Separating Business Process from User Interaction Utilizing Process-Aware XSLT Style-Sheets

Karl Aberer, Anwitaman Datta, Zoran Despotovic
Swiss Federal Institute of Technology (EPFL)
1015 Lausanne, Switzerland
{karl.aberer, anwitaman.datta, zoran.despotovic}@epfl.ch

Abstract

In the web context, it is difficult to disentangle presentation from process logic, and sometimes even data is not separate from the presentation. Consequently, it becomes crucial to define an abstract model for business processes, and their mapping into an active user interface presentation, using the principle of separation of concern between the process logic, data and its presentation aspects. We endeavor to extend declarative (rule based) XSLT to accommodate the separation of process information from the data structure and presentation, and thus propose to design process aware stylesheets, in a minimally invasive manner. The isolation of the three otherwise entangled aspects of web-processes makes it easy to develop and maintain web-applications in a more independent manner, where each individual developer can focus on his/her primary responsibility, like describing the process, maintaining a database, or creating user interface (web-pages) without being levied any substantial effort to learn new technology.
Keywords. *Web-based information systems, e-commerce systems, user interface, XSLT, workflow management, information commerce.*

1 Introduction and Motivation

One of the main reasons for the success of XML, as well as its predecessor SGML, is the separation of document structure from document layout. In this way, the logical structure of the document content or the document data is represented in a layout-independent format. The generation of the layout, e.g. in HTML, is performed by specialized document processing languages, like XSLT [3]. This approach has the well-known advantages of separation of concerns: development and processing of the data can make use of application-specific data schemas ("application markup"). Thus changes in the presentation do not affect the logical structure of the documents (a form of data independence). Multiple layouts for different media can be generated from the same logical documents.

Considering the function of a service-oriented Web site, like an e-commerce site, we must be aware that it goes beyond the presentation of document content and data. In fact it represents an active document supporting interactions among users and backend systems. There exist nowadays various implementation approaches of such dynamic Web pages, like CGI scripts, active server pages, servlets and JSP, just to name the most important ones. The business processes that are implemented using these techniques are hidden in the application code (and often the logical data structures as well). On the other hand the area of workflow management systems has demonstrated that an abstract representation of complex business processes is extremely beneficial in their development, maintenance and implementation. Also here we find an independence principle realized, namely the independence of the process structure, usually expressed in terms of an abstract process model, like Petri-Nets or state charts, from the implementation of the processes, by associating various system and user resources with the activities to be performed.

This brings us to the question, whether the independence of the implementation of the active parts of a Web page, in terms of what interactions a user may perform within a business process, from the actual abstract specification of the process structure would not be a valuable goal to achieve and if yes, how it may be reached. This is the problem that we intend to address in this paper. We will show that such a separation of concerns is indeed possible and that it can be achieved in fact, in a Web context, with minimally invasive extensions of existing approaches, in particular the one of XSLT. In other words, we will extend XSLT in a way, such that both process and data structure can be transformed into active Web pages. Doing so, we also intend to maintain the declarative nature of XSLT, allowing a rule-oriented specification of the layout-oriented transformations.

We will achieve, following the principle of orthogonality, a separation of concerns along two dimensions:

- Content and process: Web applications consist of the content presented and the process underlying the inter-

action with the user. Only the ensemble of the two allows to fully characterize a service-oriented Web site.
- Structure and presentation: Structure provides a logical view of the application, whereas presentation is concerned with the user interaction. For both, content and process, we want to achieve independence of structure and presentation.

Separation of concerns facilitates the development of service-oriented Web sites by making it more modular. Changes that affect one concern do not affect the other ones. Examples would be:

- changes to the user interface do not affect the logical representation of the data as well as the processes.
- changes to the process logic (e.g. the control flow) can be made independently of the specifications of the layout transformations, and even can be done without affecting existing transformations (and similarly for the data structures, as it is known for XSLT).
- control of the process flow (workflow logic) and generation of user interface layout can be independently distributed as required (client, server).
- developers can concentrate on the aspects they are interested in (process structure, data structures, layout). When developing the process and data structures, layout aspects can be ignored, and when developing the layout only a partial understanding of the structured specification is required (e.g. only activity types but not their detailed dependencies). Also using more abstract models for specifying the structural models facilitates their understanding.

Declarative specifications increase the modularity of specifications and allow for increased ease of the evolution in applications. Thus we will follow the XSLT paradigm. We will pay attention to avoid any dependencies on heavyweight infrastructure, like application servers. Our architecture will consist mainly of an XSLT interpreter that is extended by a process control component. It will be lightweight, making implementation of complete applications, like C2C business processes, which are purely client-based, perceivable. Hence our approach will also be an ideal platform to implement business processes in P2P environments.

The rest of the paper is organized as follows. In Section 2 we discuss related work, then in Section 3 we provide a generic overview of basic abstractions we devised. Section 4 contains the description of the workflow model we use. Section 5 introduces details of the implementation architecture explained with an illustrative example in section 6. Implementation related details are given in Section 7. Section 8 concludes the paper with a summary and indicating the scope of future work.

2 Related Approaches

After it started as a network-based information system offering only static content (pure HTML), the web has now become a platform for highly sophisticated applications delivering dynamic content to its users. In this section we discuss different solutions that enable dynamic behaviour of the web - both commercial solutions as well as some of the latest research on that issue. In particular, we will evaluate these approaches with respect to the separation of concern principles that we have addressed in the introduction.

The separation of the content from the presentation does not seem to be a major problem nowadays. The appearance of XML is usually (and wrongly) tied with solving this problem. Essentially, any Web-DBPL integrator tool (such as Microsoft's Internet Database Connector (IDC) or Allaire's Cold Fusion Web Database Construction Kit) or any Web database publishing tool (Microsoft Access is a typical example) provide a solution for this problem. No matter what kind of database connectivity they use (proprietary or not) they can be looked at as belonging to a large group of scripting (or programming) language based HTML extensions, main representatives of which are ASP or JSP as well as Java servlet based solutions. Also a number of research efforts were directed to support the design of data-intensive Web sites. However user interaction is considered there for the purpose of data navigation only (see for example [4] for a comprehensive overview).

But the real problems start when it comes to the separation of the business processes from the presentation. Let us discuss some relevant solutions.

The first group of solutions - CGI scripts and servlets[1] - does not offer anything to help developers separate the business processes from the presentation. With servlets for example, the entire page must be composed in a servlet. More precisely, HTML tags and presentation code are embedded within Java code. The most important consequence is that if a developer wanted to make any change in the appearance of the page he would have to edit and recompile the servlet, which obviously presents not only a development but also maintenance and evolution nightmare.

A considerable improvement was brought by server side scripting solutions - ASP and JSP in the first place. Essentially, JSP [6] offers a possibility to encapsulate the application logic into software components (JavaBeans) with well-defined interfaces expressed in XML (tag libraries) that can be easily used by the page designers. Thus, application logic is separated from the presentation and both application logic and presentation can be changed without affecting each other. But there is still one problem. JSP does not support process definition on a high abstraction level which makes it difficult (if not impossible) to specify process-aware UI mapping with JSP. Client side scripting can help here but such solutions become cumbersome even for simple business processes.

[1] Servlets are usually considered in literature as a replacement for CGI because they are in many aspects a better solution. From the perspective of the problem that we are considering here, however, they belong to the same group.

An interesting approach that bears some similarity to ours is the WSUI initiative [2]. WSUI defines a web component model that couples network services with interaction and presentation. The presentation of a component is defined as a set of XSLT stylesheets that work independently of the business logic of the component, which essentially means that, as with JSP, the second of our two objectives is achieved. But there are some unnatural limitations of the approach taken there. Namely, the layout of the components is not defined in terms of the state the component can be in, but in terms of events that it can respond to.

Diaz et al [8] present an interesting approach that advocates the same principles as ours. They essentially provide a possibility to define a process of WSUI components. Compared to our approach, the main difference is the fact that they start in the design from active HTML pages and bind to those the process logic explicitly, whereas we intend to generate those pages from a rule-based specification. Thus our approach is more flexible and efficient in particular when generating multiple layouts for different platforms.

3 Abstract Process Description

The separation of the business process logic from the implementation of the activities is usually considered as the main contribution of the workflow management technology. The business process is specified at a high level of abstraction, using either GUI tools or workflow definition languages. The main result of this step is the definition of control and data flow dependencies among activities. Activities themselves are defined separately and their definition is not a part of the process definition. In runtime a workflow engine invokes the activity implementations according to the dependencies defined in the process specification. The workflow management approach ascertains that process definition and the implementation of the activities can be maintained and changed independently.

3.1 The Role of Interactive Web Documents in Workflow Implementation

One possibility to implement a workflow activity, that is performed interactively by a human, is to provide an interactive Web document (a "Web page"). The document allows a user to provide workflow-relevant data and thus decide on the further process execution. Interactions of the user initiate the return of workflow-relevant data to the workflow engine and indicate the completion of activities.

On the other hand interactive Web documents serve the purpose of data presentation to an interactive user. If the interactive Web document is implementing a workflow activity data is typically related to the purpose of this activity, e.g. by presenting data relevant for making decisions. Users can also interact with the interactive Web document in order to navigate through the data as it is presented. By using a mechanism for layout generation (like XSLT) data structure is separated from layout for presentation and navigation.

Thus an interactive web document clearly plays two roles in the implementation of workflow activities: with respect to separating content from presentation it serves as presentation medium, and with respect to separating activity implementation from workflow structure it serves as activity implementation. This double role should also be reflected in the mechanisms used to generate interactive Web documents. XSLT addresses only generation of presentation format from structured data. We intend to extend XSLT to also generate interaction capabilities for activity implementation, in order to separate process structure from activity implementation.

3.2 Implementing Workflow Activitites using Interactive Web Documents

The way data is presented to a user within an Ecommerce application depends on the current state of the workflow process. Therefore we have to provide for the UI developer using XSLT a means to express this dependency in the stylesheets he/she is developing. On the other hand when implementing a workflow activity an interactive Web document must be enabled to interact with the workflow engine. Therefore we have to extend XSLT also to include interactions with the workflow engine. It may seem that, with such an extension of the stylesheet functionality, we are sacrificing the ease of the UI creation process and putting more burdens on the UI designer. However, we do not think that this is the case if we provide the UI developer with clear and simple abstractions of the workflow and use simple constructs for interacting with the workflow engine.

We can identify two main ingredients of a workflow: activities (with dependencies among them, control flow) and the data passed between the actions (data flow). Additional concepts such as participants (whether human beings or IT applications) are used to further determine the concept of activities. Thus, the UI designer needs first of all to understand about the workflow, the type of activities occuring and the workflow-relevant data. For the activities this requires knowledge of the activity names and invocation parameters. This gives a set $B = \{b(x_1, x_2, \ldots, x_k) \mid b \text{ is a business activity}\}$. For the workflow relevant data this requires the knowledge of workflow-relevant data objects $D = \{d_1, d_2, \ldots, d_n\}$. What a UI designer doesn't need to bother about are the details of the process definitions, i.e. the different data and control flow dependencies.

Based on this knowledge he must be able to interact with the workflow engine at least in the following ways.

1. Initiate the performance of business activities in B. For doing that he must in particular be able to provide values for the invocation parameters of the activities

from set B derived from the earlier interactions with the user or the WF engine.

2. Obtain and manipulate the values of workflow relevant data in D. This data is required for the implementation of activities.
3. The evaluation of conditions on the process state. These are predicates $C = \{p(x_1, x_2, \ldots, x_k)\}$ where p are (Boolean combinations of) predicates on the activity execution state. The most basic predicates on the activity execution state is to determine whether an activity has been executed or not. Depending on the activity state model of the WF model this can be refined to take into account more activity states.

For a more fine-grained interaction of the interactive Web document with the workflow engine additional functionality can be provided, for example, functions for analysis of workflow history. The specific instantiation of these interactions depends on the underlying workflow model and the capabilities of the workflow interpreter used. More on this will be given in Section 4, when we describe the specific workflow model that we use and its semantics.

3.3 Process-Aware XSLT Style Sheets

We give now a pattern of how to extend the XSLT style sheet mechanism in order to make it interacting with a workflow process. It covers the interaction possibilities a UI designer needs that have been mentioned before. This pattern should be applicable for most kinds of workflow models and interpreters.

Conditional sections of style sheets depending on the workflow process and data state. This enables the UI designer to make the document rendition dependent on the workflow execution. Syntactically we represent this as a new element type

<xslt-wf:for-each select="Rep($p(x_1,\ldots,x_k)$)">
... </xslt-wf:for-each>[2]

The representation of a predicate $p(x_1, \ldots, x_k)$, determined by the function Rep, depends on the specific implementation. We will give later an example were we use for the sake of compatibility with XSLT an XPath-oriented representation. The important point is that a predicate may be true for multiple instantiations of the parameters $x_1, \ldots, x_k$. In such a case, as for ordinary XSLT templates, the content of <xslt-wf:for-each> is instantiated for every instantiation of the parameters once. The order of these instantiations is implementation-dependent. We may also just test whether a predicate is satisfied for specific parameters by using

<xslt-wf:if test="Rep($p(x_1,\ldots,x_k)$)">
...</xslt-wf:if>

[2] We use the namespace prefix xslt-wf only in this generic description section. Each specific workflow model will have its own namespace prefix.

The subelements of <xslt-wf:if> are then included when there exist parameters, for which the predicate evaluates true. In contrast to the use of <xslt-wf:for-each> and the use of <xsl:if> in ordinary XSL we will also allow a template rule to be contained within an <xslt-wf:if> element, such that the rule becomes only applicable if the test predicate is satisfied in the workflow state.

Retrieval of data from workflow state. When processing an <xslt-wf:for-each> element the processing clearly will depend on the values of $x_1, \ldots, x_k$. Therefore we provide within the scope of the element these values as implicitly defined XSLT variables with name Rep(x_i), i.e. it is available as if it were defined as

<xsl:variable name="Rep(x_i)" select="..."/>

and takes the value of the parameter for which the template condition is evaluated to true. For retrieval of other data from the workflow state we also may explicitly retrieve these values into variables by

<xslt-wf:variable name=".." select="Rep(*query*)"/>

or inline it directly into the resulting document by

<xslt-wf:value-of select="Rep(*query*)"/>.

In both cases the expressiveness of expressions $query$ and their representation in XSLT by Rep($query$) depends on the concrete implementation of the mechanism. Natural examples of languages for Rep($query$) are the use of XPath and XQuery.

Execution of a workflow action as part of a user interaction. This enables the UI designer to couple active elements of the user interface (like forms and links) with workflow actions. Syntactically we represent this by introducing a new element type:

<xslt-wf:submit action="$b(x_1,\ldots,x_k)$"
actionElement="ID">
... </xslt-wf:submit>

The submit element binds the execution of the action b to an HTML form element that is identified by identifier ID and that is contained within the element. We make here use of the XML ID/IDREF mechanism for linking. The variable values $x_1, \ldots, x_k$ are either taken from the referenced action element (e.g. HTML form) or from an XSLT variable, either explicitly defined or implicitly defined by the <xslt-wf:for-each> element.

4 Business Offer Language (BOL)

4.1 BOL Overview

We use the Business Offer Language, BOL [1, 10], to encode our process description. BOL is a flexible rule-based abstraction to specify business processes. It is particularly well suited for information commerce as its main abstraction primitive is the exchange of an information good. A

complete description of BOL and BOL encoding is neither the purpose nor scope of this paper. Here we describe the basic concepts of BOL tailored to suit the understanding of rest of the paper.

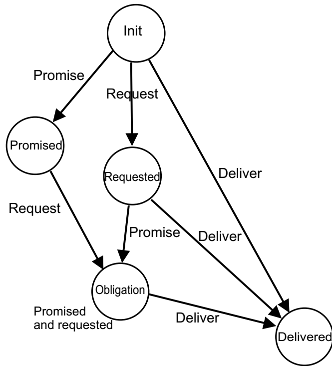

Figure 1: State transitions diagram

Four notions play the central role in BOL - goods, roles, actions, and rules.

The notion of **information good** includes all goods that are exchanged electronically in a given information commerce business process. Possible examples range from registrations or newsletters, to payments and certificates. Each information good is generally characterized by a set of attributes or parameters: $\mathtt{G}(\mathtt{p}_1 : \mathtt{T}_1, \ldots, \mathtt{p}_n : \mathtt{T}_n)$, where their domains $\mathtt{T}_i$ include all standard primitive data types as well as domain specific types. For example, the good $\mathtt{registration}(\mathtt{n} : \mathtt{Name})$ has one parameter drawn from the set of all valid (human) names.

In order to specify such an information good exchange action, we need to describe the **provider** and **receiver** of the information goods. Thus all the participants of such exchanges are modeled by **roles**. In a typical portal example, we may consider "Website" and "Visitor" as the roles.

Since the provider and receiver are autonomous, a minimal mechanism is required in order to establish an agreement on information good exchanges. This is done using **actions** (namely `request`, `promise`, and `deliver`) on information goods, where a (`request` + `promise`) pair on any particular information good implies the obligation to deliver the same.

The state transition diagram shown in Figure 1 associates with every information good an elementary business process. In this process all information goods are first by default in an initial state ("init") and can move by different transition sequences to the final delivered state. Note that not for all paths necessarily an obligation occurs. Obligation is only a conceptual notion. Adherence to the obligation in the real life transactions, and any legal actions in the event of breach of obligation by real life participants playing the role of BOL Role(s) are beyond the scope of BOL, and are case specific. However the BOL contract and history log may be used as evidence in any legal proceeding.

In order to create processes with dependencies among different information goods we must have a possibility to create dependencies among executions of different actions. For that purpose there exist two types of rules. Actions that correspond to an explicit interaction of the participants, i.e. a message exchange, are enabled by rules of the form

$\mathtt{state} \rightarrow \mathtt{[action]}$

where the `state` expression is a Boolean combination on the state predicates `promised`, `requested` and `delivered`, on relations that are defined on the parameter domains of the information goods and on the temporal parameters that occur in the state predicates. An empty predicate `state` evaluates to true. Implicit actions (or state changes) that are not connected to an interaction but result from other state changes are described by so-called substitution rules

$\mathtt{state} \rightarrow \mathtt{impliedState}$

```
roles:
Visitor, Website;

goods:
Registration (MemberID : STRING, Email : EMAIL) :
                    Visitor -> Website,
Payment (CreditCardInformation : INTEGER) :
                    Visitor -> Website,
Book (Title : STRING, Number : INTEGER) :
                    Website -> Visitor;

init:
promised(Website,Visitor, Book(Title, Price, 1), START),
requested(Visitor, Website, Registration(MemberID, EMAIL),
          START);

rules:
-> [deliver(Visitor, Website, Registration(MemberID, Email))]

delivered(Visitor, Website,Registration((MemberID, Email), t)
-> [deliver(Visitor, Website, Payment(CreditCardInformation))]

promised(Website, Visitor, Book(Title, Number), t)
-> [request(Visitor, Website, Book(Title, Number))]

requested(Visitor, Website, Book(Title, Number), t)
-> [deliver(Website, Visitor, Book(Title, Number))]

substitution:
requested(Visitor, Website, Payment(CreditCardInformation,t)
and delivered(Website, Visitor, Book(Title,Number), NOW)
and Number < 20 and NOW < t+1month
=> promised(Website, Visitor, Book(Title, Number+1),t)

delivered(Visitor, Website, Payment(CreditCardInformation))
=> promised(Website, Visitor, Book(Title, Number),t)

delivered(Visitor, Website, Registration((MemberID, Email),t)
=> requested(Website, Visitor, Payment(CreditCardInformation), t)
```

Figure 2: Sample business process

Example Scenario: A website offers electronic books to its members. Membership is free, and the visitors get K electronic books for free. However, free access to books expires after a period of one month since registration. After accessing K books, or expiry of the first month, whichever occurs earlier, the visitor needs to provide valid credit card information to the website. Then the visitor may request as

many books he likes. The corresponding BOL specification is given in Figure 2. We make a few important observations regarding this example. Only four different interactions can occur, as specified in `rules`. Since those are explicit communication actions among the participants they will be of particular importance in the implementation of the user interfaces. Most of the business logic is encoded into the substitution rules. These specification are mostly relevant for the workflow engine. There exists in this example one case where an obligation occurs: whenever the Website has promised to deliver a book and the Visitor requests a book, the delivery of the book becomes obligatory.

The expressions and rules are interpreted by the BOL interpreter which checks for the validity of incoming actions, maintains present workflow state and the workflow history, and determine potential actions (from the rules and workflow history).

4.2 BOL-aware XSLT Extensions

In section 3.1 we described our basic XSLT extensions enabling workflow-aware XSLT stylesheets. In this section we describe BOL specific instantiations related to the constructs introduced. The workflow relevant data objects correspond to the goods that have been defined. For each good type multiple instances of the good can be instantiated at the same time. Therefore the state of the goods can be given as relations.

$$D = \{R_G \mid G(x_1, \ldots, x_{n_G}) \in Goods\},\ R_G = \{(x_1, \ldots, x_{n_G})\}$$

Note that tuples in these relations may contain NULL values if the corresponding good parameters have not been set in the course of the execution of the process corresponding to one good (see Figure 1). The set of actions is given by

$$B = \bigcup_{G(x_1, \ldots, x_{n_G}) \in Goods} \{request(G(x_1, \ldots, x_{n_G})), promise(G(x_1, \ldots, x_{n_G})), deliver(G(x_1, \ldots, x_{n_G})), set(G(x_1, \ldots, x_{n_G}))\}$$

The semantics of the actions $request$, $promise$, and $deliver$ is the execution of the corresponding BOL action on the good G. The set action inserts (or updates) a data tuple into the corresponding tuple relation R_G. This allows to specify action parameters in a stepwise, interactive manner before actually submitting an action. Different types of atomic predicates can be used to check the workflow execution state. First, we have predicates to check whether a specific action has been performed for a good. These have the good parameters and the execution time as parameters:

$$C_1 = \bigcup_{G \in Goods} \{requested(G(x_1, \ldots, x_{n_G}), t), promised(G(x_1, \ldots, x_{n_G}), t), delivered(G(x_1, \ldots, x_{n_G}), t)\}$$

Second, we have predicates that check whether the execution of actions on certain good types is enabled according to the BOL rules. These predicates have no parameters.

$$C_2 = \bigcup_{G \in Goods} \{enabled(promise(G)), enabled(request(G)), enabled(deliver(G))\}$$

Third, we have a predicate to check the status of the workflow execution with one free parameter returning the error status value.

$$C_3 = \{error(x)\}$$

As far as the representation function `Rep` for predicates is concerned we use the following XPath-like notation: For a predicate in C_1 we have
`Rep(`$requested(G(x_1, \ldots, x_{n_G}), t)$`) = /requested/G.`
The parameters are uniquely determined and need therefore not to be explicitly mentioned. Within the scope of the element, where the predicate is introduced, we may access the parameters through variables with default name `$G.name`, where `name` is the name of the good parameter used in the BOL good specification. For the time parameter we use the name `$requested.G`.
For a predicate in C_2 we have
`Rep(`$enabled(request(G))$`) = /enabled/request/G`
and for C_3 we have `Rep(`$error(x)$`) = /error`. We may also construct Boolean combinations of these predicates in `<xslt-bol:if>` element conditions, as with ordinary XSL. For obtaining explicitly data values using `xslt-wf:value-of` or `xslt-wf:variable` we use also XPath expressions assuming a proper XML encoding of the additional state information made available by the workflow engine is given. As far as the XSLT representation is concerned namespace `xslt-wf` is replaced by `xslt-bol`. We give now a few examples for the use of the XSLT extensions interfacing with the BOL workflow engine.

Example: First we give an example for the evaluation of a condition within a `<xslt-bol:for-each>` element. The nesting determines of how the interleaving of the processing of registrations and payments is ordered.

```
<xslt-bol:for-each select=
                "/delivered/Registration">
   <xslt-bol:for-each select=
                "/enabled/request/Payment">
      ...
      <xsl:if test="contains($Registration.Email,
               'epfl.ch')and $requested.Registration
               ='01.01.2001'">
         You will be billed in Swiss Francs
      </xsl:if>
      <xsl:if test="$Payment.CreditCardInfo=''">
         // xslt-bol:submit example as shown next.
      </xsl:if>
     ...
   </xslt-bol:for-each>
</xslt-bol:for-each>
```

Within the scope of the element the value of `$Registration.Email` is available for processing and we can use it like an XSLT variable. Using these variables in the condition has the same effect as including, for example,

`<xsl:variable name='$Registration.Email' select='zoran@epfl.ch'/>` where `zoran@epfl.ch` is the value of the workflow variable. In order to check whether the workflow variable has been set we can check whether its value equals the NULL value represented as empty string `''`.

`xslt-bol:submit` elements provide an abstraction to the UI designer to create active elements in a desired look-and-feel manner for BOL actions and to pass the user's inputs to the BOL interpreter. Here we give an example:

```
<xslt-bol:submit action="deliver(Payment(
     $CreditCardForm.CreditCardInformation))"
     actionElement="pay" from="Visitor"
     to="Website">
 <form name="CreditCardForm">
   Credit Card Information:
  <INPUT type="text" name="CreditCardInformation"/>
  <INPUT type="submit" value="Submit Credit
        Card Information" actionID="pay"/>
 </form>
</xslt-bol:submit>
```

The `action` attribute specifies a complete BOL action that is to be performed. The type of action and the BOL goods along with the parameter passing specifications must be provided. Parameter passing specifications bind actual parameter values to the BOL goods parameters. Parameters defined both in XSL or in form elements and parameters implicitly defined within our workflow extension elements (`xslt-bol`) can be used in the action invocation. The `actionElement` attribute contains the unique identifier and determines, by means of the `actionID` attribute of an `INPUT` element, the rendition element which is associated with the specified BOL action. In the above example, a simple form submit element is tied with this action. Consequently, the UI will have a rendition of a simple submit button of the form. The 'event' of clicking of this button will be associated, transparently to the UI designer, with the passing of relevant information to the BOL interpreter. In BOL every action must be directed from one participant to another. The `from` and `to` attributes specify the originating and target participant of the action respectively.

There might be scenarios where the UI designer may want to obtain some additional information about the current state of the process. For example, he may want to know how many actions of certain type(s) have been already performed. For that purpose we use `xslt-bol:value-of` primitive. For example

```
<xslt-bol:value-of
          select="/History/Book/Delivered/Count">
```

provides a means to count the number of `deliver` actions on a BOL good called `Book` assuming a log is made available by the workflow engine with a proper XML encoding. If access to WF-Interpreter's history log is not made available, then such information may be obtained only by modifying the business description, like by appending additional parameters associated with the good `Book` (in this case).

5 Implementation Architecture

The diagram shown in Figure 3 summarizes the architecture of our approach. For the purpose of describing business processes we use the business offer language (BOL) that was introduced in the previous section and is interpreted by the workflow engine. The key component is the XSLT-BOL interpreter which interfaces in a state-dependent manner between the workflow engine and the user interface interpreter (a Web browser, for example). The UI designer provides to the XSLT-BOL interpreter two files, called XSLT-BOL and Root-XML, that essentially specify the mapping from the process states to the presentation. The Root-XML file contains static information about look and feel and possibly some data relevant to the process and user interface. The XSLT-BOL contains the process-aware template rules as introduced before. Based on the state of the process, XSLT-BOL is transformed by the XSLT-BOL interpreter into different XSLT stylesheets that, when applied to the Root-XML, produce the final rendition.

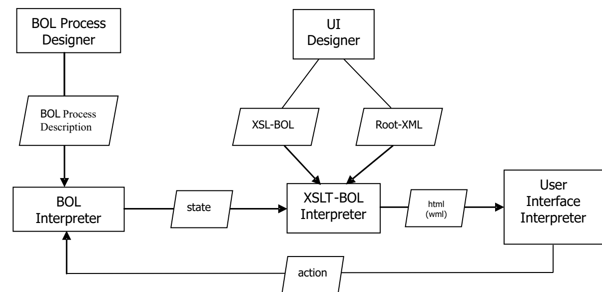

Figure 3: Architecture of the solution

In runtime, the XSLT-BOL interpreter performs the following steps to create the active user interface.

First, pre-processing the XSLT-BOL: the XSLT-BOL Interpreter is used to pre-process the UI designer's XSLT-BOL file, and transform it into a conventional XSLT file. In this step it evaluates state-based conditions and instantiates the XSLT templates accordingly and introduces the necessary state variables into the stylesheet.

Second, transforming Root-XML with the projected XSLT: The Root-XML file is then transformed using the generated XSLT stylesheet. However there is a subtle manipulation at this stage, whereby, the `xslt-bol:submit` primitives are modified to create ordinary elements of the presentation technology, for example, forms, if the presentation technology is of the html family. In this step also the necessary code for interacting with the workflow engine is introduced (using JavaScript). In addition two more issues can be addressed at this step to support the development of consistent user interfaces:

1. To assure correctness, only actions corresponding to valid BOL actions (at a given instant) are made active, ascertaining that only valid actions may be performed.
2. If the UI designer forgets to provide `xslt-bol:submit` and associated rendition for any particular BOL action, a default is created for the same, in order to ensure completeness. This ensures that all valid actions may be performed at the UI.

A possible variation of this approach is instead of generating directly the final rendition from the Root-XML file to generate from the Root-XML file an intermediary format that can be further processed (again using stylesheets) for specific end-point technololgies (like HTML and WML).

6 Application

Now we demonstrate the use of the basic mechanism introduced with a concrete example. By doing so we also discuss a number of methodological issues that have to be addressed when using process-aware XSLT templates. We base the discussion on the example introduced in Section 4.

6.1 Knowledge Required by the UI Designer

Though the UI designer need not understand the whole BOL description, he needs some minimal knowledge of the process description semantics. In our process description, we have the following

1. BOL Goods (parameters associated with the BOL goods are enclosed in brackets)
   - `Registration(MemberID, Email)`
   - `Book (Title, Number)`
   - `Payment (CreditCardInformation)`
2. The BOL roles participating in this process
   - `Visitor`
   - `Website`
3. The actions that can be performed and thus require interaction between the involved participants. These are the actions occuring in the rules section on the left-hand side.
   - `deliver(Visitor, Website, Registration(MemberID, Email))`
   - `deliver(Visitor, Website, Payment(CreditCardInformation))`
   - `request(Visitor,Website, Book(Title,Number))`
   - `deliver(Website,Visitor, Book(Title,Number))`

The BOL rules guiding the process are not necessarily to be known by a user interface designer in detail (though their knowledge might be of help).

6.2 Role of Root-XML File

The Root-XML file is supposed to factor out all static data that is common to all generated Web documents and that contains no presentation-specific code. Among these kind of data we find the following.

Common elements: There is certain information, which the UI designer may potentially want to reuse. It is preferable to have such static information, or other data, in the Root-XML. For example, a welcome message for the `Visitor`, or a message on the services provided by the `Website`, copyright information, disclaimers, an error message, may be put in this category. Other contents of this category may be some ordinary HTML (like forms). Though it is not illegal to have such content in our Root-XML, the drawback in having such end-point technology specific content is that it will defeat the purpose of using the same Root-XML and different processing templates to create User Interfaces for variants of technologies.

Data: In our example, it will be necessary to have a catalogue of `Books` that the `Website` offers. So, one keeps the `catalogue` information (data) in the Root-XML.

Layout structure: The logical structure of the Website and the common structure underlying the Web page layout is kept in the Root-XML. This ensures that the designer will use a consistent organization of the Web page throughout the process execution.

```
<ROOTXML>
  <CATALOGUE>
    <BOOK> <TITLE> ... </TITLE> </BOOK>
    <BOOK> <TITLE> ... </TITLE> </BOOK>
  </CATALOGUE>
  <MAINPAGE>
    <HEADER/>
    <REGISTRATION/>
    <DELIVERYSTATUS/>
    <FOOTER/>
  </MAINPAGE>
  <MESSAGES>
    <ERROR> Illegal operation!</ERROR>
  </MESSAGES>
</ROOTXML>
```

Since there is no restriction on this document, Root-XML potentially can be any well formed XML document, from a trivial document to any complex document the UI designer may wish to create. The example above gives a possible Root-XML file for our sample business process.

6.3 BOL State Aware Templates

Now we give a small template rule for the state when good Registration has not been delivered. The template creates a form in the UI while processing the `Registration` element in Root-XML so that the `Visitor` may deliver `Registration`. The fact that the form supports the action to deliver the BOL goods `Registration` is encoded in the `xslt-bol:submit` element. This form is instantiated conditionally, depending on whether the good `Registration` is yet to be delivered. Further, the header, which appears above the `MemberShipForm` form, gives the visitor some information about what he is supposed to

do. Such information, as we will see, will keep changing in our example scenario, in a state based manner. We also see how `href` elements may be used to bind user interactions to business actions.

```
<xslt-bol:if test="/enabled/deliver/Registration">
  <xsl:template match="HEADER">
    <b>You are a new member. Please register to
    access e-books.</b>
  </xsl:template>
  <xsl:template match="REGISTRATION">
    <xslt-bol:submit action="deliver(Registration(
        $MemberShipForm.MemberID,
        $MemberShipForm.Email))" from="Visitor"
        to="Website" actionElement="register">
     <form name="MemberShipForm">
          <input type="text" name="MemberID"/>
          <input type="text" name="Email"/>
          <a href="none" actionID="register">
             Register</a>
          your MemberID and Email.
     </form>
    </xslt-bol:submit>
  </xsl:template>
</xslt-bol:if>
```

6.4 Data Flow between Style Sheet and Workflow

The following is an example where a workflow template is matched and the value is transferred from the workflow state to the web document (`$Book.Number`). It also enables the user to select a book from the catalogue. Further, it demonstrates of how a data collection in the root document can be used to interactively select workflow relevant data values. The value retrieved into the XSL variable `$BookTitle` is then provided as a parameter to the activity invocation.

```
<xslt-bol:if test="/enabled/request/Book">
   <xsl:template match="HEADER"> You have accessed
      <xslt-bol:value-of select="$Book.Number"/>
       book(s) for free since registration.
       You still may freely access following books:
   </xsl:template>
   <xsl:template match="CATALOGUE">
      <xsl:apply-templates/>
   </xsl:template>
   <xsl:template match="BOOK">
      <xsl:variable name="BookTitle"
          select="./TITLE/*"/>
      <xslt-bol:submit action="request(Book
          ($BookTitle,$Book.Number))"
          from="Visitor" to="Website"
          actionElement="bookselect">
        <a href="none" actionID="bookselect">
           <xsl:value-of select="$BookTitle"/>
        </a>
      </xslt-bol:submit>
   </xsl:template>
</xslt-bol:if>
```

6.5 Avoiding Process Logic in Style Sheets

Let us now look at another template that covers the case when a book cannot be requested.

```
<xslt-bol:if test="not(/enabled/request/Book)">
  <xsl:template match="HEADER">
    <b> Oops! No more free Books.
        Please give a valid Credit Card Number.
    </b>
  </xsl:template>
  <xsl:template match="REGISTRATION">
    <xslt-bol:submit action="deliver(Payment(
       $CreditCardInfoForm.CreditCardInformation))"
       from="Visitor" to="Website"
       actionElement="payment">
     <form name="CreditCardInfoForm">
       <TEXTAREA name="CreditCardInformation"/>
       <INPUT type="submit" value="Submit Credit
       Card Information" actionID="payment" />
     </form>
    </xslt-bol:submit>
  </xsl:template>
</xslt-bol:if>
```

The UI designer knows that free books are available to `Visitor`, limited by some number and time. After that the user needs to deliver a valid credit card information. Depending on the state, the UI designer needs to provide a means to request a book, or deliver credit card information. He distinguishes this based on which actions are enabled. Alternatively, he might have reconstructed from the workflow state the business conditions, like checking how many books have been delivered or when the registration has been performed. This is not illegal to do, it will simply be bad design. It introduces unnecessary dependencies between the workflow specification and the layout specification.

6.6 Eliminating Repetitive Conditions

The encapsulation of ordinary and XSLT-BOL elements can possibly be nested.

```
<xsl:template match="MAINPAGE">
  <xslt-bol:if test="/error">
    <html><head>
      <title>ERROR !!!</title></head>
      <body bgcolor="red">
       <h3><xsl:value-of select="/error"/></h3>
      </body>
    </html>
  </xslt-bol:if>
  <xslt-bol:if condition="not(/error)">
    <html><head>
      <title>Electronic Books Online</title></head>
      <body bgcolor="azure">
       <xsl:apply-templates>
      </body>
    </html>
  </xslt-bol:if>
</xsl:template>
```

If there is similar behavior for multiple nodes (of Root-XML) depending on their state, we may like to have a `xslt-bol:if` encapsulating `xsl:template`. This was used in several of the previous examples. In such a case these templates are only applicable if the state predicate in the `xslt-bol:if` element evaluates true. On the other hand, if the same context node of Root-XML needs to be processed differently in different BOL states, we may encapsulate `xslt-bol:if` or `xslt-bol:for-each`

inside conventional `xsl:template`. We illustrate this point in the example of this subsection. In the above example, the UI designer wants to render the presentation in red, whenever an error occurs. Further he wishes to show the error message in bold (html header), and doesn't want to further process anything else. So, if an error occurs after the good `Registration` has been delivered, there will be other XSLT templates enclosed in a state-dependent `xslt-bol:if` that are applicable, however they won't be applied in this case since there is no `xsl:apply-template` for this case. On the other hand, if no error occurs, the UI will have an azure rendition, and the other templates will be processed `(xsl:apply-template)` according to the template rules applicable in the given state.

7 Summary and Further Work

In this paper we proposed a generic mechanism to make XSLT process-aware. We started with an abstract process description and a mechanism to translate it to an active UI using the stylesheet approach. Then we demonstrated the practicality of the approach using BOL as the process description language supported by a process interpreter and a process aware style sheet interpreter. We extended the already existing features of XSLT in a minimally invasive manner. The present BOL process interpreter as well as the XSLT-BOL interpreter are organized as groups of XSLT based components, and thus are easily integrated together, where the XSLT-BOL interpreter preprocesses BOL-aware XSLT stylesheets and transforms the Root-XML document (as described in Section 5). The proper use of the mechanism is an issue orthogonal to the mechanism itself. It allows both 'good' and 'bad' designs. Most importantly, it allows to separate concerns and supports orthogonality and modularity of the design of process-oriented web applications.

There are several aspects that need to be addressed in our future work, in particular implementation aspects. As the current implementation is heavily based on mechanism that are inherently slow (servlet and XML-XSLT processing) one of our primary concerns will be to evaluate the performance and try to improve it by refining the XSLT based part of the implementation. If a server processes multiple BOL process instances simultaneously and there are multiple users who are to be catered the same content we may optimize the processing by reusing renditions that have been produced for different process instances.

References

[1] K. Aberer, A. Wombacher: *A Language for Information Commerce Processes*, Third International Workshop on Advanced Issues of E-Commerce and Web-based Information Systems San Jose, California, USA, June 21-22, 2001.

[2] E. Anuf, M. Chaston, D. Moses: *Web Service User Interface WSUI 1.0*, http://www.wsui.org/doc/20011031/WD-wsui-20011031.html, 2001.

[3] J. Clark: *WXSL Transformations (XSLT) Version 1.0*, http://www.w3.org/TR/xslt, 1999.

[4] P. Fraternali: *Tools and Approaches for Developing Data-Intensive Web Applications: A Survey* ACM Computing Surveys, Vol. 31, No. 3, September 1999.

[5] G. Huck, I. Macherius and P. Fankhauser: *PDOM: Lightweight Persistency Support for the Document Object Model*, Proc. of the 1999 OOPSLA Workshop "Java and Databases: Persistence Options; on the 14th Annual ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99), Denver, 1999.

[6] E. Pelegri-Llopart, L. Cable: *JavaServer Pages Specification, version 1.1*, http://java.sun.com/products/jsp, 1999.

[7] E. Di Nitto, A. Wombacher, M. Jazayeri, M. Hauswirth, Z. Mikls, I. Podnar: *An Architecture for Information Commerce Systems*, Sixth International Conference on Telecommunications (ConTEL 2001), Zagreb, Croatia, 2001.

[8] J.J. Rodriguez, O. Diaz: *Seamless Integration of Inquiry and Transaction Tasks in Web Applications, Juan Jose Rodriguez, Oscar Diaz*, Proc. 9th IFIP 2.6 Working Conference on Database Semantics, Hong Kong, 2001.

[9] A. Wombacher, P. Kostaki, K. Aberer: *WebXIce: An Infrastructure for Information Commerce on the Web*, Proceedings of the 34th International Hawaiian Conference on System Sciences, Maui, USA, 2001.

[10] A. Wombacher, K. Aberer: *Modelling the ICE standard with a formal language for information commerce*, Proc. of 2nd International Conference on Electronic Commerce and Web Technologies (EC-Web 2001), Munich, Germany, September 4-6, 2001.

[11] WfMC Members, *Workflow Management Reference Model*, The Workflow Management Coalition, http://www.wfmc.org/standards/docs.htm.