illustrates this approach for the running example.

An important note. Strictly speaking, style and layout parameters are closer to PSM concerns than to PIM ones. Indeed, a proper rendering PIM should provide some general guidelines that are later mapped into CSS-like parameters. As stated in [33] *"many of the usability problems can be addressed automatically. For example the navigation scheme of a Web Application can be provided automatically based on general guidelines given by the modeler. Of course, a solution at this level of abstraction can only be applied to a small subset of existing problems and therefore it should be easy and fast to come up with alternatives or extensions for the code generation templates"*. Here, the transformation hides the guidelines to map from the PIM rendering constructs to the platform-specific CSS-like constructs.

However, guidelines are distilled from experience, and we do not have enough

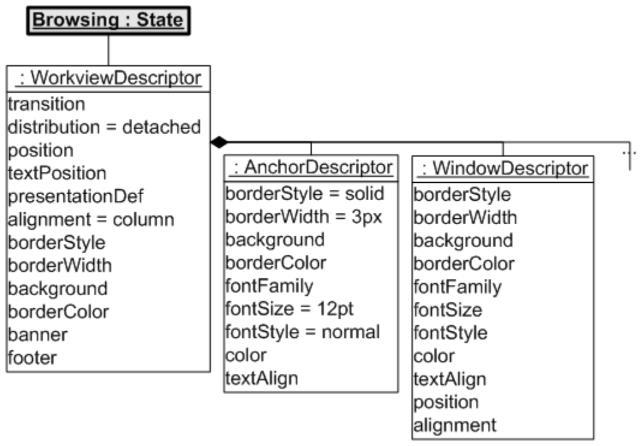


Figure 4.10: Excerpt of another alternative RENDERING model.

acquaintance to provide those rendering guidelines yet. To move rendering parameters up is then a temporal solution till enough experience is collected that permits to abstract away from CSS-like parameters.

A RENDERING model for the sample case

Figure 4.9 specifies the RENDERING model for the *Browsing* statechart. For the sake of clarity, related to each rendering construct the figure also shows its corresponding state of the ORCHESTRATION model (the states are shown in a different shape). The root state, *Browsing*, holds the portal skin where the font type and size for displaying portlet information is set to *times* and *12pt*, respectively, in the *WindowDescriptor*. Of note, the *IEEESearch* skin redefines these parameters to *courier* and *14pt*, respectively. More to the point, the *Browsing* skin sets that anchors must be together (*distribution* attribute in the *WorkviewDescriptor*) and shown with a *solid 5px border-line* and in an *italic font-style* (in the *AnchorDescriptor*). This specification is overridden by the *PaperSearch* skin and set to a *dotted 7px border line*. The rest of the states without explicit skin description inherit rendering guidelines from the root state skin. The portal page for the state *configuration3* is shown in Figure 4.7.

RENDERING and ORCHESTRATION models are orthogonal. That is, the very same ORCHESTRATION model can be presented along distinct skins to better fit the user profile, and the other way around, the same skin can be used for distinct ORCHESTRATION models. The former situation is illustrated for the running example.

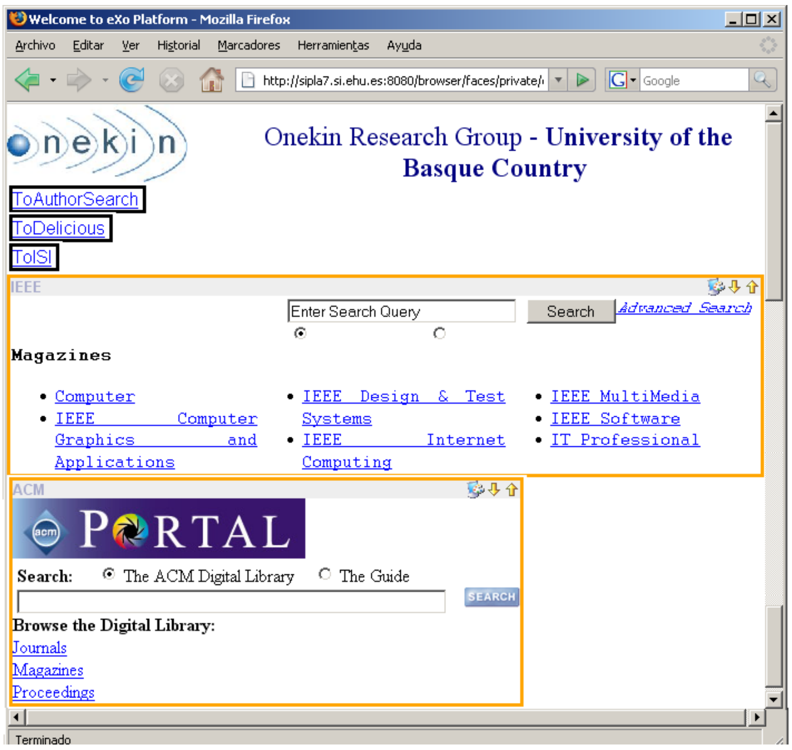

Welcome to eXo Platform - Mozilla Firefox
Archivo Editar Ver Historial Marcadores Herramientas Ayuda
http://sipla7.si.ehu.es:8080/browser/faces/private/
Google
onekin
Onekin Research Group - University of the Basque Country
ToAuthorSearch
ToDelicious
ToISI
IEEE
Enter Search Query
Search
Advanced Search
Magazines
Computer
IEEE Computer Graphics and Applications
IEEE Design & Test Systems
IEEE Internet Computing
IEEE MultiMedia
IEEE Software
IT Professional
ACM
PORTAL
Search: The ACM Digital Library The Guide
SEARCH
Browse the Digital Library:
Journals
Magazines
Proceedings
Terminado

Figure 4.11: Another presentation counterpart of the state configuration *{Browsing, Search, PaperSearch, IEEESearch, ACMSearch}*.

The statechart in Figure 4.6 can have two alternative RENDERING models, namely, those shown in Figure 4.9 and Figure 4.10 (for the sake of clarity, only different values are shown). In the latter case, the portal page related to the *configuration3* would have been as shown in Figure 4.11. Anchors are detached in different rows, and in *normal 12pt* font with a *solid border line thinner* than before. Moreover the portal does not show any helping text.

4.4.4 Portal personalization

In order to avoid misunderstandings, a clear distinction must be made first between systems that are customizable or adaptable and adaptive or personalized systems. They differ in the way the adaptation is performed. According to Nielsen *"Customization* is under direct user control", i.e., the user can configure an interface and create a profile manually, and *"Personalization* is driven by the computer which tries to serve up individualized pages to the user based on some form of model of that user's needs", i.e., the user is seen to be as more passive and it is the system that

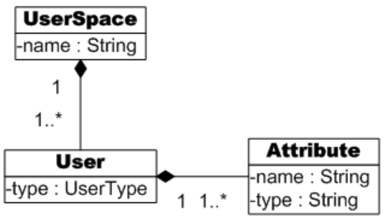


Figure 4.12: The USER metamodel.

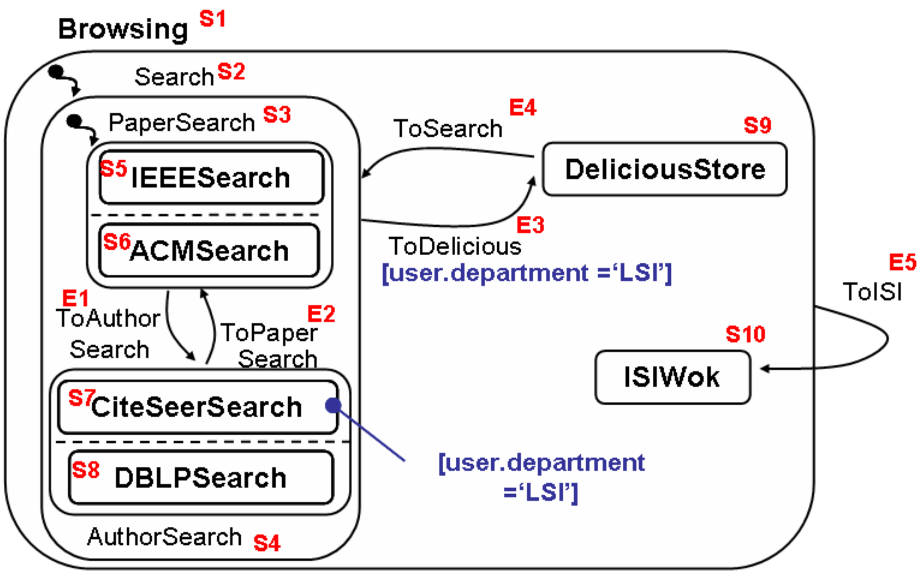


Figure 4.13: An ORCHESTRATION model with personalization conditions.

monitors, analyzes and reacts to the user's behaviour [68].

Personalization of Web sites has become an important issue in Web modelling methods, and it has been studied from different points of view, namely, personalization based on user attributes or their preferences and interests, personalization dependent on domain information, personalization of navigational behaviour, and content personalization. Web portals being particular Web sites, the SOP metamodel of our portal design approach also takes personalization into account and next paragraphs describe the details, i.e., how the previous metamodels have been completed with this aim.

In the context of Web design methods, to model personalization two models are used: a *personalization model*, where personalization rules are defined to store information needed to personalize, and to specify different personalization policies, and a *user model*, which allows to store data about the beliefs and knowledge the system

Personalization patterns	**SOP metamodel**
Link personalization	Orchestration personalization on transition
Structure personalization	Orchestration personalization on state
Content personalization	Depends on each portlet design
Behaviour personalization	Depends on each portlet design

Table 4.2: Personalization patterns and the SOP metamodel.

has about the end-user. In our approach, only the user model has been added, and the personalization model is scattered among previous metamodels, they have been modified to be able to include personalization conditions.

For the user model only aspects related to user static attributes have been taken into account. We have only considered simple users and their P3P-like attributes [111], those that can be filled in when the user registers in the Web portal. We have not considered the hierarchy among users, i.e., user groups, nor dynamic aspects as the number of visited pages, the time of connection, and the like. Therefore, the user space is composed of the set of portal users, and these are described through some attributes (see the USER metamodel in Figure 4.12). This metamodel will constitute the fourth package of the prior SOP metamodel.

According to [8] motivations for personalization can be divided into (1) those that are primarily to facilitate the work (i.e., enabling access to information content, accommodating work goals, and accommodating individual differences), and (2) those that are primarily to accommodate social requirements (e.g., eliciting an emotional response and expressing identity). Our proposal takes into account both and proposes orchestration personalization, related to enabling access to content or functionality, and rendering personalization, related to modifying portal presentation to accommodate individual differences or an identity, e.g., for visually impaired people or people joined to a certain group.

Moreover, Schwabe *et al.* [95] distinguish several personalization patterns, among them, *Link*, *Structure*, *Content* and *Behaviour* personalizations. Those patterns propose personalizing, respectively, the link's end-point, the Web application's structure, the content for a particular information item and other functions apart from navigation; and they make personalization based on user-related information.

As explained in Subsection 4.4.2, for the ORCHESTRATION metamodel the UML Statechart package has been used. This specification includes the *Constraint* metaclass. *Constraint* is related to the *State* and *Transition* metaclasses, thus a statechart can include constraints linked to states and transitions. In this way, it will

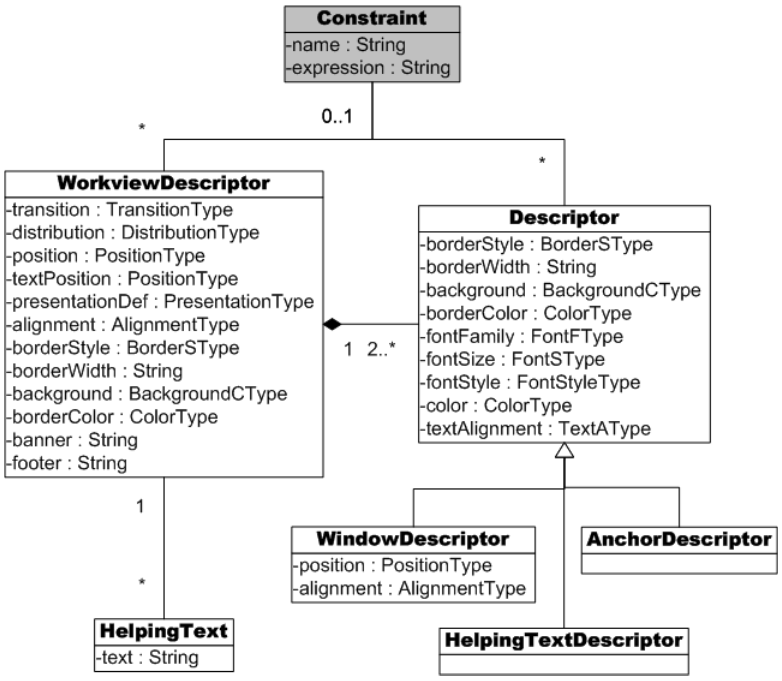


Figure 4.14: The RENDERING metamodel with personalization.

be possible to limit the access to a state or the use of a transition. As an example, Figure 4.13 shows a statechart with two constraints related to the *CiteSeerSearch* state and the *ToDelicious* transition. The former indicates the portlet related to the *CiteSeerSearch* state will be rendered only to users of LSI department and the latter specifies that only those users will be able to save references in the Delicious store, i.e to use the *ToDelicious* transition. Therefore, as for orchestration personalization, our proposal uses *Constraint* metaclass to implement Structure and Link personalizations. Content and Behaviour personalizations are related to providing individualized content or responses to a particular operation, respectively. In our approach this functionality is in portlet hands, so it depends on personalization characteristics of each portlet. Table 4.2 summarizes the approach.

As for Presentation, it offers lots of possibilities for personalization, for instance, changes in the language in which the content of the Web portal will be shown depending on the origin of the user, changes in font size depending on the needs of visually impaired people, and changes in portal layout depending on the device where it will be shown. In order to include the personalization aspect in the RENDERING metamodel, we have followed the same idea as UML specification: adding a *Constraint*

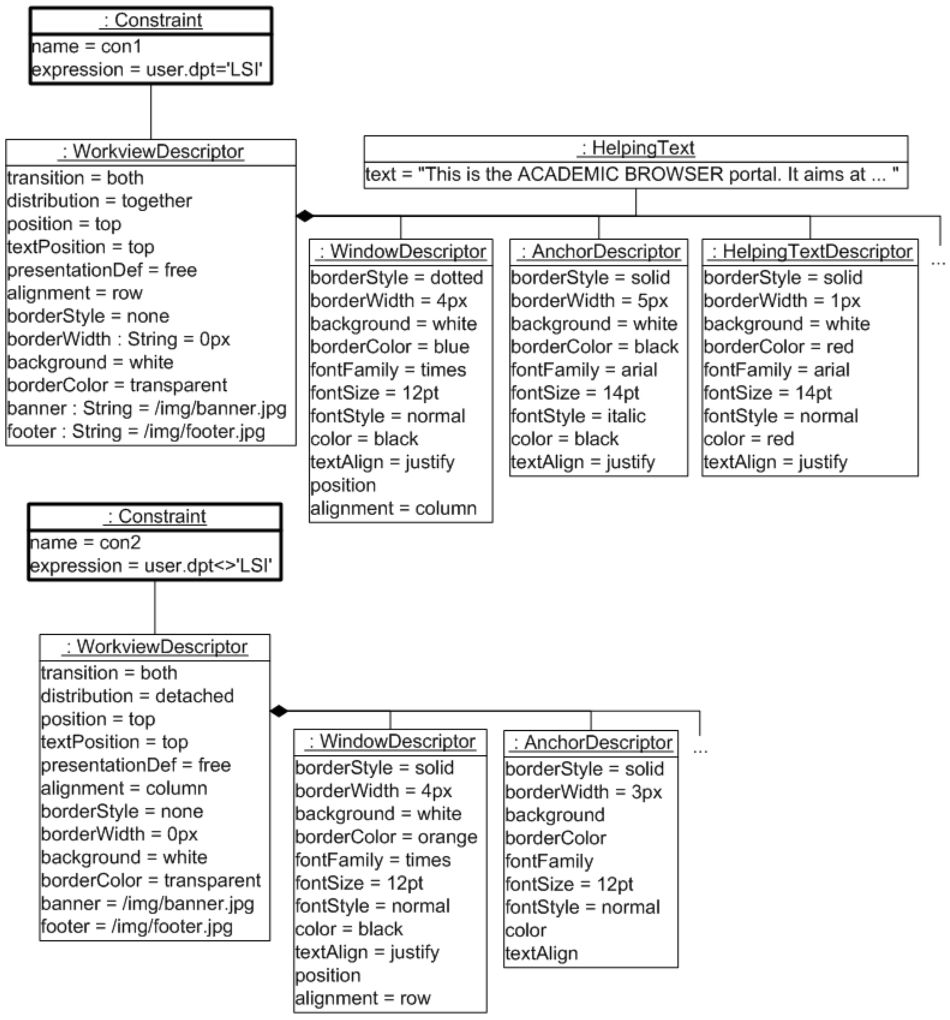
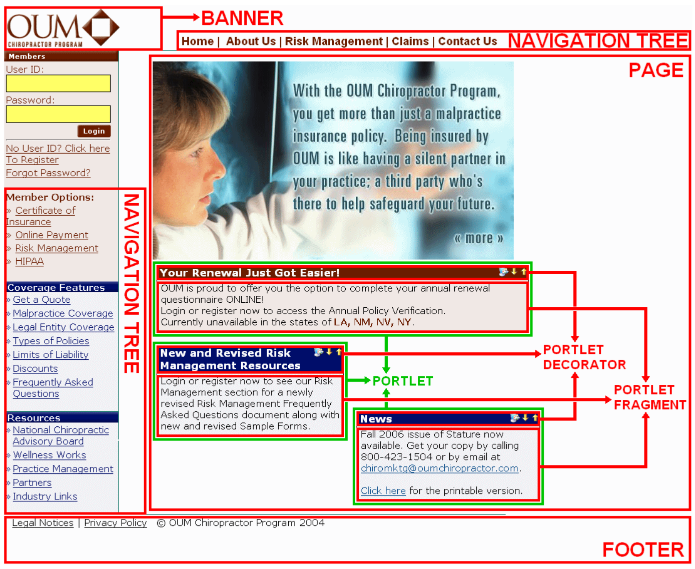


Figure 4.15: A RENDERING model with personalization conditions.

metaclass (see Figure 4.14). This new metaclass is related to the *WorkviewDescriptor* and *Descriptor* metaclasses. Therefore, the overall portal layout, in general, or the presentation of a specific portlet or anchor can be conditioned by the user's attribute values. As an example, Figure 4.15 shows a RENDERING model that describes two different workviews, depending on the department of the user. Those workviews correspond to portal pages shown in Figures 4.7 and 4.11, respectively.

Summing up, this personalization approach is concentrated on portal design dependent on user static characteristics, stored in the USER model, therefore a condition expression is enough. Other website personalization approaches, as [32, 54], focus the attention on both user-specific information and domain-dependent infor-



With the OUM Chiropractor Program, you get more than just a malpractice insurance policy. Being insured by OUM is like having a silent partner in your practice; a third party who's there to help safeguard your future.

Your Renewal Just Got Easier!

OUM is proud to offer you the option to complete your annual renewal questionnaire ONLINE!
Login or register now to access the Annual Policy Verification.
Currently unavailable in the states of LA, NM, NV, NY.

New and Revised Risk Management Resources

Login or register now to see our Risk Management section for a newly revised Risk Management Frequently Asked Questions document along with new and revised Sample Forms.

News

Fall 2006 issue of Stature now available. Get your copy by calling 800-423-1504 or by email at chiromktg@oumchiropractor.com.

Click here for the printable version.

Legal Notices | Privacy Policy © OUM Chiropractor Program 2004

Figure 4.16: A sample *eXo* portal page.

mation (e.g., number of clicks on a certain anchor or using a specific functionality) and they need ECA-like rules, to specify the actions to perform the personalization. Besides personalization rules, those approaches need acquisition rules to monitor the execution environment and obtain the required knowledge to personalize.

4.5 The EXO model: a PSM for the *eXo* platform

According to the official *eXo* site (*www.exoplatform.com*), this platform was the first portlet container to be JSR-168 certified in 2003, and it is currently one of the most popular, freeware portal frameworks. The work presented in this dissertation is based on *eXo version 1* [28].

An *eXo* portal is a compound of four main types of artefacts: the *portal* itself, *pages*, *containers* and *portlets*. A portal encompasses a set of pages which in turn, hold containers, which finally, keep the portlets. Figure 4.16 shows how an *eXo* page looks like.

A page is conformed along two directives: the *portal template* and the *page content*. The former specifies a layout in terms of rows and columns. A common pattern is depicted in Figure 4.16: a banner, a footer, a navigation tree (an index to the main portal pages), and the *page content*. Whereas the *portal template* is shared by all pages, the page content is specific for each page. It is also described through a set

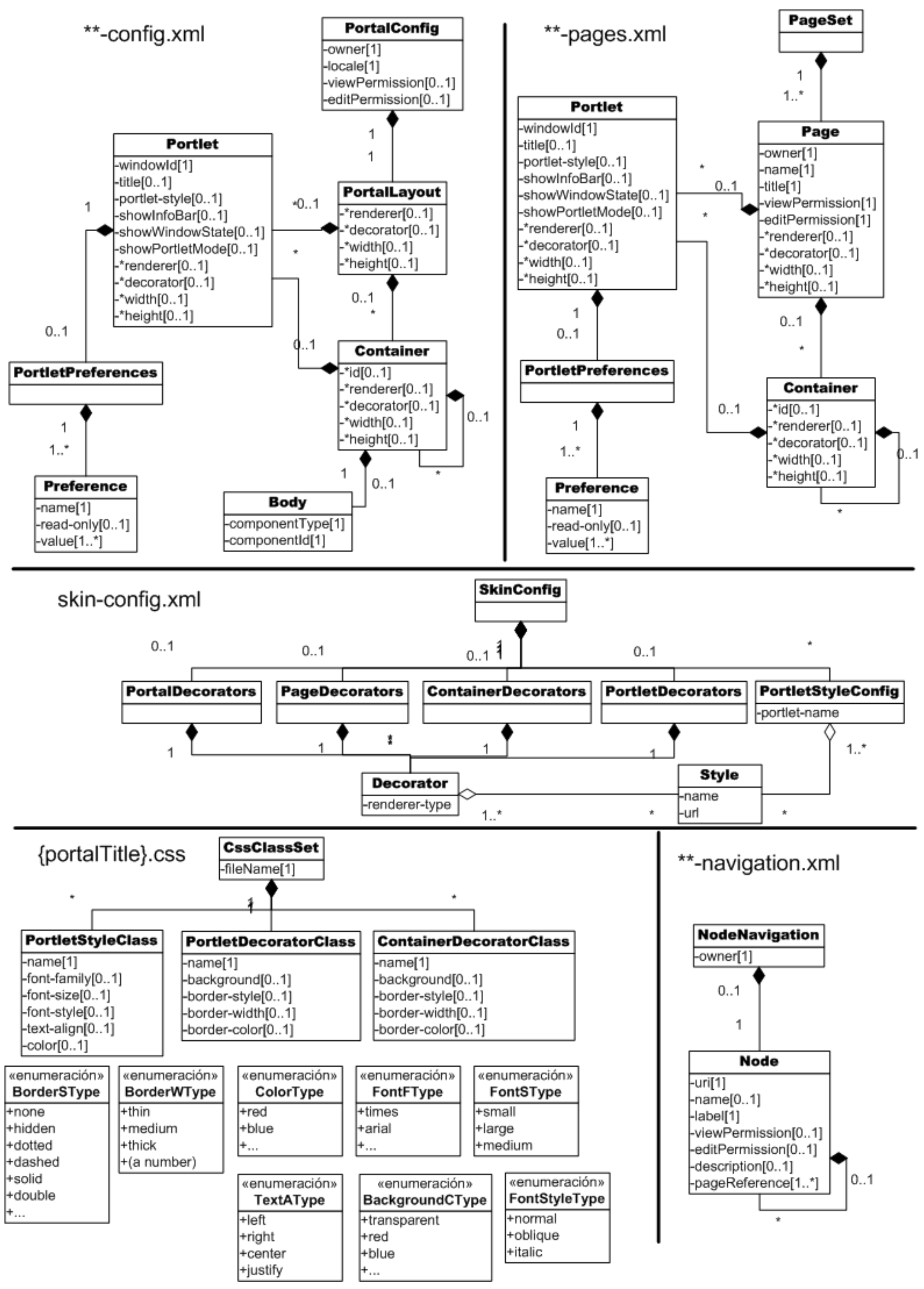


Figure 4.17: The EXO metamodel.

of nested rows and columns where each cell contains a portlet. In this way, the *page content* is built up by aggregating the presentation of the contained portlets. A *page content* is bound to one or several nodes of the *navigation tree*. By clicking on these nodes, the user moves along the distinct *page contents*.

Both the *portal template* and the *page content* are specified as XML files, namely (see Figure 4.17):

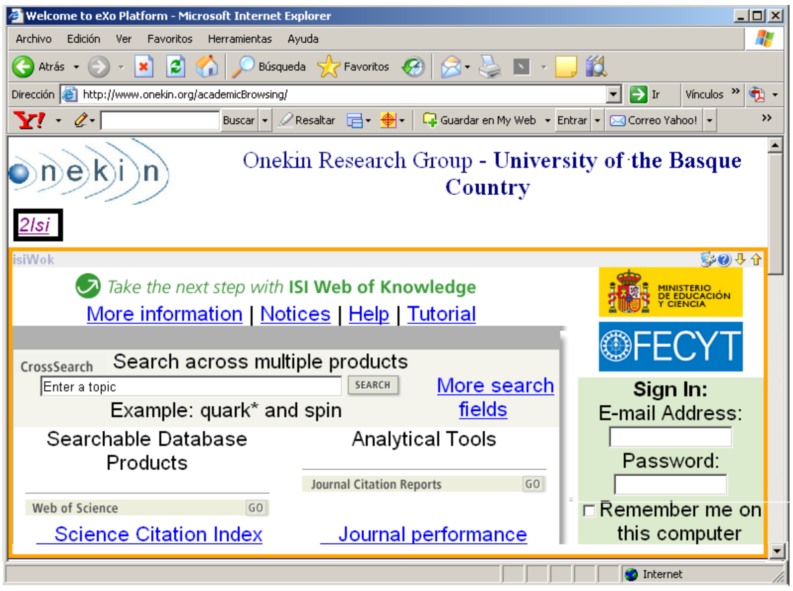
- ***-config.xml*, which describes the *portal template*, e.g., whether the portal has banner, footer, or does not have, or how portlets are to be arranged in rows or columns. The root element of this file is *PortalConfig* which contains a *PortalLayout*. From then on, the layout is described in a tree-like way as a containment hierarchy where the leaves of the hierarchy are either portlets or a body (i.e., a kind of canvas that holds *page content*).

- ***-pages.xml*, which describes the set of *page contents*. They sit at the bottom of the layout hierarchy. A *page content* can in turn hold other containers and portlets.

- ***-navigation.xml*, which describes the hierarchical relationship among the *page content*, i.e., which other pages are reachable when in a given *page content*. This basically defines the “navigation tree” shown on the left of Figure 4.16.

The previous files describe the layout. Aesthetic parameters (e.g., color, fonts, etc.) are set through decorators. Since a portal is a compound of four main types of artefact (i.e., the portal itself, containers, pages and portlets), a decorator is defined for each type of artefact: *portlet decorators*, *container decorators*, *page decorators* and *portal decorators*. Moreover, portlet’s fragments can also be the subject of special CSS which are known as *“portlet styles”*. Therefore, presentation wise, a portlet presentation is governed by the decorator (i.e., the component that surrounds the portlet body) and the portlet style, the latter guides the presentation of the fragments (i.e., the markup rendered by the portlet) (see Figure 4.16). The aesthetic of the portal is then set through *skin-config.xml* file and some *CSS files*. The former describes the decorators and the others contain values of the style parameters.

The information rendered to the user depends on both the user role and the portal state. The user configuration of an *eXo* portal is described in the *organization-configuration.xml* file. This file defines the preconfigured groups and users, and the


Welcome to eXo Platform - Microsoft Internet Explorer
Archivo Edición Ver Favoritos Herramientas Ayuda
Atrás Búsqueda Favoritos
Dirección http://www.onekin.org/academicBrowsing/
Ir Vínculos
Buscar Resaltar Guardar en My Web Entrar Correo Yahoo!
Internet

onekin

Onekin Research Group - University of the Basque Country

2lsi

isiWok

Take the next step with ISI Web of Knowledge

More information | Notices | Help | Tutorial

CrossSearch Search across multiple products

Enter a topic SEARCH More search fields

Example: quark* and spin

Searchable Database Products

Analytical Tools

Journal Citation Reports GO

Web of Science GO

Science Citation Index

Journal performance

MINISTERIO DE EDUCACIÓN Y CIENCIA

FECYT

Sign In:

E-mail Address:

Password:

Remember me on this computer

Figure 4.18: Presentation counterpart of the state configuration *{Browsing, ISIWoK}*.

relationships among them. All the layout, content and navigation can be personalized based on the user profile. To this end, distinct files ***-config.xml*, ***-pages.xml* and ***-navigation.xml* can be provided in a user basis. Actually, "**" stands for the username (e.g., *john-config.xml*).

We have described the specification of all those files in the EXO metamodel (see Figure 4.17), with one subpackage for each file type.

4.6 Transformation definition

Model transformation is the process of converting one model, called *source model*, to another model, i.e., the *target model*, of the same system [76]. Model transformation languages are used to specify the mapping between constructs of the source models into constructs of the target model along the so-called *transformation pattern* [12]. This section illustrates this pattern where the *annotated statechart* and the *eXo platform* play the role of the source and target models, respectively. Broadly speaking, the transformation engine takes an annotated statechart as an input, works out its state configurations and finally, outputs a set of pages conforming the *eXo* portal. Figures 4.18 and 4.7 depict the *eXo* pages for *configuration1 {Browsing, ISIWoK}* and *configuration3 {Browsing, Search, PaperSearch, IEEESearch, ACMSearch}*, respectively.

Transformations express a correspondence between elements of the source meta-

	task constructs	**orchestration constructs**	**rendering constructs**	***eXo* files**
rule1		state	windowDescriptor	****.css
rule2	task	state-configuration	workviewDescriptor	**-pages.xml

Table 4.3: Mapping SOP constructs to *eXo* files through distinct transformation rules.

model into elements of the target metamodel. Yet, this correspondence is frequently not unique but distinct ways can exist to map the same source model into alternative target models. Specifically, statecharts can be mapped into *eXo* constructs in two different ways: the *interpreted approach* and the *compiled approach*.

The *interpreted approach* simulates the dynamics of the statechart through a statechart engine code which obtains the portal page on demand. Specifically, at a given moment, the Web portal is presenting a Web page which corresponds to the current state configuration. When a user rises a GUI event (e.g., by clicking on a link), the Web portal transmits it to the transition associated to the triggered event. Then, the corresponding guard condition is evaluated, and if satisfied, the statechart sets the target state as active, which in turn, leads to a new state configuration. This new configuration sets the portlets to be displayed, and a new page is generated in accordance with the associated state skins. This page is finally rendered back to the user, and this ends the loop. However, this approach happens to suffer from efficiency problems for large statecharts. A main issue is that the test for activated transitions is time-consuming, and it should be at workout time and again as the user navigates throughout the portal.

By contrast, *the compiled approach*, works out all the pages at generation time. A Web page is generated from each state configuration, where the page anchors correspond to the transitions available to this state configuration. This approach improves efficiency as generation does not happen at portal enactment time. It also provides a solution more akin to current *eXo* development practices where users are accustomed to the so-called “navigation tree”, an index to the main portal pages. On the downside, the compiled approach prevents some adaptation from taking place at run time (e.g., some transitions can depend on some execution parameters such as the price of ticket just bought). This section focuses on the compiled approach.

Transformation wise, a main challenge is the containment structure of statecharts: a state can contain lower-level states. Specifically, state configurations are not given by the designer but need to be worked out by the transformer itself. This implies

```
top_rule 'StateMachine2CssClassSet' do
 from SOP::Orchestration::StateMachine
 to EXO::Css::CssClassSet
 mapping do |state_machine, class_set|
    class_set.name = state_machine.workspace.name
    class_set.styles = [state_machine.root_state] + state_machine.simple_states
    class_set.decorators = [state_machine.root_state] + state_machine.simple_states
    ...
 end
end
rule 'SimpleState2Style' do
 from SOP::Orchestration::State
 to EXO::Css::PortletStyleClass
 filter { |state| state.isSimple }
 mapping do |state, stylePortlet|
    stylePortlet.name = state.name + "PStyle"
    stylePortlet.fontFamily = state.windowDescriptor.fontFamily
    ...
 end
end
rule 'SimpleState2Decorator' do
 from SOP::Orchestration::State
 to EXO::Css::PortletDecoratorClass
 filter { |state| state.isSimple }
 mapping do |state, decorator|
    decorator.name = state.name + "PDecorator"
    decorator.background = state.windowDescriptor.background
    ...
 end
end
...
```

Figure 4.19: Mapping from simple state to *decorator* and *style* CSS classes in RubyTL.

recursive transformation rules that traverse the tree-like structure of the statechart. At the time of the implementation, popular transformation languages such as QVT [77] were not expressive enough to specify the required transformations, and we finally used RubyTL [99] as the transformation language.

The *SOP-to-EXO mapping* is fully implemented along 69 mapping rules. Next subsections provide a sample for two representative rules which generate a CSS file and an "*eXo page*" file, respectively (see Table 4.3). Annexe A provides further details about the whole transformation.

4.6.1 Mapping from simple states to CSS classes

Figure 4.19 depicts the *StateMachine2CssClassSet* rule. This rule takes as an input a *StateMachine* from the SOP metamodel (specifically, from the ORCHESTRATION subpackage of the SOP metamodel, see Figures 4.3 and 4.5) and returns a *CssClass-Set* element from the EXO metamodel (see Figure 4.17). That element represents a CSS file, which governs portlet presentation.

```
.ieeeSearchPDecorator-decorator {
   background: white;
   border-style: dotted;
   border-color: blue;
   border-width: 4px;
}
.ieeeSearchPStyle-portlet {
   font-family: Courier;
   color: black;
   font-size: 14pt;
   font-style: normal;
   text-align: justify;
}
.acmSearchPStyle-portlet {
   font-family: Times;
   color: black;
   font-size: 12pt;
   font-style: normal;
   text-align: justify;
}
```

Figure 4.20: Snippet of *browsing.css* file.

Portlet presentation includes the portlet markup itself and the decorator surrounding this markup (see Figure 4.16). The style guidelines for presenting the markup and the decorator are set through CSS classes, specifically, the *PortletStyleClass* and the *PortletDecoratorClass*, respectively (see Figure 4.17). The content of these classes is obtained through the state skins distributed all along the state hierarchy.

The first rule has three bindings. The first binding uses the workspace associated to the statechart in order to name the *CssClassSet* element (remember, this element corresponds to the CSS file to be generated). The next two bindings are resolved by the execution of the *SimpleState2Style* and *SimpleState2Decorator* rules, respectively. The rules are implicitly invoked through a mechanism similar to XSLT templates but now, the matching is based on metamodel types rather than XML tags. Hence, the assignment *"class_set.styles=...state_machine.simple_states"* triggers the rule *SimpleState2Style* for each simple state. This rule creates the *style* CSS class for the portlet counterpart of the state: the name is obtained after the state name adding *"PStyle"* as a suffix, whereas the CSS attributes (e.g., *fontFamily*) are taken from the *WindowDescriptor* of the corresponding state skin.

It is most important to note that the hierarchical definition of state skins permits a given state to have a partial definition (or no definition at all) for its state skin. The complete definition is obtained at transformation time by obtaining missing properties from its ancestors. For instance, *ACMSearch* does not have any associated state skin whereas *IEEESearch* only provides the *fontFamily* (courier) and the *fontSize*

(14pt) (see Figure 4.9). However, *eXo* forces to have full-fledged *IEEESearchPStyle* and *IEEESearchPDecorator* CSS classes. Therefore, the missing attributes are recursively obtained by looking up to the upper states that contain the *IEEESearch* state (i.e., *Browsing*, *Search* and *PaperSearch*), and attribute values at the upper states are overridden by those values at lower states. The outcome for these two states is shown in Figure 4.20. It shows that the *font-size* for the *IEEESearch* portlet is 14pt, like specified in the *WindowDescriptor* of the *IEEESearch* state, but *font-size* for the *ACMSearch* portlet is 12 pt, i.e., the default value, inherited from the *WindowDescriptor* of the *Browsing* state.

Implementation wise, a helper function (e.g., *windowDescriptor*) is defined that supports that look-up process. Hence, the expression *state.windowDescriptor.fontFamily* found in these rules, recovers the *fontFamily* value for the current state regardless of whether the font is locally attached to the state in its corresponding *WindowDescriptor* or "inherited" from upper levels.

4.6.2 Mapping from state configurations to *eXo* pages file

As stated previously, the work presented in this dissertation uses a compiled approach to transform statecharts to *eXo* pages, i.e., an *eXo* page is generated from each state configuration rather than constructing the page dynamically at run time. This transformation is outlined in two figures, 4.21 and 4.22, for the sake of legibility.

The transformation starts in line (13) of Figure 4.22 and takes a *SOP::Orchestration::StateMachine* as input (i.e., a *StateMachine* element of the ORCHESTRATION viewpoint or subpackage in the SOP metamodel) and returns an *EXO::Pages::PageSet* as output (i.e., a *PageSet* element of the *Pages* subpackage in the EXO metamodel). Line (14) creates a new *pageSet* object.

Line (16) introduces a *phase*. A phase is a parametrized transformation module that groups a set of related rules. This mechanism is used to group those rules that account for the PIM notion of *configuration*. Specifically, the *page* phase in Figure 4.21 has a *current_config* as its IN parameter, and returns an IN-OUT parameter named *page_set* that holds the *PageSet* object. This phase is enacted for each state configuration. To this end, the *all_configurations* function (in line (15)) works out all possible state configurations from the statechart model. For each configuration, the *page* phase is explicitly executed (in line (16)), passing the current configuration as its IN parameter.

The *page* phase starts with the implicit triggering of its first rule, i.e., *Configura-*

```
phase 'page'
  parameter :current_config
  parameter :page_set
  rule 'Configuration2Page' do
   from SOP::Orchestration::StateConfiguration
   to EXO::Pages::Page
   filter { |conf| conf == current_config }
   mapping do |configuration, page|
       page_set.pages = page
(1)     page.name = "/" + configuration.page_name
         .... <assignment of constant values>
(2)     page.decorator = configuration.state_machine.workspace.name + "PageDecorator"
(3)     page.container = configuration
(4)     page.container = configuration.state_machine.child_states.select { |s| s.isOrthogonal }
(5)     page.portlet = configuration.state_machine.child_states.select { |s| s.generate_portlet? }.
                                   select { |s| configuration.states.include?(s) }
   end
  end
  rule 'AndState2Container' do
   from SOP::Orchestration::State
   to EXO::Pages::Container
   filter { |state| current_config.states.include?(state)  &&  state.isOrthogonal }
   mapping do |state, container|
       container.renderer = state.windowDescriptor.containerRenderer
(6)     container.decorator = state.state_machine.workspace.name + "TransparentDecorator"
(7)     container.subcontainers = current_config.orthogonal_sub_states(state)
(8)     container.portlets = current_config.simple_sub_states(state)
   end
  end
  rule 'SimpleState2Portlet' do
   from SOP::Orchestration::State
   to EXO::Pages::Portlet
   filter { |state| state.isSimple }
   mapping do |state, portlet|
       portlet.renderer = "PortletRenderer"
(9)     portlet.decorator = state.name + "PDecorator"
(10)    portlet.portletStyle = state.name + "PStyle"
       portlet.windowId = "#{state.task.portlet.displayName}/#{state.task.portlet.portletName}/#{state.name}"
        ...
   end
  end
  rule 'Container4Transitions' do
   from SOP::Orchestration::StateConfiguration
   to EXO::Pages::Container
(11) filter { |configuration| configuration.state_machine.workviewDescriptor.distribution=="together"}
   mapping do |configuration, container|
       container.renderer = "ContainerColumnRenderer"
       container.decorator = configuration.state_machine.workspace.name + "TransparentDecorator"
(12)    container.portlets = configuration.all_transitions
   end
  end
  rule 'Transition2Portlet' do
   from SOP::Orchestration::Transition
   to EXO::Pages::Portlet
   mapping do |transition, portlet|
      portlet.renderer = "PortletRenderer"
      portlet.decorator = transition.name + "AnchorDecorator"
      portlet.portletStyle = transition.name + "AnchorStyle"
      portlet.windowId = transition.portlet.displayName + "/" +
                    transition.portlet.portletName + "/" +
                    transition.name +TransitionId.next.to_s
    ...
   end
  end
 ...
end
```

Figure 4.21: Mapping from state configurations to *eXo* pages file. Anchor rendering strategy is *together* and *top.*

```
explicit_execution do
(13)  SOP::Orchestration::StateMachine.all_objects.each do |state_machine|
(14)     page_set = EXO::Pages::PageSet.new
(15)     state_machine.all_configurations.each |configuration|
(16)        execute_phase 'page',
              :current_config => configuration,
              :page_set => page_set
         end
      end
end
```

Figure 4.22: Mapping from state configurations to *eXo* pages file. (cont.)

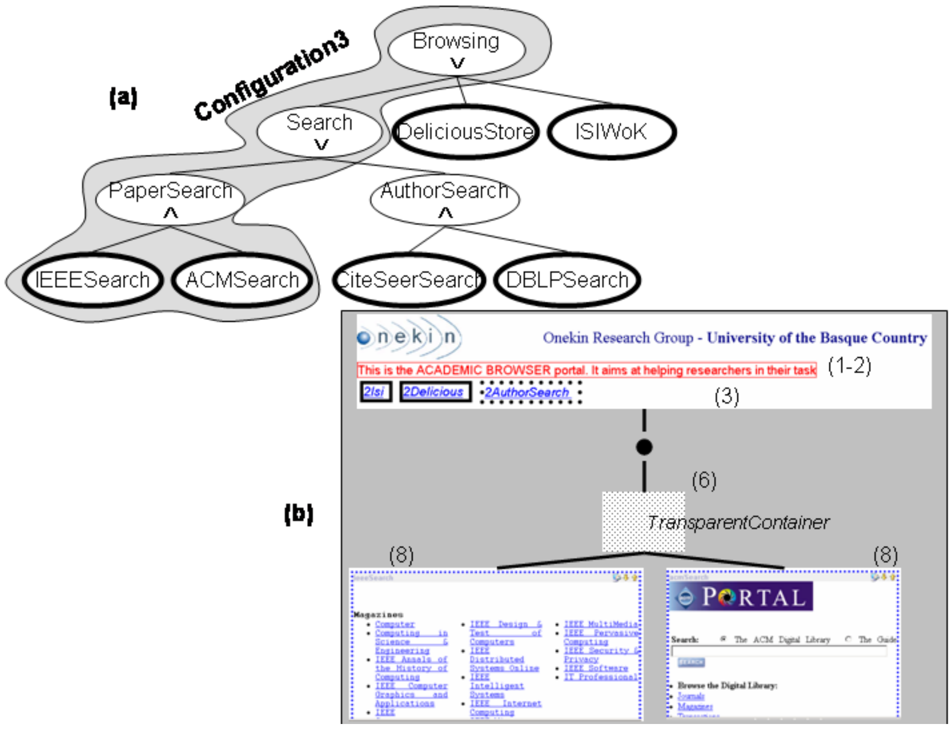


Figure 4.23: *Configuration3* and its presentation counterpart.

tion2Page rule (see Figure 4.21). Again the tree-like structure of configuration leads to a recursive transformation. First, a new page is created: its name and decorator are generated in lines (1) and (2), respectively, and it holds the results of transforming the child states. If the child is simple (and not contained in an AND state) then, a portlet is generated (line (5) that causes the triggering of the *SimpleState2Portlet* rule). If the child is an AND state then, a container is generated (line (4) that causes the triggering of the *AndState2Container* rule) whose content is the result of recursively transforming their child substates (line (7)). Finally, OR states have no impact on the page composition, and the transformation just propagates to their children.

That is for states. Now *transitions* whose PSM counterparts are *anchors*. According with the RENDERING metamodel, active anchors can be placed either together (*distribution = 'together'*) or side-by-side to the portlet being the PSM counterpart of the transition's origin state (*distribution = 'detached'*). Orthogonally, depending on the *position* attribute value, anchors can be placed left, right, up or down relative to the page/portlet. This leads us to eight different types of transformations. In Figure 4.21, the example considers the 'together'/'top' combination only: the *Container4Transitions* rule is triggered to generate the container (in line (3), before the rules related to states; moreover that rule has an appropriate filter in line (11)), and after, one anchor is described for each transition, using the *Transition2Portlet* rule, triggered in line (12) with the binding. Of note, rules for the 'detached' case are also recursive as anchors are placed along the containers hierarchy.

As an example, consider *configuration3* whose active states are *{Browsing, Search, PaperSearch, IEEESearch, ACMSearch}*. Figure 4.23(a) shows the containment relationship between these states, indicating the kind of relationship (i.e., AND or OR). Transformation proceeds from top to bottom and Figure 4.23(b)[3] shows the generation of the presentation counterpart that results from this state configuration whose complete presentation page is shown in Figure 4.7.

The process goes as follows:

1. the root, i.e., *Browsing* state, outputs a container that provides a first skin for the portal (lines (1) and (2) in Figure 4.21). Moreover, since anchors have a *together* distribution and a *top* position, PSM anchors are displayed at this very top decorator (line (3) of Figures 4.21 and 4.23(b)). This container also holds the results of transforming the *Browsing*'s child, i.e., *Search* state.

2. *Search* is an OR state. An OR state does not have a presentation counterpart, and process continues with *Search*'s child, i.e., *PaperSearch*.

3. *PaperSearch* is an AND state. An AND state is mapped to a transparent decorator (line (6)) that forces its content (generated after *PaperSearch* substates) be displayed side-by-side. *PaperSearch* is a conjunction of two simple states (i.e., *ACMSearch* and *IEEESearch*) which stand for two portlets (line (8)).

4. *ACMSearch* and *IEEESearch* are simple states. A simple state produces a portlet description with a reference to a *PortletDecorator* class and a *PortletStyle*

[3]Numbers in Figure 4.23(b) point to operations in Figure 4.21. These operations generate the corresponding PSM widgets.

```
<page-set>
 <page renderer="PageRowRenderer" decorator="browsingPageDecorator">
  <name>/home</name>
  <container renderer="ContainerColumnRenderer" decorator="browsingTransparentDecorator">
   <portlet renderer="PortletRenderer" decorator="ToAuthorSearchAnchorDecorator">
    <portlet-style>ToAuthorSearchAnchorStyle</portlet-style>
    <windowId>@owner@:/navigationstep/step/ToAuthorSearch1</windowId>
    <portlet-preferences>
      <value>ToAuthorSearch</value>
      ...
    </portlet-preferences>
   </portlet>
   ...
  </container>
  <container renderer="ContainerRowRenderer" decorator="browsingTransparentDecorator">
   <portlet renderer="PortletRenderer" decorator="IEEESearchPDecorator">
    <portlet-style>IEEESearchPStyle</portlet-style>
    <title>IEEE</title>
    <windowId>@owner@:/ieeeLibrary/ieeeLibrary/IEEESearch</windowId>
   </portlet>
   <portlet renderer="PortletRenderer" decorator="ACMSearchPDecorator">
    <portlet-style>ACMSearchPStyle</portlet-style>
    <title>ACM</title>
    <windowId>@owner@:/acmLibrary/acmLibrary/ACMSearch</windowId>
   </portlet>
  </container>
 </page>
 ...
</page-set>
```

Figure 4.24: Snippet of the *template-pages.xml* file.

class (lines (9) and (10) in Figure 4.21) that hold CSS parameters for portlet markup presentation.

A snippet of the final outcome for *configuration3* is depicted in Figure 4.24. The content of *decorator* and *style* classes (*IEEESearchPDecorator*, *IEEESearchPStyle* and the like) has been previously generated by the transformation rule introduced in Subsection 4.6.1.

It is worth emphasizing that RubyTL is an embedded language in Ruby. The benefits brought are illustrated in line (14) of Figure 4.22, where an object is explicitly created, and in line (5) of Figure 4.21, where Ruby facilities to traverse collections are used. Moreover, since RubyTL is a hybrid language, bindings and rules provide a clean way to set the mappings in a declarative manner (e.g., in line (4)), while it is also possible to write imperative code where needed (as shown in line (14)). For the sake of legibility of the rules, the possibility of using auxiliary functions is another advantage of RubyTL. Lines (5) and (8) of Figure 4.21 have the same aim, i.e., triggering the *SimpleState2Portlet* rule, but to look for the simple substates of a given AND state, the latter uses the *simple_sub_states()* auxiliary function, whereas

```
context SOP::Orchestration::State do
  inv 'maximized-only-for-one-simple-state' do
    self.isOrthogonal.implies(
      self.child_states.any? { |s| s.isSimple && s.task.portlet.windowState = = 'maximized' }.
           implies(self.child_states.select { |s| s.isSimple }.size = = 1))
  end
end
```

Figure 4.25: Validation rule.

the former uses a long declarative expression.

4.7 Realizing the MDD benefits

Different works address the advantages of MDD in general [96], and to Web development, in particular [33]. Rather than going back to their arguments, this section tries to provide examples of how these advantages turn true for the purpose of this specific project.

It is commonly stated that a main advantage of MDD is to be able to react efficiently and with low costs to technology changes. Although we are conscious about the benefits of platform portability, our troubles do not come so much for technology changes as for inefficient programming and maintenance. Our main motivation rests on the important productivity and quality gains that the model+transformation paradigm can yield as opposed to direct code programming. The rest of the section is devoted to illustrate these gains based on the experience gained during this project.

Analysis. MDD treats models (e.g., statecharts) not just as documentation but as a crucial part of the solution. From an analysis perspective, the main gains come from the verification techniques that statecharts permit and that would have been much harder to achieve if validation were conducted at the *eXo*-code level. PIM offers possibilities for analysis, verification, optimization, parallelization, and transformation in terms of PIM constructs that would be much harder or unfeasible if a programming language had been used [59]. Indeed, statecharts have long being used as a simulation technique, and distinct techniques and tools are available to validate distinct formal properties.

Additionally, portal-specific properties can also be declaratively specified as opposed to the convoluted description that an *eXo* implementation would require. The window-state restriction is a case in point. This restriction states that a portlet with a window state "maximized" must be shown alone in the portal page. The window state is a WSRP property which is captured by the WSRP model. On the other hand, the ORCHESTRATION model implicitly conveys how many portlets are shown simultaneously: if there is an AND state, all its substates must be rendered together. Therefore, the *window-state restriction* can be stated as follows: *IF the windowState of the wsrp portlet related to one task is set to "maximized" AND this task pertains to an "and" state THEN, the number of simple substates in that "and" state must be 1*. This constraint could be described in *OCL* in a declarative way. However, we did not have an *OCL* engine to enforce this constraint, so a RubyTL rule counterpart was specified (see Figure 4.25). This constraint can now be validated against the portal model (i.e., the SOP model), hence, detecting inconsistencies at design time. The important point to notice is that this constraint could have been very cumbersome to enforce directly on *eXo* artefacts!!

Design. Transformations express a correspondence between elements of the source metamodel into elements of the target metamodel. Yet, this correspondence is frequently not unique but distinct ways exist to map the same source model into alternative target models. These design options differ not so much in the functionality supported but on the so-called non-functional features. For instance, when mapping statecharts two options were considered: interpreted versus compiled. These options do not have a major impact on the functionality of the system but they *do* affect non-functional characteristics such as performance or extensibility (see Section 4.6). By using model+transformation rather than direct coding, MDD captures as part of the transformations, the distinct design alternatives and, what is most important, the criteria to be used to prioritize one over the others.

For instance, it could have been possible to go for the compiled alternative if the number of state/transitions were above a certain threshold so that portal performance does not suffer from large statecharts. By contrary, if the statechart did not reach this number *and* it is likely to add new portlets (i.e., new simple states) then, the interpreted option would be a better bet since the additional overhead of interpreting the statechart is compensated by its facility to extend it right away. The important point is that now these criteria are captured in a single place: the transformation.

Implementation. Using MDD practices, most code is generated and derived from a model. This is especially beneficial in the presence of *code clones*, i.e., code fragments repeated in source files of the application. This situation arises at distinct times during the *eXo* portal development. For instance:

- the same portlet can appear in distinct pages (i.e., a simple state can belong to different state configurations). A page description (i.e., the *template-pages.xml* file in Figure 4.24) includes the description of the portlets that form the page (i.e., the <*portlet*> element). Therefore, the <*portlet*> element for the repeating portlet is a *code clone*, i.e., it appears in each page description that renders this portlet.

- portlets, better said, their corresponding simple states, can form higher abstraction units (e.g., *IEEESearch* and *ACMSearch* are abstracted into the *PaperSearch* state). Those portlets that belong to the same abstraction unit can share some presentation parameters. As the notion of abstraction unit is missed in *eXo*, those presentation parameters that provide the look-and-feel of the abstraction need to be repeated for each portlet that realizes the abstraction unit. Likewise, other characteristics of the abstraction (e.g., outgoing transitions) give also rise to *code clones* being distributed all along the page descriptions in *eXo*.

Now more repetitive and error-prone activities are moved to the transformation rules, which focus most of the care and testing. Hence, the chances for implementation pitfalls are reduced and development is streamlined.

Maintenance. Web portals are subject to a continuous evolution. They will have to evolve in response to not only the specific requirements of their stakeholders, but also to the changes of the underneath component framework or the Web platform. Thus, for example, after our proposal, *eXo* platform has released its version 2.0 and it presents differences in the means to configure the presentation of Web portal pages (i.e., the part related to *skin-config* in the EXO metamodel, see Figure 4.17). In order to adapt our approach to the new release, we will have to modify only the EXO metamodel, all portal models will remain without change. Moreover, in RubyTL the feature of *phasing*, that is, modularizing the rules in phases, let us to refactor the rule phase related to rendering without modify the rest of rules.

4.8 Related work

The work presented in this chapter is an MDD approach for Web portal designing, Web portals as a means of integration of portlets, i.e Web components. The work related to this approach can be classified into three main fields, i.e., integration of services, MDD for Web application development, and modelling approach.

Data-intensive model	**Service-oriented model**
Structural model	Task model
Navigation space model	Customized task model
Navigation structure model	Orchestration model
Static presentation model	Rendering model
Dynamic presentation model	–implicit–

Table 4.4: Models for data-intensive portals vs. Models for service-oriented portals.

4.8.1 Integration

Integration can involve an underlying database, external applications (a.k.a. back-end integration) or presentation components (a.k.a. front-end integration). The former can be illustrated through work on data-intensive Web applications [31, 17, 58], i.e., Web applications that run on top of a back-end database system. These works normally start with a *structural model* of the entities involved in the application which normally reflects the database entities. Around this model, other models are introduced, namely, *the navigation space model*, which indicates the objects that can be visited by navigation through the application; *the navigation structure model*, which defines a road map on top of the navigation space; *the static presentation model*, which describes where and how navigation artefacts will be presented to the user; and *the dynamic presentation model*, which addresses the behaviour of the presentational objects, i.e., the changes on the user interface when the user interacts with the system [40]. Table 4.4 indicates a tentative mapping between these models and the ones introduced in this chapter.

These works are for data-intensive Web applications, and hence, the conceptual model is the starting point on which the rest of the perspectives are constructed, i.e., they rely on the existence of a common conceptual model (a.k.a structural model). However, such a premise is not always true in a service-oriented scenario which questions the holistic and linear conception that characterizes traditional database development. In the past, data modelling carried an expectation of unification –a prerequisite for effective communication and data sharing was the agreement of a single common data model. However, in a service-oriented approach partners can be reluctant to disclose their data schemas, and two parties can communicate if they can agree on a proper mapping between their respective data models without the existence of a common structural model. Actually, our approach is *process-centered* rather than *data-centered*.

A recent extension of WebML, a Web modelling language that contemplates the interplay between Web applications and Web services, is also contemplating process-intensive applications [11]. To this end, the Conceptual Design phase of process-intensive applications includes the *Process Design* task, focusing on the high level schematization of the processes underlying the application, and the *Process Distribution* task, which addresses the allocation of subprocesses to different peers, and therefore occurs only when there are several Web servers involved in the process enactment. Process and distribution influence data and hypertext design, which should take into account process requirements. This moves to *front-end integration* where the presentation layer is used to govern the process. It is worth noticing that in [11] Brambilla *et al.* consider a multi-user process where Web applications support a process view for a particular user role. From this perspective, front-end integration (i.e., those based on anchor activation) falls short to enforce constraints between activities assigned to different users. Thus, synchronization across site views is obtained by having activities record their progress in a database, and using conditional navigation (based on the values actually found in the database).

The work presented in this chapter differs from WebML in both the starting setting and the design formalism. Our approach starts with a "palette of components" (i.e., portlets) rather than a conceptual model, and statecharts are used as the main formalism. Statecharts are so far expressive enough to capture task flows, and, as opposed to *ad-hoc* notations, bring all the experience on validation and verification techniques, and tooling that are so important when developing Web portals.

Workflow Management Systems are currently slightly evolving towards service-oriented architectures based on Web Services. However, here integration is at the back-end where the browser just provides a view for what is happening at the back-end. This approach was pioneered by *WWWorkflow* [2]. Designed for intranets where the users exploit WWW browsers as the only client software, its architecture contains three main parts: the workflow engine, the workflow database and a CGI-gateway to the WWW. The latter allows users to interact with the system through a triple frame presented in their browsers. However, unlike our approach, only one activity is available at a time, and all the flow dependencies are enforced at the back-end.

Finally, Web applications as conglomerates of presentation components is a more elusive subject. It has been recently addressed in [115] where a framework is introduced for integrating components by combining their presentation front-ends. A composite application consists of one or more components and a specification of the composition model (i.e., integration logics that coordinate the components at runtime). The work stresses the importance of an event-driven architecture where the composition model includes event subscription information to facilitate the communication among presentation components. *Component coordination* is specified through events and in the application all components are presented side-by-side. The work presented in this chapter focuses on portlets and *portlet orchestration*. Both works have similarities: the component model is similar to our task model, with portlets, and the composition model in this chapter is described through statecharts where events are restricted to describe anchor navigation, i.e., intra-portal events instead of intra-portlet events. The work in [115] can be considered a Domain-Specific Language for composite Web applications. By contrast, the approach of this chapter attempts to build upon existing formalisms (e.g., statecharts) rather than coming with a new language, and then, to map this formalism to concrete frameworks where the one of [115] can be included as a PSM. At the current stage of technology where presentation integration is still immature, an MDD approach as the one presented in this chapter, facilitates easy platform portability and user adoption by building on existing (meta) models.

4.8.2 MDD for Web application development

Like the work of this chapter, the WebSA approach [61] also establishes an instance of the MDA development process. Firstly, the Web application specification is divided into two viewpoints: the functional-perspective (i.e., Content, Navigation Structure, Business Processes and Presentation models), and the Subsystem Model and the Configuration Model, which define the software architecture of the Web Application. Then, those viewpoints are integrated by means of transformations between models. The first transformation step merges the elements of the architectural models with those of the functional models, and translates them into the Integration Model, the second transformation step maps the platform specific implementation models (e.g., J2EE or .NET) from the Integration Model. The transformations are specified with a visual and textual notation, known as QVT-Partners [86], which is a proposal for Query/Views/Transformations (QVT) of OMG. The SOP model of the approach presented in this chapter is similar to the Integration Model and is also generated by a merge transformation, however the former is conformed only by functional aspects (that is, tasks, orchestration and rendering), and thus, it is more abstract, i.e., it does not have descriptions about ports, component connectors, server pages, and the like.

In the same way, Muller *et al.* [67] apply also the MDA vision to Web engineering. Their work describes a metamodel specific to dynamic Web page composition and navigation. This metamodel is composed of three platform independent packages, i.e., the Business Model, the Hypertext Model, and the Presentation Model, related to the typical content, navigation and presentation models, respectively. The authors have also implemented *Xion*, an action language based on OCL, used in the business model to express the methods or to extract information, and in the hypertext model whenever a constraint or a behaviour has to be expressed. This code helps to drive the generation of the final Web application. The Web application is generated through a PIM to PSM translation which has two PSM steps: from PIM to a platform-dependent layer and from this one to a technology dependent layer. Due to reusing the business logic from third-party portlets and annotating the orchestration statecharts with rendering information, our approach does not need any additional mechanism to express behaviour or extract information. Models are not obscured by adding transformation-like code.

MDA aside, Nunes and Schwabe [69] show the combination of Model Driven Design and Domain Specific Languages (DSL's) for the rapid prototyping of Web applications. Their HyperDe environment allows the implementation of Web applications designed using the SHDM method, a model-driven approach based on conceptual, navigational and interface models. The HyperDe also has a Domain Specific Language, and thus, for instance, the developer can include some DSL code in the navigational model for retrieving values that flow in the interface. This DSL code can manipulate both models and SHDM's metamodel. Therefore, instead of automated transformations between models to generate code, HyperDe allows to write DSL code by directly manipulating the models that specify the application. Developers will have to use this type of code for the business logic that is specific to the application or for the code in the templates that render the interfaces to the application. This chapter has been concentrated on a specific Web application, that is, Web portals constituted by reused portlets. The burden of business logic is on portlets so the designer has not to be worried about that. And as for the interface description, it is distributed among the portlets (reused) and the rendering model of the portal. This model has enough parameters so that the designer can configure the presentation of the portal without introducing specific code. Therefore, an MDA approach, with an automatic transformation from models to code, is really enough.

4.8.3 Modelling approach

For the description of the concepts and semantics used to model Web applications, Koch and Kraus [56] present a metamodel for the UWE methodology, and Moreno and Vallecillo [64] describe another metamodel to facilitate the integration of Web applications with third-party systems, e.g., portlets. Both approaches have chosen to define the metamodels as UML profiles.

A UML profile is a UML extension mechanism that allows for the definition of stereotypes, tagged values and OCL constraints to describe new modelling elements. One of the advantages of defining profiles is that they will be supported by standard UML CASE-tools with support for the UML extension mechanisms.

Another approach to define new metamodels is the use of MOF [79] and that is exactly the approach taken in the work of this chapter. Desfray [20] explains that choosing a MOF-based technique is justified when *"(1) the domain is well defined, and has a unique well accepted main set of concepts; (2) a model realized under the domain is not subject to be transferred into other domains; and (3) there is no need to combine the domain with other domains"*. Web portal domain fits the latter two conditions. And about the first one, given that Web portal design is still a research field, we cannot say that there is a total agreement in the domain and their concepts, and as if that were not enough, many times the proposed metamodels are subject to change. Therefore, for now it will probably be the same defining them using a MOF-based technique or a UML profile-based technique.

4.9 Conclusion

The new releases of the WSRP [74] and JSR-286 [49] standards promise to achieve portlet interoperability. This will certainly fuel the transition from *content syndication* to *portlet syndication*. In this context, portlet-oriented methodologies will be highly sought.

This chapter illustrates the use of MDD techniques to achieve such scenario where Web portal construction is now a pipeline-like process of model transformations. Specifically, the designer abstracts away from portal pages, and describes Web portal behaviour and its presentation in terms of annotated statecharts that are then gradually realized as *eXo* artefacts. The chapter strives to illustrate the benefits brought by MDD in general, and the use of statecharts in particular, for the design of portlet-intensive portals. Advantages are reported for the analysis, design, implementation and maintenance of the Web portal.

This chapter looks at portlets as “passive components” whereby orchestration is led by the Web portal (i.e., the portlet container). Moving from one portlet to the next is achieved just using portal resources: traditional anchors that realize navigation between pages which hold the portlets. Portlets themselves are unaware of this navigation. We refer to this orchestration as “navigational orchestration”. But this is not enough.

Portlets can also be “active components”, which consume/produce events risen in their environment, i.e., the Web portal. Such perspective, provides a complementary view to navigational orchestration. Besides direct user interactions, now portlets can also subscribe to events generated by other portlets or even the portal itself. This is a main departure from traditional Web design and implementation. This is the topic of the next chapter.

Chapter 5

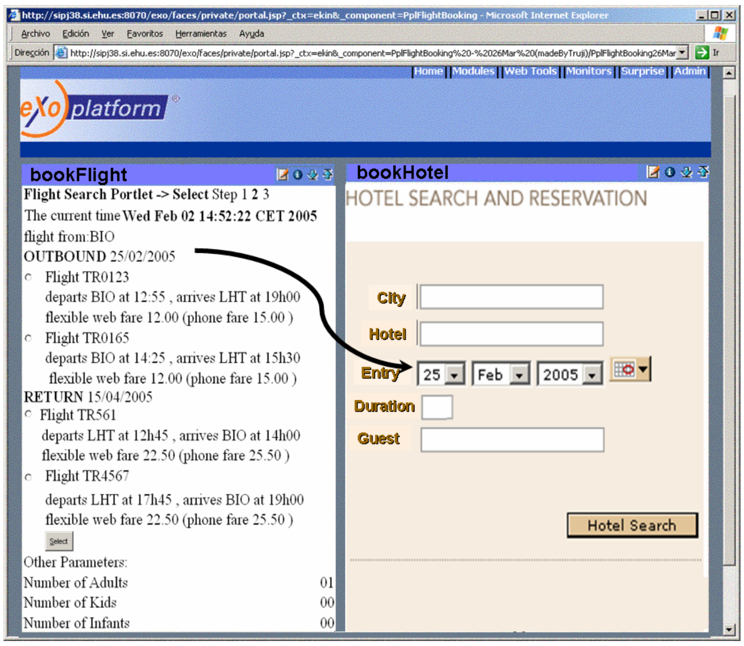
Portlet Interoperability Through Deep Annotation

5.1 Introduction

According to the IEEE Standard Computer Dictionary, interoperability means *"the ability of two or more systems or components to exchange information and to use the information that has been exchanged"*. The previous chapter has **not** considered "interoperable portlets". Rather portlets behave as "passive components". Portlets are black-boxes with no-interaction between the portlets themselves.

However, aggregating portlets into a Web portal is more than merely invoking these services, or arranging their fragments together in the same portal page (i.e., the so-called *"side-by-side"* aggregation). Information contained in one portlet will surely be required in another, and forcing the individual user to manually copy and key in data from source to target portlets would lead to frustration, lost productivity, and inevitable mistakes.

This chapter addresses portlet interoperation, i.e., the exchange of information between portlets. Current approaches to portlet interoperation rely on the existence of a common data structure that supports data exchange. However, such approach defeats the view of portlets as SOA-enablers. That is, interoperation should be achieved between portlets with different origins (i.e., different providers or producers), and this


http://sipj38.si.ehu.es:8070/exo/faces/private/portal.jsp?_ctx=ekin&_component=PplFlightBooking - Microsoft Internet Explorer
Archivo Edición Ver Favoritos Herramientas Ayuda
Dirección http://sipj38.si.ehu.es:8070/exo/faces/private/portal.jsp?_ctx=ekin&_component=PplFlightBooking%20-%2026Mar%20(madeByTruji)/PplFlightBooking26Mar
Ir
Home Modules Web Tools Monitors Surprise Admin
eXo platform
bookFlight
bookHotel
Select
Hotel Search

Flight Search Portlet -> Select Step 1 2 3
The current time Wed Feb 02 14:52:22 CET 2005
flight from:BIO
OUTBOUND 25/02/2005
- Flight TR0123
  departs BIO at 12:55 , arrives LHT at 19h00
  flexible web fare 12.00 (phone fare 15.00)
- Flight TR0165
  departs BIO at 14:25 , arrives LHT at 15h30
  flexible web fare 12.00 (phone fare 15.00)

RETURN 15/04/2005
- Flight TR561
  departs LHT at 12h45 , arrives BIO at 14h00
  flexible web fare 22.50 (phone fare 25.50)
- Flight TR4567
  departs LHT at 17h45 , arrives BIO at 19h00
  flexible web fare 22.50 (phone fare 25.50)

Other Parameters:

Number of Adults	01
Number of Kids	00
Number of Infants	00

HOTEL SEARCH AND RESERVATION

City	
Hotel	
Entry	25 Feb 2005
Duration	
Guest	

Figure 5.1: Two portlets side-by-side: *bookFlight* (left), and *bookHotel* (right).

means that no common data structure will be available (in the same way that traditional Web Services do not rely on the existence of such data structure to exchange data).

Therefore, we propose a front-end approach to portlet interoperation. That is, the visual part of a portlet, the fragments, are supplemented with information about what these fragments render. This requires the creation, either manually or semi-automatically, of metadata from existing information, a process known as *annotation* [35]. Specifically, we consider the so-called deep annotation as particularly valid for portlet interoperation due to the controlled and cooperative environment that characterizes the portal setting. The *portlet producer* can extend a portlet markup, a fragment, with data about the processes whose rendering this fragment supports. Then, the *portlet consumer* (e.g., a portal) can use deep annotation to map an output process in fragment A to an input process in fragment B. This mapping results in fragment B having its input form (or other "input" widget) filled up.

The rest of the chapter is organized as follows. First, we set the problem statement with the help of an example. Section 5.3 describes the notion of annotation. Next, Section 5.4 outlines the deep annotation process, particularized for the portlet interoperability scenario. Sections 5.5, 5.6, and 5.7 present details about that process.

Related work is presented in Section 5.8. Finally, Section 5.9 draws the conclusion.

5.2 A sample case

Consider a Web portal for helping customers to arrange their trips. That portal aggregates two portlets, one for flight booking, *bookFlight*, and the other for hotel booking, *bookHotel* (see Figure 5.1). Each portlet has a different provider.

Aggregating these two portlets together is more than merely invoking these services, or arranging their fragments together in the same portal page (i.e., the so-called *"side-by-side"* aggregation). Rather, data contained in one portlet (e.g., *arrival-date* for *bookFlight*) will surely be required by the other (e.g., *entry-date* for *bookHotel*), and forcing the user to manually copy and key in data from source to target portlets leads to frustration, lost productivity, and inevitable mistakes. Indeed, the webmaster wants these two portlets to interoperate so that data can flow smoothly from the former to the latter. That is, *bookHotel* can render the fragment which prompts for the *entry-date* already filled up from the *arrival-date* obtained after enacting *bookFlight*.

For portlet interoperation (i.e., data exchange) distinct mechanisms have been proposed which can be classified as data-based and API-based. The former permits distinct portlets share a common piece of information but within the scope of the same producer. Portlets which pertain to distinct producers remain isolated. On the other hand, the API-based approach facilitates a programmatic interface for portlets to communicate their state to interested parties. Unfortunately, at the time of our research and proposal, there was not yet an agreement on how to standardize this mechanism. Later, in December 2007, the final draft of the Java Portlet Specification (JSR-286) [49] was proposed. It introduces the notion of portlet event, as a means of communication among portlets. The portlet should declare in the *portlet.xml* deployment descriptor all events that it would like to receive and the ones it would like to initiate. Events could be either portal or portlet container generated or the result of a user interaction with other portlets. However, this does not solve semantic mismatches that can arise between the different interfaces.

Tetlow *et al.* point out [102] how Semantic Web technologies can be applied in Systems and Software Engineering. Automatic searching of Web services, model checking, standardization of the terminology used in different models, interoperability in distributed systems and heterogeneous data sources such as data warehouses are some of the areas where Semantic Web can contribute. In that context, in this work we propose another area such as the portlet interoperability. Roman *et al.* [89] established that *"Semantic markup shall be exploited to automate the tasks of Web service discovery, composition and invocation, thus enabling seamless interoperation between them while keeping human intervention to a minimum"*. Thus, our approach is to enable the seamless interoperation among portlets using *deep annotation*.

5.3 Semantic Web and Annotation

According to the *New Oxford Dictionary of English* an annotation is a note of explanation or comment added to a text or diagram. Annotations have been used extensively in several fields as programming languages, hypermedia and the like. As an example, in Java, annotations give the ability to provide hints directly in the source code which can later be used by the compiler, by other parts of the code, or by another tool; annotations can be used both at compile time and at runtime.

Within the Semantic Web context annotations are also present. To quote the Semantic Web Agreement Group *"The Semantic Web is a Web that includes documents, or portions of documents, describing explicit relationships between things and containing semantic information intended for automated processing by our machines"*. Those explicit relations and semantic information are precisely annotations, semantic annotations. These annotations will be defined formally using ontologies, another of the basic components of the Semantic Web. For the Web *"an ontology is a document or file that formally defines the relations among terms. The most typical kind of ontology for the Web has a taxonomy and a set of inference rules"* [6]. The taxonomy defines classes of objects and relations among them. Through the inference rules the computer can manipulate the terms in an effective manner, that is, it can deduce information.

On the other hand, as Berners-Lee *et al.* [6] established, in the Semantic Web *"the information is given well-defined meaning, better enabling computers and people to work in cooperation"*. Besides, they pointed out that *"The real power of the Semantic Web will be realized when people create many programs that collect Web content from diverse sources, process the information and exchange the results with other programs. Even agents that were not expressly designed to work together can transfer data among themselves when the data come with semantics"* [6]. So, annotations based on a common ontology can provide a common framework for the integration of different programs or systems, that is, they can provide interoperability.

As it has been defined in the Chapter 2, portlets are third-party applications within a Web portal. However, aggregating portlets into a portal is more than merely invoking these services or arranging their fragments "side-by-side". Information contained in one portlet will surely be required in another. Therefore portlet interoperability should be a must to carry out in the portal design. And, given that according to Berners-Lee *et al.* program interoperability may be carried out by annotations, our proposal is using annotations for portlet interoperability, more specifically, for exchange of metadata, i.e., semantic meaning of data, among portlets in order to share data.

This requires first the creation, either manually or semi-automatically, of metadata from existing information, a process known as *annotation* [35]. Most of the approaches to annotation build on the assumption that the information sources are static (e.g., static HTML pages), i.e., they provide metadata about the surface of what is being annotated, e.g., an HTML page. However, this is not always the case for Web pages nor is it for portlets. As stated by Handschuh *et al.* [36], *"for dynamic Web pages (e.g., ones that are generated from a database...) it does not seem to be useful to manually annotate every single page. Rather one wants to annotate the database in order to reuse it for one's own Semantic Web purpose"*. This leads to the notion of *deep annotation*.

Deep annotation has been proposed by Handschuh *et al.* [37] as an annotation process that *"utilizes information proper, information structures and information context in order to derive mappings between information structures"*. This process is called deep annotation *"as its purpose is not to provide semantic annotation about the surface of what is being annotated, this would be the Web page, but about the semantic structures in the background"* [1]. For dynamic Web pages, now, the page also conveys the tables, attributes and the query used to recover the content being rendered in the page. Thus, the HTML "surface" is used to obtain the underlying structure, e.g., the database schema. This information structure/context (i.e., tables, attributes, query) can now be annotated (i.e., mapped) to the information structures/-context of the client, and in so doing, permits the client, i.e., the *querying party*, to consult the database without the help of the HTML "surface" (e.g., through a Web service).

According with the proponents, deep annotation involves three actors: the *back-end owner* (e.g., the database administrator), the *annotator*, and the *querying party*. If the back-end resource is a database as illustrated in [37], then these actors interact as follows:

1. The *backend owner* produces server-side Web page markup according to the database's information structures. The outcome is a set of HTML pages that convey not only the data but also which database columns provide the data (among other aspects).

2. The *annotator* produces client-side annotations which conform to the client ontology and the server-side markup. In this context, an annotation is a set of instantiations related to a (client) ontology and referring to a (server-based) HTML document.

3. The *annotator* publishes the client ontology and the mapping rules derived from annotations. The goal of the mapping process is to give interested parties access to the source data. All information, including the structure of all tables involved in a Web site query, must be published so that users can retrieve data.

4. The *querying party* loads client's ontology and mapping rules, and uses them to query the information source via a Web service API, and without the intervention of the HTML page.

Similar to dynamic Web pages, Web portal pages are generated dynamically from portlet fragments. Each Web portal page is a conglomerate of portlet fragments, fragments that can be different depending on the specific portlet state. Therefore, Web portals are not a candidate to manually annotate every single page either. Thus, using a deep annotation approach also, portlets could be characterized by their domain ontologies and portlet interoperability could be achieved through mappings of instances of these ontologies. This is the first idea of our proposal, which we will describe in more detail in next sections.

5.4 Outline of using annotation for portlet interoperability

Consider our running example. We left the webmaster trying to figure out how to achieve that *bookHotel* can render the fragment which prompts for the *entry-date* already filled up from the *arrival-date* obtained after enacting *bookFlight* (see Figure 5.1). In order to use the *arrival-date* datum, the *bookHotel* portlet should understand its semantics or meaning. And in this step is where we see the role of *deep annotation*.

The key aspects of the approach can be summarized as follows:

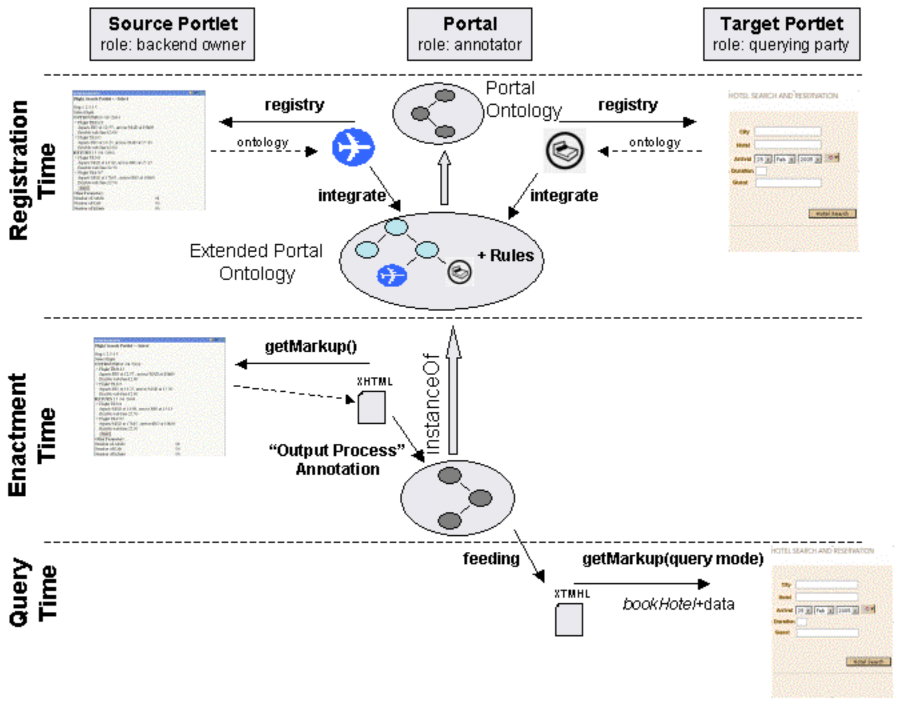


Figure 5.2: An architecture for deep annotation adapted for the portlet case.

1. Portlets are characterized by their ontologies. Although none of the portlet standards (i.e., WSRP [71] and JSR-168 [50]) contemplate this option, the extensibility mechanisms available in both standards can be used to extend the portlet description with an additional *ontology* property. Besides facilitating portlet interoperability, all the benefits of using explicit ontologies (e.g., better documentation, search, knowledge acquisition [48]) are brought to the portlet realm. The *bookFlight* portlet should then publish its ontology, describing the semantics of its concepts and relations. On the other hand, given that portlets are independent, the second portlet, *bookHotel*, has got no reason to know the former ontology. It should be up to the portal to know the semantics from both portlets and to act as a mediator between both sides, using also an own ontology.

2. Portlet fragments extend their markups with information about the processes these fragments support. So far, the fragment markup is geared towards rendering (e.g., XHTML). Now, this markup also conveys information about the underlying processes. This idea comes from previous works on deep annota-

tion.

3. Portlet interoperability is achieved through mappings of the ontology instances. Mapping is necessary as portlet producers can have their own ontologies, and mapping is required to indicate how instantiations from one portlet "flows" to a neighboring portlet.

As seen in Section 5.3, deep annotation approach in [37] is based on three roles: backend owner, querying party and annotator. In the previous portal scenario, the *backend owner* corresponds to the source portlet *bookFlight;* the *querying party* maps to the target portlet *bookHotel*; and the *annotator* role is played by the Web portal. Figure 5.2 gives an overview of this approach:

- At registration time, the portal designer loads the ontologies for the distinct portlets, and integrates them with the portal's ontology.
- At enactment time, fragment information is used to produce annotations according with the portal's ontology. The portal keeps track of the distinct interactions with the portlets in terms of instantiations of the portal's ontology.
- At query time, target portlets can use those instantiations "to feed" their fragments.

Compared with back-end approaches, this mechanism makes explicit what is hidden in the data-based approach, and unlike the API-based proposal, requires no agreement with other portlet producers.

As noted by Handschuh *et al.* [37], deep annotation relies on the cooperation of the markup producer who has to embed the "underlying information structure" into the HTML markup. Indeed, our approach rests on fragments being supplemented with information about the underlying processes. We argue that this assumption (i.e., producers cooperation) is valid here. The argument is two-fold. First, the additional effort required by this extra markup pays off in terms of achieving portlet interoperability. This in turn, leads to improve the user experience of the Web portals where these portlets are syndicated. Hence, portal masters will favor those portlet producers that facilitate this feature. Second, the mistrust to share the ontology can be overcome by requiring prior registration. It is a common scenario to require a Web portal to register with the producer prior to use its portlets (e.g., for charging matters).

Registration ensures a controlled environment where the producer can feel confident when disclosing its ontology.

As it has been advanced in the example, our approach raises the following issues:

1. Defining the ontologies for the portlets and the portal.
2. Fragment annotation, i.e., producing a set of instantiations related to the portal's ontology and referring to the fragment markups of a source portlet.
3. Fragment querying, i.e., "feeding" the markup of a target portlet from annotations kept by the portal.

Next sections address these concerns with the help of the running example.

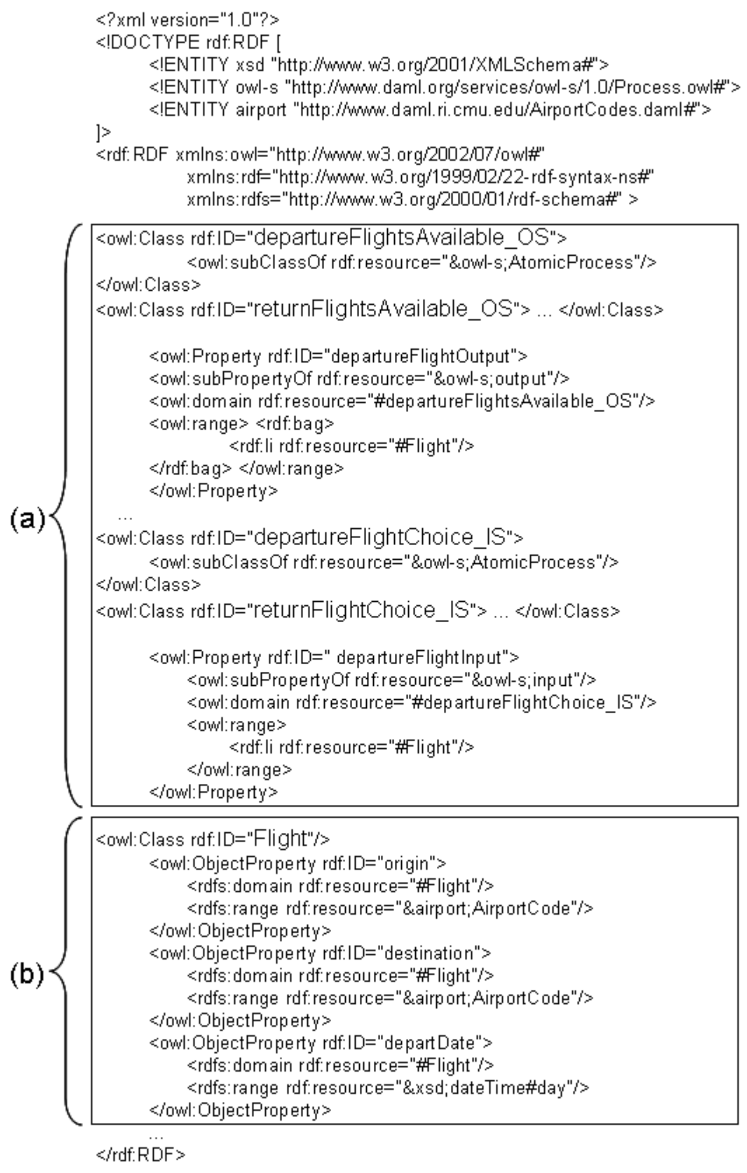
5.5 Portlet ontology and Portal ontology

For the purpose of this work and in order to describe the semantics of portlets and Web portals two types of ontologies are defined: portlet's ontology and portal's ontology. The former describes the concepts of the portlet domain and operations that work with those concepts, i.e., operations that need data, and operations that output data. Therefore the portlet's ontology is a compound of domain and task ontologies. As for portal's ontology, it only describes the events in the Web portal. As a conglomerate of portlets, a Web portal does not have any other role but a mediator among portlets, so the Web portal is only characterized by the events which happen in a portlet and that might be of interest to another portlet in the same Web portal.

5.5.1 Portlet ontology

A portlet carries out a multi-step process, rendering a fragment with each step. In that fragment some result data are shown, or otherwise some data are requested to the user, i.e., data to produce the result data. Therefore, in this approach a portlet is characterized by the set of input and output processes that can occur along its lifecycle (i.e., the task ontology) and by data the portlet provides and requests through those processes (i.e., the domain ontology). The portlet's ontology is composed of both sub-ontologies.

To describe both input and output operations, *OWL-S Atomic Processes* is used as the baseline ontology [110]. OWL-S is an initiative of the Semantic Web community to facilitate automatic discovery, invocation, composition, interoperation and



```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
        <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">
        <!ENTITY owl-s "http://www.daml.org/services/owl-s/1.0/Process.owl#">
        <!ENTITY airport "http://www.daml.ri.cmu.edu/AirportCodes.daml#">
]>
<rdf:RDF xmlns:owl="http://www.w3.org/2002/07/owl#"
            xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
            xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#" >
<owl:Class rdf:ID="departureFlightsAvailable_OS">
            <owl:subClassOf rdf:resource="&owl-s;AtomicProcess"/>
</owl:Class>
<owl:Class rdf:ID="returnFlightsAvailable_OS"> ... </owl:Class>

        <owl:Property rdf:ID="departureFlightOutput">
        <owl:subPropertyOf rdf:resource="&owl-s;output"/>
        <owl:domain rdf:resource="#departureFlightsAvailable_OS"/>
        <owl:range> <rdf:bag>
                <rdf:li rdf:resource="#Flight"/>
        </rdf:bag> </owl:range>
        </owl:Property>
    ...
<owl:Class rdf:ID="departureFlightChoice_IS">
        <owl:subClassOf rdf:resource="&owl-s;AtomicProcess"/>
</owl:Class>
<owl:Class rdf:ID="returnFlightChoice_IS"> ... </owl:Class>

        <owl:Property rdf:ID=" departureFlightInput">
            <owl:subPropertyOf rdf:resource="&owl-s;input"/>
            <owl:domain rdf:resource="#departureFlightChoice_IS"/>
            <owl:range>
                <rdf:li rdf:resource="#Flight"/>
            </owl:range>
        </owl:Property>
<owl:Class rdf:ID="Flight"/>
        <owl:ObjectProperty rdf:ID="origin">
            <rdfs:domain rdf:resource="#Flight"/>
            <rdfs:range rdf:resource="&airport;AirportCode"/>
        </owl:ObjectProperty>
        <owl:ObjectProperty rdf:ID="destination">
            <rdfs:domain rdf:resource="#Flight"/>
            <rdfs:range rdf:resource="&airport;AirportCode"/>
        </owl:ObjectProperty>
        <owl:ObjectProperty rdf:ID="departDate">
            <rdfs:domain rdf:resource="#Flight"/>
            <rdfs:range rdf:resource="&xsd;dateTime#day"/>
        </owl:ObjectProperty>
            ...
</rdf:RDF>
```

Figure 5.3: The portlet's ontology: task ontology (a) + domain ontology (b).

monitoring of Web services through their semantic description. OWL-S is an OWL ontology conceptually divided into three sub-ontologies for specifying what a service does (*profile*), how the service works (*process*) and how the service is implemented (*grounding*). The task ontology focuses on the process side.

As an example, consider the *bookFlight* portlet. This portlet comprises a set of fragments that realizes a multi-step process that ends with the booking of a flight.

The first fragment collects the *departureAirport, flightDates* and so on. Available flights matching these criteria are rendered in the second fragment where the user is prompted to select one of these flights. And so on. The portlet's ontology, *bookFlightOnto*, reflects this step-chain as a collection of input and output OWL-S atomic processes: *returnFlightsAvailable_OS*, *departureFlightChoice_IS* and the like. Figure 5.3 shows an excerpt of this ontology where the suffix *OS* (output service) and *IS* (input service) denote output and input Atomic Processes, respectively[1].

Moreover, the domain ontology describes the concepts included in the parameters of input/output processes. It describes the concepts, its properties and their relations. For the previous example, Figure 5.3 shows the *Flight* class with its properties *origin*, *destination* and, so on. This concept is used in the description of *returnFlightsAvailable_OS* and *returnFlightChoice_IS* processes, for example.

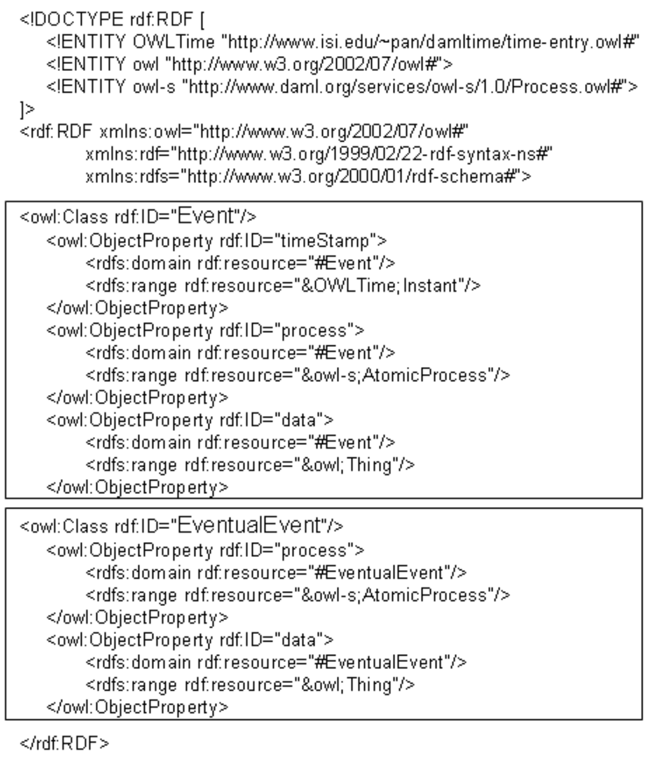
This basic task ontology can now be extended to specify the order in which processes proceed or the relationships between their parameters. For instance, it can be stated that *departureFlightsAvailable_OS* should precede *departureFlightChoice_IS*, and that, at enactment time, the flight in *departureFlightInput* parameter of the latter should be one of the values returned as the *departureFlightOutput* parameter of *departureFlightsAvailable_OS*. To this end, orchestration languages can be used [87]. That improvement is left for future works.

5.5.2 Portal ontology

For our work the Web portal is just an aggregator for portlets with no content on its own. The portal acts as a controller. Based on this perspective, all that matters are the *events* that occur during portlet's enactment. Those events are related to the input and output processes characterizing a portlet, as an example, an event of the *bookFlight* portlet is rendering a list of available departure flights, i.e., execution of the *departureFlightsAvailable_OS* process.

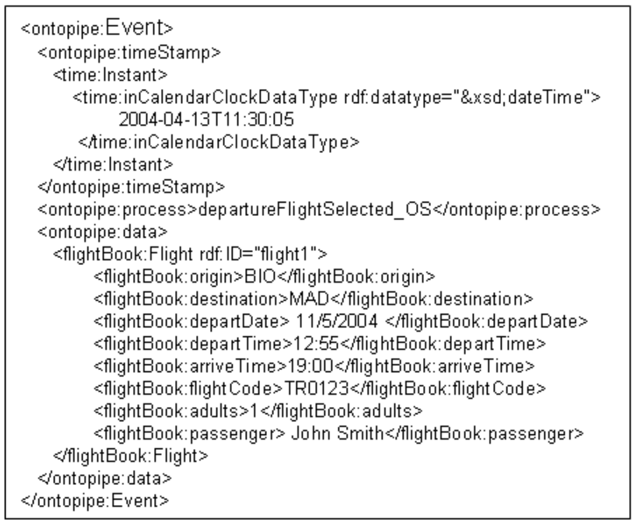
Among events we distinguish of two types: events that have already happened and events that might occur in the future but have not happened yet. We call the latter *eventual events* and they capture the permitted range of actions an end-user can click-on at a given moment and depending on past events. Section 5.6 describes the

[1]It should be noted that for stable domains, this ontology can be standardized in the same way that EDI technologies force the standardization of document formats. The Open Travel Alliance, *www.opentravel.org*, is a case in point. This consortium defines XML Schemas and corresponding usage scenarios for messages that support business activities in the travel industry. This standard can be "OWL-ized", and used for deep annotating travel websites.



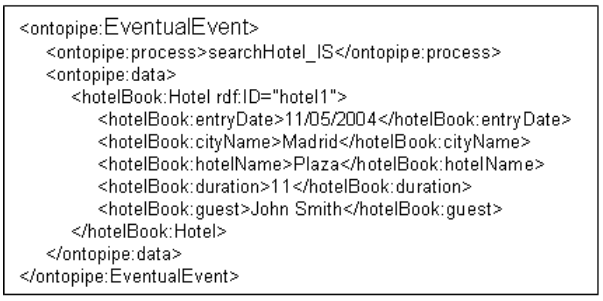
```
<!DOCTYPE rdf:RDF [
    <!ENTITY OWLTime "http://www.isi.edu/~pan/damltime/time-entry.owl#"
    <!ENTITY owl "http://www.w3.org/2002/07/owl#">
    <!ENTITY owl-s "http://www.daml.org/services/owl-s/1.0/Process.owl#">
]>
<rdf:RDF xmlns:owl="http://www.w3.org/2002/07/owl#"
         xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

<owl:Class rdf:ID="Event"/>
    <owl:ObjectProperty rdf:ID="timeStamp">
         <rdfs:domain rdf:resource="#Event"/>
         <rdfs:range rdf:resource="&OWLTime;Instant"/>
    </owl:ObjectProperty>
    <owl:ObjectProperty rdf:ID="process">
         <rdfs:domain rdf:resource="#Event"/>
         <rdfs:range rdf:resource="&owl-s;AtomicProcess"/>
    </owl:ObjectProperty>
    <owl:ObjectProperty rdf:ID="data">
         <rdfs:domain rdf:resource="#Event"/>
         <rdfs:range rdf:resource="&owl;Thing"/>
    </owl:ObjectProperty>

<owl:Class rdf:ID="EventualEvent"/>
    <owl:ObjectProperty rdf:ID="process">
         <rdfs:domain rdf:resource="#EventualEvent"/>
         <rdfs:range rdf:resource="&owl-s;AtomicProcess"/>
    </owl:ObjectProperty>
    <owl:ObjectProperty rdf:ID="data">
         <rdfs:domain rdf:resource="#EventualEvent"/>
         <rdfs:range rdf:resource="&owl;Thing"/>
    </owl:ObjectProperty>

</rdf:RDF>
```

Figure 5.4: The portal's ontology (an excerpt).



```
<ontopipe:Event>
  <ontopipe:timeStamp>
    <time:Instant>
      <time:inCalendarClockDataType rdf:datatype="&xsd;dateTime">
            2004-04-13T11:30:05
       </time:inCalendarClockDataType>
    </time:Instant>
  </ontopipe:timeStamp>
  <ontopipe:process>departureFlightSelected_OS</ontopipe:process>
  <ontopipe:data>
    <flightBook:Flight rdf:ID="flight1">
         <flightBook:origin>BIO</flightBook:origin>
         <flightBook:destination>MAD</flightBook:destination>
         <flightBook:departDate> 11/5/2004 </flightBook:departDate>
         <flightBook:departTime>12:55</flightBook:departTime>
         <flightBook:arriveTime>19:00</flightBook:arriveTime>
         <flightBook:flightCode>TR0123</flightBook:flightCode>
         <flightBook:adults>1</flightBook:adults>
         <flightBook:passenger> John Smith</flightBook:passenger>
    </flightBook:Flight>
  </ontopipe:data>
</ontopipe:Event>
```

Figure 5.5: *Event* instantiation.



```
<ontopipe:EventualEvent>
   <ontopipe:process>searchHotel_IS</ontopipe:process>
   <ontopipe:data>
      <hotelBook:Hotel rdf:ID="hotel1">
         <hotelBook:entryDate>11/05/2004</hotelBook:entryDate>
         <hotelBook:cityName>Madrid</hotelBook:cityName>
         <hotelBook:hotelName>Plaza</hotelBook:hotelName>
         <hotelBook:duration>11</hotelBook:duration>
         <hotelBook:guest>John Smith</hotelBook:guest>
      </hotelBook:Hotel>
   </ontopipe:data>
</ontopipe:EventualEvent>
```

Figure 5.6: *Eventual Event* instantiation.

rationales behind the notion of eventual event more specifically, but now it is a short sketch. Eventual event is the mechanism used to describe interoperability among portlets. Eventual event E_2 of portlet P_2 indicates that event E_1 of portlet P_1 has happened and that if the user wished, the event E_2 might happen too, using data from portlet P_1. In our example, the booking of a flight may eventually lead to the booking of a hotel. So when the user books a flight, the booking of a hotel is then an eventual event (storing some of the data from flight booking). Once the hotel is booked (action carried out by the end-user, using the stored data or not), it becomes an event.

Hence, the portal's ontology includes two main classes: the *Event* class and the *EventualEvent* class (see Figure 5.4). The former describes a happening of interest, and its description includes the following properties: the *process* being enacted, which keeps an *OWL-S AtomicProcess*; the *timestamp* at which this process was enacted whose range is *OWLTime Instant* [84]; and, the *data* of the process, which holds a *Thing*. *EventualEvent*, an event that might happen, has two properties: the *process* that might be enacted, which keeps an *OWL-S AtomicProcess*; and, the possible *data* of the process, which holds a *Thing*. Figures 5.5 and 5.6 show, respectively, an instance of an event and an eventual event.

It is worth mentioning that the *process* properties keep an *OWL-S AtomicProcess* (see Figure 5.4). In the *Event* class, *process* values will be output atomic processes from portlets' task ontologies, and in the *EventualEvent* class, input atomic processes. As for the *data* property, it keeps a *Thing* (see Figure 5.4). This "thing" stands for any of the domain classes of the portlet ontologies. For instance, a thing can be a flight, a city, a hotel, etc. As these domain classes come from distinct ontologies, the portal master must solve first potential mismatches and ontology mappings between the different portlet ontologies. Mapping may become necessary as distinct communities can have their own terms and regulations (e.g., the *bookFlight* portlet follows the

Open Travel Alliance standards whereas *bookHotel* conforms to the normative of a different committee). Ontology mapping is a tough issue whose implications are outside the scope of this thesis. But ontology mapping is a must to achieve portlet interoperability, no matter which approach is used.

5.6 Annotation process

Handschuh *et al.* [36] define “deep annotation” as *“an annotation process that utilizes information proper, information structures and information context in order to derive mappings between information structures”*. Moreover, they describe that process as *“the treatment of dynamic Web documents by annotation of the underlying .database when the database owner is cooperatively participating in the Semantic Web”*. In a similar way Agarwal *et al.* [1] add that this process is called deep annotation *“as its purpose is not to provide semantic annotation about the surface of what is being annotated, this would be the Web page, but about the semantic structures in the background”*. Bearing in mind those five underlined characteristics, we revise next features of our proposal, to conclude that ours is also a deep-annotation approach.

1. The treatment is of portlet fragments. Broadly speaking, a portlet fragment is a chunk of XHTML code (or any other rendering language). So far, the portlet producer delivers this fragment with the only purpose of being readily rendered by the portal.

2. In order that the portal can connect its portlets, those underlying portlets must be annotated.

3. Portlet producers have to cooperate, including annotations in portlet fragments. Those annotations are instantiations of the corresponding portlet ontologies.

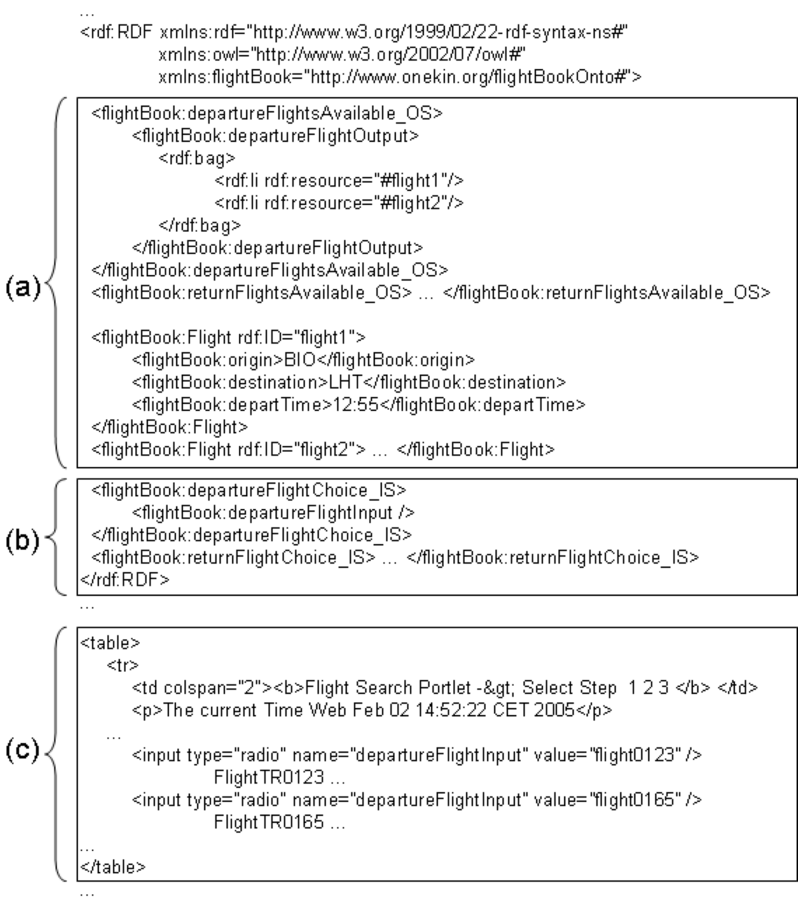
4. Portlet annotations describe the semantics of tasks carried out by the portlet and their data, there are not annotations about the presentation layer.

5. Some mapping rules must be defined in the Web portal. Those rules will define the dataflow from one portlet to another and will use the portlet annotations. The Web portal will use previous rules to make and store more annotations. These annotations are instantiations of the portal ontology.

As defined in Section 5.3 an annotation is *“a set of instantiations related to an ontology”*, and Handschuh *et al.* [36] add *“and referring to an HTML document”*. In

	Who	**How**	**What**	**When**	**Where**
remote annotation	portlet designer	—	portlet ontology	portlet design time	portlet fragment
local annotation	portal	automatic	portal ontology	enactment time	portal inference engine

Table 5.1: Annotation characterization.



```
...
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:owl="http://www.w3.org/2002/07/owl#"
         xmlns:flightBook="http://www.onekin.org/flightBookOnto#">
 <flightBook:departureFlightsAvailable_OS>
      <flightBook:departureFlightOutput>
         <rdf:bag>
               <rdf:li rdf:resource="#flight1"/>
               <rdf:li rdf:resource="#flight2"/>
         </rdf:bag>
      </flightBook:departureFlightOutput>
 </flightBook:departureFlightsAvailable_OS>
 <flightBook:returnFlightsAvailable_OS> ... </flightBook:returnFlightsAvailable_OS>

 <flightBook:Flight rdf:ID="flight1">
      <flightBook:origin>BIO</flightBook:origin>
      <flightBook:destination>LHT</flightBook:destination>
      <flightBook:departTime>12:55</flightBook:departTime>
 </flightBook:Flight>
 <flightBook:Flight rdf:ID="flight2"> ... </flightBook:Flight>
 <flightBook:departureFlightChoice_IS>
      <flightBook:departureFlightInput />
 </flightBook:departureFlightChoice_IS>
 <flightBook:returnFlightChoice_IS> ... </flightBook:returnFlightChoice_IS>
</rdf:RDF>
...
<table>
    <tr>
      <td colspan="2"><b>Flight Search Portlet -&gt; Select Step  1 2 3 </b> </td>
      <p>The current Time Web Feb 02 14:52:22 CET 2005</p>
    ...
      <input type="radio" name="departureFlightInput" value="flight0123" />
               FlightTR0123 ...
      <input type="radio" name="departureFlightInput" value="flight0165" />
               FlightTR0165 ...
...
</table>
...
```

Figure 5.7: The markup of the sample fragment of *bookFlight* (an excerpt).

our context we distinguish two types of annotations, we call them *remote* and *local*. The former 'related to a portlet fragment' and the latter 'related to the enactment of a portlet fragment in the portal'. By "remote annotation" we mean third-party, that is, the portal designer does not have anything to say about it. On the other hand, in the local annotation it is the Web portal which makes the annotations. Table 5.1

summarizes the characteristics of both annotation types in our approach. Subsections 5.6.1 and 5.6.3 describe them thoroughly. Subsection 5.6.2 introduces the process of generation of mapping rules.

5.6.1 Remote annotation with portlet ontology

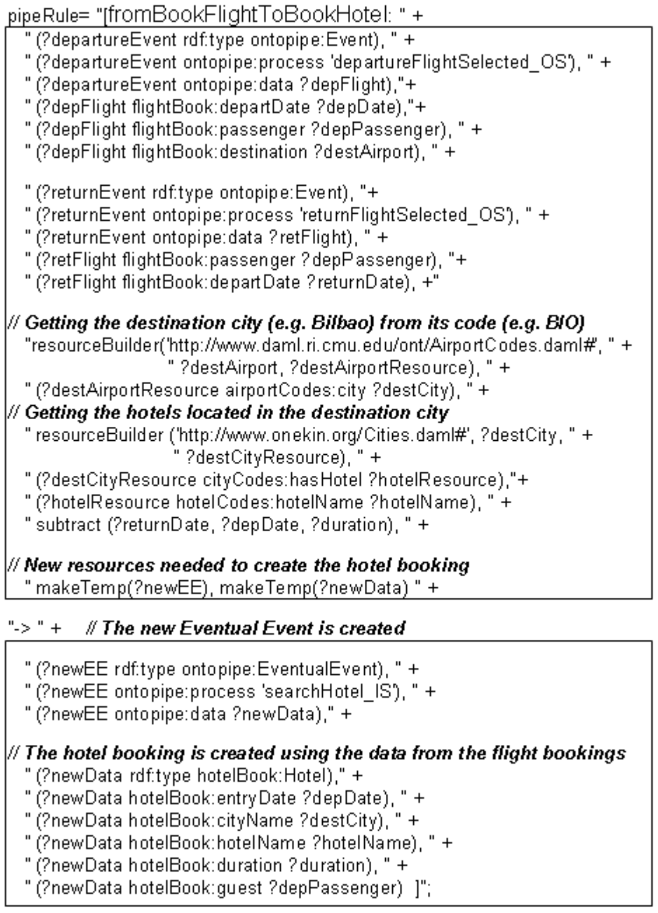
Portlet annotation process must be carried out by the portlet developer during design or development time, according with the corresponding portlet ontology. Whether it is an automatic or manual process is not important for the portal design. At registration time the portal designer will have to know if the portlet will collaborate in the inter-portlet communication. If that is so, she will know that at enactment time the portlet fragments will include instantiations of the portlet ontology, i.e., descriptions of output and input processes (see Subsection 5.5.1). Those annotations will be used in the portal annotation process.

Consider our sample fragment of the *bookFlight* portlet (see Figure 5.1). A snippet of its markup is given in Figure 5.7 where three distinct parts can be distinguished, namely:

- structure/context information markup (see Figure 5.7 (a)). Specifically, for each "output" markup chunk (i.e., the one that renders a meaningful set of data), an additional markup (i.e., an annotation) is inlaid where the outcome is conceived as the result of a function. Our sample fragment conveys two output Atomic Processes (i.e., *departureFlightsAvailable_OS*, and *returnFlightsAvailable_OS*). Each Atomic Process comprises its actual parameters. Process parameters correspond to instantiations of the domain ontology of the portlet, i.e., *flightBookOnto*. The *flightBook* namespace is introduced with this purpose.

- query-oriented markup (see Figure 5.7 (b)), which embeds the type of queries this portlet can make. These queries correspond to widgets such as entry forms which, so far, can only be "answered" by the end-user. For each "input" widget an additional markup is introduced where the widgets are conceived as the realization of an input-only atomic process of the portlet's ontology. Our sample fragment includes two input Atomic Processes (i.e., *departureFlightChoice_IS* and *returnFlightChoice_IS*).

- rendering-oriented markup (see Figure 5.7 (c)), whose purpose is to be interpreted by the browser.



```
pipeRule= "[fromBookFlightToBookHotel: " +
  " (?departureEvent rdf:type ontopipe:Event), " +
  " (?departureEvent ontopipe:process 'departureFlightSelected_OS'), " +
  " (?departureEvent ontopipe:data ?depFlight),"+
  " (?depFlight flightBook:departDate ?depDate),"+
  " (?depFlight flightBook:passenger ?depPassenger), " +
  " (?depFlight flightBook:destination ?destAirport), " +

  " (?returnEvent rdf:type ontopipe:Event), "+
  " (?returnEvent ontopipe:process 'returnFlightSelected_OS'), " +
  " (?returnEvent ontopipe:data ?retFlight), " +
  " (?retFlight flightBook:passenger ?depPassenger), "+
  " (?retFlight flightBook:departDate ?returnDate), +"

// Getting the destination city (e.g. Bilbao) from its code (e.g. BIO)
  "resourceBuilder('http://www.daml.ri.cmu.edu/ont/AirportCodes.daml#', " +
                        " ?destAirport, ?destAirportResource), " +
  " (?destAirportResource airportCodes:city ?destCity), " +
// Getting the hotels located in the destination city
  " resourceBuilder ('http://www.onekin.org/Cities.daml#', ?destCity, " +
                        " ?destCityResource), " +
  " (?destCityResource cityCodes:hasHotel ?hotelResource),"+
  " (?hotelResource hotelCodes:hotelName ?hotelName), " +
  " subtract (?returnDate, ?depDate, ?duration), " +

// New resources needed to create the hotel booking
  " makeTemp(?newEE), makeTemp(?newData) " +

"-> " +    // The new Eventual Event is created

  " (?newEE rdf:type ontopipe:EventualEvent), " +
  " (?newEE ontopipe:process 'searchHotel_IS'), " +
  " (?newEE ontopipe:data ?newData)," +

// The hotel booking is created using the data from the flight bookings
  " (?newData rdf:type hotelBook:Hotel)," +
  " (?newData hotelBook:entryDate ?depDate), " +
  " (?newData hotelBook:cityName ?destCity), " +
  " (?newData hotelBook:hotelName ?hotelName), " +
  " (?newData hotelBook:duration ?duration), " +
  " (?newData hotelBook:guest ?depPassenger)  ]";
```

Figure 5.8: The inference rule related to a pipe from *bookFlight* to *bookHotel*.

5.6.2 Mapping rules

A Web portal is seen as a *collage* of portlet fragments. Each fragment can prompt the user for distinct courses of actions: the *bookFlight* fragment is waiting for the user to select a flight, the *bookHotel* fragment is prompting the user for the date of entrance, and so on. The aim of our approach is to use deep annotation for "feeding" portlet fragments automatically. By "feeding" we mean the process of inlaying data into a current fragment, more specifically in its input widgets. These data are obtained from other fragments, that is, from data rendered by other portlets. In our example, (some) data about the booking of a hotel can be obtained from the previous booking of a flight. In order to get that, we can define a **pipe** from *bookFlight* to *bookHotel*. A pipe describes a dataflow from the source portlet to the target portlet.

Defining a pipe between two portlets means that the semantics of those portlets must be related, that is, a mapping between their ontologies is needed. More specifically, let *Ps* and *Pt* be two portlets which play the role of the source and the target, respectively. A pipe ***Ps—Pt*** is a mapping that specifies how parameters of an *Input* Atomic Process at *Pt* can be obtained from the actual values of an *Output* Atomic Process at *P*s. This definition is decided by the portal designer at portlet registration time. In the example in order to define a pipe from *bookFlight* to *bookHotel*, the designer would define an explicit mapping from *departureFlightSelected_OS* process of the *bookFlight* portlet to *searchHotel_IS* process of the *bookHotel* portlet.

Pipes between portlets, that is, mapping rules between their ontologies, will be used during enactment time by the Web portal, in the role of mediator among portlets. It is the Web portal which automatically feeds the portlet input widgets. In order to do that other mappings are needed, from the semantics of portlet processes to Web portal semantics, that is, events and eventual events. More specifically, given that an event instance represents a happening in the Web portal, and an output process in the portlet represents that some data have been rendered in the Web portal, there is an implicit mapping between *Output* Atomic Process and *Event* concepts. On the other hand, since eventual events are happenings that have not yet occurred, and an input process means that the portlet is prompting the user for some data, there is a mapping between *Input* Atomic Process and *EventualEvent* concepts.

Given the mappings among portlet and portal ontologies, now we can also define a pipe ***Ps—Pt*** as a mapping that specifies how parameters of an *eventual event* for *Pt* can be obtained from the actual values of an *event* for *P*s.

The definition of a pipe between two portlets (or more specifically, the definition of a mapping rule among their processes) will generate an inference rule implementation-wise. Figure 5.8 shows the inference rule corresponding to *bookFlight—bookHotel* pipe. This rule is described *à la PROLOG* using *Jena* [41]. *Jena* is a Java framework for building Semantic Web applications. The framework includes both an RDF and OWL APIs as well as persistent storage for ontologies and statements. In the example, a *Rule* object is defined which includes a name, a list of premises, a list of conclusions, and an optional direction. The premise includes triples, that check the existence of RDF statements in the Jena repository, built-in user-defined functions (e.g., *subtract*), and a set of predefined functions (e.g., *makeTemp*). The rule conclusion generates a new instantiation of *EventualEvent*.

In general, the source of the pipe could be more than one portlet, and composition policies and data consumption policies could be designed.

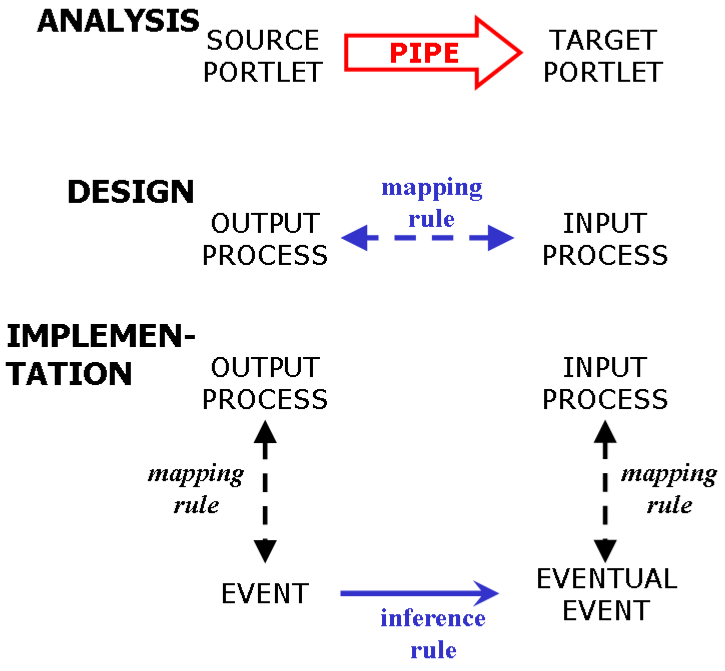


Figure 5.9: Pipe definition through definition of mapping and inference rules.

Figure 5.9 summarizes the relation between the pipe rule between portlets and the mapping rules and the inference rules among portlet and portal ontologies.

While mappings among portlet and portal ontologies are static in the Web portal definition, mappings among portlet ontologies (i.e., pipes) depend on domain semantics. However as a portlet's processes are known in advance, during the design, the set of pipes are also pre-established as part of the Web portal environment. Both types of mappings will be used by the Web portal in its annotation process.

5.6.3 Local annotation with portal ontology

On the contrary to portlet annotation, annotation process related to portal ontology is carried out at enactment time. The Web portal is the annotator and it uses the mapping rules (see Subsection 5.6.2) to carry it out automatically. It is a deep annotating process, that is, a mapping from the information structures found in the portlet's markup to the information structures of the portal. Therefore, in this case, annotations will be *Event* and *EventualEvent* instantiations and they will be stored in the inference engine of the Web portal.

Based on *"output process – event"* mapping (see Figure 5.9), the portal generates a new *Event* annotation of the portal ontology when a source portlet renders data, more specifically, when the portlet fragment contains an annotation of output atomic process. Data in this annotation will serve to define data of the new event. In the example, the booking of a flight, that is, the *departureFlightSelected_OS* process of the last fragment of the *bookFlight* portlet will generate an event instance, shown in Figure 5.5. The annotation process for *EventualEvents* is different, and it uses generated event annotations and the inference rules (see Subsection 5.6.2). Next, Subsection 5.7.1 will describe it, namely, the annotation process used to feed portlet input widgets.

5.7 Querying process

In a traditional setting, deep annotation permits *querying parties* to interact with the background structure without the help of the HTML "surface". By contrast, we do not want to get rid of the HTML surface. One of the added-values of a portlet when compared with traditional Web services is that it comprises the GUI, and we want to keep this interface. We want to interplay with the HTML surface, but with an enhanced HTML surface where entry forms are already filled up. More specifically, we want that some portlet input widgets are filled up with data from other portlets.

Handschuh *et al.* [36] establishes that mapping rules *"enable third parties (querying party) to access and query the (underlying) database on the basis of the semantic that is defined in the (client) ontology"*. In our approach, portlets with input widgets are the querying parties to access and to query data of source portlets. In the query they use their proper ontology, with input processes. This querying process implements the interoperability among portlets. In so doing, the end-user interacts with a portlet but the effects span along multiple neighboring portlets. However, it should be stressed that "feeding" is not a substitution for end-user interaction. That is, it is always up to the end-user to decide whether the hotel is booked with the parameters obtained from *bookFlight* or not.

Subsection 5.7.2 gives some details about querying process, but first Subsection 5.7.1 describes the second step in portal annotation through inference rule. Subsection 5.7.3 discusses about different approaches for piping rules.

5.7.1 Completing portal ontology annotation

As we described in Subsection 5.6.3, while portlets render their fragments, the Web portal stores *Event* instantiations with data shown in those fragments. However, the portal annotation process has a second step to generate the *EventualEvent* instantiations.

This step is carried out using the inference rules (see Subsection 5.6.2) pre-established in the Web portal environment. The eventual events, since they represent happenings that have not occurred yet, get their data from those stored past events.

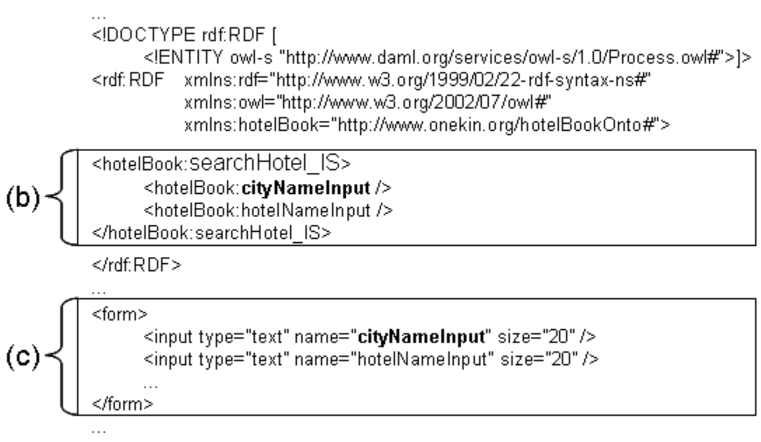
In our example, taking into account the event annotation shown in Figure 5.5, the inference rule *FromBookFlightToBookHotel* will generate the eventual event instantiation as shown in Figure 5.6. In the sample, a *searchHotel_IS* eventual event is obtained from a pair of *departureFlightSelected_OS* and *returnFlightSelected_OS* events. Specifically, the rule checks the existence of a pair of departure and return flight events associated to the same passenger and uses a user-defined function, *subtract*, to calculate the duration of the stay at the hotel.

The final part of the premise uses a predefined function, *makeTemp*, to indicate the creation of two new instances, *newEE* and *newData*, whose properties are assigned in the conclusion of the rule. The former is an *eventualEvent* of type *searchHotel_IS* whose *data* property corresponds to an instantiation of *hotelBook*. The properties of this instance are in turn obtained from the variables which have been instantiated in the premise of the rule, for example, values of *entryDate* and *guest* properties are the same as values of *departDate* and *passenger* parameters in the first event instance, and *cityName* parameter gets its value based on datum in *destination* parameter.

As we will see next, these eventual events will be used in the querying process, that is, to feed those portlet fragments asking for data.

5.7.2 Querying at enactment time

In our context, querying is the process carried out by a portlet when it prompts for data through an input widget, e.g., an entry form. Portlets with input widgets are the querying parties to access and query data of source portlets, and in the query they use their proper ontology with input processes. To this end, a convention is needed to identify which widget obtains the value of which process property. This is achieved by identifying the widget parameter from the input process property of the portlet ontology. Then, based on *"input process – eventual event"* mappings (see Figure 5.9), the portal is able to obtain required data. The querying process finishes with



```
...
<!DOCTYPE rdf:RDF [
        <!ENTITY owl-s "http://www.daml.org/services/owl-s/1.0/Process.owl#">]>
<rdf:RDF   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
           xmlns:owl="http://www.w3.org/2002/07/owl#"
           xmlns:hotelBook="http://www.onekin.org/hotelBookOnto#">
<hotelBook:searchHotel_IS>
        <hotelBook:cityNameInput />
        <hotelBook:hotelNameInput />
</hotelBook:searchHotel_IS>
</rdf:RDF>
...
<form>
        <input type="text" name="cityNameInput" size="20" />
        <input type="text" name="hotelNameInput" size="20" />
        ...
</form>
...
```

Figure 5.10: The markup of the sample fragment of *bookHotel* (an excerpt).

"feeding", that is, an operation that fills up the widget with data from the parameters of an eventual event instance. Now, it is up to the user to accept these values or provide her own.

Figure 5.10 shows a snippet of a fragment of the *bookHotel* portlet (its rendering can be seen in Figure 5.1). The form inputs are identified from the process properties (e.g., *cityNameInput*). Through the *"input process – eventual event"* mapping the portal will identify the corresponding eventual event for the *searchHotel_IS* process (like that in Figure 5.6) and it will get the *cityName* content ('Madrid'). Then, the portal will invoke the *getMarkup()* operation of the querying portlet. To this end, the *getMarkup()* operation has been extended with an eventual-event parameter. On reception, the portlet producer proceeds to feed the current fragment with this parameter content, and then returns the result to the portal as usual. In the current implementation, feeding is implemented as an XSLT stylesheet. A template locates the corresponding *<input>* element in the XHTML markup, and introduces a *value* attribute whose content comes from the eventual-event parameter.

5.7.3 More on piping rules

Web portals exhibit eclectic navigation styles from hypertext-based to totally constrained ones. The former *"lets users explore a body of information freely, by following the available links without obeying to predefined sequences of actions. The power of hypertext is in their feature-rich interfaces for navigating in a non-linear way a collection of related data."* [10]. This is in contrast with workflows, i.e., software systems for directing the work of users, by superimposing control over their activities and supplying only the data needed to accomplish the currently ongoing tasks. In workflow systems, the sequence of possible actions is predetermined and the user is accompanied through the activities according to the workflow specification. Depending on the task at hand, Web portals can be anyway in between these two extremes of the navigation spectrum.

Querying, i.e., the process of making the data flow along one of the pre-established pipes, serves navigation. The time at which querying is enacted can be tuned to the navigation style that better fits the task at hand. Two options are possible, namely:

- *forward style*. By triggering piping rules in a forward mode, the target portlet is fed by the source portlet as soon as the source portlet is enacted. As soon as an event is risen, this happening is piped to all neighboring portlets. In so doing, you are conducting the user towards the next tasks to be fulfilled, i.e., the portlets at the end of the pipe,

- *backward style*. Triggering piping rules in a backward mode implies the dataflow occurring on demand. Here, the happening of an event is not immediately propagated to the piped portlets. There is no update on the fragments of the target portlets. The end-user is not distracted, and he or she can feed the target portlet on demand. Implementation-wise, this is achieved by extending the portlet decorator with an extra icon.

Jena2 includes a general purpose rule-based reasoner which is used to implement the OWL reasoner. This reasoner supports rule-based inference over RDF graphs, and provides forward chaining, backward chaining and a hybrid execution model[2]. The designer should be aware that the triggering mode can influence not only the moment at which the derived data are obtained but the data being derived as well. This stems from event occurrences being inserted in the Jena database continuously as the user interacts with the portlets.

5.8 Related Work

5.8.1 About interoperability or dataflow

Portlet interoperation has been addressed in [91] where the authors propose the use of a custom JSP tag library in order to enable portlets to be a source of data. Moreover,

[2]For clarity sake, the example uses a forward rule.

the target portlet is defined in a WSDL file with a custom extension to describe the actions which can consume data transferred from other portlets. At execution time, a click-able icon is inserted into the portlet fragment. By clicking on this icon, the user enacts the flow of data from the source portlet to the target portlet. Hence, this approach follows a "backward style" of navigation, and piping information is described in the WSDL file. By contrast, our approach uses the fragment markup to convey this information, and uses ontologies to facilitate portlet interoperation. Additionally, the use of inference rules enables sophisticated ways of piping that are "declaratively" described using Jena rules.

This work also relates to Web service composition and orchestration. In the SELF-SERV architecture [4], the composition of Web services is encoded using statecharts. With the statechart, the service deployer generates the post-processing and precondition tables, and this information is distributed among the participating services. During the definition of the composite service, the producer decides if the value of the input of a component is obtained from the output of another component or requested from the user. By contrast, our approach is centralized (i.e., all flow information, the piping rules, are kept in a single place, the Web portal), and it is always up to the end-user to accept the values suggested by the piping flow. This is akin to the portal manners where content is centralized, and freely browsed by the user.

Yu *et al.* [115] agree with us saying that *"integration does not just mean to put the GUIs side by side: interactions need to be coordinated so"*. In order to get this coordination they propose a framework with two models, that is, a model for presentation components and an event-based composition model. The former characterizes a component with *"the notion of state (which defines what the composite application can see and control in terms of changes to the UI), operations to request state changes, and events to notify state changes, mainly occurring due to user interactions"*. The final draft of the Java Portlet Specification (JSR-286) [49] also introduces the notion of portlet events. Events could be either portal or portlet container generated or the result of a user interaction with other portlets. The portlet should declare in its *portlet.xml* deployment descriptor all events that it would like to receive and the ones it would like to initiate. In our approach the component, that is, the portlet, is described using input and output processes, in some sense, similar to operations and events, respectively. The difference is our 'events' do not notify an UI state change, but the production, i.e., the rendering, of some data. More specifically, in [115] user actions are the ones which trigger component-defined events, and they are related to presentation changes in any case, still in our approach the business logic of the component is which decides to notify that new data have been processed and rendered to the user, the change is more related to the business logic.

Moreover, in Yu's approach the component communication is immediate, *"the middleware captures an event from a source component and automatically dispatches it to the designated operations of other components, based on the event listener specifications in the composition model"*. In our approach, using the inference engine, the mapping between input and output processes can be achieved later, when the target portlet is being rendered, because target and source portlets can be shown separately. That apart, inference engine is also useful when more transformations are needed in data mappings, as shown in the example with the "duration" attribute. Yu *et al.* propose the use of XSLT sytlesheets or scripting languages, but we consider using ontologies and semantics can be more profitable, because it offers more opportunities, apart from syntactic transformations.

5.8.2 About semantic approach

Paolucci *et al.* [85] and Sirin *et al.* [97] use a semantic approach for Web service location and composition. DAML-based ontologies are used to describe the inputs and outputs of the services. The semantic match between a service's outputs and another service's input are determined by the minimal distance between concepts in a taxonomy tree. This is similar to our piping in which "matching" between portlets is achieving through the help of the ontology.

IRS-III [23] is an implemented infrastructure which allows the description, publication and execution of Semantic Web services based on the Web Service Modeling Ontology (WSMO) [89]. Cabral and Domingue [14] present an approach for mediation in IRS-III. This mediation framework implements data mediation, goal mediation and process mediation of Semantic Web services. The Process Mediator component executes Web service choreography and orchestration, and it handles mismatches that occur during the invocation or composition of a Web service. The Process Mediator uses GG-mediators and WW-mediators, two kinds of WSMO mediators. The WW-mediator can provide mappings between the input values of the Web services in the orchestration, i.e., it establishes interoperability between Web services. The GG-mediator connects goals that characterize the Web services. These mediators are described using states and transitions, and they play a similar role to inference rules in our approach. They are used to resolve the mismatching of service semantics.

Agarwal *et al.* [1] describe the use of deep annotation for Web service integration. WSDL files are extended with an ontology which is used to describe input and output parameters. The service consumer acts as a querying party by mapping the Web service ontology with its own. A framework, *OntoMat-Service*, generates the mapping rules between the consumer ontology and the ontologies referred to in the WSDL documents. At enactment time, the data for the Web services are retrieved automatically from the client's ontology. From this perspective, our work explores the use of a rule-based approach where the flow is based not just on the matching between parameters but in richer flow policies.

Mrissa *et al.* [65] also propose the use of annotation in WSDL for Web service mediation. They first annotate WSDL descriptions with semantic metadata for capturing contextual information, and then they propose a rule-based mediation mechanism for Web services composition. Besides domain ontologies they define context ontologies. Those describe all the modifiers, static and dynamic, that Web service providers associate to a concept. The rule engine, using Jena 2 as in our approach, applies appropriate conversion functions and inferring rules to the data transmitted, so that the value of the source modifier matches the target context. Mediation functionality is implemented through a mediator Web service, which is generated in a model-driven process. The composition and dataflow is described in a WS-BPEL process and the mediator Web service is located between every two composed Web services [66]. By contrast, in our approach portlet composition is described in the portal pages, putting portlets side-by-side, and dataflow is described through an inference rule and the annotations in composition partners, i.e., portlets. Both, rule and annotations, must be described by designers, there is no automatic generation.

5.9 Conclusion

Enhancing the user experience is one of the hallmarks of Web portals. This implies for the user to perceive a Web portal as an integrated workplace where data flow smoothly among the distinct portlets being framed by the portal. Thus, for example, the user can see some text boxes in a portlet form already filled in with data from other portlets or data previously typed by the user in the context of other portlets. This chapter has presented the use of Semantic Web and, specifically, deep annotation to portlet interoperation.

We argue that deep annotation is particularly valid for portlet interoperation due to the controlled and cooperative environment that characterizes the portal setting. The *portlet producer* can extend a portlet markup, a fragment, with data about the processes whose rendering this fragment supports. Then, the *portlet consumer* (e.g., a Web portal) can use deep annotation to map an output process in fragment A to an input process in fragment B. This mapping results in fragment B having its input form (or other “input” widget) filled up.

This approach then relies on portlet developers to include annotations in portlet fragments. We do not think the latter is an optimistic supposition because the reuse of third-party portlets will depend on the facilities that they offered to be included in external Web portals.

Chapter 6

Conclusion

This chapter reviews our main contributions and proposes new topics for future work.

6.1 Main Contributions

This dissertation has centered around portlet-centric Web portals, and more specifically, the design and the implementation of dataflow among portlets.

6.1.1 Reuse-based Design Method for Web portals

It goes without saying the importance of design for software development. *When the domain complexity is not handled in the design, it won't matter that the infrastructural technology is well conceived* [26]. Initially design methods for Web applications centered in the concerns of content, navigation and presentation. Some of them evolved taking into account business processes and integrated them with the former aspects [11, 90, 57]. A business process defines a sequence of activities to be executed, and those methods describe processes using activity diagrams. Depending on the methods, these diagrams are used in different ways to enrich or derive data and navigation models. However, this dissertation takes into account business processes in a coarser granularity.

Our aim was proposing a design method *with reuse*, following previous approaches of Sametinger [92] and Jacobson [46]. Therefore, we can consider that ours is a bottom-up alternative for Web portal design. Thus, the construction of a new Web portal is based on gathering the appropriate components, i.e. portlets, that are already available. Once "the portlet palette" is obtained, the rest of design steps take that as the baseline. Such design takes statecharts as the main conduit for describing how portlets are gathered together. Additionally, the rendering model specifies the aesthetic presentation of each state. The rendering model takes advantage of hierarchical construction of orchestration model to specify the presentation in a stepwise manner. Thus, aesthetic parameters are obtained through an inheritance-like mechanism based on the state hierarchy.

Therefore, the main outcome of the design is an *annotated statechart* which accounts for the main portal design decisions, i.e. tasks, structural organization and workflow organization and aesthetic concerns.

This proposal was presented in the 7th International Conference on Web Information Systems Engineering (WISE 2006), Wuhan, China: *Modeling Portlet Aggregation Through Statecharts.* O. Díaz, A. Irastorza, M. Azanza, and F.M. Villoria. Lecture Notes in Computer Science - 4255, pages 265-276 (ISBN: 3-540-48105-2). This international conference had an acceptance ratio of 20% (37 out of 183).

Moreover, a preliminary work about different approaches for component-based design was presented in Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2002), Madrid, Spain: *Component-Based Design: Alternatives During Partition Phase* A. Irastorza, A. Jaime, and O. Díaz. Proceedings of the conference, pages 263-271 (ISBN: 0-9700776-4-5). This international conference had an acceptance ratio of about 50%.

6.1.2 Code-generation for Web portals

The proposal presented in Chapter 4 for design of Web portals is based on Web components. It improves productivity and decreases the cost of production, because it isolates functionality and allows for debugging and upgrading it in an independent way. Besides productivity improvement, among our concerns is also the product quality.

We consider that the development quality is improved through two efforts: code reusing and code generation. On one hand, we assume that portlets, as Web components, should be included in a repository, should be reliable, appropriately documented, self-content and sufficiently proved, and they should also use standard architectures. Therefore, the use of such components would be a guarantee for the quality of the obtained Web portal.

On the other hand, code generation is also a technique that strives to improve not only time-to-market but also the quality of the final product by embodying best practices in the transformation. From this perspective, mechanisms were needed to move from statechart models (used during design) to implementation concepts, like Web page, anchor, CSS class, and so on. To this end, an MDD approach is used, specifically, two metamodels are introduced with their corresponding transformations. The first metamodel, called SOP, is a PIM (Platform-Independent Model) and comprises three packages, i.e. task, orchestration and rendering (one for each concern in the Web portal design). The second metamodel is PSM (Platform-Specific Model): the *eXo* platform. The work is completed with the RubyTL implementation of the transformation of PIM-to-PSM and PSM-to-code. The transformation execution is truly automatic with no user involvement required.

This proposal was accepted for publication in the Journal of Information and Software Technology (26th November 2007): *From page-centric to portlet-centric Web development: Easing the transition using MDD*. O. Díaz, A. Irastorza, J. Sánchez-Cuadrado, and L.M. Alonso. (doi:10.1016/j.infsof.2007.11.006). In 2007 this journal had an impact factor of 0.581 © Journal Citation Reports 2008.

6.1.3 Dataflow among portlets

Apart from being an access point to data, one of the most important features of Web portals is its capability for integration. This thesis has been focused on portlet-centric Web portals, so here integration means *portlet interoperability*, that is, the ability of two or more portlets to exchange information.

If the aim of the former contributions was facilitating the work of designers and developers, now the focus is on end-users. Information shown in one portlet will surely be required in another, and forcing the end-user to manually copy and key in data from source to target portlets leads to frustration, and inevitable mistakes.

Although the final draft of the Java Portlet Specification (JSR-286) [49], with the notion of portlet events as a means of communication among portlets, was proposed at the end of 2007, at the time of this work, there was not yet an agreement on how to standardize the portlet interoperability. Our proposal was a front-end approach, where presentation fragments of portlets were annotated with information about the rendering process, that is, information about which data, and how, are shown or request to the end-user. It is a seamless integration approach, where the user does not need to have an active attitude so that the dataflow takes place. OWL rules are which make inference of data, i.e. carry out the flow.

This proposal was presented in the 14th International Conference on World Wide Web (WWW 2005), Chiba, Japan: *Improving Portlet Interoperability Through Deep Annotation.* O. Díaz, J. Iturrioz, and A. Irastorza. Proceedings of the conference, pages 372-381 (ISBN: 1-59593-046-9). This international conference had an acceptance ratio of 14% (77 out of 550).

6.2 Future Work

The three basic ideas presented so far has been implemented and proved as feasible using some sample cases. However more work should be carried out in the Web portal design approach. More specifically, we should collaborate with different industrial partners in order to apply our design approach on real problems. Furthermore, this work might be extended in several areas we present next.

- **Complete SOP metamodel**. The design approach, materialized in the SOP metamodel, is focused on functionality, but a Web portal can contain much more. It can include advertisements, structured and non-structured contents, and the like, so the SOP model resulting from the design process should take them into account. Moreover, apart from a *user-driven control-flow*, a *task-driven control-flow* could also be studied. What we mean by *task-driven control-flow* is that the end of a task could trigger the beginning of another task in the Web portal, without the participation of the user. Modelling would need another type of transitions for this control-flow. Furthermore, our proposal for both rendering and personalization models are preliminary, and more study is required, for example, to take into account usability guidelines and to look for the way to describe a CIM (Computation-Independent Model) for them and then a transformation to the SOP metamodel.

- **MDD approach for dataflow design**. Chapter 5 describes our approach, based on deep annotation, for dataflow implementation, however surely portal designers will miss a means to describe dataflow at design-level. Further work is needed to find a model for this design aspect, and then, to apply an MDD-approach so that inference rules for dataflow can be generated automatically from that model. More specifically, first the WSRP metamodel presented in Chapter 4 should be extended with the description of input and output processes (used in Chapter 5 to carry out the dataflow) and the association among them (in order to represent the design of dataflow). Moreover, a metamodel for the inference rules should also be defined, and then, a transformation from the WSRP-extended metamodel to the new metamodel of inference rules.

- **Dataflow model**. Dataflow is implemented through the inference from data of one portlet to data of another portlet, but more complex semantic rules could be defined, e.g., inference rules with events from different source portlets as participants. Besides, the inference rule model should be improved to include several policies on data events, such us, consumption, deletion and so on.

- **Platform and model evolution**. MDE promises that both development and maintenance effort can be reduced by working at the model level. While distinct experiences provide evidences of the fulfillment of this promise for model-driven development, model maintenance has not received so much attention. However, system maintenance is reckoned to be the most consuming software activity, and MDE has yet to demonstrate that really pays off when facing maintenance. Although, seamless platform evolution was one of the raison d'etre of MDE, it is also true that MDE introduces distinct "abstract platforms" which pose stringent demands when any of these platforms (or their mappings) need to be evolved. Somehow, we have already suffered from this pain. The proposal of this thesis and the metamodel for *eXo* platform are based on version 1.0 of that platform, but version 2.0 has already been released. This problem leads us to another research area more focused on MDD. We should work about how to apply techniques on metamodel evolution to our approach, and how to modify the SOP or EXO metamodels, so that changes in design models can be automatically propagated to the corresponding Web portals already working, or developers can upgrade working portals to the new platform version.

6.3 Some final thoughts

Model-driven engineering departs from traditional software engineering, with its mostly monolithic development platform. Instead of one or a few programming languages, MDE development introduces a multitude of languages that are themselves artefacts of the development process. Traditional projects require competence in the PSM. The very same project using MDD, requires competence in both the PSM and the PIM as well as being knowledgeable about transformation language and MDD environments. Interesting enough, this drawback is seldom mentioned despite being a main stumbling block for companies to embrace this paradigm shift.

We "severely" suffer such drawback. Building portlet-based portals require in itself to master a broad range of technologies. In our case, we had to cope with *eXo* platform, WSRP and Java as the PSMs. But this was not enough. We needed to become familiarized with state machines as the PIM. Finally, model transformation required to become aware of QVT and ATL, which were finally overridden by RubyTL. This implies a large bulk of technologies and platforms just to begin to grasp the solution space.

In a real setting, such heterogeneity calls for a separation between distinct groups. Research wise, this is more difficult to achieve as the research contribution sometimes lies in between. That is, we regard as a main contribution of this dissertation, the *engineering practice* that demonstrates the feasibility of using statecharts as a PIM for *eXo* portals. From this viewpoint, the contribution is not so about statecharts nor the *eXo* platform. There were already there. Neither is it about model transformations, though some improvements in RubyTL were identified during this thesis. Rather, a main challenge rested on combining all technologies and practices to solve a **non-trivial** problem.

Appendix A

SOP-to-EXO Transformations

A.1 Introduction[1]

This thesis dissertation presents an MDD approach for Web portal construction, following the MDA proposal. Thus, two metamodels have been designed, a PIM named SOP and a PSM named EXO (both described in Chapter 4), and the transformation from SOP to EXO has been implemented in the RubyTL language [99]. The main features of that transformation were also presented in Chapter 4, and this appendix describes the transformation rules thoroughly.

Figure A.1 depicts the SOP metamodel, with its three viewpoints integrated, i.e., TASK, ORCHESTRATION and RENDERING. It also shows the relation among the concepts from different viewpoints: (a) a *Workspace* and a *Task* in TASK are related to one *StateMachine* and one *SimpleState*, respectively, in ORCHESTRATION; (b) a *StateMachine* in ORCHESTRATION is related to one *WorkviewDescriptor* and three *Descriptors* in RENDERING, a *Transition* can be related to one *AnchorDescriptor*, a *State* can be related to one *WindowDescriptor* and one *AnchorDescriptor*, and moreover it can be related to one 'helping text' (therefore, the overall description of the portal is at statemachine level, and at state or at transition level special description can be placed, namely, the special description for a specific state, or its inner substates, or a specific transition).

[1]My special gratitude to Luis M. Alonso for his help and support in the creation of this annexe.

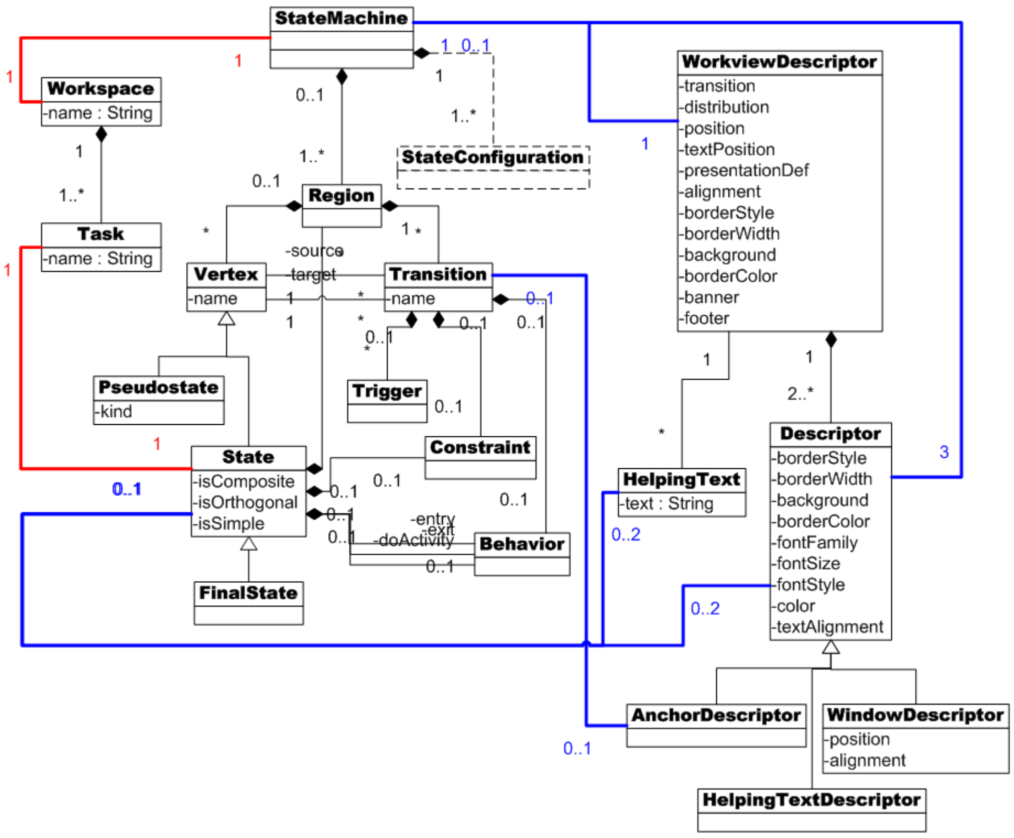


Figure A.1: The complete SOP metamodel.

EXO metamodel is composed of five packages: CONFIG, PAGES, NAVIGATION, CSS, and SKIN_CONFIG (see Figure A.2). Each one corresponds to one of the files to configure the Web portal (***_config.xml*, ***_pages.xml*, ***_navigation.xml*, *{portalName}.css*, and *skin_config.xml*, respectively[2]).

Moreover, a third metamodel has also been defined, the WSRP metamodel (see Figure A.3). It describes the set of WSRP portlets will be used in the new portal implementation, namely, the portlets which will implement the task collected in the portal design.

In order to get the XML and CSS files of the new *eXo* portal, i.e., its code, the transformation process includes two steps, the first one implements model-to-model transformations, specifically, from SOP-to-EXO, and the second one, model-to-code transformations. Figure A.4 depicts the overall process. It takes a SOP model and a WSRP model as input and gets five *eXo* files. The process to get the ***_config.xml*,

[2]The ** symbol stands for username, since there is a configuration file set for each registered user.

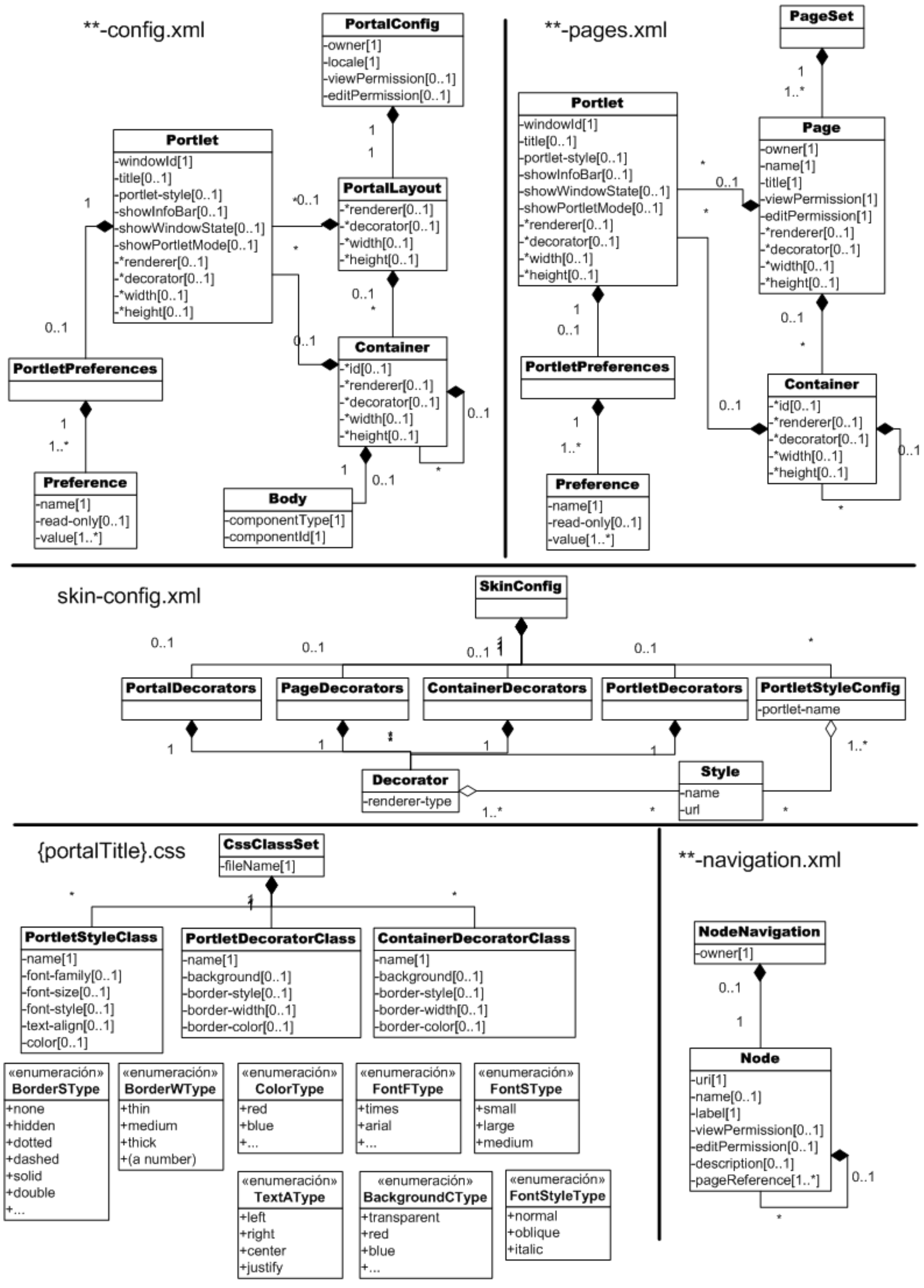


Figure A.2: The EXO metamodel.

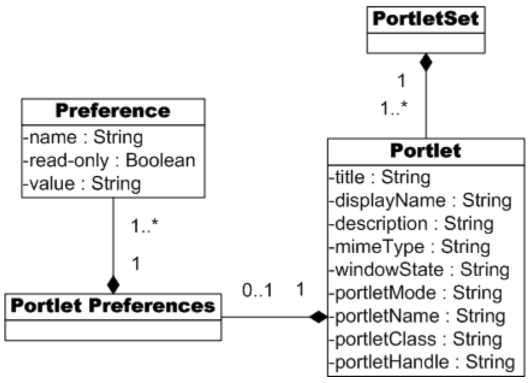


Figure A.3: The WSRP metamodel.

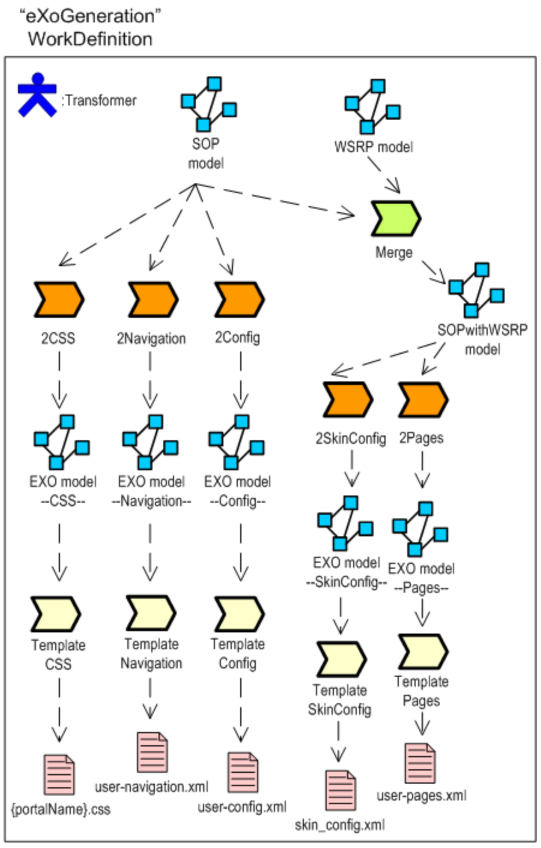


Figure A.4: Transformation process.

***_navigation.xml*, and *{portalName}.css* files is completely automatic and straight. In order to get the ***_pages.xml* and *skin_config.xml* files, first a merge process must be carried out to specify the relationship of each simple state in the ORCHESTRATION part with a portlet in the WSRP model, provided that they have equal names, then the rest of the process takes place as before.

Listing A.1: The *template-pages.2code* file.

```
main do
  compose_file 'pages.xml' do
   EXO::Pages::PageSet.all_objects do |obj|
          apply_template 'templates/template-pages.rtemplate', :obj => obj
   end
  end
end
```

Listing A.2: The *template-pages.rtemplate* file.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<page-set xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <%obj.pages.each do |page|%>
   <page renderer="<%=page.renderer%>" decorator="<%=page.decorator%>">
      <name><%=page.name%></name>
      <viewPermission><%=page.viewPermission%></viewPermission>
      ...
      <%if page.textPortlet %>
      <portlet renderer="<%=page.textPortlet.renderer%>"
               decorator="<%=page.textPortlet.decorator%>">
         <portlet-style><%=page.textPortlet.portletStyle%></portlet-style>
         <title><%=page.textPortlet.title%></title>
         ...
      </portlet>
      <%end

      container4states=page.container.collect
      container4transiciones=container4states.delete_at(0)

      if obj.layout == "top"
            container=container4transiciones
      %>
      <container renderer="<%=container.renderer%>"
                 decorator="<%=container.decorator%>"><%
         container.portlets.each do |p|
         %>
            <portlet renderer="<%=p.renderer%>"
                     decorator="<%=p.decorator%>">
               <portlet-style><%=p.portletStyle%></portlet-style><%
               if p.title != nil:%><title><%=p.title%></title><%end%>
               <windowId>@owner@:/<%=p.windowId%></windowId><%
               ...
            </portlet><%
         end #each
         %>
      </container><%
      end #if

      colaContainers = container4states
      colaContainers.reverse!
      while colaContainers.any?{|x| true}
         container = colaContainers.at(-1)
         colaContainers.delete_at(-1)

         if container != nil
```

```
          %>
            <container renderer="<%=container.renderer%>"
                      decorator="<%=container.decorator%>"><%
               container.portlets.each do |p|
                %>
                 <portlet renderer="<%=p.renderer%>"
                          decorator="<%=p.decorator%>">
                     <portlet-style><%=p.portletStyle%></portlet-style><%
                     <windowId>@owner@:/<%=p.windowId%></windowId><%
                     ...
                 </portlet><%
                end ### each

              colaContainers = colaContainers + [nil] +
                               container.subcontainers.collect.reverse
          %>
            </container><%
          end #if
      end ### while colaContainers.any?

      page.statePortlets.each do |p|
       %>
        <portlet renderer="<%=p.renderer%>" decorator="<%=p.decorator%>">
           <portlet-style><%=p.portletStyle%></portlet-style><%
           ...
           <windowId>@owner@:/<%=p.windowId%></windowId><%
             if p.preferences.any?
              %>
               <portlet-preferences><%
                 p.preferences.each{|l| l.preference.each{|pref| %>
                   <preference>
                     <name><%=pref.name%></name>
                     <value><%=pref.value%></value>
                   </preference><%
                 }}%>
               </portlet-preferences><%
             end%>
        </portlet><%
      end #each

      if obj.layout == "bottom"
           ...
      end #if
         ...
   %>
    </page>
   <%end%>
</page-set>
```

Following sections describe the model-to-model transformations thoroughly (i.e., the '2...' activities in Figure A.4). Each activity has been implemented in RubyTL as a set of transformation rules grouped in a modular element, named *phase* [98].

As for the model-to-code transformations (i.e., the *'Template...'* activities in Figure A.4), they are implemented through jsp-templates. Listing A.1 depicts the Ruby code which calls the template to be applied to `PageSet` element, from `EXO::Pages` package. An excerpt of that template is shown in Listing A.2. It builds the structure of the ***-pages.xml* file, and at the same time it scans the elements of the EXO model, specifically its `Pages` package, and when it is necessary it gets a value from the model (e.g., `page.title`, `page.textPortlet.renderer`, and so on).

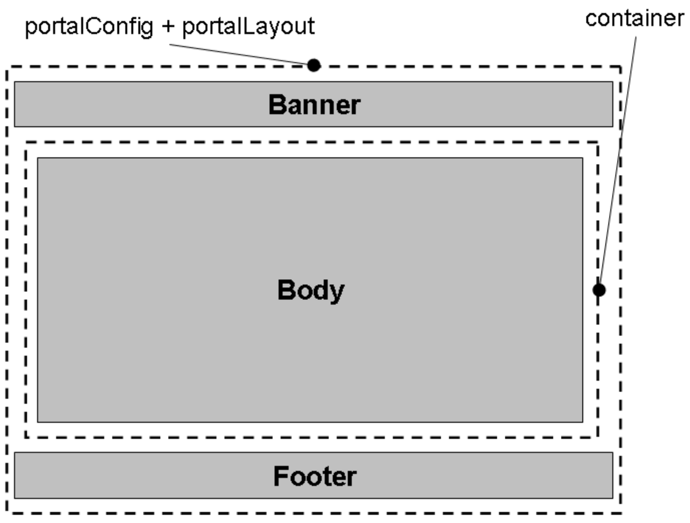


Figure A.5: A Portal layout.

A.2 Transformation for CONFIG package

Listing A.3 depicts the rule set which creates the CONFIG package of an EXO model, i.e., the part corresponding to the ***_config.xml* file. The set of portlets and containers composing the portal layout can have several combinations, the transformation in this prototype generates a concrete layout, shown in Figure A.5. This portal layout contains a banner and a footer, and, in the middle, a container including a body element (this element will contain the *eXo* pages, generated in a later transformation process). `Body`, `Container`, `PortalLayout` and so on are elements of the EXO metamodel (see Figure A.2). In *eXo*, banners and footers are shown using portlets.

Specifically, for a `StateMachine` in the `ORCHESTRATION` a `PortalConfig` element in the `EXO::Config` package is generated. It contains one single `Portal-Layout` object, which is generated by the `Rule4PortalLayout` rule (line 42). The structure of that object is completed firing another two rules `Rule4Portal-LayoutContainer` and `Rule4PortalLayoutPortlet`. The former (in line 95) generates the `Container` element for the `Body` element and the latter (in line 64) generates two `Portlet` objects, for the banner and the footer. In both cases this transformation uses the `DisplayStaticContent` portlet (in line 70), offered by *eXo* platform to show static content. The transformation could have been parametrized in order to be able to configure the use of different portlets. Each portlet has one `PortletPreferences` object, which is generated and completed by the `Rule4-`

`BannerFooterPreferences` rule (line 117).

The transformation has the following parameters: *view_permission*, *edit_permission*, and *locale*. Their values come from the `EXO__Config_parameters` global variable, defined when the transformation is launched. The `EXO__Config_default_parameters` local variable stores a map with their default values, i.e., *owner*, *owner* and *en*, respectively (line 8). These values are overridden when the first variable has its proper values (line 17). The `Rule4PortalConfig` rule will use those parameter values to complete the content of `PortalConfig` element.

Listing A.3: The *sop2config.rb* file.

```
1  use_library 'helper://state_machine'
2  use_library 'helper://presentation'

4  transformation 'SOP2Config'

6    #Transformation parameters

8    EXO__Config_default_values = { #
9      :view_permission => 'owner',
10     :edit_permission => 'owner',
11     :locale          => 'en'
12   }

14   #Take default values, if not given
15   $EXO__Config_parameters ||= EXO__Config_default_values
16   $EXO__Config_parameters =
17     EXO__Config_default_values.merge($EXO__Config_parameters) #

19   VIEW_PERMISSION = $EXO__Config_parameters[:view_permission]
20   EDIT_PERMISSION = $EXO__Config_parameters[:edit_permission]
21   LOCALE = $EXO__Config_parameters[:locale]


24 phase 'EXO__Config__PortalConfig__generation' do

26   # Each StateMachine is transformed into a PortalConfig object,
27   # containing the reference to a PortalLayout.
28   #
29   top_rule 'Rule4PortalConfig' do
30     from    SOP::Orchestration::StateMachine
31     to      EXO::Config::PortalConfig

33     mapping do |state_machine, portal_config|
34       portal_config.locale = LOCALE
35       portal_config.viewPermission = VIEW_PERMISSION
36       portal_config.editPermission = EDIT_PERMISSION
37       portal_config.layout = state_machine          #fires Rule4PortalLayout
38    end
39   end


42   rule 'Rule4PortalLayout' do #
43     from    SOP::Orchestration::StateMachine
44     to      EXO::Config::PortalLayout

46     mapping do |state_machine, portal_layout|
47       portal_layout.renderer = state_machine.workview.portalRenderer
48       portal_layout.decorator = state_machine.portal_title + "CDecorator"
49       portal_layout.width = "100%"
50       portal_layout.height = "100%"
```

```
      portal_layout.portlets    = #
        #fires Rule4PortalLayoutPortlets
        state_machine
      portal_layout.containers = portal_layout #fires Rule4PortalLayoutContainer
  end
end

# The PortalLayout contained in the PortalConfig object, contains two
# Portlet references, for banner and footer, respectively.
# The source must be StateMachine since it is needed to get
# some design values and to set the atributes
#
rule 'Rule4PortalLayoutPortlet' do #
  from     SOP::Orchestration::StateMachine
  to       EXO::Config::Portlet, EXO::Config::Portlet

  mapping do |state_machine, banner, footer|
    banner.showInfoBar = "true"
    banner.windowId = "@owner@:/content/DisplayStaticContent/banner" #

    footer.showInfoBar = "false"
    footer.windowId = "@owner@:/content/DisplayStaticContent/footer"

    banner.preferences = banner #fires Rule4BannerFooterPreferences #
    footer.preferences = footer #fires Rule4BannerFooterPreferences #

    #add another value to each Preference
    tmp_uri =
      "uri=war:/#{state_machine.portal_title}/static-content/" +
      "#{state_machine.workview.banner}"
    banner.preferences[0].preferences[0].values << #
      EXO::Config::Value.new(:value => tmp_uri)
    tmp_uri =
      "uri=war:/#{state_machine.portal_title}/static-content/" +
      "#{state_machine.workview.footer}"
    footer.preferences[0].preferences[0].values << #
      EXO::Config::Value.new(:value => tmp_uri)
  end
end

# The PortalLayout contained in the PortalConfig object,
# contains one Container object
#
rule 'Rule4PortalLayoutContainer' do #
  from     EXO::Config::PortalLayout
  to       EXO::Config::Container

  mapping do |state_machine, container|
    container.decorator = "default"
    container.body = container
  end
end

rule 'Rule4PortalLayoutContainerBody' do
  from     EXO::Config::Container
  to       EXO::Config::Body

  mapping do |container, body|
    body.componentType = "page-node"
    body.componentId = "/"
  end
end


#Banner, Footer Preferences
rule 'Rule4BannerFooterPreferences' do #
  from     EXO::Config::Portlet
  to       EXO::Config::PortletPreferences
```

```
    mapping do |portlet, portlet_preferences|
      portlet_preferences.preferences = portlet_preferences
   end
  end

  rule 'Rule4BannerFooterPreference' do
    from     EXO::Config::PortletPreferences
    to       EXO::Config::Preference

    mapping do |portlet_preferences, preference|
      preference.name = "default"
      preference.values = preference
   end
  end

  rule 'Rule4BannerFooterPreferenceValues' do #
    from     EXO::Config::Preference
    to       EXO::Config::Value, EXO::Config::Value

    mapping do |preference, a, b|
      a.value = "title=Default␣Content"
      b.value = "encoding=UTF-8"
    end
  end
end
```

A.3 Transformation for CSS package

The portal look-and-feel is described through several CSS classes that can be defined along one or more CSS files. This approach has decided to collect all CSS classes in one file, represented by `CssClassSet` metaclass in the EXO metamodel. That file will have the same name as the Web portal, decided at design level, in the SOP model. Listing A.4 depicts the rule set which creates the CSS package of an EXO model, which corresponds to the CSS file. Before giving more details about the rules, we will explain the used naming convention and some features of *eXo*, implicit in the transformation.

In *eXo* portals almost everything is implemented with portlets. As said in the previous section, portal banner and footer are shown through a portlet, moreover another portlet, named *navigationStep*, implements the transitions among portal pages and a third portlet, named *showText*, shows the text that at design level we called 'helping text'.

Otherwise, *eXo* platform has four UI component types, namely, *portal*, *page*, *container* and *portlet*. Each component has CSS classes to describe its presentation. As shown in the EXO metamodel, those classes are classified in three groups: decorator classes for containers, and portal and pages, as special containers (described through the `ContainerDecoratorClass` metaclass), decorator classes for portlets (described with the `PortletDecoratorClass` metaclass) and font-style classes for portlet content (described with the `PortletStyleClass` metaclass). Each page and each container can have its own decorator class, but in the prototype presented in this thesis only one class is generated in each case (i.e., every page, and every container, has the same presentation). Both decorator classes have the portal name as a prefix in their names, and then, as suffix, `TransparentDecorator` in the container decorator[3] and `PageDecorator` in the page decorator.

As for portlets, the naming convention is to put `Decorator` as suffix for decorator classes and `Style` for font-style classes. Moreover, those class names will have the portal name, the portlet name or the transition name as prefix, depending on the specific portlet:

- The *showText* portlet will have only one presentation throughout the portal, so only one decorator class and one font-style class are generated. Therefore, the portal name is used as prefix. For sake of legibility, moreover `TextP` is added in the middle of the name.
- The *navigationStep* portlet is used for transitions, and its presentation can be different each time, depending on the design. Therefore, one decorator class (and one font-style class) must be generated for each transition. The transition name is used as prefix to distinguish each class name. Moreover, `Anchor` is added to indicate that those are classes for transitions, or anchors in implementation-wise.
- Functionality tasks are implemented through several portlets, each one with its presentation requirements depending on the design. Therefore, one decorator class (and one font-style class) is generated for each one and the portlet name is used as prefix (`P` is added in the middle to distinguish from previous classes).

Table A.1 sums up what has been described.

Transformation in Listing A.4 generates a `CssClassSet` element in the `EXO:-:CSS` package for a `StateMachine` in the `ORCHESTRATION` of a `SOP` model. Transformation starts with the `Rule4CssClassSet` top rule, in line 10; this rule also fires the generation of the elements contained in the `CssClassSet`. Specifically, it fires the `Rule4Container` rule (in line 34) to generate the container

[3]In this approach containers are used to structure the composition of portlets, side-by-side or one below the other. They do not have specific presentation features, therefore they are no visible and we give them a transparent decorator (with no color in the background, no borderline, and so on). That is why we use `TransparentDecorator` as the name of the class.

	Decorator	**Style**
Page	*portalName*`PageDecorator`	—
Container	*portalName*`TransparentDecorator`	—
Portlet (tasks)	*portletName*`PDecorator`	*portletName*`PStyle`
Portlet (anchors)	*transitionName*`AnchorDecorator`	*transitionName*`AnchorStyle`
Portlet (helpingText)	*portalName*`TextPDecorator`	*portalName*`TextPStyle`

Table A.1: Naming convention for CSS classes of UI components.

decorators (as said before, their names are `{portalName}PageDecorator` and `{portalName}TransparentDecorator`) and then the rules to generate decorator and font-style classes for different portlets, namely,

- The `Rule4TextDecorator` and `Rule4TextStyle` rules generate two CSS classes named `{portalName}TextPDecorator` and `{portalName}-TextPStyle`, respectively. They are the classes for the *showText* portlet.
- The `SimpleState2Decorator` and `SimpleState2Style` rules (in lines 98 and 115, respectively) are fired once for each simple state (since they are related to portal tasks, i.e., portlets).
- The `Transition2Decorator` and `Transition2Style` rules (in lines 133 and 149, respectively) are fired once for each transition. All of them will be used with the *navigationStep* portlet. As said before, this portlet implements transitions among portal pages and in the Web portal is rendered as an anchor (using the hypermedia terminology). Since presentation features of each anchor can vary depending on its design, more specifically, on the `AnchorDescriptor` values, a class pair is generated for each transition/anchor, in other words, for each use of the *navigationStep* portlet.

The values for attributes in the `{portalName}PageDecorator` class are copied from the `WorkviewDescriptor` element (in the `RENDERING` package) associated with the `StateMachine`. Attribute values of portlet classes come from the `HelpingTextDescriptor`, `WindowDescriptor`, and `AnchorDescriptor` elements (in the `RENDERING` package) which are associated with `stateMachine`, `state` and `transition`, respectively. Given the hierarchical definition of statecharts, some states or transitions may not have a rendering descriptor associated or they may be lacking in some attributes. In such cases, they inherit rendering values from descriptors associated with their ancestors. This inheritance is achieved through some auxiliary functions, such as the `window_descriptor` and `anchor_descriptor` functions, used in the `SimpleState2Style` and `Transition2Style` rules (in lines 115 and 149, respectively), for example.

Listing A.4: The *sop2css.rb* file.

```
1  use_library 'helper://state_machine'

3  transformation 'SOP2CSS'

5  phase 'sop2CSS' do

7    # Each StateMachine is transformed into a CssClassSet, that
8    # contains references to CSS classes.
9    #
10   top_rule 'Rule4CssClassSet' do #
11     from    SOP::Orchestration::StateMachine
12     to      EXO::CSS::CssClassSet

14     mapping do |state_machine, cl_set|
15       cl_set.name       = state_machine.portal_title
16       cl_set.containers = state_machine       #fires Rule4Container
17       cl_set.decorators =
18         #fires rules
19         # Rule4TextDecorator SimpleState2Decorator Transition2Decorator
20         [state_machine] +
21         state_machine.simple_states + state_machine.all_transitions
22       cl_set.portlets   =
23         #fires rules
24         #    Rule4TextStyle SimpleState2Style Transition2Style
25         [state_machine] +
26         state_machine.simple_states + state_machine.all_transitions
27     end
28   end

30   # Each StateMachine generates two ContainerDecoratorClass objects named:
31   #       portal_title+PageDecorator
32   #       portal_title+TransparentDecorator
33   #
34   rule 'Rule4Container' do #
35     from    SOP::Orchestration::StateMachine
36     to      EXO::CSS::ContainerDecoratorClass,
37             EXO::CSS::ContainerDecoratorClass

39     mapping do |state_machine, page, transparent|
40       page.name        =   #
41         state_machine.portal_title + "PageDecorator"
42       page.borderStyle = state_machine.workview.borderStyle
43       page.borderWidth = state_machine.workview.borderWidth
44       page.background  = state_machine.workview.background
45       page.borderColor = state_machine.workview.borderColor

47       transparent.name       = #
48         state_machine.portal_title + "TransparentDecorator"
49       transparent.borderStyle= "solid"
50       transparent.borderWidth= "4px"
51       transparent.background = "transparent"
52       transparent.borderColor= "white"
53     end
54   end


57   # Each StateMachine generates one PortletDecoratorClass object named
58   #       portal_title+TextPDecorator
59   #
60   rule 'Rule4TextDecorator' do  #
```

```
  from     SOP::Orchestration::StateMachine
  to       EXO::CSS::PortletDecoratorClass

  mapping do |state_machine, textP|
   textP.name         = #
      state_machine.portal_title + "TextPDecorator"
    textP.background  = state_machine.text_descriptor.background
    textP.borderStyle = state_machine.text_descriptor.borderStyle
    textP.borderWidth = state_machine.text_descriptor.borderWidth
    textP.borderColor = state_machine.text_descriptor.borderColor
  end
end

# Each StateMachine generates one PortletStyleClass object named
#       portal_title+TextPStyle
#
rule 'Rule4TextStyle' do #
  from     SOP::Orchestration::StateMachine
  to       EXO::CSS::PortletStyleClass

  mapping do |state_machine, clTextPStyle|
    clPStyle.name        = #
      state_machine.portal_title + "PStyle"
    clTextPStyle.name        = #
      state_machine.portal_title + "TextPStyle"
    clTextPStyle.fontFamily = state_machine.text_descriptor.fontFamily
    clTextPStyle.fontSize   = state_machine.text_descriptor.fontSize
    clTextPStyle.fontStyle  = state_machine.text_descriptor.fontStyle
    clTextPStyle.color      = state_machine.text_descriptor.color
    clTextPStyle.textAlign  = state_machine.text_descriptor.textAlign
 end
end


# Each SimpleState generates a PortletDecoratorClass object named
#       state_name+PDecorator
#
rule 'SimpleState2Decorator' do  #
  from     SOP::Orchestration::State
  to       EXO::CSS::PortletDecoratorClass
  filter do |state| state.isSimple end

  mapping do |state, decorator|
    decorator.name        = state.name + "PDecorator" #
    decorator.background  = state.window_descriptor.background
    decorator.borderStyle = state.window_descriptor.borderStyle
    decorator.borderWidth = state.window_descriptor.borderWidth
    decorator.borderColor = state.window_descriptor.borderColor
  end
end

# Each SimpleState generates a PortletStyleClass object named
#       state_name+PStyle
#
rule 'SimpleState2Style' do #
  from     SOP::Orchestration::State
  to       EXO::CSS::PortletStyleClass
  filter do |state| state.isSimple end

  mapping do |state, stylePortlet|
    stylePortlet.name       = state.name + "PStyle" #
    stylePortlet.fontFamily = state.window_descriptor.fontFamily
    stylePortlet.fontSize   = state.window_descriptor.fontSize
    stylePortlet.fontStyle  = state.window_descriptor.fontStyle
    stylePortlet.textAlign  = state.window_descriptor.textAlign
    stylePortlet.color      = state.window_descriptor.color
  end
end
```

```
  # Each transition generates a PortletDecoratorClass object named
  #       transition_name+AnchorDecorator
  #
  rule 'Transition2Decorator' do #
    from    SOP::Orchestration::Transition
    to      EXO::CSS::PortletDecoratorClass

    mapping do |transition, decorator|
      decorator.name        = transition.name + "AnchorDecorator"#
      decorator.background  = transition.anchor_descriptor.background
      decorator.borderStyle = transition.anchor_descriptor.borderStyle
      decorator.borderWidth = transition.anchor_descriptor.borderWidth
      decorator.borderColor = transition.anchor_descriptor.borderColor
    end
  end

  # Each transition generates a PortletDecoratorClass object named
  #       transition_name+AnchorStyle
  #
  rule 'Transition2Style' do #
    from    SOP::Orchestration::Transition
    to      EXO::CSS::PortletStyleClass

    mapping do |transition, portlet|
      portlet.name       = transition.name + "AnchorStyle" #
      portlet.fontFamily = transition.anchor_descriptor.fontFamily
      portlet.fontSize   = transition.anchor_descriptor.fontSize
      portlet.fontStyle  = transition.anchor_descriptor.fontStyle
      portlet.textAlign  = transition.anchor_descriptor.textAlign
      portlet.color      = transition.anchor_descriptor.color
    end
  end
end
```

A.4 Transformation for SKIN_CONFIG package

The *skin_config.xml* configuration file in *eXo* platform constitutes an index of all the CSS class names available for the UI elements rendered in the portal. In other words, it collects all the CSS classes (their names) generated with the transformation shown in Section A.3. These classes are classified according with the UI element which they characterize. As shown in the EXO metamodel (see Figure A.2), there are five main groups: the group of style classes for portlets and the groups of UI element decorators, i.e., portal decorators, page decorators, container decorators and portlet decorators. Each group constitutes a list with the names of CSS classes and the paths of the files which contain them. Figure A.5 depicts an excerpt of a *skin_config.xml* file. The *Browsing.css* file stores all the CSS classes for the portal elements, except the default CSS classes, contained in *default-xxx.css* files and offered by the *eXo* platform by default. In this example *Browsing* is the name of the Web portal.

UI element decorators can be classified by the renderer-type. The *eXo* platform also offers some by default, like, *PortalRenderer*, *PageRowRenderer*, *PageColumn-Renderer*, *PortletRenderer*, and so on. For example, using a *PageRowRenderer* type decorator for pages suggests that elements inside a page will be located one below the other, in rows. The prototype presented in this thesis uses those basic renderer types.

Listing A.5: Snippet of a *skin_config.xml* file.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<skin-config>
  <portal-decorators>
    <decorator>
      <renderer-type>PortalRenderer</renderer-type>
      <style name="default" url="/Browsing/skin/portal/default-portal.css" />
    </decorator>
  </portal-decorators>
  <page-decorators>
    <decorator>
      <renderer-type>PageRowRenderer</renderer-type>
      <style name="default" url="/Browsing/skin/page/default-page.css" />
      <style name="BrowsingPageDecorator" url="/Browsing/skin/Browsing.css" />
    </decorator>
    <decorator>
     <renderer-type>PageColumnRenderer</renderer-type>
      ...
   </decorator>
  </page-decorators>
  <container-decorators>
    <decorator>
      <renderer-type>ContainerColumnRenderer</renderer-type>
      <style name="default" url="/Browsing/skin/container/default-container.css" />
       <style name="BrowsingTransparentDecorator" url="/Browsing/skin/Browsing.css" />
    </decorator>
    <decorator>
      <renderer-type>ContainerRowRenderer</renderer-type>
       ...
    </decorator>
  </container-decorators>
  <portlet-decorators>
    <decorator>
      <renderer-type>PortletRenderer</renderer-type>
  <style name="default" url="/Browsing/skin/portlet/decorators/default-decorator.css" />
      <style name="BrowsingTextPDecorator" url="/Browsing/skin/Browsing.css" />
      <style name="ieeeSearchPDecorator" url="/Browsing/skin/Browsing.css" />
      <style name="acmSearchPDecorator" url="/Browsing/skin/Browsing.css" />
      <style name="ToAuthorSearchAnchorDecorator" url="/Browsing/skin/Browsing.css" />
      <style name="ToPaperSearchAnchorDecorator" url="/Browsing/skin/Browsing.css" />
     ...
    </decorator>
  </portlet-decorators>
  <portlet-style-config>
    <portlet-name>default</portlet-name>
  <style name="default" url="/Browsing/skin/portlet/styles/default-portlet.css" />
  </portlet-style-config>
  <portlet-style-config>
    <portlet-name>showText/showText</portlet-name>
    <style name="BrowsingTextPStyle" url="/Browsing/skin/Browsing.css" />
  </portlet-style-config>
  <portlet-style-config>
    <portlet-name>ieeeLibrary/ieeeLibrary</portlet-name>
    <style name="ieeeSearchPStyle" url="/Browsing/skin/Browsing.css" />
  </portlet-style-config>
  <portlet-style-config>
    <portlet-name>navigationstep/step</portlet-name>
    <style name="ToAuthorSearchAnchorStyle" url="/Browsing/skin/Browsing.css" />
  </portlet-style-config>
  ...
</skin-config>
```

Listing A.6 shows the rule set which creates the SKINCONFIG package of an EXO model, i.e., the part corresponding to the *skin_config.xml* file. Specifically, for a `StateMachine` in the `ORCHESTRATION` the top rule generates a `SkinConfig` element in the `EXO::Skin_Config` package. The only data needed for the generation are the names of simple states and transitions, and the portal name (achieved through the relation between the statemachine and the workspace).

The generated `SkinConfig` element is made up of one `PortalDecorators` object, one `PageDecorators` object, one `ContainerDecorators` object, one `PortletDecorators` object and several `PortletStyleConfig` objects. More specifically:

- The `PortalDecorators` object contains one `Decorator` element, generated by the `PortalDecoratorsGeneration` rule (in line 190). The `Style` element contained in the `Decorator` object is created using an explicit assignment in line 199.

- The `PageDecorators` object contains two `Decorator` elements, generated by the `PageDecoratorsGeneration` rule (in line 210), and their renderer-types are *PageRowRenderer* and *PageColumnRenderer*, respectively. Both elements contain two `Style` elements, whose names are `default` and `{portalName}PageDecorator`.

- The `ContainerDecorators` object contains two `Decorator` elements, generated by the `ContainerDecoratorsGeneration` rule (in line 248), and their renderer-types are *ContainerRowRenderer* and *ContainerColumn-Renderer*, respectively. Both elements contain two `Style` elements, whose names are `default` and `{portalName}TransparentDecorator`.

- The `PortletDecorators` object contains one `Decorator` element whose renderer-type is *PortletRenderer* and is generated by the `PortletDecorators-Generation` rule (in line 289). This elements have several `Style` elements, `default` and `{portalName}TextPDecorator`, generated by the `PortletDecoratorStyles` rule (in line 307), and one element for every simple state and every transition in the `StateMachine`. State styles are generated by the `State2PortletDecoratorStyle` rule in line 329 and transition styles are generated by the `Transition2PortletDecorator-Style` rule in line 342.

- The `PortletStyleConfig` objects correspond to the *showText* portlet, the *navigationStep* portlet, and the portlets for tasks. The `PortletStyle-Configs` and `PortletStyleConfigs__HelpingText` rules (in lines 94 and 109, respectively) create the `default` and `{portalName}TextPStyle` styles. They have a filter, thus the triggering of one of them depends on whether the portal design includes or not a 'helpig text'. Otherwise, the style for every simple state in the statemachine is generated by the `PortletStyle-Config4State` rule in line 138 and style for every transition by the `Portlet-StyleConfig4Transition` rule in line 164.

Listing A.6: The *sop2skinConfig.rb* file.

```
1  use_library 'helper://state_machine'
2  use_library 'helper://presentation'

4  transformation 'SOP2SkinConfig'
5  #   A StateMachine maps to a SkinConfig made up of several
6  #   PortalDecorator, PageDecorator, ContainerDecorator
7  #   objects wich are aggregated in corresponding containers:
8  #   PortalDecorators, PageDecorators, ContainerDecorators
9  #   That SkinConfig object also contains PortletStyleConfig objects.
10 #

12 phase 'EXO__Skin_config__SkinConfig' do #

14   #Some rules other than the top rule
15   #need access to the StateMachine being mapped.
16   #This local variable is set when the first rule is started.
17   state_machine__EXO__Skin_Config = nil

19   #StateMachine maps to PortletStyleConfig objects and several
20   #      PortalDecorator, PageDecorator, ContainerDecorator
21   #wich are aggregated in corresponding containers.
22   #This rule generates the aggregating objects
23   #and the PortletStyleConfig objects
24   #
25   top_rule 'SkinConfig' do
26     from     SOP::Orchestration::StateMachine
27     to       EXO::Skin_config::SkinConfig

29     mapping do |state_machine, skin_config|
30       state_machine__EXO__Skin_Config = state_machine

32       skin_config.portalDecorators = state_machine     #fires PortalDecorators #
33       skin_config.pageDecorators = state_machine       #fires PageDecorators #
34       skin_config.containerDecorators = state_machine #fires ContainerDecorators #
35       skin_config.portletDecorators = state_machine   #fires PortletDecorators #

37       skin_config.portletStyleConfigs = #
38         [state_machine]+               #fires either:
39                                        #      PortletStyleConfigs
40                                        #      PortletStyleConfigs__HelpingText
41         state_machine.simple_states+  #fires PortletStyleConfig4State
42         state_machine.all_transitions #fires PortletStyleConfig4Transition
43     end
44   end

46   # trivial rules generating
47   #      PortalDecorator, PageDecorator, ContainerDecorator
```

```
rule 'PortalDecorators' do
  from    SOP::Orchestration::StateMachine
  to      EXO::Skin_config::PortalDecorators

  mapping do |state_machine, portal_decorators|
    portal_decorators.decorators = #fires PortalDecoratorsGeneration
      portal_decorators
  end
end

rule 'PageDecorators' do
  from    SOP::Orchestration::StateMachine
  to      EXO::Skin_config::PageDecorators

  mapping do |state_machine, page_decorators|
    page_decorators.decorators = page_decorators
  end
end

rule 'ContainerDecorators' do
  from    SOP::Orchestration::StateMachine
  to      EXO::Skin_config::ContainerDecorators

  mapping do |state_machine, container_decorators|
    container_decorators.decorators = container_decorators
  end
end

rule 'PortletDecorators' do
  from    SOP::Orchestration::StateMachine
  to      EXO::Skin_config::PortletDecorators

  mapping do |state_machine, portlet_decorators|
    portlet_decorators.decorators = portlet_decorators
  end
end


#Rules generating PortletStyleConfig objects.
#
#The StateMachine generates one or two PortletStyleConfig objects
#depending on the existence of a HelpingText.
#The following two rules have filters which are
#mutually exclusive, so at most one of them will be fired

rule 'PortletStyleConfigs' do #
  from    SOP::Orchestration::StateMachine
  to      EXO::Skin_config::PortletStyleConfig
  filter do |state_machine| !state_machine.helping_text end

  mapping do |state_machine, default|
    default.portletName = "default"
    tmp_url =
      "/#{state_machine.portal_title}"+
      "/skin/portlet/styles/default-portlet.css"
    default.styles = EXO::Skin_config::Style.new(:name => "default",
                                                 :url => tmp_url)
  end
end

rule 'PortletStyleConfigs__HelpingText' do #
  from    SOP::Orchestration::StateMachine
  to      EXO::Skin_config::PortletStyleConfig,
          EXO::Skin_config::PortletStyleConfig
  filter do |state_machine| state_machine.helping_text end

  mapping do |state_machine, default, helping_text|
    default.portletName = "default"
```

```
      tmp_url =
        "/#{state_machine.portal_title}"+
        "/skin/portlet/styles/default-portlet.css"
      default.styles = EXO::Skin_config::Style.new(:name => "default",
                                                   :url => tmp_url)
      tmp_portlet = state_machine.helping_text.portlet
      helping_text.portletName = tmp_portlet.displayName + "/" + tmp_portlet.portletName
      tmp_name = "#{state_machine.portal_title}TextPStyle"
      tmp_url =
        "/#{state_machine.portal_title}/skin/"+
        "#{state_machine.portal_title}.css"
      helping_text.styles = EXO::Skin_config::Style.new(:name => tmp__name,
                                                        :url => tmp_url)
    end
  end


  #Every simple state generates a PortletStyleConfig,
  #which is contained in the SkinConfig object
  #(see top rule)
  #
  rule 'PortletStyleConfig4State' do #
    from     SOP::Orchestration::State
    to       EXO::Skin_config::PortletStyleConfig
    filter   do |s| s.isSimple end

    mapping do |state, portlet_style_config|
     portlet_style_config.portletName =
        state.portlet.displayName + "/" + state.portlet.portletName

     #Doing this with another rule would eventually lead to
     #several rules with the same source and target types: State and Style
     #Moreover, it is an almost trivial generation
     tmp_name = state.name + "PStyle"
     tmp_url =
        "/#{state.state_machine.portal_title}/skin/"+
        "#{state.state_machine.portal_title}.css"
     portlet_style_config.styles << EXO::Skin_config::Style.new(:name => tmp_name,
                                                               :url => tmp_url)
    end
  end


  #Every transition generates a PortletStyleConfig,
  #which is contained in the SkinConfig object
  #(see top rule)
  #
  rule 'PortletStyleConfig4Transition' do #
    from     SOP::Orchestration::Transition
    to       EXO::Skin_config::PortletStyleConfig

    mapping do |tr, portlet_style_config|
      portlet_style_config.portletName =
        tr.portlet.displayName + "/" + tr.portlet.portletName

      #The same as above:
      #we don't want to end up with several rules
      #with the same source and target types (Transition and Style)
      tmp_name = tr.name + "AnchorStyle"
      tmp_url =
        "/#{tr.state_machine.portal_title}/skin/"+
        "#{tr.state_machine.portal_title}.css"
      portlet_style_config.styles <<
        EXO::Skin_config::Style.new(:name => tmp_name,
                                    :url => tmp_url)
    end
  end
```

```
#The PortalDecorators object, whose generation was fired
#in the top rule, contains the reference of a Decorator object
#generated by this rule
#
rule 'PortalDecoratorsGeneration' do #
  from    EXO::Skin_config::PortalDecorators
  to      EXO::Skin_config::Decorator

  mapping do |portal_decorators, decorator|
    decorator.rendererType = "PortalRenderer"
    tmp_url =
      "/#{state_machine__EXO__Skin_Config.portal_title}" +
      "/skin/portal/default-portal.css"
    decorator.styles <<
      EXO::Skin_config::Style.new(:name => "default",
                                  :url => tmp_url)
  end
end


#The PageDecorators object whose generation was fired
#in the top rule contains the reference of two Decorator objects
#generated by this rule
#
rule 'PageDecoratorsGeneration' do #
  from    EXO::Skin_config::PageDecorators
  to      EXO::Skin_config::Decorator, EXO::Skin_config::Decorator

  mapping do |page_decorators, decorator_a, decorator_b|
    decorator_a.rendererType = "PageRowRenderer"
    decorator_a.styles       = page_decorators
    decorator_b.rendererType = "PageColumnRenderer"
    decorator_b.styles       = page_decorators
  end
end

copy_rule 'PageDecoratorStyles' do
  from    EXO::Skin_config::PageDecorators    #the Syles generated are contained in
                                              #a Decorator object but
                                              #the source must be PageDecorators
                                              #to prevent the definition of several rules
                                              #with the same source and target types
  to      EXO::Skin_config::Style, EXO::Skin_config::Style

  mapping do |page_decorators, style_a, style_b|
    style_a.name = "default"
    style_a.url  =
      "/#{state_machine__EXO__Skin_Config.portal_title}"+
      "/skin/page/default-page.css"
    style_b.name =
      state_machine__EXO__Skin_Config.portal_title + "PageDecorator"
    style_b.url  =
      "/#{state_machine__EXO__Skin_Config.portal_title}/skin/"+
      "#{state_machine__EXO__Skin_Config.portal_title}.css"
  end
end


#The ContainerDecorators object whose generation was fired
#in the top rule contains the reference of two Decorator objects
#generated by this rule
#
rule 'ContainerDecoratorsGeneration' do #
  from    EXO::Skin_config::ContainerDecorators
  to      EXO::Skin_config::Decorator, EXO::Skin_config::Decorator

  mapping do |container_decorators, decorator_a, decorator_b|
    decorator_a.rendererType = "ContainerRowRenderer"
    decorator_b.rendererType = "ContainerColumnRenderer"
    decorator_a.styles      = container_decorators #fires ContainerDecoratorsStyle
    decorator_b.styles      = container_decorators #fires ContainerDecoratorsStyle
  end
end

#Although the Syles are contained in a Decorator object,
#the source must be ContainerDecorators,
#so preventing the definition of several rules
#with the same source and target types
#    Decorator -> Style
#
copy_rule 'ContainerDecoratorsStyle' do
  from    EXO::Skin_config::ContainerDecorators
  to      EXO::Skin_config::Style, EXO::Skin_config::Style

  mapping do |decorator, style_d, style_a|
    style_d.name = "default"
    style_d.url  =
      "/#{state_machine__EXO__Skin_Config.portal_title}"+
      "/skin/container/default-container.css"

    style_a.name =
      state_machine__EXO__Skin_Config.portal_title + "TransparentDecorator"
    style_a.url  =
      "/#{state_machine__EXO__Skin_Config.portal_title}"+
      "/skin/#{state_machine__EXO__Skin_Config.portal_title}.css"
  end
end


#The PortletDecorators object whose generation was fired
#in the top rule contains the reference of a Decorator object
#generated by this rule
#
rule 'PortletDecoratorsGeneration' do #
  from    EXO::Skin_config::PortletDecorators
  to      EXO::Skin_config::Decorator

  mapping do |portlet_decorators, decorator|
    decorator.rendererType = "PortletRenderer"
    decorator.styles      = portlet_decorators  #fires PortletDecoratorStyles #

    # State2PortletDecoratorStyle and Transition2PortletDecoratorStyle rules
    # are explicitly fired
    state_machine__EXO__Skin_Config.simple_states.each{|st|
      decorator.styles << State2PortletDecoratorStyle(st)}

    state_machine__EXO__Skin_Config.all_transitions.each{|tr|
      decorator.styles << Transition2PortletDecoratorStyle(tr)}
  end
end

rule 'PortletDecoratorStyles' do #
  from    EXO::Skin_config::PortletDecorators
  to      EXO::Skin_config::Style, EXO::Skin_config::Style

  mapping do |portlet_decorators, style_a, style_b|
    style_a.name = "default"
    style_a.url =
      "/#{state_machine__EXO__Skin_Config.portal_title}" +
      "/skin/portlet/decorators/default-decorator.css"

    style_b.name =
      state_machine__EXO__Skin_Config.portal_title + "TextPDecorator"
    style_b.url =
      "/#{state_machine__EXO__Skin_Config.portal_title}" +
      "/skin/#{state_machine__EXO__Skin_Config.portal_title}.css"
  end
end
```

Orchestration + Rendering elements	*eXo* elements
state configuration	page
simple state	portlet
orthogonal state (and)	container
composite state (or)	—
transition	portlet
helping text	portlet

Table A.2: Relation among design elements and *eXo* platform elements.

```
  #Each state generates a Style, whose reference
  #is included in a Decorator (see PortletDecorators, above)
  #
  rule 'State2PortletDecoratorStyle' do #
    from    SOP::Orchestration::State
    to      EXO::Skin_config::Style
    filter do |state| state.isSimple end

    mapping do |state, style|
      style.name = state.name + "PDecorator"
      style.url =
        "/#{state.state_machine.portal_title}/skin/"+
        "#{state.state_machine.portal_title}.css"
    end
  end

  rule 'Transition2PortletDecoratorStyle' do #
    from    SOP::Orchestration::Transition
    to      EXO::Skin_config::Style
    filter do |tr| tr.source.is_state? end

    mapping do |tr, style|
      style.name = tr.name + "AnchorDecorator"
      style.url =
        "/#{tr.state_machine.portal_title}/skin/"+
        "#{tr.state_machine.portal_title}.css"
    end
  end

end
```

A.5 Transformation for PAGES package

In *eXo* platform a Web portal, apart from header and footer elements, is composed of a set of pages. *eXo* page is the UI element shown in the Body container (see Figure A.5). As the EXO metamodel depicts (see Figure A.2), a page is composed of several portlets and containers. Containers structure the presentation of portlets, i.e., they can be used, for example, to put two portlets one next to the other, and a third one below. On the other hand, the approach of this dissertation work is to design the Web portals, i.e., to design the organization of its pages, using statecharts. In statecharts, A *state*

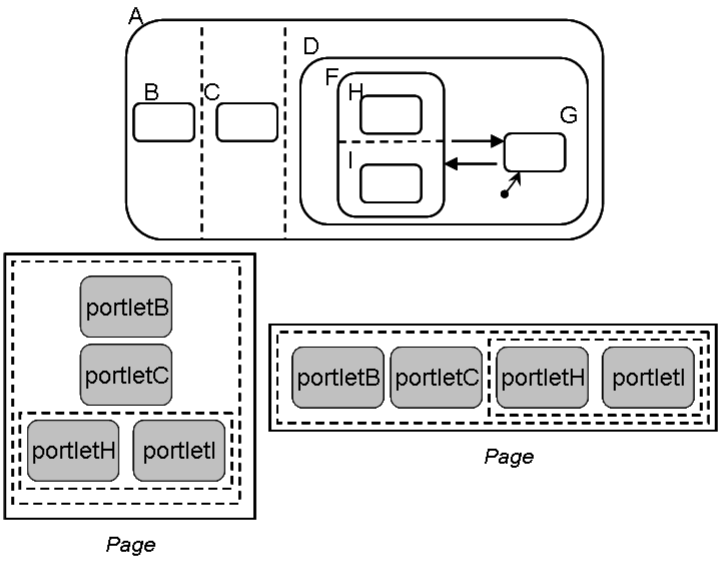


Figure A.6: A statechart example and two alternative pages corresponding to its {A, B, C, D, F, H, I} state configuration.

configuration is the set of states active at a given time [39]. In a composite state (OR-type state) its substates are active one at a time, instead when an orthogonal state (AND-type state) is active, every substate is also active. The approach presented in this thesis uses the state configuration concept to define an *eXo* page. The relation among design elements (specifically, orchestration and rendering elements) and *eXo* elements is summarized in Table A.2. For the aim of this approach, i.e., for the structure of the corresponding page, three distinct types of state configurations are distinguished:

- **{SimpleState}**$^+$. According to the statechart metamodel, a statemachine can be composed of orthogonal regions which contain simple states, therefore, in that case there is an only state configuration, and it is a set of those simple states. At *eXo* level that state configuration corresponds to a page with a set of portlets, without any other container.

- **{ORstate}*****{SimpleState}**. The state configuration is composed of several nested OR states, and in the end one simple state (only one, because in OR states only one of its substates is active at a time). That state configuration is transformed into a page which contains an only portlet related to that simple state, and without any other container.

- **{ORstate, ANDstate, SimpleState}*****{SimpleState}**$^+$. The most general case, when the state configuration is composed of several nested states, and in the end some simple states (there can be more than one, because of the active sub-states of an AND state). Figure A.6 depicts a statechart example where its {A, **B**, **C**, D, F, **H**, **I**} state configuration fits this expression. Its corresponding page is composed of portlets, besides two containers. Their aim is to structure the presentation of portlets, in rows or columns. Figure A.6 also depicts two alternative pages related to that state configuration. The dotted squares correspond to the containers, and they are no visible for the user.

As Figure A.1 depicts, `StateConfiguration` in `ORCHESTRATION` package is a derived metaclass. For each `SOP` model its values will be worked out and then the transformation (see Listing A.7) will use those state configurations to generate *eXo* pages.

Although the main element to build the page set of an *eXo* portal is the statechart, some attributes of the `WorkviewDescriptor` (in the `RENDERING` package), related to the `StateMachine` element, are also needed to make some layout decisions. Using the hypermedia terminology anchors are the links, i.e., the elements to navigate among pages. Moreover portlets, containers and anchors are arranged along a table-like structure. Attributes in `WorkviewDescriptor`, namely, *transition*, *distribution*, *position*, and *alignment*, specify different alternatives for the arrangement among them: ***transition*** indicates how anchors are realized (the value options include *button* or *helping text* where the transition is achieved by clicking on the underlined text); ***distribution***, indicates how to locate anchors along the portal page, and options include *together* (i.e. anchors are all located together, regardless of their orchestration counterparts, i.e. transitions) and *detached* (i.e. anchor *A* is located beside window[4] *W* if *A* stands for a transition that leaves from the state whose counterpart is *W*); ***position***, indicates whether anchors are placed at the *top*, *bottom*, *left* or *right* of either the page (when `distribution`=together) or the associated window (when `distribution`=detached); ***alignment***, indicates how windows are rendered together (values are *column*, i.e one below the other, and *row*, i.e. one by the other).

Depending on those attribute values, several design combinations can be defined and the transformation rules will be different in each case. The transformation prototype presented in this appendix takes into account the combination where *transition=anchor*, *distribution=together*, and *position=top*. Therefore, all anchors will

[4]Window is the rendering counterpart of orchestration state and implementation portlet.

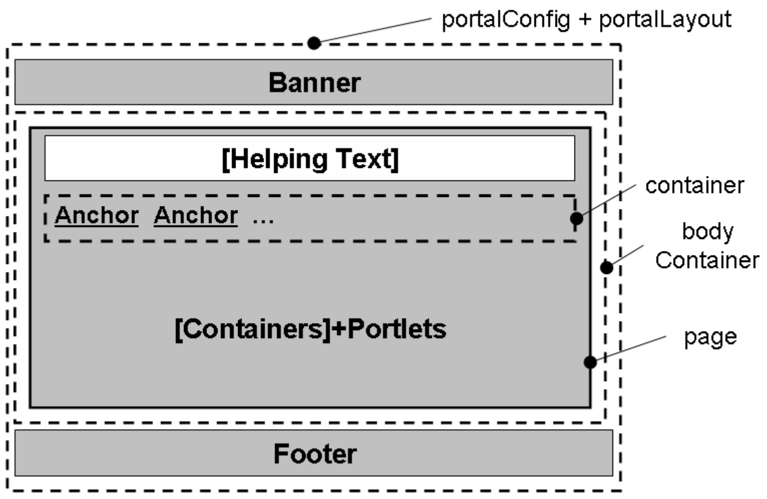


Figure A.7: A portal layout, when design attribute values are: *transition=anchor*, *distribution=together*, and *position=top*.

appear together and above task portlets. Moreover helping text, if it is included in the portal design, will be above those anchors. Figure A.7 depicts the portal layout under these conditions. The only difference from one page to another is the distribution of portlets and their containers. It is exactly the statechart and its state configurations which determine that distribution. Moreover, as Figure A.6 depicts, the same state configuration can be transformed into different pages, in this case depending on the *alignment* attribute. For the page on the left, the `WindowDescriptor` of the A state has the 'column' value in the *alignment* attribute, whereas for the page on the right the value is 'row'. In both cases, the value for the F state is 'row'.

Each *eXo* page has a name for identifying it internally. The naming convention sets that '*/home*' must be the name of the main page, which is the first page the user sees when logs in. The rest of pages will have any name with '/' as prefix.

Listing A.7 depicts the rules which generate the PAGES package of an EXO model, i.e., the part corresponding to the ***_pages.xml* file. The rules transform the `StateMachine` element (in the `ORCHESTRATION` package of the `SOP` metamodel) into a `PageSet` element in the `EXO::Pages` package. That element is made up of several `Page` objects, one for each state configuration of the `StateMachine`. Each `Page` element is generated by the `Configuration2Page` rule, in line 50. As said previously, the page names are *'/home'* or *'/a_name'*, depending on whether the page is or not the main: a conditional statement in Ruby is used to make a decision. In the second case the configuration name is reused. As for the `renderer` attribute, the `{portalName}PageDecorator` value is assigned, and recall that this decorator class is described in the *skin_config.xml* file, generated through the transformation rules described in Section A.3.

Moreover, every page is also populated with portlets and some nested containers (see Figure A.7):

- First, the `PagePortlet` rule is fired in line 69 to generate the description for the portlet which must show the called 'helping text', described in the `HelpingText` object of the `RENDERING` of the source model. This object is optional in the design, therefore the fire of the rule is under a guard condition. The `PagePortlet` rule uses the `Preferences4PagePortlet` and `PortletPreferences4PagePortletImp` rules to complete its task, i.e., to define the portlet preferences.
- In line 76 the `Container4Transiciones` and `State2Container` rules are fired, to generate the anchors and the portlets corresponding to transitions and simple states, respectively.
  - The `Container4Transiciones` rule, in line 129, first generates a container which is only a means to structure all the anchors. Therefore, its decorator is the CSS class which specifies the 'transparent' description, namely, the class named `{portalName}TransparentDecorator` in the transformation of Section A.3. Moreover, for each transition the `Transition2Portlet` rule is fired in line 138. The `all_transitions` auxiliary function gets the transitions whose source state belongs to the current state configuration. The `Transition2Portlet` rule, in line 147, generates the description for the portlet implementing the anchor functionality in the portal. In that description it assigns the names of the decorator and style classes generated in the transformation of Section A.3 and it uses the `PortletPreferences4Transition` and `Portlet-Preferences4TransitionImp` rules to define the portlet preferences. One of these preferences is the target page of the anchor, i.e., the page where the anchor goes to, and it is calculated in line 185 using an auxiliary function named `fire`.
  - The `State2Container` rule is fired for the first orthogonal state (AND-state) of the state configuration, if there is one. In line 202, that rule generates a container for functional portlets (corresponding to simple

states). Like anchor containers, the decorator for these containers is transparent. Next, if the substates of the current state are simple the `SimpleState2Portlet` rule is fired for each one, in line 212, and if it contains other orthogonal substates, then the `State2Container` rule is fired recursively, in line 215. Thus, successive nested `Container` objects are generated. The `SimpleState2Portlet` rule, in line 224, generates the description for the functional portlet corresponding to the current simple state.

- When the state configuration has simple states not contained in any orthogonal state (AND-state), their corresponding portlets must be placed without any specific container, apart from the page itself. Therefore, the `SimpleState-2Portlet` rule is fired directly, in line 72.

In the portlet description the `windowId` attribute is composed of three elements: a display name, a portlet name and an identifier. The first two are names specified by its developer in the portlet descriptor, and the identifier is any value to distinguish the different instances or uses of the same portlet in the portal. For the functional portlets (corresponding to tasks) the state name is used as identifier, and for anchors, given that the *navigationStep* portlet is used always and the same transition name can appear more than once, a number is used. The `TransitionId` variable, in line 26, holds the reference of an auxiliary `IdGenerator` object, out of the `EXO::Pages` package. This object is used to obtain successive integer values to build different `windowId` values, in line 159.

As stated above, the generation of the nested containers in a page depends on the state configuration firing the generation of the page itself; the `current_config-___EXO___Pages` variable is used to hold that configuration, in line 67, across the rules of the phase.

Listing A.7: The *sop2pages.rb* file.

```
1  use_library 'helper://state_machine'
2  use_library 'helper://presentation'
3  use_library 'helper://util'

5  transformation 'SOP2Pages'

7    #Transformation parameters

9    EXO__Pages_default_values = { #
10     :view_permission => 'any',
11     :edit_permission => 'owner'
12   }
```

```
  #Take default values if not given
  $EXO__Pages_parameters ||= EXO__Pages_default_values
  $EXO__Pages_parameters =
    EXO__Pages_default_values.merge($EXO__Pages_parameters) #

  VIEW_PERMISSION = $EXO__Pages_parameters[:view_permission]
  EDIT_PERMISSION = $EXO__Pages_parameters[:edit_permission]


phase 'EXO__Pages__PageSet__generation' do

  TextId = IdGenerator.new
  TransitionId = IdGenerator.new
  current_config__EXO__Pages = nil


  # Each StateMachine is transformed into a PageSet, that
  # contains references to Page objects.
  # There is one Page for every configuration
  #
  top_rule 'Rule4PageSet' do
    from     SOP::Orchestration::StateMachine
    to       EXO::Pages::PageSet

    mapping do |state_machine, page_set|
      page_set.layout = state_machine.workview.position
      page_set.pages  = #
        #fires Configuration2Page: once for every configuration
        [state_machine.initial_configuration] +
        state_machine.all_configurations.find_all{|c| !c.is_initial?}
    end
  end

  #The Page object for a given configuration.
  #It contains another portlet, if there is a SOP::Rendering::HelpingText
  #
  rule 'Configuration2Page' do #
    from     SOP::Orchestration::Configuration
    to       EXO::Pages::Page

    mapping do |configuration, page|
      page.renderer = "PageRowRenderer"
      page.name     = if configuration.is_initial?: "/home"
                      else "/" + configuration.page_name
                      end
      page.title    = if configuration.is_initial?: configuration.owner.portal_title
                      else configuration.page_name
                      end
      page.viewPermission = VIEW_PERMISSION
      page.editPermission = EDIT_PERMISSION
      page.decorator      = configuration.owner.portal_title + "PageDecorator"

      #make current configuration available to subsequently fired rules
      current_config__EXO__Pages = configuration #

      page.textPortlet  = #fires PagePortlet #
        configuration  if configuration.owner.helping_text #

      page.statePortlets  = #
        #fires SimpleState2Portlet: once for every simple root state
        configuration.simple_root_states

      page.container      = #
        [configuration] +                    #fires Container4Transiciones
        configuration.orthogonal_root_states #fires State2Container
    end
  end

  #Portlet included in a page if there exists a SOP::Rendering::HelpingText
```

```
#
rule 'PagePortlet' do #
  from    SOP::Orchestration::Configuration
  to      EXO::Pages::Portlet

  mapping do |configuration, page_portlet|
    page_portlet.renderer      = "PortletRenderer"
    page_portlet.decorator     =
      configuration.owner.portal_title + "TextPDecorator"
    page_portlet.portletStyle =
      configuration.owner.portal_title + "TextPStyle"
    page_portlet.showInfoBar   = "false"
    tmp = configuration.owner.helping_text.portlet
    page_portlet.title         = tmp.windowTitle
    page_portlet.windowId      = tmp.displayName + "/" + tmp.portletName +
                                 "/" + TextId.next.to_s
    page_portlet.preferences   = page_portlet
  end
end

rule 'Preferences4PagePortlet' do
  from    EXO::Pages::Portlet
  to      EXO::Pages::PortletPreferences

  mapping do |page_portlet, preferences|
    preferences.preference =  #fires PortletPreferences4PagePortletImp
      preferences
  end
end

rule 'PortletPreferences4PagePortletImp' do
  from    EXO::Pages::PortletPreferences
  to      EXO::Pages::Preference
  mapping do |preferences, preference|
    preference.name  = "text"
    preference.value = current_config__EXO__Pages.owner.helping_text
  end
end


#Every page is populated with a portlet for
#every transition originating in any of the
#configuration states.
#All such portlets are included in a container,
#generated by this rule.
#
rule 'Container4Transiciones' do #
  from    SOP::Orchestration::Configuration
  to      EXO::Pages::Container

  mapping do |configuration, container|
    container.renderer  =
      configuration.owner.workview.portalRenderer
    container.decorator =
      configuration.owner.portal_title + "TransparentDecorator"
    container.portlets = #
     #fires Transition2Portlet
      configuration.all_transitions
  end
end

#Every transition originating in any of the states
#in the current configuration is transformed into a portlet.
#
copy_rule 'Transition2Portlet' do  #
  from    SOP::Orchestration::Transition
  to      EXO::Pages::Portlet

  mapping do |transition, portlet|
    portlet.renderer   = "PortletRenderer"
    portlet.decorator  = transition.name + "AnchorDecorator"
    portlet.portletStyle = transition.name + "AnchorStyle"
    portlet.showInfoBar  = "false"
    portlet.title   = "Step␣Navigation"
    portlet.windowId     = transition.portlet.displayName + "/" +
                           transition.portlet.portletName + "/" +
                           TransitionId.next.to_s #
    portlet.preferences   =  #
      #fires PortletPreferences4Transition
      transition
  end
end

copy_rule 'PortletPreferences4Transition' do #
  from     SOP::Orchestration::Transition
  to       EXO::Pages::PortletPreferences

  mapping do |transition, preferences|
    preferences.preference = #fires PortletPreferences4TransitionImp
      transition
  end
end

copy_rule 'PortletPreferences4TransitionImp' do
  from     SOP::Orchestration::Transition
  to       EXO::Pages::Preference, EXO::Pages::Preference
  mapping do |transition, pref_a, pref_b|
    pref_a.name  = "transitionTitle"
    pref_a.value = transition.name

    pref_b.name  = "pageName"

    tmp_target_conf = transition.fire(current_config__EXO__Pages) #
    #the configuration when yhe given transition is fired
    #in the current configuration

    pref_b.value = tmp_target_conf.page_name
  end
end


#Every orthogonal state in the current configuration generates a container.
#This container is populated with one portlet
#for every simple state in the configuration,
#which is contained in that orthogonal state.
#Besides this, the container is also populated with
#the containers generated by any orthogonal state in the same configuration
#which is contained in the originating orthogonal state.
#
copy_rule 'State2Container' do #
  from    SOP::Orchestration::State
  to      EXO::Pages::Container
  filter do |state| state.isOrthogonal end

  mapping do |state, container|
    container.renderer       =
      state.window_descriptor.portlet4statesContainerRenderer
    container.decorator      =
      state.state_machine.portal_title + "TransparentDecorator"
    container.portlets = #
      #fires SimpleState2Portlet
      current_config__EXO__Pages.simple_sub_states(state)
    container.container = #
      #fires State2Container
      current_config__EXO__Pages.orthogonal_sub_states(state)
  end
end
```

```
  # Transforms a SimpleState into a portlet. Every SimpleState is always
  # mapped into a portlet, included in a container or in a page.
  #
  copy_rule 'SimpleState2Portlet' do #
    from    SOP::Orchestration::State
    to      EXO::Pages::Portlet
    filter do |state| state.isSimple end

    mapping do |state, portlet|
      portlet.renderer     = "PortletRenderer"
      portlet.decorator    = state.name + "PDecorator"
      portlet.portletStyle = state.name + "PStyle"
      portlet.showInfoBar  = "false"
      portlet.title        =
        state.portlet.windowTitle if state.portlet.showInfoBar
      portlet.windowId     =
        "#{state.portlet.displayName}/#{state.portlet.portletName}/#{state.name}"
    end
  end
end
```

A.6 Transformation for NAVIGATION package

Just as the *skin_config.xml* file constitutes an index for the CSS classes defined in the *{portalName}.css* file, the ***-navigation.xml* file constitutes an index for the pages defined in the ***-pages.xml* file. Although for the aim of the prototype presented in this appendix a list of the pages is enough, the index must have a hierarchical structure, as the EXO metamodel (in Figure A.2) shows. Therefore, in this case the main page, related to the initial state configuration, defines the root node and the rest of pages are its subnodes. Listing A.8 depicts the rule set which creates the NAVIGATION package of an EXO model, i.e., the part corresponding to the ***_navigation.xml* file.

Specifically, the `StateMachine` in the `SOP:Orchestration` package is transformed into a `NodeNavigation` element in the `EXO::Navigation` package, using the `Rule4NodeNavigation` rule, in line 23. This `NodeNavigation` element is made up of several nested `Node` objects, in this prototype one root `Node` and its subnodes: the first one generated by the `Rule4MainNode` rule (in line 35), and the latter generated by the `Node4Configuracion` rule, fired in line 47 for each state configuration, except for the initial one.

Apart from `label`, `name`, `uri` and `reference`, the node description is completed with the `edit` and `view permission` values, two attributes which indicate who has permission to edit and view that node, respectively. The chosen default values are `any` and `owner`. They are stored in a map whose reference is kept in the local variable `EXO__Navigation_default_parameters` (in line 7). If the `EXO__Navigation_parameters` global variable is not defined with specific values when the transformation is launched, it will be created and assigned the value of `EXO__Navigation_default_parameters` (in line 15), otherwise the default values are overridden.

Listing A.8: The *sop2navigation.rb* file.

```
1  use_library 'helper://state_machine'

3  transformation 'sop2navigation'

5    #Transformation parameters

7    EXO__Navigation_default_values = { #
8      :view_permission => 'any',
9      :edit_permission => 'owner'
10   }

12   #Take default values if not given
13   $EXO__Navigation_parameters ||= EXO__Navigation_default_values #
14   $EXO__Navigation_parameters =
15     EXO__Navigation_default_values.merge($EXO__Navigation_parameters) #

17   VIEW_PERMISSION = $EXO__Navigation_parameters[:view_permission]
18   EDIT_PERMISSION = $EXO__Navigation_parameters[:edit_permission]


21 phase 'sop2navigation' do

23   top_rule 'Rule4NodeNavigation' do #
24     from     SOP::Orchestration::StateMachine
25     to       EXO::Navigation::NodeNavigation

27     mapping do |state_machine, node_navigation|
28       node_navigation.nodes     = state_machine #fires Rule4MainNode #
29     end
30   end

32   #This rule generates the only Node object
33   #contained directly in the NodeNavigation
34   #
35   rule 'Rule4MainNode' do #
36     from     SOP::Orchestration::StateMachine
37     to       EXO::Navigation::Node

39     mapping do |state_machine, node_home|
40       node_home.title          = 'home'
41       node_home.viewPermission = VIEW_PERMISSION
42       node_home.editPermission = EDIT_PERMISSION
43       node_home.label          = 'home'
44       node_home.uri            = "/"
45       node_home.name           = "home"
46       node_home.pageReference  = "home"
47       node_home.nodes          =  #
48         #fires Node4Configuracion once for every configuration,
49         #except for the initial one
50         state_machine.all_configurations.reject{|c| c.is_initial?}
51     end
52   end

54   #One Node objtect is generated for every configuration,
55   #except for the initial one
56   #
57   rule 'Node4Configuracion' do #
58     from     SOP::Orchestration::Configuration
```

```
    to        EXO::Navigation::Node
    filter  do |configuration| !configuration.is_initial? end

    mapping do |configuration, node|
      node.title           = configuration.page_name
      node.viewPermission  = VIEW_PERMISSION
      node.editPermission  = EDIT_PERMISSION
      node.label           = configuration.page_name
      node.uri             = configuration.page_name
      node.name            = configuration.page_name
      node.pageReference   = configuration.page_name
    end
  end
end
```

Bibliography

[1] S. Agarwal, S. Handschuh, and S. Staab. Surfing the Service Web. In *International Semantic Web Conference*, volume 2870 of *Lecture Notes in Computer Science (LNCS)*, pages 211–226. Springer, October 2003. 91, 100, 112

[2] C. Ames, S. Burleigh, and S. Mitchell. WWWorkflow: World Wide Web based Workflow. In *Hawaii International Conference on System Sciences*, January 1997. 81

[3] androMDA Team. androMDA. Available from http://www.andromda.org/ (December 2007). 31

[4] B. Benatallah, M. Dumas, Q.Z. Sheng, and A.H.H. Ngu. Declarative Composition and Peer-to-Peer Provisioning of Dynamic Web Services. In *18th International Conference on Data Engineering (ICDE'02)*. IEEE, 2002. 110

[5] D. Berardi, D. Calvanese, G. de Giacomo, M. Lenzerini, and M. Mecella. Automatic Composition of E-services that Export their Behavior. In *1st International Conference on Service Oriented Computing (ICSOC 2003)*, volume 2910 of *Lecture Notes in Computer Science (LNCS)*, pages 43–58. Springer, 2003. 53

[6] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American Magazine*, 284(5):34–43, May 2001. 90

[7] J. Blattman, N. Krishnan, D. Polla, and M. Sum. Open-Source Portal Initiative at Sun: Portlet Repository, 2006. Available from http://developers.sun.com/portalserver/reference/techart/portlet-repository.html (May 2008). 18

[8] J. Blom. Personalization - A Taxonomy. In *Conference on Human Factors in Computing Systems CHI'00*, pages 313–314. ACM, April 2000. 62

[9] J. Boye. Portals: from idea to reality - the dangers of the current state of portals in the marketplace. In *Online Information Conference*, pages 103–106, 2005. 11

[10] M. Brambilla, S. Ceri, S. Comai, and P. Fraternali. Specification and Design of Workflow-driven Hypertexts. *Journal of Web Engineering*, 1(1), 2002. 108

[11] M. Brambilla, S. Ceri, P. Fraternali, and I. Manolescu. Process Modeling in Web Applications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(4):360–409, October 2006. 81, 115

[12] J. Bézivin. In Search of a Basic Principle for Model Driven Engineering. *UPGRADE, Novótica*, 2, April 2004. 68

[13] J. Bézivin. On the Unification Power of Models. *Software and System Modeling (SoSym)*, 4(2):171–188, May 2005. 26, 29, 30, 31

[14] L. Cabral and J. Domingue. Mediation of Semantic Web Services in IRS-III. In *Workshop on Mediation in Semantic Web Services (MEDIATE) at International Conference on Service Oriented Computing (ICSOC)*, Amsterdam, The Netherlands, 2005. 111

[15] F. Casati and M. C. Shan. Dynamic and adaptive composition of e-services. *Information Systems*, 26(3):143–163, May 2001. 53

[16] B. Castle. Introduction to Web Services for Remote Portlets. use WSRP in Service-Oriented Architecture, 2005. Available from http://www-128.ibm.com/developerworks/webservices/library/ws-wsrp/ (May 2008). 18

[17] S. Ceri, P. Fraternali, and A. Bongio. Conceptual Modeling of Data-Intensive Web Applications. *IEEE Internet Computing*, 6(4):20–30, July 2002. 80

[18] Compuware. OptimalJ. Available from http://www.compuware.com/products/optimalj/ (December 2007). 31

[19] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. In *Workshop on Generative Techniques in the Context of Model-Driven Architecture (OOPSLA03)*, October 2003. 35, 36

[20] P. Desfray. UML Profiles versus Metamodeling extensions: An ongoing debate. In *OMG Workshop, UML in the .com Enterprise: Modeling*

CORBA, Components, XML/XMI and Metadata, November 2000. Available from http://www.omg.org/news/meetings/workshops/uml_presentations.htm (November 2007). 84

[21] O. Díaz, J. Iturrioz, and A. Irastorza. Improving Portlet Interoperability through Deep Annotation. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 372–381, New York, NY, USA, 2005. ACM Press. 48

[22] Oscar Díaz and Iñaki Paz. Turning web applications into portlets: Raising the issues. In Wojciech Cellary and Hiroshi Esaki, editors, *Symposium on Applications and the Internet (SAINT2005)*, pages 31–37. IEEE Computer Society, 2005. 49

[23] J. Domingue, L. Cabral, F. Hakimpour, D. Sell, and E. Motta. IRS-III: A Platform and Infrastructure for Creating WSMO-based Semantic Web Services. In *Workshop on WSMO Implementations (WIW)*, volume 113, Frankfurt, Germany, 2004. 111

[24] DSM. Domain-Specific Modelling Forum. Available from http://www.dsmforum.org/ (December 2007). 33

[25] ESWC 4th European Semantic Web Conference. OWL-S Experiences and Future Developments workshop, June 2007. Available from http://www.eswc2007.org/workshops.cfm (April 2008). 7

[26] E. Evans. *Domain-Driven Design. Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004. 115

[27] Exo. Exo Community. Available from http://www.exoplatform.org (November 2007). 5, 56

[28] eXo. Exo Platform version1. Available from http://docs.exoplatform.org/exo-documents/exo.site/index.html (October 2007). 65

[29] M.C. Ferreira de Oliveira, M.A. Santos Turine, and P.C. Masiero. A Statechart-Based Model for Hypermedia Applications. *ACM Transactions on Information Systems*, 19(1):28–52, January 2001. 44, 53

[30] J.M. Firestone. *Enterprise Information Portals and Knowledge Management*. KMCI Press, 2002. 10

[31] P. Fraternali and P. Paolini. Model-driven development of Web applications: the AutoWeb system. *ACM Transactions on Information Systems (TOIS)*, 18(4):323–382, October 2000. 80

[32] I. Garrigós, J. Gómez, P. Barna, and G-J. Houben. A Reusable Personalization Model in Web Application Design. In *International Workshop on Web Information Systems Modelling (at ICWE2005)*, July 2005. 64

[33] R. Gitzel, A. Korthaus, and M. Schader. Using established Web Engineering knowledge in model-driven approaches. *Science of Computer Programming*, 66(2):105–124, April 2007. 58, 77

[34] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004. 34

[35] S. Handschuh and S. Staab (eds.). *Annotation for the Semantic Web*. IOS Press, 2003. 6, 88, 91

[36] S. Handschuh, S. Staab, and R. Volz. On Deep Annotation. In *12th International Conference on the World Wide Web (WWW2003)*. ACM, May 2003. 6, 91, 100, 106

[37] S. Handschuh, R. Volz, and S. Staab. Annotation for the Deep Web. *IEEE Intelligent Systems*, 18(5):42–48, September/October 2003. 91, 94

[38] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996. 53

[39] D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman. On the Formal Semantics of Statecharts. In *2nd IEEE Symposium on Logic in Computer Science*, pages 54–64. IEEE Computer Society, 1987. 4, 53, 54, 144

[40] R. Hennicker and N. Koch. Modeling the User Interface of Web Applications with UML. In *Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists. Workshop of the pUML-Group at the UML 2001*, pages 158–172, 2001. 80

[41] Hewlett-Packard. Jena: a Java framework for writing Semantic Web applications, 2003. Available from http://www.hpl.hp.com/semweb/jena.htm (March 2008). 104

[42] Honeywell Inc. DOME (the DOmain Modeling Environment). Available from http://www.htc.honeywell.com/dome/index.htm (September 2007). 34

[43] IBM. WebSphere, May 2006. Available from http://www.ibm.com/websphere. 56

[44] ISIS Institute for Software Integrated Systems (Vandervilt University). GME (The Generic Modeling Environment). Available from http://www.isis.vanderbilt.edu/projects/gme/index.html (September 2007). 34

[45] Interactive Objects. ArcStyler. Available from http://www.interactive-objects.com/products/arcstyler (December 2007). 31

[46] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Oranization for Business success*. Addison-Wesley, 1997. 116

[47] A. Jafari and M. Sheehan. *Designing Portals: Opportunities and Challenges*. IRM Press, 2003. 1, 2

[48] R. Jasper and M. Uschold. A Framework for Understanding and Classifying Ontology Applications. In *IJCAI99 Workshop on Ontologies and Problem Solving Methods KRR5*, August 1999. 7, 93

[49] Java Community Process. JSR 286: Portlet Specification 2.0. Available from http://www.jcp.org/en/jsr/detail?id=286 (March 2008). 2, 7, 84, 89, 110, 117

[50] Java Community Process. JSR 168: Portlet specification, October 2003. Available from http://www.jcp.org/en/jsr/detail?id=168 (March 2008). 7, 12, 93

[51] S. Kelly. Domain-specific languages versus generic modeling languages, May 2007. Available from http://www.ddj.com/architect/199500627 (December 2007). 33, 34

[52] S. Kent. Model Driven Engineering. In *3th International Conference on Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science (LNCS)*, pages 286 – 298. Springer-Verlag, 2002. 26, 27

[53] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained. The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003. 28, 29, 30

[54] N. Koch. Software Engineering for Adaptive Hypermedia Systems. Reference Model, Modeling Techniques and Development Process, December 2000. PhD Thesis. 64

[55] N. Koch. Transformation Techniques in the Model-Driven Development Process of UWE. In *6th International Conference on Web Engineering. 2nd International Workshop on Model Driven Web Engineering (MDWE'06)*, volume 155. ACM, July 2006. 35, 46

[56] N. Koch and A. Kraus. Towards a Common Metamodel for the Development of Web Applications. In *3rd International Conference on Web Engineering (ICWE 2003)*, volume 2722 of *Lecture Notes in Computer Science (LNCS)*, pages 497–506. Springer Verlag, 2003. 84

[57] N. Koch, A. Kraus, C. Cachero, and S. Meliá. Integration on Business Processes in Web Application Models. *Journal of Web Engineering*, 3(1):22–49, May 2004. 115

[58] N. Koch, A. Kraus, and R. Hennicker. The Authoring Process of the UML-based Web Engineering Approach. In *1st International Workshop on Web-Oriented Software Technology*, June 2001. 80

[59] C.W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2), June 1992. 77

[60] G.P. Marquis. Application of traditional system design techniques to web site design. *Information and Software Technology*, 44(9):507–512, 2002. 13

[61] S. Meliá, A. Kraus, and N. Koch. MDA Transformations Applied to Web Application Development. In *5th International Conference on Web Engineering (ICWE 2005), Sidney, Australia*, volume 3579 of *Lecture Notes in Computer Science (LNCS)*, pages 465–471. Springer-Verlag, July 2005. 82

[62] MetaCase. MetaEdit+. Available from http://www.metacase.com/ (September 2007). 34

[63] J. Miller and J. Mukerji. Model Driven Architecture (MDA), July 2001. Technical Report ormsc/2001-07-01, Object Management Group (OMG), Architecture Board ORMSC. 35

[64] N. Moreno and A. Vallecillo. A Model-Based Approach for Integrating Third Party Systems with Web Applications. In *5th International Conference on Web Engineering (ICWE 2005), Sydney, Australia*, volume 3579 of *Lecture Notes in Computer Science (LNCS)*, pages 441–452, Berlin, 2005. Springer. 46, 84

[65] M. Mrissa, C. Ghedira, D. Benslimane, and Z. Maamar. A Context Model for Semantic Mediation in Web Services Composition. In *Conceptual Modeling - ER 2006*, volume 4215 of *Lecture Notes in Computer Science (LNCS)*, pages 12–25. Springer, Berlin, October 2006. 112

[66] M. Mrissa, C. Ghedira, D. Benslimane, Z. Maamar, F. Rosenberg, and S. Dustdar. A Context-Based Mediation Approach to Compose Semantic Web Services. *ACM Transactions on Internet Technology*, 8(1), November 2007. 112

[67] P. Muller, P. Studer, F. Fondement, and J. Bèzivin. Platform independent Web application modeling and development with Netsilon. *Software & System Modeling*, 4(4):424–442, November 2005. 82

[68] J. Nielsen. Personalization is Over-Rated, 1998. Alertbox. Jakob's column on Web Usability. Available from http://www.useit.com/alertbox/981004.html (June 2008). 61

[69] D. A. Nunes and D. Schwabe. Rapid Prototyping of Web Applications Combining Domain Specific Languages and Model Driven Design. In *6th International Conference on Web Engineering (ICWE 2006)*, volume 263 of *ACM International Conference Proceeding Series*, pages 153–160, New York, USA, July 2006. ACM. 83

[70] OASIS. Electronic Business using eXtensible Markup Language (ebXML), 2003. Available from http://www.ebxml.org/ (July 2007). 48

[71] OASIS. Web Service for Remote Portlets Specification (WSRP) Version 1.0, 2003. Available from http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrp (March 2008). 7, 15, 18, 48, 93

[72] OASIS. WSRP-ebXML Registry Technical Note. Using ebXML Registry to Publish, Manage and Discover WSRP Artifacts, 2005. Available from http://www.ebxml.org/ (June 2008). 48

[73] OASIS. Web Services Business Process Execution Language (WSBPEL) Version 2.0, 2007. Available from http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel (May 2008). 4

[74] OASIS. Web Service for Remote Portlets Specification (WSRP) Version 2.0, 2008. Available from http://docs.oasis-open.org/wsrp/v2/wsrp-2.0-spec.html (April 2008). 2, 6, 7, 84

[75] Object Management Group (OMG). The Common Warehouse Model 1.1. (CWM), 2003. Available from http://www.omg.org/cgi-bin/doc?formal/03-03-02 (June 2008). 36

[76] Object Management Group (OMG). MDA Guide Version 1.0.1, june 2003. Available from http://www.omg.org/docs/omg/03-06-01.pdf (April 2007). 4, 28, 35, 44, 45, 46, 68

[77] Object Management Group (OMG). MOF QVT Final Adopted Specification, November 2005. Available from http://www.omg.org/docs/ptc/05-11-01.pdf (April 2008). 8, 36, 37, 70

[78] Object Management Group (OMG). Unified Modeling Language (UML): Superstructure, August 2005. Available from http://www.omg.org/cgi-bin/doc?formal/05-07-04 (April 2007). 7, 53

[79] Object Management Group (OMG). Meta Object Facility (MOF) Core Specification ver2.0, January 2006. Available from http://www.omg.org/docs/formal/06-01-01.pdf (July 2007). 84

[80] Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, July 2007. Available from http://www.omg.org/docs/ptc/07-07-07.pdf (April 2008). 7

[81] Object Management Group (OMG). Software Process Engineering Metamodel (SPEM), version 1.1, January 2007. Available from http://www.omg.org/technology/documents/formal/spem.htm. 47

[82] A. Olivé. *Conceptual Modeling of Information Systems*. Springer, October 2007. 53

[83] Oracle. Oracle Portal. Available from http://www.oracle.com/appserver/portal_home.html (November 2007). 56

[84] F. Pan and J. R. Hobbs. Time in OWL-S. In *1st International Semantic Web Services Symposium*, March 2004. 99

[85] M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara. Semantic Matching of Web Services Capabilities. In *1st International Semantic Web Conference*, pages 333–347, Berlin, June 2002. Springer-Verlag. 51, 111

[86] QVT Partners. Revised submission for MOF 2.0 Query/ Views/ Transformations RFP, 2003. Available from http://qvtp.org/downloads/1.1/qvtpartners1.1.pdf (November 2007). 82

[87] C. Peltz. Web Service Orchestration and Choreography: A look at WSCI and BPEL4WS. *Web Services Journal*, 03(7), July 2003. 97

[88] E. Reshef. Building Interactive Web Services with WSIA & WSRP. *Web Services Journal*, pages 2–6, December 2002. Available from http://webservices.sys-con.com/read/39627.htm (May 2008). 13

[89] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web Service Modeling Ontology. *Applied Ontology*, 1(1):77 – 106, 2005. 90, 111

[90] G. Rossi, H. Schmid, and F. Lyardet. Engineering Business Processes in Web Applications: Modeling and Navigation issues. In *Third International Workshop on Web-Oriented Software Technologies (IWWOST03)*, pages 81–89, Oviedo (Spain), 2003. 115

[91] A. Roy-Chowdhury, S. Ramaswamy, and X. Xu. Using Click-to-Action to Provide User-Controlled Integration of Portlets, December 2002. Available from http://www.ibm.com/developerworks/websphere/library/techarticles/0212_roy/roy.html (March 2008). 109

[92] J. Sametinger. *Software Engineering with Reusable Components*. Springer, 1997. 116

[93] D.C. Schmidt. Why Software Reuse has Failed and How to Make It Work for You. Available from http://www.cs.wustl.edu/ schmidt/reuse-lessons.html (September 2007). 25

[94] D.C. Schmidt. Model Driven Engineering. *IEEE Computer*, 39(2):41–47, February 2006. 25

[95] D. Schwabe, R.M. Guimaraes, and G. Rossi. Cohesive Design of Personalized Web Applications. *IEEE Internet Computing*, 6(2):34–43, March 2002. 62

[96] B. Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003. 77

[97] E. Sirin, J. Hendler, and B. Parsia. Semi-automatic Composition of Web Services using Semantic Descriptions. In *1st Workshop on Web Services: Modeling, Architecture and Infrastructure. In conjunction with ICEIS 2003*, pages 17–24. ICEIS Press, April 2003. 51, 111

[98] J. Sánchez Cuadrado and J. García Molina. A Phasing Mechanism for Model Transformation Languages. In *ACM Symposium on Applied Computing. Seoul, Korea*, pages 1020–1024. ACM, March 2007. 41, 126

[99] J. Sánchez Cuadrado, J. García Molina, and M. Menárguez Tortosa. RubyTL: A Practical, Extensible Transformation Language. In *Model Driven Architecture - Foundations and Applications. 2nd European Conference, ECMDA-FA2006. Bilbao, Spain*, pages 158–172. Springer-Verlag, July 2006. 7, 37, 38, 39, 41, 70, 121

[100] T. Stahl, M. Völter, J. Bettin, A. Haase, and S. Helsen. *Model Driven Software Development: Technology, Engineering, Management*. Wiley, June 2006. 31

[101] H. Strauss. All About Web Portals: A Home Page Doth Not a Portal Make. In *Web Portal and Higher Education. Technologies to Make IT Personal*, pages 33–40. Jossey-Bass, A Wiley Company, 2000. 10

[102] P. Tetlow, J. Pan, D. Oberle, E. Wallace, M. Uschold, and E. Kendall. Ontology Driven Architectures and Potential Uses of the Semantic Web in Systems and Software Engineering, 2006. Available from http://www.w3.org/2001/sw/BestPractices/SE/ODA/ (March 2008). 89

[103] The Delphi Group. Portal Lifecycle Management: Addressing the Hidden Cost of Portal Ownership, 2001. Available from http://www.mongoosetech.com/downloads/ portal_ownership.pdf. 3, 43

[104] D. Thomas. Programming Ruby. The Pragmatic Programmers' Guide, 2004. Available from http://www.rubycentral.com/book/ (April 2007). 37

[105] J-P. Tolvanen. Domain-Specific Modeling: How to Start Defining Your Own Language, February 2006. Available from http://www.devx.com/enterprise/Article/30550 (December 2007). 32, 34

[106] A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000. 32

[107] A. van Deursen, E. Visser, and J. Warmer. Model-Driven Software Evolution. a Research Agenda. In *CSMR Workshop on Model-Driven Software Evolution (MoDSE 2007)*, pages 41–49, Amsterdam, The Netherlands, March 2007. 25

[108] W3C Consortium. Cascading Style Sheet (CSS), 1998. Available from http://www.w3.org/TR/REC-CSS2/ (March 2007). 16, 56

[109] W3C Consortium. XSL Transformations (XSLT), 1999. Available from http://www.w3.org/TR/xslt (May 2008). 36

[110] W3C Consortium. OWL-S: Semantic Markup for Web Services, 2004. Available from http://www.w3.org/Submission/OWL-S/ (June 2008). 7, 95

[111] W3C Consortium. Platform For Privacy Preferences (P3P), 2006. Available from http://www.w3.org/TR/P3P11/ (June 2008). 62

[112] W3C Consortium. Web Services Description Language WSDL 2.0, 2007. Available from http://www.w3.org/TR/wsdl20/ (June 2008). 48

[113] C. Walker. Types of portal: a definition, 2006. Available from http://www.steptwo.com.au/papers/cmb_portaldefinitions/index.html (May 2008). 10

[114] S. Wong. Web Services: The Next Evolution of Application Integration, 2001. Available from http://www.ebizq.net/topics/web_services/features/1526.html (May 2008). 13

[115] J. Yu, B. Benatallah, R. Saint-Paul, F. Casati, F. Daniel, and M. Matera. A Framework for Rapid Integration of Presentation Components. In *16th International Conference on the World Wide Web (WWW2007)*, pages 923–932, May 2007. 81, 82, 110

Acknowledgements

So much time wishing to come this day and so many things I would like to say... and now perhaps I will not find the suitable words.

First and foremost, I would like to thank my supervisor Oscar Díaz for investing a great deal of time in this work and for encouraging me those times when I saw all black. He has taught me the value of being persevering in our research works and keeping open eyes to coming innovations.

I would also like specially thank Luis M. Alonso for his great support and push with the implementation in RubyTL and his help in the creation of the annexe of this dissertation.

The help of Arturo Jaime and Maider Azanza has been also invaluable through long *couch sessions*, they helped me to realize that I was not alone and I was not the only one in the desert.

They are all members of Onekin research group at the University of the Basque Country, like they are those, currents and formers, whom I would also like to express my gratitude to: Cristóbal Arellano, Iker Azpeitia, Sergio Fernández, Felipe Ibañez, Jon Iturrioz, Felipe Martín, Iñaki Paz, Sandy Pérez, Gorka Puente, Juan J. Rodríguez, and Salvador Trujillo. They have always been very friendly and helpful. I only wish to be able to pay them back in their own coin.

Despite the demand of this work, there was still life out there. I would like to thank my family and friends in Ordizia for suffering my melancholy and silences. They have always helped me to see more opportunities work aside. From now on, I will turn Ordizia into something more than the town where I sleep, and I will be able to spend more time with them. I promise.

And, in general, I would like to thank everybody, in the lunch time and in the faculty, who has understood me when sometimes I did not want to speak about my thesis and has cheered me up with a conversation or just a pat on the back.

Thanks, in all sincerity, to all of you.

Bukatu da, bukatu da....
Bukatu da, akabo!!
...
ala ez?
ai ama!!!