

Integrating Web services into Web sites: the notion of workview*

Juan J. Rodríguez, Oscar Díaz
Dpto. de Lenguajes y Sistemas Informáticos
University of the Basque Country
Apdo. 649 - 20080 San Sebastián (Spain)
E-mail: <jibrojj,jipdigao>@si.ehu.es

October 19, 2001

1 Introduction

The evolution of web sites can be seen as an effort to cope with increasingly complex integration scenarios. First generation web sites are used as *content integration* platforms, usually for a single company. The second generation, usually referred to as e-commerce, required the integration of back-end OLTP systems with web front ends, still within a single organisation. Finally, the third generation (e.g. marketplaces) has to address content and operation integration in a multi-organisation setting. This vision has been fueled by the notion of Web services. Web services are self-contained, self-describing modular applications that can be published, located and invoked from anywhere across the Web. They will move Internet applications toward a service-oriented architecture than can perform functions, from simple requests to complicated business processes. Where HTML allows us to share text and XML allows us to share data, Web services permit us to share services regardless of the underlying platform. However, this is not enough. Services are not isolated units but rather, they are assembled into higher compounds that realise entire business processes. Service integration is becoming an increasing concern among practitioners [7][8].

1.1 Problem statement

Although these initiatives help to realise service composition, it is not clear how this composition affects the final outlook of the Web site that acts as the container of the service. If each service is invoked from a different web site, the presentation counterpart will be straightforward: a form that collects the service parameters and some buttons to trigger the service. However, if distinct services can be invoked from the very same site, the flow dependencies that govern the process could influence the final presentation of the site. As an example, consider the purchase order process where the processing of the purchase order by the supplier must precede the acceptance of the shipping request. Interface-wise, this control-flow dependency can be enforced by disabling the “*shippingRequestAcceptance*” button until the “*purchaseOrder*” button has been clicked on.

*This research was partially supported by the Secretaría de Estado de Política Científica y Tecnológica of the Spanish Government under contract TIC 1999-1048-C02-02.

Unfortunately, when service dependencies are moved to the presentation layer, they end up being hard-coded, “melted” with the rest of the page code. For instance, some of these dependencies can be enforced through page navigation. For example, the flow dependency “*login can only be issued once during a session*” is commonly enforced by providing a separate login page that once logged, is no longer available. If more than one service can be enacted from the very same page, flow dependencies are usually hard-coded in some obscure scripting code. This situation deteriorates if parameter passing between services is required. Then, the programmer has to resort to cookies or other mechanism to overcome the stateless nature of the *http* protocol. This ends up in the flow dependencies being hard-coded, implicit and scattered around distinct pages. This hinders not only development, but most important, maintenance and evolution of web applications. Such situation stems from the lack of abstractions that impede current web programming.

1.2 Contribution of the paper

Based on the above observations, this work proposes a separate and declarative specification of the control and data dependencies that regulate service enactment in web applications: the *workview*. Using an XML language, the designer defines in a separate *workview* XML document, the dependencies that harness the set of services that can be invoked from a site, regardless of the page from where the service can be enacted or its results rendered. This notion allows to distinguish service dependencies from how and where these services are enacted. In so doing, it affords service designers and page designers the freedom to work independently. The latter are able to focus on content presentation and navigation without being involved in the intricacies of service dependencies. The page designer has only to decide where a service can be enacted and rendered, but she is not burdened with flow dependency enforcement and parameter passing concerns. These issues fall within the realm of the *workview* runtime.

In our opinion, the notion of *workview* brings the following advantages:

1. It alleviates the application server workload as some of the flow dependencies can be enforced at the interface side
2. It eases development and maintenance. The declarative and modular description of flow dependencies facilitates the addition and removal of flow dependencies during the site life-span. As the site incorporates new services or additional business policies that need to be enforced, flow dependencies can be added/removed with little or no impact on the rest of the site. And site evolution is an increasing demand among practitioners due to the dynamic and competitive business environments that characterised most web applications.
3. It improves coherence and thus, usability of the site. Most sites offer loosely-coupled services where the role of the Web site is almost restricted to be a front-end for invoking services. However, if the number of services is large, it is fundamental to struggle for the site to appear to function as a single whole. A *workview* provides a sense of coherence and unity among the distinct services within a site.
4. It promotes separation of concerns, a well-known strategy for speeding up complex software projects. Web development is a mixture between print publishing and software development, between marketing and computing, between art and technology [3]. This setting calls for knowledge and expertise from many different disciplines and requires heterogeneous teams. Separation of concerns eases group working by keeping each aspect

separate: content layout and presentation concerns are the duty of usability practitioners while service definition and integration can be addressed by knowledgeable engineers on EAI technologies. Therefore, this work strives to separate content presentation from service delivery on the grounds that the underlying technologies and requirements demand distinct skills.

The rest of the paper is structured as follows. Section 2 presents workview elicitation. Section 3 presents the Web Service Workview Language. Workview evolution is the topic of section 4. How workviews are binded to HTML pages is addressed in section 5. Finally, conclusions are given. A car-retailer site is used as an example throughout the paper. This example can be enacted at <http://simpl68.si.ehu.es/wswl/examples/car-retailer>. Besides the application itself, this site contains the complete specification of the services, the workview definition and an example of a hosting HTML page.

2 Workview elicitation

The notion of workview strives to provide a sense of coherence and unity among the distinct services that can be accomplished throughout a site. This coherence stems from the distinct dependencies that tie these services together. Consider our running example: a *car-retailer* site. This site supports the purchase of a car (i.e. the “*purchaseRequest*” service through which a remote purchase service is invoked), the possibility to apply for a credit, (i.e. the “*creditRequest*” service) and the login of a user (i.e. the “*login*” service). As for dependencies, they range from control restrictions (e.g. car configuration should precede car purchase), data flow constraints (e.g. the name of the user requesting the credit should be obtained from the car buyer), or authorisation rules (e.g. only preferential customers have access to the credit request service).

During analysis, the application requirements are examined, and a conceptual model of the system to be built is produced. The analysis artifacts include class diagrams, sequence diagrams, state diagrams and activity diagrams. Workview analysts strive to elicitate control and data flow dependencies which are depicted through a service-dependency graph. Next subsections look at these dependencies and how they reflect application requirements.

2.1 Control-flow dependencies

Control-flow dependencies can be described either procedural or declarative. In the first case, constructs like those found in programming languages are used to specify execution control. The most important operators are sequential execution ($t1;t2$), conditional execution ($c?t1:t2$), loops ($while\ b:\ w$) and parallel execution ($\parallel\ t1,t2\dots$). Sagas [2], Interactions [5] among research prototypes, or FlowMark [4] as a commercial product, are systems which exhibit this approach. WSFL and XLANG also follow this approach. By contrast, a declarative description constrains the space of possible interactions through both temporal and existence conditions [1]. For instance, a temporal ordering of tasks can express that $t1$ must take place before $t2$ if both $t1$ and $t2$ take place ($t1 < t2$). The existence condition ($t1 \Rightarrow t2$) denotes that if $t1$ occurs then $t2$ must also take place but without imposing any restriction on the moment when $t2$ should happen.

We found the declarative approach more appropriate in a GUI context where a common situation is for a service to require some necessary condition to be available, and once available, it is let to the user the freedom to invoke it (e.g. once an item has been added to the cart, the

```

<history>
  <login timeStamp="1">
    <userName>Smith</userName>
    <password>****</password>
    <alternative>correct</alternative>
    <userRole>Client</userRole>
  </login>
  <purchaseRequest timeStamp="2">
    <buyer>Smith</buyer>
    <carInfo>
      <carPrice>14,400</carPrice>
      ...
    </carInfo>
  </purchaseRequest>
  <purchaseRequest timeStamp="3">
    <buyer>Smith</buyer>
    <carInfo>
      <carPrice>18,700</carPrice>
      ...
    </carInfo>
  </purchaseRequest>
  <creditRequest timeStamp="4">
    <userName>Smith</userName>
    <creditProvider>AK45W</creditProvider>
    <creditAmount>14,400</creditAmount>
  </creditRequest>
</history>

```

Figure 1: A history state.

“*sending the order*” service becomes available). In our opinion, this approach naturally fits the event-driven nature that characterise GUI applications. Besides it accounts for flexibility and maintainability at the price of complex debugging. Whereas a procedural description allows to have a clear picture of the event flow at compile time, this is not the case for enabled-based descriptions where a global scheduler is responsible to keep informed each service of its availability. A similar approach has also been proposed for deductive information system modelling[6].

Our approach is to describe control-flow dependencies as necessary conditions (as opposed to sufficient conditions) on the flow of services previously enacted. This flow of ordered occurrences is known as the *history*. Each service enactment causes an entry on the history log. As an example, consider a session where the user first logs in, then, invokes two “*purchaseRequest*” services, and finally, issues a “*creditRequest*”. The resulting state of the history is shown in figure 1. The log is supported by an XML document which has `<history>` as its root element and, as sub-elements, the name of the service enacted and its actual parameters (i.e. the service *occurrences*). Additionally, an entry can have a session or an application life-span. The former is removed as soon as the session ends. By contrast, application entries are persistent and thus, are shared among distinct sessions of the same application.

Therefore, control-flow dependencies are stated as predicates on the history log. As this log is an XML file, these predicates are expressed as XPath expressions. Some possible dependencies on the “*creditRequest*” service are listed in table 1.

For simplicity sake, the examples shown throughout this paper just involve two services.

Control dependency	The dependency as a predicate on the history	Lifespan
<i>creditRequest</i> is enabled ... only once during a session	not(history/creditRequest)	session
<i>creditRequest</i> is enabled...provided the same user has bought a car during this session	history/purchaseRequest[buyer=history/login/user]	session
<i>creditRequest</i> is enabled...no more than 100 times during the application lifetime	count(history/creditRequest) < 100	application
<i>creditRequest</i> is enabled...only once a day	not(history/creditRequest[day(@timeStamp)= atari:day(history(login/@timeStamp))])	app, session
<i>creditRequest</i> is enabled...only two times by the same user	count(history/creditRequest[userName = history/login/user]) < 2	app,session
<i>login</i> can only be successfully issued once	not(history/login)	session
<i>purchaseRequest</i> is enabled... if the user has logged in	history/login	session
<i>creditRequest</i> is enabled.. if the user has made a purchase above \$100 during . the application	history/purchaseRequest[number(carPrice)>100]	application

Table 1: Control-flow dependencies for the *car-retailer* site (the bold ones have been implemented in this example)

Notice however that the enabling predicate is any XPath expression on the history log. So, the availability of a service can be based on any conjunction/disjunction of occurrence which can be filtered based on the values of their actual parameters. It is also worth noticing that the service itself is unaware of those dependencies. Dependencies belong to the workview which provides the context in which the services are invoked. Indeed, the same service can participate in distinct workviews, and thus, be subject to distinct constraints.

For our *car-retailer* site, the following control-flow dependencies need to be enforced:

- a customer can only apply once for a credit, that is, the “*creditRequest*” service is enabled only if the user has never applied for a credit in this or any previous session.
- the “*login*” service can only be successfully issued once within a session.
- the “*purchaseRequest*” service is enabled only if “*login*” has been successfully invoked within the session.
- the “*creditRequest*” service is enabled only if the user has made a purchase above \$100, anytime.

Figure 2 displays the distinct control dependencies involved in the *car-retailer* workview. Nodes stands for services. An arc is drawn from node *S1* to node *S2* to show that *S1* constrains somehow *S2*. A “+” indicates an existence condition whereas a “-“ checks that the service has not yet been enacted.

2.2 Data-flow dependencies

Service parameters can be obtained by directly querying the user. Alternatively, these parameters can also be retrieved from previous service occurrences kept in the history. As an

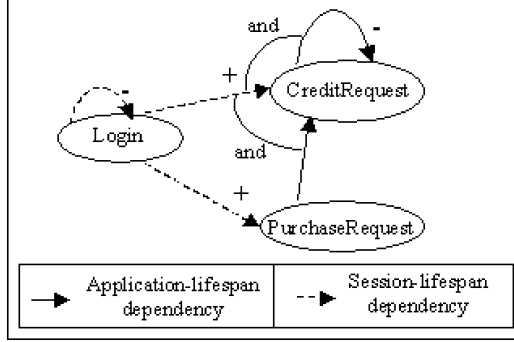


Figure 2: A dependency graph for the *car-retailer* workview.

example, consider the “*creditRequest*” service. It has three parameters: *creditProvider*, *creditAmount*, and *userName*. While the former should be input directly by the user, *creditAmount* and *userName* can be obtained from previous occurrences of the *login* and *purchaseRequest* services, respectively. These requirements state a data-flow dependency.

A data-flow dependency does not end by indicating the mapping and mapped attribute. We also consider three further aspects: the binding policy, the selection policy and the consumption policy.

The binding policy. It indicates whether the mapping is “*compulsory*” or “*optional*”. In our example, the *userName* is compulsory obtained from the *login*; no option is given to the user to change it. Alternatively, the *creditAmount* value is also obtained from the history but it is not fixed, that is, it is presented to the user as an option, but the user is free to change it.

The selection policy. As a motivating example, consider that *creditRequest* takes its *creditAmount* parameter from the *purchaseRequest* task. First, *purchaseRequest* is enacted more than once which is reflected in the history as a sequence of occurrences: $\{ purchaseRequest(...100...), purchaseRequest (...200...), purchaseRequest(...300...) \}$. If now *creditRequest* is issued, which of the available *purchaseRequest* occurrences should be used to obtain the *creditAmount* parameter? This question poses the need for a selection policy which tackles the situation where several services occurrences of the same type are available in the history. In this case, two selection policies are introduced to avoid any ambiguity: “*mostRecent*” which selects the latest occurrence, and “*oldest*” which selects the occurrence which appears first in the history. In our example, defining a “*mostRecent*” selection policy will make the system to request a credit of \$100 while the “*oldest*” alternative would go for \$300. These keywords are shortcuts for easy access to the history (i.e. they are expanded to XPath predicates based on the occurrences’ time-stamps). If the designer needs to enforce more complex policies, he can always result to stating directly the XPath predicate himself. For instance, if the *creditAmount* is obtained not from individual purchases, but as the sum of the prices of distinct items bought so far, the designer can express such policy as follows: $sum(history/purchaseRequest/carPrice)$.

The consumption policy. Consider again that *purchaseRequest* is enacted more than once: $\{ purchaseRequest(..100..), purchaseRequest(..200..), purchaseRequest(..300..) \}$. If *creditRequest* is issued with the “*mostRecent*” selection policy, \$300 will be requested. If the user clicks on *creditRequest* again (if this be possible), should the system select the very same *creditAmount*? The specification should indicate whether the service occurrence is “consumed” (i.e. removed from the history log) when invoking the *creditRequest* or not. For instance,

Mapping parameter	Mapped parameter	lifeSpan	Selection policy	Consumption policy	Binding policy
<i>purchaseRequest.buyer</i>	<i>login.userName</i>	session	mostRecent	none	compulsory
<i>creditRequest.creditAmount</i>	<i>purchaseRequest.carPrice</i>	application	mostRecent	mostRecent	optional
<i>creditRequest.userName</i>	<i>login.userName</i>	session	mostRecent	none	compulsory

Table 2: Data-flow dependencies for the *car-retailer* site.

purchaseRequest takes the value of its parameters *userName* and *creditAmount* from previous occurrences of *login* and *purchaseRequest*, respectively. A common situation could be to keep the *login* occurrence in the history: *userName* is obtained once and again from the very same *login* occurrence, regardless of how many times *creditRequest* has been enacted. By contrast, we could be interested in “consuming” the *purchaseRequest* occurrences as they are used by *creditRequest*; in this case, each *creditRequest* requires a different *purchaseRequest* occurrence.

Ocasionally, the notion of consumption can also be relevant for control-flow dependencies. For instance, the consumption policy can support some pair-like dependencies between services. As a case in point, consider that a credit can be requested for each car being bought. In this case, an enabling predicate is not enough. Although the condition can enforce that *creditRequest* is enabled once *purchaseRequest* has been enacted, every successful invocation of *creditRequest* should consume its *purchaseRequest* counterpart so that you can issue as many *creditRequest* as cars being bought.

Table 2 sums up the distinct data-flow requirements for this application:

- the “*buyer*” of “*purchaseRequest*” must be obtained from the *login*’s “*userName*” as many times as required. If the user logs in several times during the session, the last enactment will be considered.
- the “*creditAmount*” of “*creditRequest*” can be obtained from the *purchaseRequest*’s “*carPrice*” only once. If different *purchaseRequest* are available, the last occurrence will be taken.
- the “*userName*” of “*creditRequest*” must be obtained from the *login*’s “*userName*” as many times as required. If the user logs in several times during the session, the last enactment will be considered.

2.3 Other dependencies

Besides the previous dependencies, authorisation and temporal restrictions also need to be enforced in web sites. The former restricts the access to some services based on the user’s profile (e.g. the user account, her age, her role and the like). It requires the existence of some profiling mechanism. As for temporal restrictions, they define time intervals during which the service is either enabled or disabled.

For the *car-retailer* site, the following additional dependencies are enforced:

- “*creditRequest*” is only available for customers that do not have “*anonymous*” as their username. By default, it is available.
- “*creditRequest*” is available on Sundays.

The advantage of jointly specifying all these distinct types of dependencies is to have a wholistic view of how the set of services behaves. For instance, if only preference customers can enact

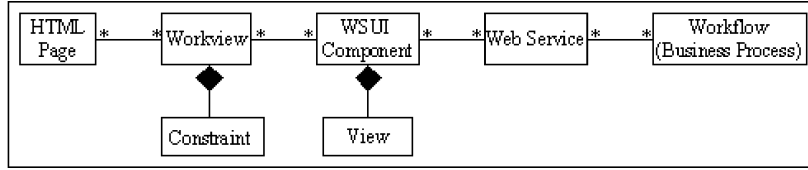


Figure 3: The workview meta-model.

carConfiguration (an authorisation constraint), and *carConfiguration* enables *carPurchase* (a control-flow dependency), the designer can infer that *purchaseRequest* is going to be available only for preference customers. If this is not a desirable situation, the designer could resort to a data-mapping dependency between *carConfiguration* and *purchaseRequest* with “optional” as the binding policy, or, alternatively, she can change the authorisation constraint. The point is that unexpected side effects can be more easily detected at design time rather than when the system is in operation.

Figure 3 outlines the relationships between the workview construct and the other notions. Next section addresses workview definition through the *Web Service Workview Language* (WSWL).

3 Workview definition

This section proposes an XML language for workview definition. Figure 4 gives an example for the *car-retailer* site. The identifier and description of the workview are provided first followed by the services that participate in the workview. For each service, its selector and the file holding its definition are provided. The list of constraints ends the definition of the workview. The element `<dependency type= “aType” service= “aService”>` indicates the type of the constraint (i.e. “controlFlow”, “dataFlow”, “authorisation” or “temporal”) and the service being constrained. Moreover:

- control flow dependencies include an `<controlFlow>` sub-element which holds an XPath expression¹ on the history log. This expression has to be satisfied (i.e. some elements must be retrieved) for the associated service to be available.
- data flow dependencies add a `<dataFlow>` sub-element which contains one or more mapping requirements.
- authorisation dependencies are stated through the `<granted to= “aUser”>` and `<revoked from= “aUser”>` sub-element where the users to whom the services have been granted or revoked are identified through their roles or account names.
- temporal dependencies are specified through the `<available on= “aPeriod”>` and `<unavailable on= “aPeriod”>` sub-elements.

Additionally, control-flow and data-flow constraints add a `<historyManagement>` element which indicates the service occurrences to be kept or removed from the history. This behaviour is specified through the `<on service= “aService”>` element. When “aService” is completed, an appropriate action is conducted on the history, namely:

¹For simplicity sake, XPath expressions do not need to include the root of the history document. The system automatically completes it.

```

<workviewComponent id="car-retailer">
  <title>car-retailer Workview</title>
  <description>car-retailer Workview</description>
  <services>
    <sessionService id="login" definition="login.wsui"/>
    <transactionService id="purchaseRequest" definition="purchaseComponent.wsui"/>
    <transactionService id="creditRequest" definition="creditRequest.wsui"/>
  </services>
  <dependencies>
    <dependency service="login" type="controlFlow">
      <controlFlow enabledWhen="not(login[alternative='correct'])"/>
      <historyManagement>
        <on service="login">
          <register lifeSpan="session"/>
        </on>
      </historyManagement>
    </dependency>
    <dependency service="purchaseRequest" type="controlFlow">
      <controlFlow enabledWhen="login[alternative='correct']"/>
      <historyManagement>
        <on service="login">
          <register lifeSpan="session"/>
        </on>
      </historyManagement>
    </dependency>
    <dependency service="purchaseRequest" type="dataFlow">
      <dataFlow>
        <mapping parameter="buyer" select="login/UserName.text()" selectionPolicy="mostRecent" bindingPolicy="compulsory"/>
      </dataFlow>
      <historyManagement>
        <on service="login">
          <register lifeSpan="session"/>
        </on>
      </historyManagement>
    </dependency>
    <dependency service="creditRequest" type="controlFlow">
      <controlFlow enabledWhen="login[alternative='correct'] and not(creditRequest[userName=most-recent(login[alternative='correct']/UserName.text())])"/>
      <historyManagement>
        <on service="login">
          <register lifeSpan="session"/>
        </on>
        <on service="creditRequest">
          <register lifeSpan="application"/>
        </on>
      </historyManagement>
    </dependency>
    <dependency service="creditRequest" type="dataFlow">
      <dataFlow>
        <mapping parameter="userName" select="login/UserName.text()" selectionPolicy="mostRecent" bindingPolicy="compulsory"/>
        <mapping parameter="creditAmount" select="purchaseRequest/carInfo/carPrice.text()" selectionPolicy="mostRecent" bindingPolicy="optional"/>
      </dataFlow>
      <historyManagement>
        <on service="login">
          <register lifeSpan="session"/>
        </on>
        <on service="purchaseRequest">
          <register lifeSpan="application"/>
          <remove service="purchaseRequest" consumptionPolicy="mostRecent"/>
        </on>
      </historyManagement>
    </dependency>
  </dependencies>
</workviewComponent>

```

Figure 4: The *car-retailer* workview definition.

- $\langle register\ lifeSpan = "aLifeSpan" \rangle$ which records this service occurrence on the history untill the session ends (a “*session*” lifespan) or persistently (an “*application*” lifeSpan),
- $\langle remove\ service = "aService" consumptionPolicy = "aPolicy" \rangle$ which removes occurrences of “*aService*” according to the consumption policy: “*mostRecent*” or “*oldest*” which removes the *aService* occurrence held on the history which has the greatest or smallest time stamp, respectively. The “*all*” option removes every *aService* occurrence held on de history.

Therefore, part of the constraint semantics is supported through the management of the history (e.g. the life-span of the service occurrences). As a result, a separate history log is kept for each constraint. If the life-span is “*session*”, this does not pose any storage problem, as the number of service enacted within a session is not too large, and the log lifespan is that of the session. However, if the life-span is “*application*”, the history log persists between sessions, and hence, the designer should carefully check that an appropriate consumption policy is in place so that history logs are prevented from growing too much. Some garbage collection mechanism could also be useful in this context but has not yet been investigated.

References

- [1] P. Attie, M. Singh, A. Sheth, and M. Rusinkiewicz. Specifying and enforcing intertask dependencies. In *Conf. on Very Large Data Bases (VLDB)*, pages 134–145, 1993.
- [2] H. Garcia-Molina and K. Salem. Sagas. In *ACM SIGMOD Intl. Conf. on Managment of Data*, pages 249–259, 1987.
- [3] A. Ginige and S. Murugesan. Web engineering: An introduction. *IEEE MultiMedia*, pages 15–18, January - March 2001.
- [4] F. Leymann and W. Altenhuber. Managing business processes as an information resource. *IBM Systems Journal*, 2(33):326–348, 1994.
- [5] M. H. Nodine, N. Nakos, and S. B. Zdonik. Specifying flexible tasks in a multidatabase. In *2nd Int. Conf. on Cooperative Information Systems*, pages 3–14, 1994.
- [6] A. Olivé. A comparison of the operational and deductive approaches to conceptual information systems modelling. In *Proc. IFIP-86*, pages 91–96, Dublin, 1986.
- [7] W. Rajput. *E-Commerce Systems Architecture and Applications*. Artech House Publishers, 2000.
- [8] R. Kalakota M. Robinson. *e-Business: Roadmap for Success*. Addison-Wesley, 1999.