From page-centric to portlet-centric Web development: easing the transition using MDD*

Oscar Díaz, Arantza Irastorza, Jesús Sánchez Cuadrado*, Luis M. Alonso

February 10, 2009

ONEKIN Research Group, Department of Computer Languages and Systems, University of the Basque Country, 20080 San Sebastian (Spain) PO Box 649 Tel: 34 943 018064
* GTS Research Group, Department of Computer Science and Systems, University of Murcia, Campus de Espinardo, 30100 Murcia (Spain)
oscar.diaz@ehu.es, arantza.irastorza@ehu.es, jesusc@um.es, jipalgol@ehu.es

Abstract

Portlet syndication is the next wave following the successful use of content syndication in current portals. Portlets can be regarded as Web components, and the portal as the component container where portlets are aggregated to provide higher-order applications. This perspective requires a departure from how current Web portals are envisaged. The portal is no longer perceived as a set of pages but as an integrated set of Web components 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, the mapping from portlet orchestration (design time) to page navigation (implementation time) is too tedious and error prone. For instance, 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. This work uses the *eXo* platform as the target PSM, and the PIM is based on *Hypermedia Model Based on Statecharts*. The paper shows how this approach accounts for portal validation/verification to be conducted earlier at the PIM level, and streamlines both design and implementation of *eXo* portals. A running example is used throughout the paper.

Keywords: portlets, model-driven development, service-oriented architecture, web portals

1 Introduction

Web portals offer corporations a means by which to manage and access content from disparate sources across the firm. Frequently, this content is obtained locally in a va-

*Published in *Information and Software Technology Journal*, Vol. 50 (12), Pages 1210-1231, November 2008 (doi:10.1016/j.infsof.2007.11.006)

riety of formats that the portal makes transparent and adapt to the user profile. This is the role of the portal as a content manager.

But, 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. Information contained in one application will surely be needed in another, and requiring the individual user to manually bridge these gaps leads to frustration, lost productivity, and inevitable mistakes. More to the point, an increasing amount of content comes from outside the portal itself, and the main challenge is in integrating ("composing") these heterogeneous data sources. Examples of this tendency are content syndication (RSS and related standards) and Web mashup (i.e. a Web page or application that combines data from two or more external online sources). This has lead to the so-called *portal imperative*: the emergence of portal software as a universal integration mechanism [40]. This imperative is paving the way from the first-generation, content-centric portals towards the second-generation, service-oriented portals [39].

Unfortunately, current portal applications tend to be rather monolithic in both conception and support. This can be traced back to the fact that many first generation portal development environments evolved from content management origins. For portal applications to take hold, 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 [40], *"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 lifecycle".*

An evidence of the need of componentware in a portal setting is the widespread adoption of *portlets* among commercial players. Portlets are Web applications which are packed to be delivered through third-party Web applications (e.g. a portal). As Web applications, portlets are user-facing (i.e. return markup fragments rather than data-based XML) and multi-step (i.e. they encapsulate a chain of steps rather than a one-shot delivering) [10]. They are very much like Windows applications in a user desktop, in the sense that a portlet renders markup *fragments* that are surrounded by a decoration containing controls. Oracle, IBM, and BEA are examples of providers of Integrated Development Environments (IDE) for portal construction where the notion of portlet is incorporated.

However, portal IDEs have not yet exploited the full potential of portlets. The content-management role is still prevalent in most of current IDEs. The 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.

Yet, 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* is the integration achieved (i.e. at the front end).

This makes portlets 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 portal is no longer perceived as a set of pages but as an integrated set of Web components that are now delivered through the portal (i.e. portlets). 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 paper promotes this service-oriented view of Web portals.

To this end, portal design should abstract 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 portlets 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 paper reports on an MDD approach to service-oriented portal development. It follows the MDA (*Model-Driven Architecture*) proposal [29], and defines a Platform Independent Model (PIM), a Platform Specific Model (PSM) and the mapping transformations.

The PIM abstracts *portal* and *portlets* into *workspace* and *tasks,* respectively. The workspace specifies how tasks are engaged in workflows that guide the user in accomplishing some goal. However, traditional workflows impose a rigid control on the activities and data available where tasks are predetermined. By contrast, a "deskflow" should be more easy-going where users can freely browse along the available tasks while some general guidelines can be prescribed.

Therefore, the challenge is to find a formalism ductile enough to specify both constraint and unconstrained task flows. This paper argues about the benefits of using statecharts for this purpose. Statecharts have been traditionally used for control specification, and presentation concerns fall outside. But portlets are front-end applications. Therefore, an extension to statecharts is adopted that uses the structure and execution semantics of statecharts to specify both the structural organization and the browsing semantics of portlet aggregation [14].

The proposed PIM, called SOP (it stands for *Service-Oriented Portal*), aggregates the three viewpoints of a portal: tasks, orchestration (that is, flow) and rendering or presentation. As for the PSM, the *eXo* platform [12] is used. *eXo* is an open-source enterprise portal framework that support portlets. However, the portal is still conceived in terms of pages, where portlets are regarded as subordinated modules that output page fragments. Bridging the gap between SOP and *eXo* requires the use of transformation rules that map SOP constructs into *eXo* code. To this end, the *RubyTL* transformation language [38] is used.

Last but not least, we find most important to illustrate the benefits that MDD brought to our domain. From a practitioner's viewpoint, the often-claimed MDD ben-

efits (e.g. reduced costs, improved quality, reduced development time) should be made evident through concrete example. We attempt to provide such a case through a sample case for a six-portlet portal. This *Browsing* portal is available at *http://www.onekin.org/academicBrowsing/*

The rest of the paper is structured as follows. The notion of *portlet* is first introduced in section 2, section 3 provides an overview of portal development as an MDD development process. Section 4 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 6 and 5, respectively. The transformation from the PIM to the PSM is addressed in section 7. Section 8 outlines the benefits drawn from this approach. Related work is presented in section 9. Conclusions end the paper.

2 A brief on portlets

Web Services 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.

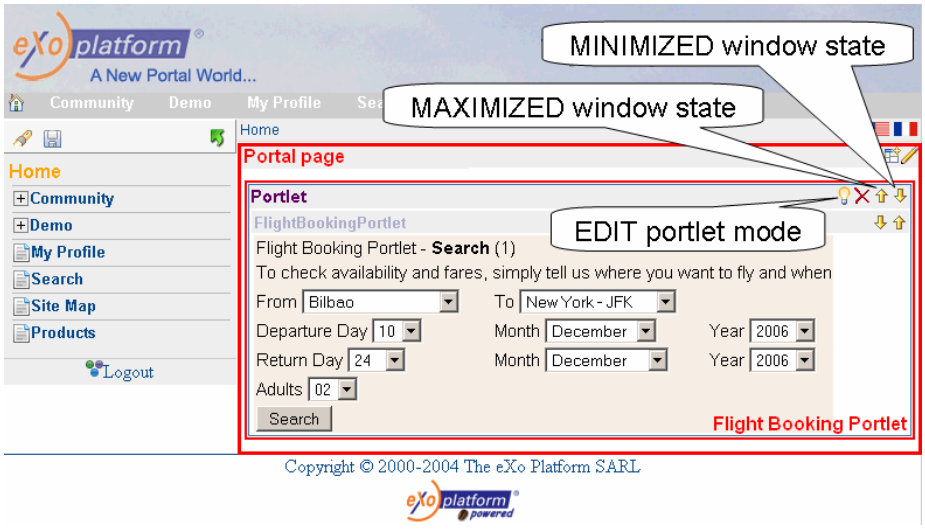

Figure 1: The *flightSearch* portlet being delivered through a portal.

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 navigation strategies are left to the calling application (i.e. the presentation layer).

But what if now we want to re-use the whole application, i.e. the business logic as well as the presentation and navigation code? It is worth noticing that presentation and navigation realization are very time consuming activities that convey costly marketing strategies that companies are interested in capitalizing when their services are offered through third-party Web applications. So far, most SOA approaches achieve integration at the back-end. Portlets open the door to achieve similar gains but now through front-end integration.

Let's go back to the flight-booking example, but now delivered as a portlet. A *flightSearch* portlet is defined that encapsulates not only the business logic but also the navigation and presentation realizations. Unlike the traditional Web-service approach, now the consumer of *flightSearch* re-uses both the presentation and the navigation. As for the presentation, portlet operations are still WSDL (*Web Services Description Language*) compliant, but now their XML results might convey not only raw data but rendering markup such as XHTML (known as "fragments" in the portlet parlance). This XHTML fragment is ready to be included within the consumer page. As for the navigation, now all interactions with a given portlet belong to the very same session, and hence, session and state management should be preserved along these interactions. Although different approaches exist, this can be the duty of the portlet producer, and hence, the consumer is relieved from the burden of complex and intricate session management and control flow. Figure 1 shows the *flightSearch* portlet when offered through a portal.

Portlets rest on two main standardization efforts: WSRP (*Web Services for Remote Portlets*) [28] and JSR168 (*Java Specification Requests: Portlet Specification*) [21]. WSRP standardizes the interfaces of the Web services a portlet producer must implement to allow another application (typically a portal) to consume its portlets. As for JSR168, it is a *Java Community Process* that standardizes an API for implementing local, WSRP-compatible portlets.

It is well-known in the component community that, the larger the component, the more reduced the reuse. Portlets tend to be statefull, 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 this data from the portal to the portlet producer. User Information Attributes Names 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) [42]. CSSs permit HTML fragments to parametrize some of its 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 is the portlet producer. 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 our *flightSearch* portlet. Its preferences can include *arrivalAirport* with values "San Sebastián", "London" or "New York", and *departureAirport* with values "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.

3 Outline: developing an *eXo* portal as an instance of the MDD development process

MDD strives to separate platform independent design from platform specific implementation and, in so doing, delaying as much as possible the dependence on specific technologies. The idea is creating 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 modeling and transforming. Consequently, MDD focuses on the construction of models, specification 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 Model-Driven Architecture (MDA) of the OMG [29]. MDA suggests building computational independent models (CIMs), platform independent models (PIMs), and platform specific models (PSMs) corresponding to different levels of abstraction or viewpoints. The computational independent viewpoint focuses on the environment of the system, and the requirements that stakeholders put on the system (details on the structure and processing are hidden or yet

undefined). The platform independent viewpoint focuses on the operation of the application while ignoring the details for a particular technological infrastructure. However, it specifies a complete, although abstract, application. Finally, the platform specific viewpoint refines the platform independent viewpoint by adding the characteristics for a specific platform.

This work focuses on PIMs and PSMs for portlet-based portals. To this end, three metamodels are introduced, namely,

- *WSRP* model, which is a PSM model of the interfaces defined by the WSRP standard,
- *SOP* model, which is a PIM model that promotes a service-oriented architecture also for 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* model, which is a PSM model 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, 5 and 6, respectively.

	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
Transf. into *eXo* model	PIM to PSM PSM to code	merge	metamodel	automatic	RubyTL

Table 1: Characteristics of model transformations in the SOP MDD process.

Transformations. Model transformation is the process of converting one or more models (a.k.a. source models) to one output model (a.k.a. the target model) of the same application [29]. Transformations are classified in automatic, semi-automatic and manual, based on the involvement of the designer during model transformation (for a complete account of transformation types refer to [8]). A transformation is automatic if it does not require user involvement. The transformation is semi-automatic if it is up to the designer to determine which elements of the source model will be transformed, and manual if the designer produces the results. Obviously, much effort should be being devoted to obtain automatic model transformations that generate complete applications without user involvement.

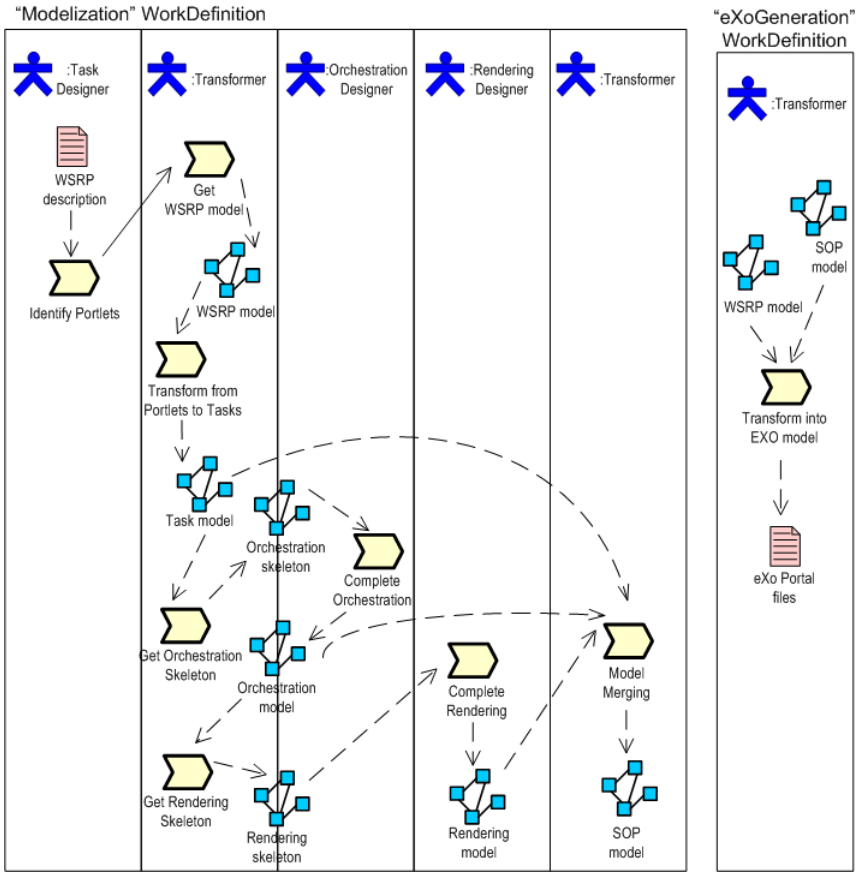

Figure 2: The SOP process.

Table 1 outlines the distinct transformations used in this work along the characteristics introduced in [22, 29]. Model transformations can be of type *CIM to PIM, PIM to PIM, PIM to PSM, PSM to PIM* and *PSM to code*. Our work use extensively *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 [26]. In our opinion, SOA also requires this *PSM to PIM* transformations to abstract from service descriptions into UML 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-compliant WSRP 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.

In order to describe the transformations, MDA proposes the use of the *Query/Views/*

Transformations Specification (QVT) [30]. However, to the best of our knowledge, current QVT implementations can not yet deliver the expressiveness required for some of the transformations described in this paper. Hence, we finally used *RubyTL* [38], a powerful, pattern-based transformation language based on *Ruby* [41]. Section 7 provides the details.

The MDD process. MDD conceives development as transformation chains where the artifacts 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 [32]. In MDD terms, SPEM is a metamodel for process modeling. Hereafter, SPEM terminology is used to specify the milestone, roles and dataflow that goes with producing an *eXo* portal from a set of WSRP-compliant portlets through a chain of model transformations (see Figure 2).

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 interwined; and the *rendering designer*, which focuses on the look-and-feel of the 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 . Interesting 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.

4 The WSRP model: a PSM for the WSRP interfaces

WSRP [28] standardizes the programmatic interfaces for portlets. Besides method signatures, WSRP 1.0 standardizes window states, CSS classes for portlet rendering, *P3P*-based user profile description, etc. For the purpose of this work it is important to note

that WSRP 1.0 treats portlets as isolated entities where interoperation between portlets occurrs beneath the GUI interface (e.g. sharing some data common to two portlets by using the so-called "portlet application")[9]. 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 portles are treated as isolated components.

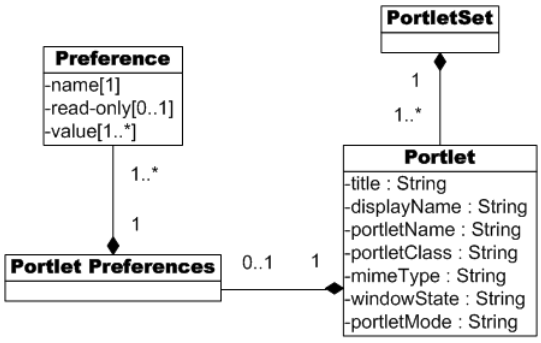

Figure 3: The WSRP metamodel.

For the purpose of this work, a metamodel is defined for WSRP portlet descriptions (see Figure 3). This metamodel builds upon the OASIS *ebXML Registry specifications for WSRP*. This ebXML [27] defines a framework for global electronic business that will allow businesses 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. Recently, portlets have been added as one of these offerings. To this end, metadata about the Portlet descriptions as well as the actual Producer Service Implementation WSDL needs to be published. Specifically, [3] proposes a way to publish of the following major WSRP artifacts: (1) WSDL description for a WSRP Producer service (2) metadata describing a Producer service and the Organization provides it (optional), and (3) metadata describing one or more WSRP Portlets hosted within Producer services (optional). For our purpose, Figure 3 just focuses on the latter. It represents the set of portlets and their description.

4.1 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;

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

- 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 [11], as it is the case for this sample problem.

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

Portal development platforms conceive portals as a compound of different types of artifacts (e.g., containers, content pages and portlets) where implementation depends on the vendor at hand. We depart from this vision and conceive portals as "universal integration mechanisms" where an increasing amount of their content comes from outside the portal itself. This brings a service-oriented perspective to portal conception where the portal is no longer perceived as a *set of pages* but as an integrated *set of services*.

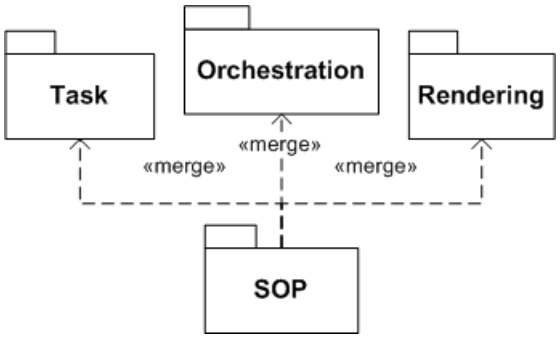

Figure 4: The SOP metamodel.

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 portal defines a workspace, i.e. a compendium of **front-end** tasks which are realized as portlets. For the purpose of this paper, portal modeling implies three viewpoints (see Figure 4):

- the *TASK model*, which describes the functionality, i.e. the set of tasks, that the 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 made available to the end user.

5.1 The TASK metamodel

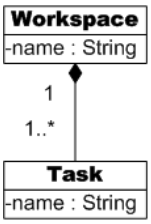

Figure 5: The TASK metamodel.

This model captures the portal as a workspace that aggregates the set of tasks that the portal will offer to the users (see Figure 5).

A TASK model for the sample case

The components in the WSRP model are abstracted as tasks. A backward transformer would take WSRP descriptions as inputs and return the corresponding TASK model. It is composed of six tasks whose names are *IEEESearch*, *ACMSearch*, *CiteSeerSearch*, *DBLPSearch*, *DeliciousStore*, *ISIWoK*, derived from the names of portlets. This sets the pieces for the *Browsing* portal.

5.2 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 artifacts to achieve a common goal. The peculiarities of the artifact (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 [35, 37]). 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 portal. Moreover, portlets tend to be statefull, and the interaction lasts for a whole session, rather than the simple request that characterizes the one-shot interaction of Web Services.

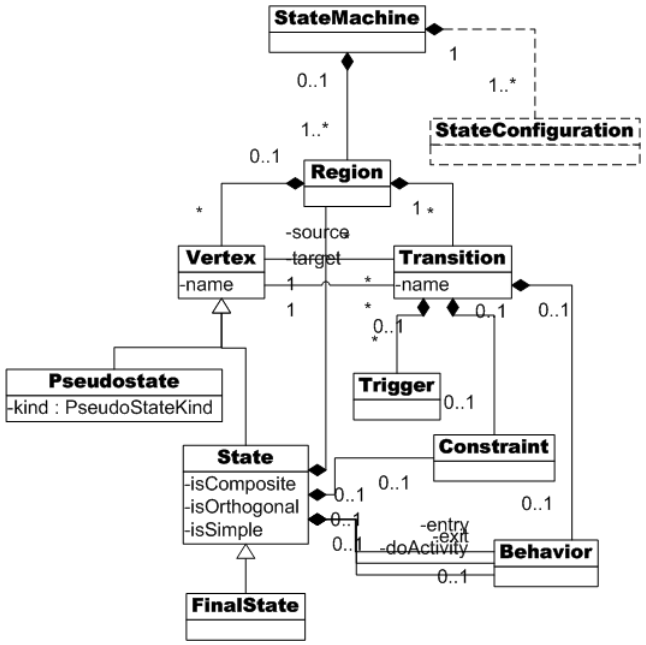

Figure 6: The ORCHESTRATION metamodel.

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

- *hypertext-like navigation style.* Portlet aggregation should permit users to explore the 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 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 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.

To accomplish these requirements, the *Hypermedia Model Based on Statecharts (HMBS)* [14] is adapted for our purposes. The use of statecharts [18, 17] or their predecessors, state-transition diagrams, is common for modeling hypermedia applications [14], Web

service composition [2, 6] and reactive systems. A hypermedia system, and hence, a 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.

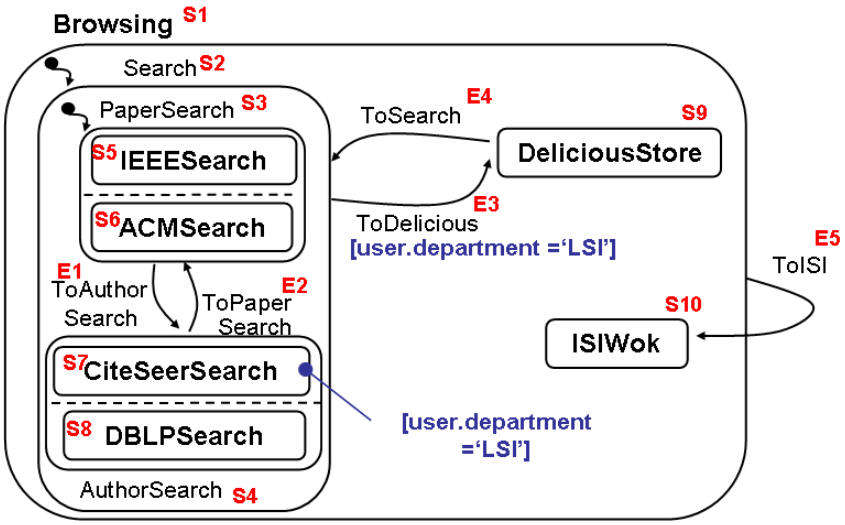

Figure 7: Statechart of the *Browsing* portal.

Therefore, statecharts are used to define the orchestration. The statechart metamodel proposed in the UML specification [31] is used as a base for the ORCHESTRATION metamodel (see in Figure 6 a simplified version, which has been extended with the notion of "*state configuration*"). Statechart extends 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 [33]).

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 till all tasks are enclosed in a root state which 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). Here, the former is addressed by permitting conditions on the user profile.

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 7 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 (but in this case, because of the condition in the statechart, *CiteSeerSearch* is only available if the user's department is LSI). Of note, the *ToDelicious* transition is only available for users of the LSI department. Therefore, *DeliciousStore* is only reachable for users with this profile.

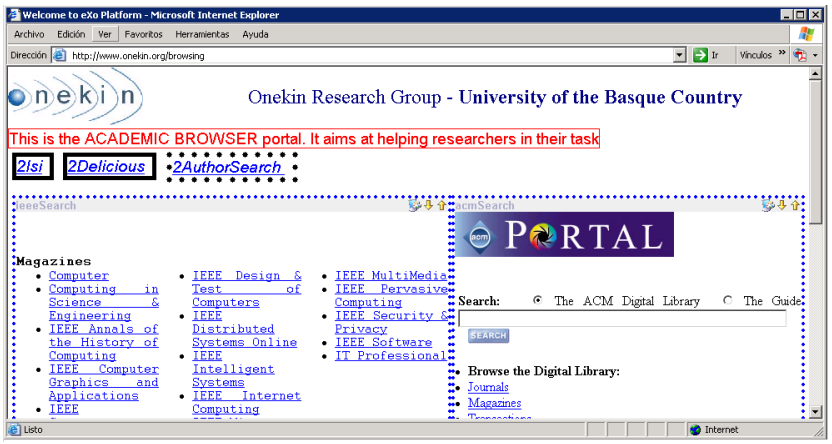

Figure 8: Presentation counterpart of the state configuration *{Browsing, Search, PaperSearch, IEEESearch, ACMSearch}*.

For the purpose of this work, it is important to recall the notion of *state configu-*

ration, i.e. the set of currently active states of the statechart [18]. 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 at Figure 7 are:

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

The importance of state configurations rests on being the PIM counterparts of portal pages (transformations will make this explicit). For our sample problem, Figure 8 shows the presentation counterpart of *configuration3*. The details of this transformation are given at section 7.

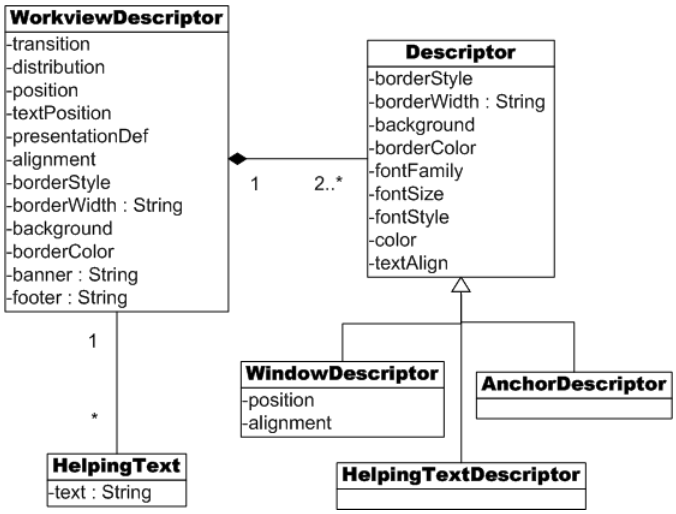

Figure 9: The RENDERING metamodel.

5.3 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 9). Additionally, the RENDERING metamodel contributes with the ***helpingText*** construct, a construct to complete the information shown in the 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. CSS files externalize presentation parameters such as *font family*, *font size*, *font style*, *color*, *background*, *border style*, *border width*, *border color*, etc. Among style parameters we include ***background-color***, ***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 indicates how anchors are realized. The 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* [13], Oracle Portal [34] or IBM's Web Sphere [20].

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 transition counterparts) 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 artifacts through *WindowDescriptor*, *AnchorDescriptor* and *HelpingTextDescriptor* (see Figure 9).

The simplicity of this metamodel comes from the fact that portlets free the 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 relationship among states and transitions, leaving outside what happens when in a given state .

According to the metamodel in Figure 9, 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 throughout the 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 state diagrams (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 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 it is still possible to completely override this skin for a particular simple state (e.g. *IEEESearch*). The 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 [16] "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 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

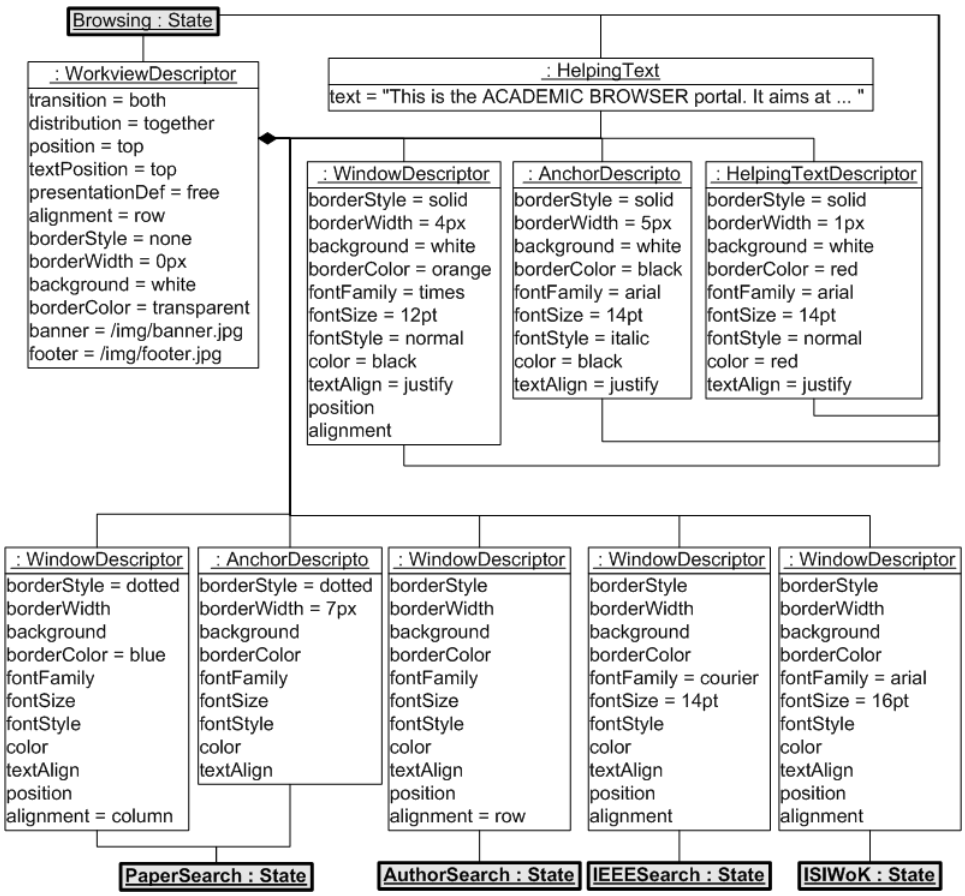

Figure 10: States and their rendering counterpart.

Figure 10 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 (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 skin description inherit rendering guidelines from the root skin. The portal page for the state *configuration3* is shown in Figure 8.

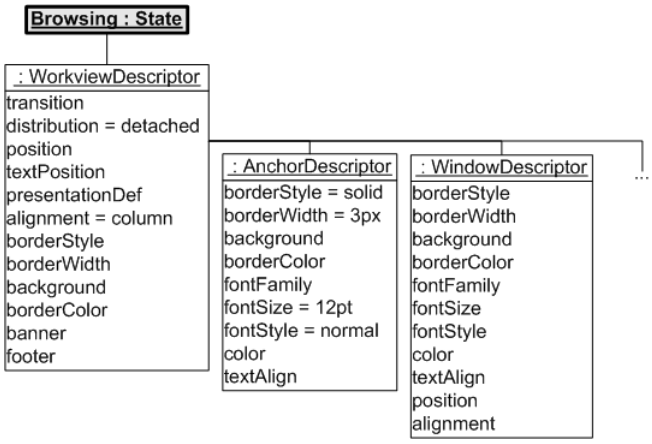

Figure 11: Piece of another alternative RENDERING model.

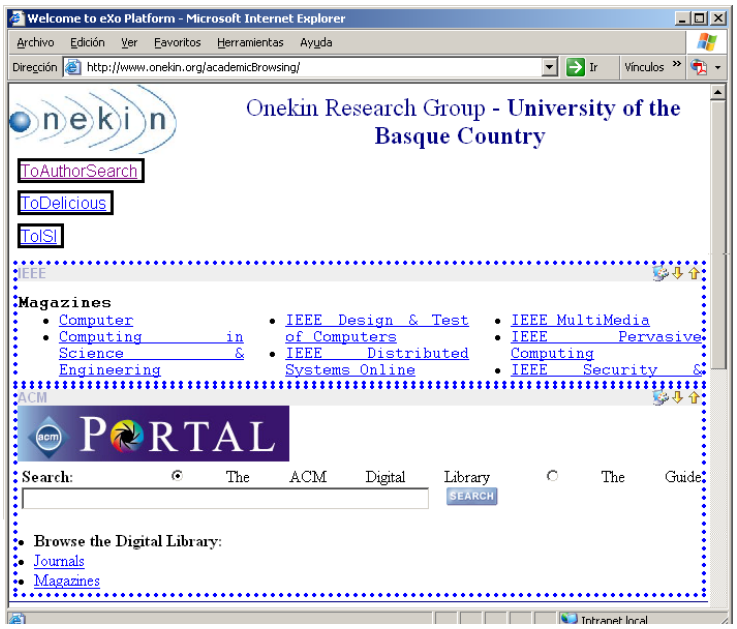

Figure 12: Alternative presentation counterpart of the state configuration *{Browsing, Search, PaperSearch, IEEESearch, ACMSearch}*.

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. The statechart in Figure 7 can have two alternative RENDERING models, namely,

those shown in Figure 10 and Figure 11 (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 12. Anchors are detached in different rows, and in normal *12pt* font with a *border line thinner* than before. Moreover the portal does not show any helping text.

6 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 *JSR168* certified in 2003, and it is currently one of the most popular, freeware portal frameworks. This work is based on *eXo version 1* [12].

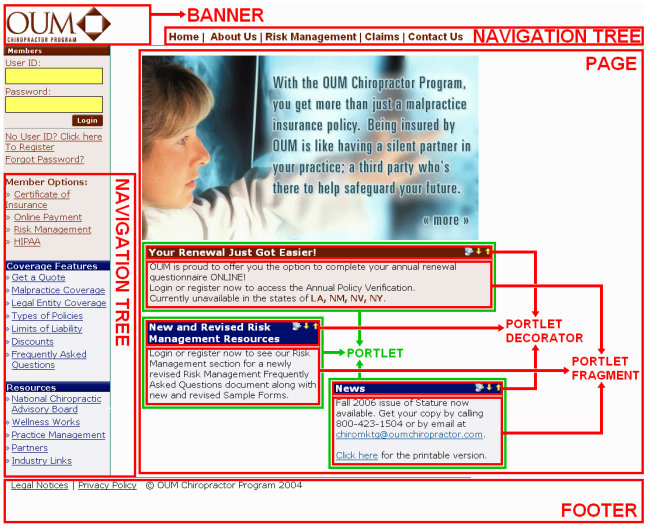

Figure 13: A sample *eXo* portal page.

An *eXo* portal is a compound of four main types of artifacts: 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 13 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 13: 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 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*.

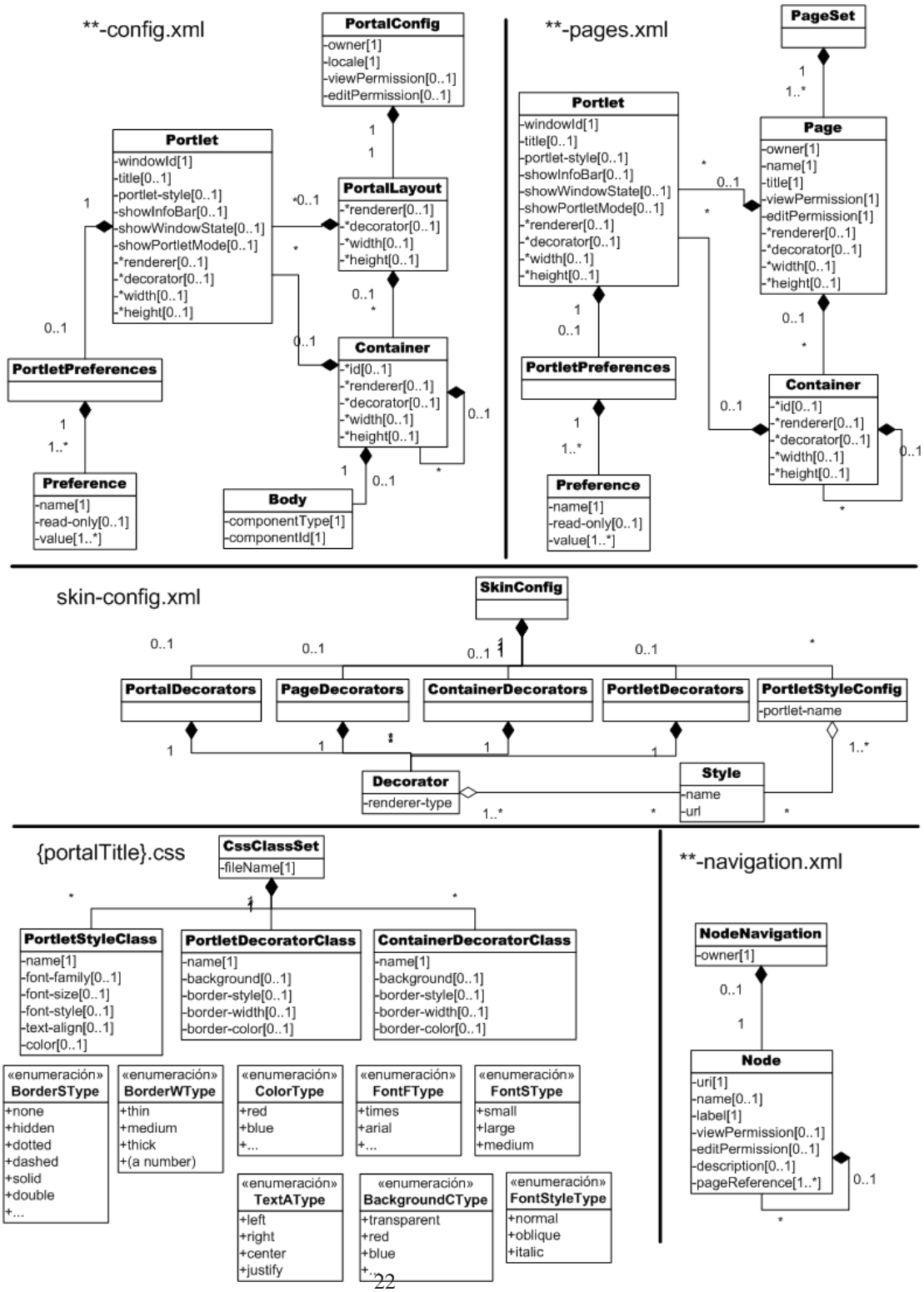

Figure 14: The EXO metamodel.

Both the *portal template* and the *page content* are specified as XML files, namely (see Figure 14):

- **-*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 13.

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 artifact (i.e. the portal itself, containers, pages and portlets), a decorator is defined for each type of artifact: *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 13). 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 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 14), with one subpackage for each file type.

7 Transformation definition

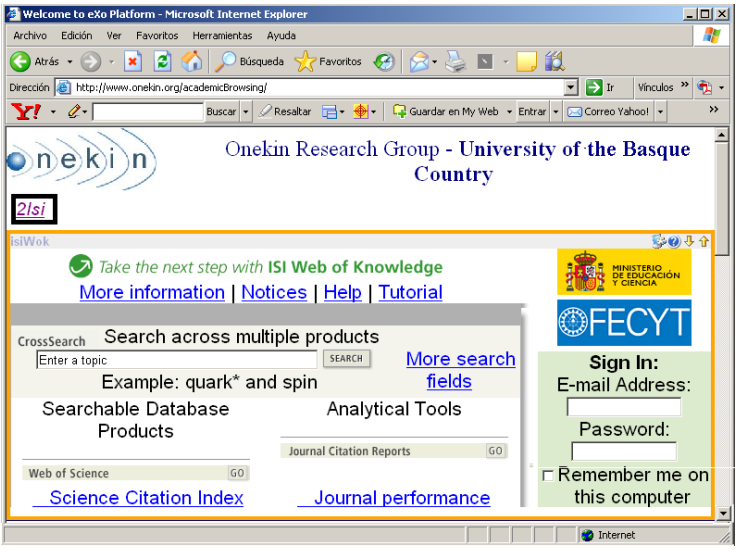

Figure 15: Presentation counterpart of the state configuration *{Browsing, ISIWoK}*.

Model transformation is the process of converting one or more models, called *source models*, to one output model, i.e. the *target model*, of the same system [29]. 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* [5]. 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 a statechart as an input, works out its state configurations and finally, outputs a set of pages conforming the *eXo* portal. Figures 15 and 8 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 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 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 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 should be 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 more akin solution 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 focus 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 recursive transformation rules that traverse the tree-like structure of the statechart. At the time of the implementation, popular transformation languages such as QVT [30] were not expressive enough to specify the required transformations, and we finally used RubyTL [38] as the transformation language.

Based on Ruby, RubyTL [38] is a domain-specific language for the domain of model transformations. It is a hybrid language that provides both declarative and imperative constructs to specify transformation definitions. A RubyTL rule includes the following clauses:

- *from*, where the constructs of the source metaclasses are indicated,
- *to*, where the constructs of the target metaclasses are specified,
- *filter*, which holds a condition over the source constructs for the transformation to be enacted
- *mapping*, which states binding relationships between source and target model constructs. A binding is a kind of assignment that indicates what needs to be transformed into what, instead of how the transformation must be performed.

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

Table 2: Mapping SOP constructs to *eXo* files through distinct transformation rules.

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 2).

7.1 Mapping from simple states to CSS classes

```
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 16: Mapping from simple state to *decorator* and *style* CSS classes in RubyTL.

Figure 16 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 and 6) and returns a *CssClassSet* element from the EXO metamodel (see Figure 14). That element represents a CSS file, which governs portlet presentation.

Portlet presentation includes the portlet markup itself and the decorator surrounding this markup (see Figure 13). The style guidelines for presenting the markup and the decorator are set through CSS classes, specifically, the *PortletStyleClass* and the *PortletDecoratorClass*, respectively (see Figure 14). These classes are obtained from the state skins distributed all along the state hierarchy.

The first rule has three bindings. The first binding uses the workspace associated to the root state of the statechart in order to name the *CssClassSet* element (remember, this element stands for 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 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 their ancestors. For instance, *ACMSearch* does not have any associated state skin whereas *IEEESearch* only provides the *fontFamily* (courier) and the *fontSize* (14pt) (see Figure 10).

```
.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 17: Snippet of *browsing.css* file.

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, where attributes at the upper states are overridden by lower states. The outcome for these two states is shown in Figure 17. Implementation wise, a helper function is defined that supports this look-up process (e.g. *windowDescriptor*). 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 or "inherited" from upper levels.

7.2 Mapping from state configurations to *eXo* pages file

```
phase 'page'
  parameter :configuration
  parameter :page_set
  rule 'Configuration2Page' do
    from SOP::Orchestration::StateConfiguration
    to EXO::Pages::Page
    filter { |conf| conf == configuration }
    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| configuration.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 = state.child_states.select { |s| s.isOrthogonal }
(8)     container.portlets = state.child_states.select { |s| s.generate_portlet? }.select { |s| configuration.states.include?(s) }
    end
  end
  rule 'SimpleState2Portlet' do
    from SOP::Orchestration::State
    to EXO::Pages::Portlet
    filter { |state| state.isSimple }
    mapping do |state, portlet|
        portlet.renderer = "PortletRenderer"
        portlet.decorator = state.name + "PDecorator"
        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
(9) filter { |configuration| configuration.state_machine.workviewDescriptor.distribution=="together"}
    mapping do |configuration, container|
        container.renderer = "ContainerColumnRenderer"
        container.decorator = configuration.state_machine.workspace.name + "TransparentDecorator"
(10)    container.portlets = configuration.states.map { |s| s.source_of_transition }.flatten
    end
  end
  rule 'Transition2Portlet' do
    from SOP::Orchestration::Transition
    to EXO::Pages::Portlet
    mapping do |transition, portlet|
      portlet.renderer = "PortletRenderer"
      portlet.decorator = transition.state_machine.workspace.name + "AnchorDecorator"
      portlet.portletStyle = transition.state_machine.workspace.name + "AnchorStyle"
      portlet.windowId = transition.portlet.displayName + "/" + transition.portlet.portletName + "/" + transition.name +
                  TransitionId.next.to_s
     ...
    end
  end
 ...
end
```

Figure 18: Mapping from state configurations to *eXo* pages file. Anchor rendering strategy is *together* and *top.*

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

Figure 19: Mapping from state configurations to *eXo* pages file. (cont.)

As stated previously, this work 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, 18 and 19, for the sake of legibility.

The transformation starts in line (11) of Figure 19 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 (12) creates new *pageSet* object.

Line (14) 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 18 has a *configuration* 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 (13)) works out all possible state configurations from the statechart model. For each configuration, the *page* phase is explicitly executed (in line (14)), passing the current configuration as its IN parameter.

The *page* phase starts with the implicit triggering of its first rule, i.e. *Configuration2Page* rule (see Figure 18). 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 place left, right, up or down relative to the page/portlet. This leads us to eight different types of transformations. In Figure 18, 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 (9)), and after, one anchor is de-

scribed for each transition, i.e using the *Transition2Portlet* rule triggered in line (10) with the binding. Of note, rules for the detached case are also recursive as anchors are placed along the containers hierarchy.

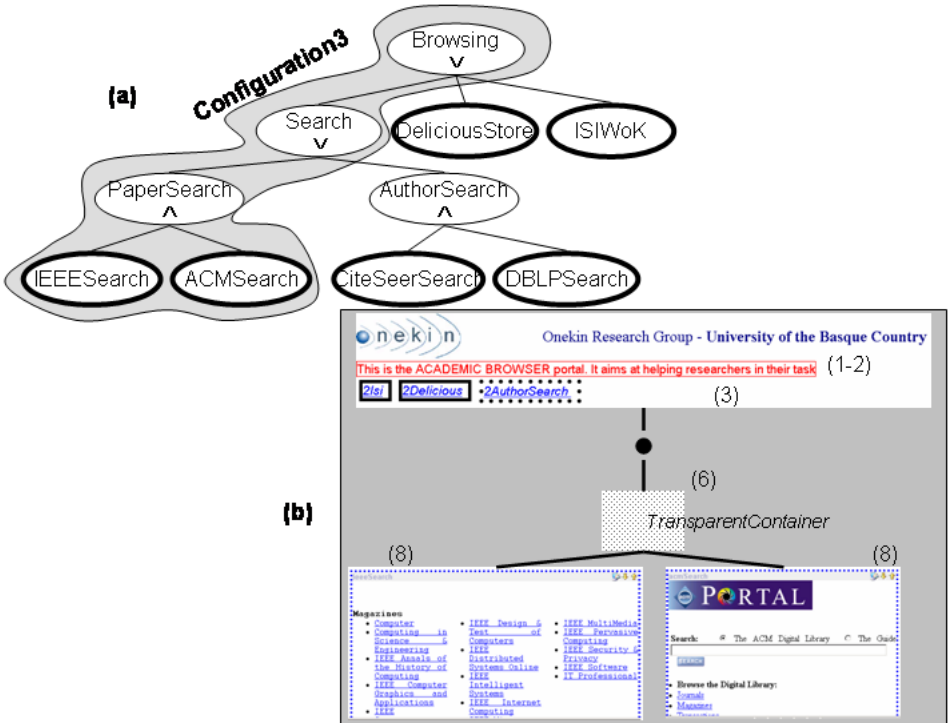

Figure 20: *Configuration3* and its presentation counterpart.

As an example, consider *configuration3* whose active states are *{Browsing, Search, PaperSearch, IEEESearch, ACMSearch}*. Figure 20(a) shows the containment relationship between these states, indicating the kind of relationship (i.e. AND, OR). Transformation proceeds from top to bottom and Figure 20(b)[2] shows the generation of the presentation counterpart that results from this state configuration whose complete presentation page is shown in Figure 8.

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 18). Moreover, since anchors have a *together* distribution and a *top* position, PSM anchors are displayed at this very top decorator (line (3) of Figures 18 and 20(b)). This container also holds the results of transforming the *Browsing*'s child, i.e. *Search* state.

[2]Numbers in Figure 20(b) point to operations in Figure 18. These operations generate the corresponding PSM widgets.

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 reference to a *PortletDecorator* class and a *PortletStyle* class that hold CSS parameters for portlet markup presentation *(*this is the *SimpleState2Portlet* transformation rule).

```
<page-set>
 <page renderer="PageRowRenderer" decorator="browsingPageDecorator">
  <name>/home</name>
  <container renderer="ContainerColumnRenderer" decorator="browsingTransparentDecorator">
   <portlet renderer="PortletRenderer" decorator="browsingAnchorDecorator">
    <portlet-style>browsingAnchorStyle</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 21: Snippet of the *template-pages.xml* file.

The final outcome for *configuration3* is depicted in Figure 21. The content of *decorator* and *style* classes has been previously generated by the transformation rule introduced in subsection 7.1.

It is worth emphasizing that RubyTL is an embedded language in Ruby. The benefits brought are illustrated in line (12) of Figure 19, where an object is explicitly created, and in line (5) of Figure 18, 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 (12)).

8 Realizing the MDD benefits

Different papers address the advantages of MDD in general [36], and to Web development, in particular [16]. 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 [24]. Indeed, statecharts have long being used as a simulation technique, and distinct techniques and tools are available to validate distinct formal properties.

```
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 22: Validation rule.

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 22). This constraint can now be validated against the portal 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* artifacts!!

Design. Transformations express a correspondence between elements of the source

model 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 where 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 7). 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 distinct times during the *eXo* portal development. For instance:

- the same portlet can appear in distinct pages (i.e. an simple state can belong to different state configurations). A page description (i.e. the *template-pages.xml* file in Figure 21) 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. outcoming 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 that focus most of the care and testing. Hence, the chances for implementation pitfalls are reduced and development is streamlined.

9 Related work

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 3: Models for data-intensive portals vs. Models for service-oriented portals.

This paper is about integration in a Web setting. 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 [15, 7, 23], 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 object 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 artifacts will be presented to the user; and *the dynamic presentation model*, which addresses the behavior of the presentational objects, i.e. the changes on the user interface when the user interacts with the system [19]. Table 3 indicates a tentative mapping between these models and the ones introduced in this paper.

These works 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 characterize traditional database development. In the past, data modeling carried an expectation of unification -a prerequisite for effective communication and data sharing was the agreement of a single common data model. However, this is no longer true in a service-oriented approach where 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.

An eclectic approach is taken in [25] for WebML, a Web modeling language that contemplates the interplay between Web applications and Web services. Here, the navigation model is enhanced to describe calls to Web services, capturing the relationship between the invocations and the data units which provide their inputs, and, respectively, capture their outputs. Moreover, Web services can be engaged in conversations. A WebML conversation *"is a collection of Web service operations, belonging to one or more Web services, used together to achieve a given application goal and orchestrated by a Web hypertext"* [25]. From this perspective, WebML's conversations play a similar role to our workspace in the sense that conversations are meaningful aggregations of lower tasks. The difference stems from conversations exhibiting a more conductive approach that the explorative nature of workspaces. Also the starting point is differ-

ent. WebML is thought for data-intensive Web applications, and hence, the conceptual model is the starting point on which the rest of the perspectives are constructed. By contrast, our approach does not address Web applications in general but Web portals in particular. In this setting, a component-based approach is more akin to the notion of the portal as a doorway to other applications. Hence, our approach is *process-centered* rather than *data-centered*. Process-centered approaches are predominant when integration happens at the middle-tier.

Integration at the middle-tier is first exhibited by Workflow Management Systems which 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* [1]. 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. A first frame displays the actual to-do-list of a user. A middle frame presents the workspace containing an HTML document describing the activity. In the bottom frame, buttons allow submission of the completion status of this activity. Unlike our approach, only one activity is available at a time, and all the flow dependencies are enforced at the back-end.

A recent extension of WebML is also contemplating process-intensive applications [4]. 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. In the case of WebML this leads to introduce two additional constructs to the navigation model, namely, the *If* unit and the *Switch* unit as control-flow constructs. It is worth noticing that 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).

This work 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.

Finally, Web applications as conglomerates of presentation components is a more elusive subject. It has been recently addressed in [43] where a framework is introduced for integrating components by combining their presentation front-ends. A composite application consists of one or more components, a specification of the composition

model (i.e. integration logics that coordinate the components at runtime) and a middleware for the execution of the component. By using adapters the model can integrate heterogeneous components. The work stress the importance of an event-driven architecture where the composition model includes event subscription information to facilitate the communication among presentation components. In addition, the composition model may contain additional data transformation logics via XSLT and integration logics in the form scripts or references to external code. Our work share similar aims but focuses on portlets. That is, (1) the component model considers portlets, (2) the composition model is described through statecharts where events are restricted to anchor navigation, and (3) the layout model benefits from the statechart hierarchical structure. The work at [43] can be considered a Domain-Specific Language for composite Web applications. By contrast, our approach attempts to build upon existing formalisms (e.g. statecharts) rather than coming with a new language, and then, map this formalism to concrete frameworks where the one of [43] 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 paper, facilitates easy platform portability and user adoption by building on existing (meta) models.

10 Conclusions

The recent release of the WSRP and JSR168 standards promises 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 work illustrates the use of MDD techniques to achieve such scenario where portal construction is now a pipeline-like process of model transformations. Specifically, the designer abstracts away from portal pages, and describes portal behavior in terms of statecharts (i.e. HMBS statecharts) that are then gradually realized as *eXo* artifacts. The paper 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 portal.

So far, our implementation does not take the full potential of ebXML registries [27], and portlet selection is conducted manually rather than through queries to the ebXML registry. However, future work includes dynamic selection of portlets from third parties similar to the one provided from traditional Web Services. Yet, this still requires the portlet market to mature.

Another follow-on is extending state diagrams to contemplate portlet events. WSRP 2.0, introduces an event-based mechanisms where portlets can subscribe to events being generated by other portlets. In this way, events can be risen by either GUI widgets or the portlets themselves when they reach a certain state. Our insights is that our approach can accommodate the novelties brought by WSRP 2.0

Acknowledgments. This work was co-supported, on the one hand, by the Spanish Ministry of Science & Education, and the European Social Fund under contract TIC2005-05610, and on the other hand, by Séneca Foundation (Murcia, Spain) with the grant 05645/PI/07.

References

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

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

[3] J. Blattman, N. Krishnan, D. Polla, and M. Sum. Open-Source Portal Initiative at Sun, Part 2: Portlet Repository, 2006. at http://developers.sun.com/prodtech/portalserver/reference/techart/portlet-repository.html (July 2007).

[4] 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.

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

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

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

[8] 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.

[9] O. Díaz, J. Iturrioz, and A. Irastorza. Improving Portlet Interoperability through Deep Annotation. In *14th International Conference on World Wide Web (WWW'05)*, pages 372–381, New York, NY, USA, 2005. ACM Press.

[10] O. Díaz and J.J. Rodríguez. Portlets as Web Components: an Introduction. *Journal of Universal Computer Science*, 10(4):454–472, Apr 2004. http://www.jucs.org/jucs_10_4/portlets_as_web_components.

[11] 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 (SAINT'05)*, pages 31–37. IEEE Computer Society, 2005.

[12] eXo. Exo Platform v1. at http://docs.exoplatform.org/exo-documents/exo.site/index.html (October 2007).

[13] Exo. Exo Community, May 2006. at http://www.exoplatform.org/ (October 2007).

[14] 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.

[15] 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.

[16] 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.

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

[18] 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, 1987.

[19] 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*, 2001.

[20] IBM. WebSphere. at http://www.ibm.com/websphere (October 2007).

[21] Java Community Process. JSR 168 portlet specification, October 2003. at http://www.jcp.org/en/jsr/detail?id=168.

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

[23] 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.

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

[25] I. Manolescu, M. Brambilla, S. Ceri, S. Comai, and P. Fraternali. Model-driven design and deployment of service-enabled web applications. *ACM Transactions on Internet Technology (ACM TOIT)*, 5(3):439–479, August 2005.

[26] N. Moreno and A. Vallecillo. A Model-Based Approach for Integrating Third Party Systems with Web Applications. In *Proceedings of the 5th International Conference on Web Engineering (ICWE'05)*, pages 441–452, 2005.

[27] OASIS. Electronic Business using eXtensible Markup Language (ebXML). at http://www.ebxml.org/ (July 2007).

[28] OASIS. Web Service for Remote Portlets Specification Version 1.0, 2003. http://www.oasis-open.org/commitees/tc_home.php?wg_abbrev=wsrp.

[29] Object Management Group (OMG). MDA Guide Version 1.0.1.omg/2003-06-01, june 2003. at http://www.omg.org/docs/omg/03-06-01.pdf (April 2007).

[30] Object Management Group (OMG). MOF QVT Final Adopted Specification /2005-11-01, 2005. at http://www.omg.org/docs/ptc/05-11-01.pdf (July 2007).

[31] Object Management Group (OMG). Unified Modeling Language: Superstructure, August 2005. at http://www.omg.org/cgi-bin/doc?formal/05-07-04 (April 2007).

[32] Object Management Group (OMG). Software Process Engineering Metamodel, version 1.1 SPEM, January 2007. at http://www.omg.org/technology/documents/formal/spem.htm.

[33] A. Olivé. *Conceptual Modeling of Information Systems*. Springer, October 2007.

[34] Oracle. Oracle Portal. at http://www.oracle.com/appserver/portal_home.html (October 2007).

[35] 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. Springer-Verlag, June 2002.

[36] B. Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003.

[37] 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.

[38] 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.

[39] H. Stern. Second-Generation Portals. *Web Services Journal*, pages 34–35, December 2001.

[40] The Delphi Group. Portal Lifecycle Management: Addressing the Hidden Cost of Portal Ownership, 2001. at http://www.mongoosetech.com/downloads/ portal_ownership.pdf.

[41] D. Thomas. Programming Ruby. the Pragmatic Programmers' Guide, 2004. at http://www.rubycentral.com/book/ (April 2007).

[42] W3C. Cascading Style Sheets (CSS). at http://www.w3.org/Style/CSS/ (April 2007).

[43] J. Yu, B. Benatallah, R. Saint-Paul, F. Casati, F. Daniel, and M. Matera. A framework for rapid integration of presentation components. In *Proceedings of the 2007 International Conference on the World Wide Web (WWW'07)*, pages 923–932, May 2007.