Hosted Universal Composition: Models, Languages and Infrastructure in mashArt

Florian Daniel[1], Fabio Casati[1], Boualem Benatallah[2], Ming-Chien Shan[3]

[1]University of Trento - Via Sommarive 14, 38050 Trento - Italy
{daniel,casati}@disi.unitn.it

[2] University of New South Wales- Sydney NSW 2052, Australia
boualem@cse.unsw.edu.au

[3] SAP Labs- 3410 Hillview Avenue, Palo Alto, CA 94304, USA
ming-chien.shan@sap.com

Abstract. Information integration, application integration and component-based software development have been among the most important research areas for decades. The last years have been characterized by a particular focus on web services, the very recent years by the advent of web mashups, a new and user-centric form of integration on the Web. However, while service composition approaches lack support for user interfaces, web mashups still lack well engineered development approaches and mature technological foundations.
In this paper, we aim to overcome both these shortcomings and propose what we call a *universal* composition approach that naturally brings together data and application services with user interfaces. We propose a unified component model and a universal, event-based composition model, both able to abstract from low-level implementation details and technology specifics. Via the *mashArt* platform, we then provide *universal composition as a service* in form of an easy-to-use graphical development tool equipped with an execution environment for fast deployment and execution of composite Web applications.

1 Introduction

The advent of Web 2.0 led to the participation of the user into the content creation and application development processes, also thanks to the wealth of social web applications (e.g., wikis, blogs, photo sharing applications, etc.) that allow users to become an active contributor of content rather than just a passive consumer, and thanks to *web mashups* [1]. Indeed, especially mashup tools enable fairly sophisticated development tasks, mostly inside the browser. They allow users to develop their own applications starting from existing content and functionality. Some applications focus on integrating RSS or Atom feeds, others on integrating RESTful services, others on simple UI widgets, etc. Many mashup approaches are innovative in that they tackle integration at the user interface level (most mashups integrate presentation content, not “just” data) and aim at simplicity more than robustness or completeness of features (up to the point that advanced web users, not only professional programmers, can develop mashups).

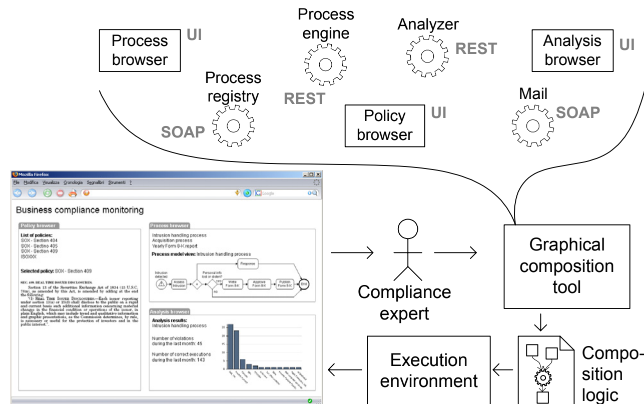

Figure 1 Reference scenario: development of a business compliance monitoring application

Inspired by and building upon research in SOA and capturing the trends of Web 2.0 and mashups, this paper introduces the concept of *universal integration*, that is, the creation of composite web applications that integrate data, application, and user interface (UI) components. Our aim is to do what service composition has done for integrating services, but to do so at all layers, not just at the application layer, and remove some of the limitations that constrained a wider adoption of workflow/service composition technologies. Universal integration can be done (and is being done) today by joining the capabilities of multiple programming languages and techniques, but it requires significant efforts and professional programmers. In this paper we provide abstractions, models and tools so that the development and deployment of universal compositions is greatly simplified, up to the extent that even non-professional programmers can do it in their web browser.

Scenario. To exemplify the needs for universal integration, in Figure 1 we present the scenario that will accompany us throughout this paper, i.e., the development of a business compliance monitoring (BCM) web application starting from existing services and components.

A company's compliance expert wants to develop a web application that allows her to correlate company policies (representing the regulations the company is subject to) with process execution data and compliance analysis data and, in case a compliance violation by a process execution is detected, send a notification email. For this purpose, she wants to integrate a variety of different components already existing inside the company: components with own UI (*Policy browser*, *Process browser*, and *Analysis browser*), SOAP web services (*Process registry*, *Process engine*), and RESTful web services (*Analyzer* and *Mail* services). In addition to the "traditional" concerns of service composition (mainly revolving around the sequential or conditional invocation of components), UI components need to be *synchronized*: user interaction with the policy browser (e.g., to select a policy) must cause the process browser UI to change (showing processes affected by the policy). In general, in composed UIs, all components may have to change at the same time as they need to display consistent information. This also means that UI components must somehow be able to react to user input (that's what they have been designed for), but also to programmatic input: in the example above, the process component should be notified of the selection in the policy

browser and change its UI accordingly. Additional challenges are related to the fact that the components are heterogeneous in nature, that developers need to master multiple communication protocols, client- and server-side programming techniques, different service and application architectures and programming languages, and must be able to integrate the event-driven philosophy of UIs with the control-flow-based philosophy of service orchestrations. These are only a few of the difficulties they encounter in their task; many others still lie in the details (e.g., how to deploy and maintain such complex integration logic).

Ideally, as shown in Figure 1, there would be a composition tool that hides the described implementation details and allows developers to graphically specify the desired composition logic, to execute it, and to obtain straight away the web application in the lower left corner of the figure. Currently, there are no integration instruments available that can cope with the described heterogeneity of components and that rely on one single integration paradigm only. Service composition approaches cannot handle UIs, and UI technologies are not designed with service integration in mind. Our compliance expert therefore falls back to various programming languages and tools or complex frameworks like J2EE and .NET along with AJAX scripting for UI, which makes applications harder to develop and maintain, and certainly beyond the reach of non-programmers. Yet, as more and more web applications offer their UI as components, open APIs toward them, or both (a la Google Maps), the importance of universal integration is likely to grow even faster in future.

Approach and contributions. In the following we describe a universal composition model and tool, called *mashArt*. MashArt aims at *empowering users with easy-to-use and flexible abstractions and techniques to create and manage composite web applications.* In particular, in this paper we make the following contributions:

- A *universal component model*, allowing the modeling of UI components, application components (e.g., services with an API) and data components (representing feeds or access to XML/relational data) using a unified model.
- A *universal composition model*, to combine the building blocks and expose the composition as a MashArt component, possibly accessible via rest/soap, and/or providing feeds, and/or having its own (composed) UI.
- The *mashArt platform* which is a service providing a number facilities for facilitating the rapid development and management of composite web applications. MashArt is entirely hosted and web-based, with zero client-side code.

In this paper we focus on the conceptual and architectural aspects of mashArt, which constitute the most innovative contributions of this work, namely the *component* and *composition model*s as well as the *development* and *runtime* part of the infrastructure. The reader is referred to the mashArt web site (*http://mashart.org/ER09*) for more technical details.

We next introduce the principles that guide our work (Section 2), and then discuss the state of the art (Section 3). In Section 4 and Section 5we introduce the mashArt unified component and composition models. Section 6 describes the platform and hosted execution environment. Section 7 provides concluding remarks.

2 Guiding principles

We aim at universal integration, and this has fundamental differences with respect to traditional composition. In particular, the fact that we aim at also integrating UI implies (i) that *synchronization*, and not (only) orchestration a la BPEL, should be adopted as interaction paradigm, (ii) that components must be able to react to both human user input and programmatic interaction, and (iii) that we must be able to design the UI of the *composite* application, not just the behavior and interaction among the components. This shows the need for a model based on state, events and synchronization more than on method calls and orchestration. We recognize in particular that *events*, *operations*, a notion of *state* and *configuration properties* are all we need to model a universal component. With respect to the design of the composite UI, we assume developers will use their favorite Web development tool (we do not aim at competing with these tools, although we do offer a simple templating mechanism for rapid development of prototype applications that run in the browser). Rather, we make it easy to embed mashArt components inside a Web application.

On the data side, we realize that *data* integration on the Web may also require different models: for example RSS feeds are naturally managed via a pipe-oriented data flow/streaming model (a-la Yahoo Pipes) rather than a variable-based approach as done in conventional service composition.

Another dimension of universality lies in the interaction protocols. MashArt aims at hiding the complexity of the specific protocol or data model supported by each component (REST, SOAP, RSS, Atom, JSON, etc) so a design goal is that from the perspective of the composer all these specificities are hidden – with the exceptions of the aspects that have a bearing on the composition (e.g., if a component is a feed, then we are aware that it operates, conceptually, by pushing content periodically or on the occurrence of certain events).

Generality and universality are often at odds with the other key design goal we have: *simplicity*. We want to enable advanced web users to create applications (an old dream of service composition languages which is still somewhat a far reaching objective). This means that mashArt must be fundamentally simpler than programming languages and current composition languages. We target the complexity of creating web pages with a web page editor, or the complexity of building a pipe with Yahoo Pipes (something that can be learned in a matter of hours rather than weeks).

To achieve simplicity we make two design decisions: first, we keep the composition model lightweight: for example, there are no complex exception or transaction mechanisms, no BPEL-style structured activities or complex dead-path elimination semantics. This still allows a model that makes it simple to define fairly sophisticated applications. Complex requirements can still be implemented but this needs to be done in an "ad hoc" manner (e.g., through proper combinations of event listeners and component logic) but there are no specialized constructs for this. Such constructs may be added over time if we realize that the majority of applications need them.

The second decision is to focus on simplicity only *from the perspective of the user* of the components, that is, the designer of the composite applications. In complex applications, complexity must reside somewhere, and we believe that as much as possible it needs to be inside the components. Components usually provide core functionalities and are reused over and over (that's one of the main goals of compo-

nents).Thus, it makes sense to have professional programmers develop and maintain components. We believe this is necessary for the mashup paradigm to really take off. For example, issues such as interaction protocols (e.g., SOAP vs. REST or others) or initialization of interactions with components (e.g., message exchanges for client authentication) must be embedded in the components.

3 State of the Art

Service composition approaches. A representative of service orchestration approaches is BPEL [6], a standard composition language by OASIS. BPEL is based on WSDL-SOAP web services, and BPEL processes are themselves exposed as web services. Control flows are expressed by means of structured activities and may include rather complex exception and transaction support. Data is passed among services via variables (Java style). So far, BPEL is the most widely accepted service composition language. Although BPEL has produced promising results that are certainly useful, it is primarily targeted at professional programmers like business process developers. Its complexity (reference [6] counts 264 pages) makes it hardly applicable for web mashups.

Many variations of BPEL have been developed, e.g., aiming at invocation of REST services [7]and at exposing BPEL processes as REST services [8]. In [9] the authors describe Bite, a BPEL-like lightweight composition language specifically developed for RESTful environments. IBM's Sharable Code platform [10] follows a different strategy for the composition of REST or SOAP services: a domain-specific programming language from which Ruby on Rails application code is generated, also comprising user interfaces for the Web. In [11], the authors combine techniques from declarative query languages and services composition to support multi-domain queries over multiple (search) services. All these approaches focus on the application and data layer; UIs can then be programmed on top of the service integration logic. mashArt features instead universal integration as a paradigm for the simple and seamless composition of UI, data, and application components. We argue that universal integration will provide benefits that are similar to those that SOA and process centric integration provided for simplifying the development of enterprise processes.

UI composition approaches. In [12] we discussed the problem of integration at the presentation layer and concluded that there are no real UI composition approaches readily available: Desktop UI component technologies such as .NET CAB [13] or Eclipse RCP [14] are highly technology-dependent and not ready for the Web. Browser plug-ins such as Java applets, Microsoft Silverlight, or Macromedia Flash can easily be embedded into HTML pages; communications among different technologies remain however cumbersome (e.g., via custom JavaScript). Java portlets [15] or WSRP [2] represent a mature and Web-friendly solution for the development of portal applications; portlets are however typically executed in an isolated fashion and communication or synchronization with other portlets or web services remains hard. In addition, portals do not provide support for service orchestration logic. The Web mashup paradigm aims at addressing the above shortcomings. Mashup development is still an ad-hoc and time-consuming process, requiring advanced programming skills

(e.g., wrapping web services, extracting contents from web sites, interpreting third-party JavaScript code, etc).

Computer-aided web engineering tools. In order to aid the development of web applications, the web engineering community has so far typically focused on model-driven design approaches. Among the most notable and advanced model-driven web engineering tools we find, for instance, WebRatio [16] and VisualWade [17]. The former is based on a web-specific visual modeling language (WebML), the latter on an object-oriented modeling notation (OO-H). Similar, but less advanced, modeling tools are also available for web modeling languages/methods like Hera, OOHDM, and UWE. All these tools provide expert web programmers with modeling abstractions and automated code generation capabilities, which are however far beyond the capabilities of our target audience, i.e., advanced web users and not web programmers.

Mashup tools. These tools typically provide easy-to-use graphical user interfaces and extensible sets of components for mashup development also by non-professional programmers. For instance, Yahoo Pipes (http://pipes.yahoo.com) focuses on data integration via RSS or Atom feeds via a data-flow composition language. UI integration is not supported. Microsoft Popfly (http://www.popfly.ms) provides a graphical user interface for the composition of both data access applications and UI components. Services orchestration is not supported. JackBe Presto (http://www.jackbe.com) adopts a Pipes-like approach for data mashups and allows a portal-like aggregation of UI widgets (mashlets) visualizing the output of such mashups.IBM QEDWiki (http://services.alphaworks.ibm.com/qedwiki) provides a wiki-based (collaborative) mechanism to glue together JavaScript or PHP-based widgets. Intel Mash Maker (http://mashmaker.intel.com) features a browser plug-in which interprets annotations inside web pages allowing the personalization of web pages with UI widgets.

Although existing mashup approaches have produced promising results, techniques that cater for simple and universal integration of web components are needed. These techniques are necessary to transition Web 2.0 programming from elite types of computing environments to environments where users leverage simple abstractions to create composite web applications over potentially rich web components developed and maintained by professional programmers. With this aim in mind, in the following we describe the mashArt models and system.

4 The mashArt Component Model

The first step toward the universal composition model is the definition of a component model. *MashArt* components wrap UI, application, and data services and expose their features/functionalities according to the mashArt component model. The model described here extends our initial UI-only component model presented in [3] to cater for universal components.

The model is based on four abstractions: state, events, operations, and properties. The *state* is represented as a set of name-value pairs. What the state exactly contains and its level of abstraction is decided by the component developer, but in general it should be such that its change represents something relevant and significant for the other components to know. For example, for our *Process browser* component, we can

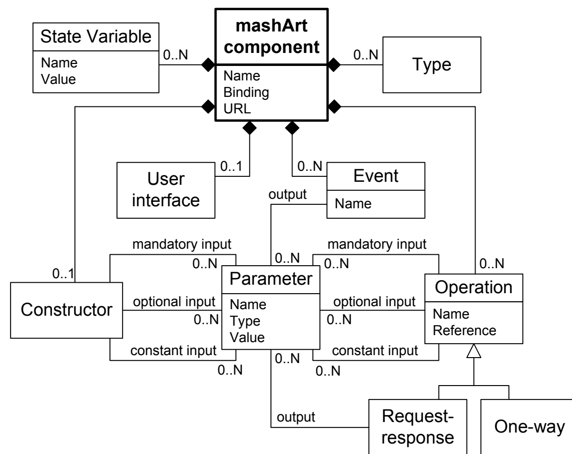

Figure 2 The mashArt component model

change the color in which the process is displayed or rearrange the process graph. This is irrelevant for the other components that need not be notified of these changes. Instead, clicking on a specific process or drilling down on a specific step may lead other components to show related information or application services to perform actions (e.g., compute compliance indicators). This is a state change we want to capture. In our case study, the state for the *Process browser* component is the process or process step that is being displayed. Modeling state for application components is something debatable as services are normally used in a stateless fashion. This is also why WSDL does not have a notion of state. However, while implementations can be stateless, from a modeling perspective it can be useful to model the state, and we believe that its omission from WSDL and WS-* standards was a mistake (with many partial attempts to correct it by introducing state machines that can be attached to service models). For example, an application component may provide relations between compliance policies and processes that need to observe the policies, and can raise a state change event each time processes need to be compliant with newly defined policies, so that other components can be informed and for example change the displayed information or compute compliance indicators for the new policy. Although not discussed here, the state is a natural bridge between application services and data-oriented services (services that essentially manipulate a data object).

Events communicate state changes and other information to the composition environment, also as name-value pairs. External notifications by SOAP services, callbacks from RESTful services, and events from UI components can be mapped to events. When events represent state changes, initiated either by the user by clicking on the component's UI or by programmatic requests (through operations, discussed below), the event data includes the new state. Other components subscribe to these events so that they can change their state appropriately (i.e., they synchronize). For instance, when selecting a process in the *Process browser* component, an event is generated that carries details about the performed selection.

Operations are the dual of events. They are the methods invoked as a result of events, and often represent state change requests. For example, the *Process browser* component will have a state change operation that can request that the component

displays a specific process. In this case, the operation parameters include the state to which the component must evolve. In general, operations consume arbitrary parameters, which, as for events, are expressed as name-value pairs to keep the model simple. *Request-response* operations also return a set of name-value pairs – the same format as the call – and allow the mapping of request-response operations of SOAP services, Get and Post requests of RESTful services, and Get requests of feeds. *One-way* operations allow the mapping of one-way operations of SOAP services, Put and Delete requests of RESTful services, and operations of UI components. The linkage between events and operations, as we will see, is done in the composition model. We found the combination of (application-specific) states, events, and operations to be a very convenient and easy to understand programming paradigm for modeling all situations that require synchronization among UI, application, or data components.

Finally, *configuration properties* include arbitrary component setup information. For example, UI components may include layout parameters, while service components may need configuration parameters, such as the username and password for login. The semantics of these properties is entirely component-specific: no "standard" is prescribed by the component model. Again, they are name-value pairs.

In addition to the characteristics described above, components have aspects that are *internal*, meaning that they are not of concern to the composition designer, but only to the programmer who creates the component. In particular, a component might need to handle the invocation of a service, both in terms of mapping between the (possibly complex) data structure that the service supports and the flat data structure of mashArt (name-value pairs), and also in terms of invocation protocol (e.g., SOAP over http). There are two options for this: The first is to develop ad hoc logic in form of a wrapper. The wrapper takes the mashArt component invocation parameters, and with arbitrary logic and using arbitrary libraries, builds the message and invokes the service as appropriate. The second is to use the built-in mashArt bindings. In this case, the component description includes component bindings such as *component/http*, *component/SOAP*, *component/RSS*, or *component/Atom*. Given a component binding, the runtime environment is able to mediate protocols and formats by means of default mapping semantics; mappings can also be customized (more details are provided in the implementation section).In summary, the mashArt model intuitively accommodates multiple component models, such as UI components, SOAP and RESTful services, RSS and Atom feeds. Figure 2 combines the previous considerations in a meta-model for mashArt components.

In Figure 3 we introduce our graphical modeling notation for mashArt components that captures the previously discussed characteristics of components, i.e., state, events, operations, and UI. *Stateless* components are represented by circles, *stateful* components by rectangular boxes. Components with *UI* are explicitly labeled as such. We use arrows to model *data flows*, which in turn allow us to express events and operations: arrows going out from a component are *events*; arrows coming in to a component are *operations*. There might be multiple events and operations associated with one component. Depending on the particular type of operation or event of a stateless service, there might be only one incoming data flow (for one-way operations), an incoming and an outgoing data flow (for request-response operations), or only an outgoing data flow (for events). Operations and events are bound to their component by means of a simple dot-notation: *component.(operation|event)*.

The actual model of a specific component is specified by means of an abstract component *descriptor*, formulated in the *mashArt Description Language* (MDL) available on the mashArt web site *http://mashart.org/ER09*). MDL is for mashArt components what WSDL is for web services, though considerably simpler and aiming at universal components.

5 Universal Composition Model

Since we target universal composition with both stateful and stateless components, as well as UI composition, which requires synchronization, and service composition, which is more orchestrational in nature, the resulting model combines features from *event-based* composition with *flow-based* composition. As we will see, these can naturally coexist without making the model overly complex.

In essence, composition is defined by linking events (or operation replies) that one component emits with operation invocations of another component. In terms of flow control, the model offers conditions on operations and split/join constructs, defined by tagging operations as optional or mandatory. Data is transferred between components following a pipe/data flow approach, rather than the variables-based approach typical of BPEL or of programming languages. The choice of the data flow model is motivated by the fact that while variables work very well for programs and are well understood by programmers, data flows appear to be easier to understand for non-programmers as they can focus on the communication between a pair of components. This is also why frameworks such as Yahoo Pipes can be used by non-programmers.

To keep the solution simple as per our requirements (yet, as complete and flexible as necessary) we had to make some compromises. For example, the model comes without any structured or complex system activities (e.g., scopes, nested scopes, subprocesses, timers) and does not include transaction management or exception handling. If more complex modeling constructs are necessary (e.g., a join construct with a special data merging function, a complex data transformation service, or a death-path elimination BPEL-style), they can be (i) implemented using the language constructs (although they could require many components and events and render the graph complex), (ii) integrated in the form of dedicated services (implemented as components), or (iii) by creating a BPEL subflow invoked by mashArt (this is supported by the tool but not described here, as it is implementation and not an original contribution). The model and the language described here provide for the necessary basic composition logic, while more complex logics are integrated without requiring any extension at the language level. As we go along and we realize that certain features are crucial, they will be added to the model.

The universal composition model is defined in the Universal Composition Language (UCL), which operates on MDL descriptors only. UCL is for universal compositions what BPEL is for web service compositions (but again, simpler and for universal compositions). A universal composition is characterized by:

- *Component declarations*: Here we declare the components used in the composition and provide references to the MDL descriptor of each component. This allows

access to all component details (e.g., the binding). Optionally, declarations may also contain the setting of constructor parameters.

- *Listeners*: Listeners are the core concept of the universal composition approach. They associate events with operations, effectively implementing simple publish-subscribe logics. Events produce parameters; operations consume them (static parameter values may be specified in the composition). Inside a listener, inputs and outputs can be arbitrarily connected (by referring to the respective IDs and parameter names) resulting into the definition of *data flows* among components. An optional condition may restrict the execution of operations; conditional statements are XPath statements expressed over the operation's input parameters. Only if the condition holds, the operation is executed.
- *Type definitions*: As for mashArt components, the structures of complex parameter values can be specified via dedicated data types.

We are now ready to compose our reference BCM application. Composing an application means connecting events and operations via data flows, and, if necessary, specifying conditions constraining the execution of operations. The graphical model in Figure 3represents for instance the "implementation" of the BCM scenario described earlier. We can see the three UI components *Policy*, *Process* and *Analysis* and the four stateless service components *Repository*, *Engine*, *Analyzer* and *Mail* (*Repository* is invoked two times). The composition has four listeners:

1. If a user selects a policy from the list of policies (*PolicySelected* event), we retrieve the list of processes associated with that policy from the repository (*Repository*.*GetProcsByPolicy* operation). Then we ask the process engine which of those processes are actually deployed in the system (*Engine.GetProcs*) and display the processes (*ShowProcesses* operation) in the *Process* component. In parallel, we also forward the retrieved processes to the *Analyzer* service, which retrieves possible analysis results for the first process (*Analyzer.GetResults*) and causes the *Analysis* component to render them.
2. By selecting another process (*ProcessSelected*) from the list rendered by the *Process* component, the user can view the respective compliance analyses (if any) by synchronizing the *Analysis* UI component (*ShowAnalysis*).
3. If a user selects a process, we retrieve the whole list of policies associated with that particular process (*Repository*.*GetPolicyByProc*) and show it in the *Policy* UI component (*ShowPolicy*).
4. Finally, if by looking at the analysis data the user detects a compliance violation (*ViolationDetected*), she can send an email to a responsible person (*Mail*.*SendMail*).

The graphical model represents the information that is necessary to understand the composition from the composer's point of view. Of particular interest for the structure of the composition is the distinction between stateful and stateless components: Stateful components handle multiple invocations during their lifetime; stateless components always represent only one invocation. This explains why the *Repository* service is placed twice in the model for its two invocations, while the *Analysis* UI component is placed only once, even though it too is invoked twice.

Regarding the semantics of the two data flows leaving the *Engine* service, it is worth noting that we allow the association of a *condition* to each operation. A *condi-*

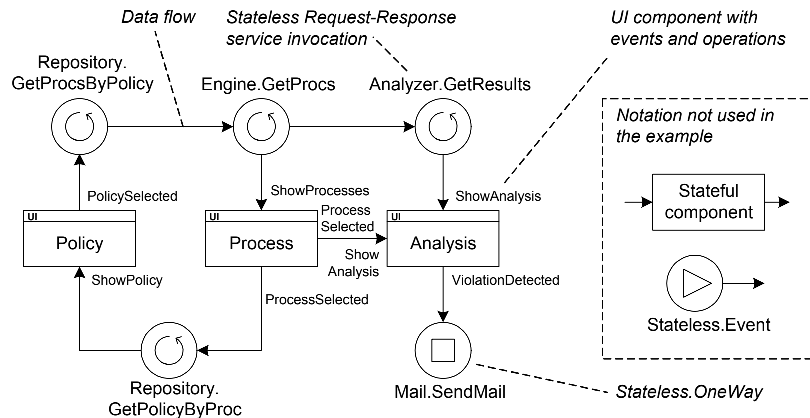

Figure 3 Composition model for the BCM application

tion is a Boolean expression over the operation's input (e.g., simple expressions over name-value pairs like in *SQL where* clauses) and constrains the execution of the operation. The two data flows in Figure 3 leaving the *Engine* service represent a *parallel branch* (conjunctive semantics); if conditions where associated with either *ShowProcesses* or *Analyzer.GetResults* the flows would represent a *conditional branch* (disjunctive semantics). A similar logic applies to operations with multiple incoming flows that can be used to model *join* constructs. Inputs may be *optional*, meaning that they are not mandatory for the execution of the operation. If only mandatory inputs are used, the semantics is conjunctive; otherwise, the semantics is disjunctive.

A branch/join inside a listener corresponds to a *synchronous* branch/join. We speak instead of an *asynchronous* branch/join, when branching and joining a flow requires defining two listeners, one with the branch and one with the join. The listener with the branch terminates with multiple operations; the listener with the join reacts to multiple events or operation results. Again, events may be optional or mandatory. If only mandatory events are used, the semantics is conjunctive; if optional events are used, the semantics is disjunctive. There is no BPEL-style dead path elimination, and in case of conjunctive joins a FIFO semantic is used for pairing events. The combination of events/operations with a graph and with optional/mandatory inputs naturally combine a pub/sub approach with an orchestration approach.

Notice that although the model in the example shows a connected graph, this is not true in general for universal compositions. Indeed, if a composition contains components that need not be synchronized, the respective listeners will be disconnected, resulting in a disconnected directed graph.

Finally, data passing does not require any variables to store intermediate results. Parameter names and data types only refer to the data and the data structures exchanged via data flows. Data transformations are defined by connecting the event or feed parameters with the parameters of the operations invoked as a result of the event triggering. More complex mappings require knowledge about the exact data type of each of the involved parameters. In general, our approach supports a variety of data transformations: (i) simple parameter mappings as described above; (ii) inline scripting, e.g., for the computation of aggregated or combined values; (iii) runtime XSLT

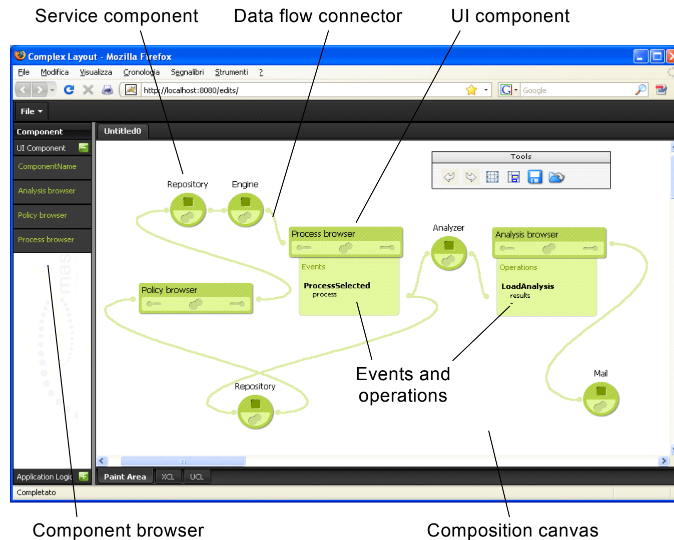

Figure 4 The mashArt editor

transformations; and (iv) dedicated data transformation services that take a data flow in input, transform it, and produce a new data flow in output. The use of the dedicated data transformation services is enabled by UCL's extensibility mechanism.

6 Implementing and Provisioning Universal Compositions

Development environment. In line with the idea of the Web as integration platform, the mashArt editor runs inside the client browser; no installation of software is required. The screenshot in Figure 4 shows how the universal composition of Figure 3 can be modeled in the editor. The modeling formalism of the editor slightly differs from the one introduced earlier, as in the editor we can also leverage interactive program features to enhance user experience (e.g., users can interactively choose events and operations from respective drop-down panels). But the expressive power of the editor is the same as discussed above.

The *list of available components* on the left hand side of the screenshot shows the components and services the user has access to in the online registry (e.g., the *Policy Browser* or the *Registry* service). The *modeling canvas* at the right hand side hosts the composition logic represented by *UI components* (the boxes), *service components*(the circles), and *listeners* (the connectors). A click on a listener allows the user to map outputs to inputs and to specify optional input parameters.

In the lower part of the screenshot, tabs allow users to switch between different views on the same composition: visual model vs. textual UCL, interactive layout vs. textual HTML, and application preview. The layout of an application is based on standard HTML templates; we provide some default layouts, own templates can easily by uploaded. Laying out an application simply means placing all UI components of the composition into placeholders of the template (again, by dragging and dropping

components). The preview panel allows the user to run the composition and test its correctness. Compositions can be stored on the mashArt server.

The implementation of the editor is based on JavaScript and the Open-jACOB Draw2D library (http://draw2d.org/draw2d/) for the graphical composition logic and AJAX for the communication between client and server. The registry on the server side, used to load components and services and to store compositions, is implemented as a RESTful web service in Java. The platform runs on Apache Tomcat.

Execution environment. In developing a mashArt execution environment, the issues that need to be solved include (i) the seamless integration of stateful and stateless components and of UI and service components, (ii) the conciliation of short-lived and long-lasting business process logics in one homogeneous environment, (iii) the consistent distribution of actual execution tasks over client and server, and (iv) the transparent handling of multiple communication protocols. We now detail these issues.

Stateful components may internally maintain state variables as well as the state in their UI, raising events upon state changes. Stateful application components may be implemented as wrappers that manage communications with an external service, the state itself, and possible correlation logic (that is, stateful wrappers may internally embed the analogous of BPEL correlation sets logic, consistently with the approach of pushing complexity to components). As for now, wrappers are implemented by component developers, even though we are implementing mechanisms for embedding state management and correlation management in MDL and UCL extensions.

Short-lived process logics are represented by listeners that involve stateful components or synchronous service invocations only. Such logics can easily be executed at the client side. Stateful components are instantiated inside the client browser or the server-side framework and run there locally. The lifetime of client-side components strictly depends on the user's browsing behavior, e.g., the user might leave the composite application by navigating to another page or by closing the browser. *Long-lasting* process logics are represented by listeners that involve asynchronous service invocations and external notifications or callbacks. Such logics typically require the availability of a web server and a constantly available runtime environment, which can only be guaranteed on the server side. The optimal *distribution* of components and tasks over client and server is another problem that needs to be addressed. For instance, UI components typically run on the client side, while we wait for notifications by an external web service on the server side. Depending on the kind of process logics and the nature of the involved components, the association of components to either the client or the server side may be computed at startup of the composite application. For now, we can handle client-side components and external notifications.

Finally, the handling of *multiple communication protocols* (e.g., SOAP and plain http) requires either the implementation of wrappers or of message adapters that mediate between the native protocols of remote services and the internal message format of the execution environment. Depending on the binding, a suitable protocol adapter is selected. For instance, the *component/http* binding allows issuing arbitrary Get, Post, Put, or Delete http calls to a specified URI. Adapters can be customized for individual components: the content that is sent is specified by a text document (e.g., a SOAP-compliant XML document) that can include references to operation parameters (surrounded by $ signs) that are replaced by the mashArt framework with the actual values at runtime. In this way, we can implement many kinds of message exchanges

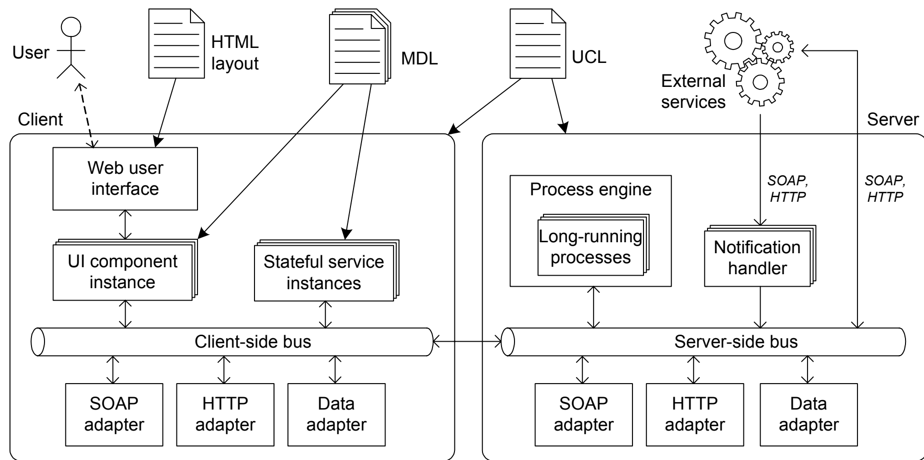

Figure 5 Universal execution framework

(e.g., SOAP- or REST-based). Reply values can be similarly mapped using XPath expressions inside the component definition.

Figure 5 contextualizes the previous considerations in the functional architecture of our execution environment. The environment is divided into a client- and a server-side part, which exchange events via a synchronization channel. On the client side, the user interacts with the application via its UI, i.e., its UI components, and thereby generates events that are intercepted by the client-side event bus. The bus implements the listeners that are executed on the client side and manage the data and SOAP-HTTP adapters. The data adapter performs data transformations, the SOAP-HTTP adapters allow the environment to communicate with external services. Stateful service instances might also use the SOAP-HTTP adapters for communication purposes. The server-side part is structured similarly, with the difference that the handling of external notifications is done via dedicated notification handlers, and long-lasting process logics that can be isolated from the client-side listeners and executed independently can be delegated to a conventional process engine (e.g., a BPEL engine).

The whole framework, i.e., UI components, listeners, data adapters, SOAP-HTTP adapters, and notification handlers are instantiated when parsing the UCL composition at application startup. The internal configuration of how to handle the individual components is achieved by parsing each component's MDL descriptor (e.g., to understand whether a component is a UI or a service component). The composite layout of the application is instantiated from the HTML template filled with the rendering of the application's UI components.

The client-side environment is an evolution of the already successfully implemented and tested UI integration framework of the Mixup project [3], that was however limited to UI components only. The environment comes with an AJAX implementation of the UCL and MDL parsers and is integrated with the mentioned online registry storing components and compositions. The server-side environment has successfully passed a prototype implementation (the effort of several Master theses) based on Java and the Tomcat web server. The integration with the external process engine (e.g., Active-BPEL) and of the client- and server-side parts is ongoing.

A first conclusion that can be drawn from our experiences is that performance does not play a major role on the client side. This is because in a given composition, only a limited number of components run on the client, and the client needs to handle only one instance of the application. On the server-side, performance becomes an issue if multiple composite applications with a high number of long-lasting processes are running in the same web server. Although we did not run scalability experiments yet, the re-use of existing and affirmed technologies, simple servlets for notification handlers, and BPEL engines for process logics will provide for the necessary scalability.

MashArt at work in the BCM example. Once components are in place and we have searched what we need from the registry (via the *registry browser*), we are ready to define universal composite applications. The mashArt ingredients that allow composition are the *graphical UCL editor* for the drag-and-drop development of UCL compositions and the *execution environment* for the hosted execution of ready compositions. Furthermore, an *online monitoring and analysis* tool provides a visual analysis of active and completed executions. The development of our BCM application would thus occur in the following steps:

1. The compliance expert starts the UCL editor and composes the *UCL logic* of the application by putting together the required components, found in the registry.
2. Still in the graphical editor, she can define the applications appearance by applying a simple *layout* template (e.g., an HTML template with <*div*>placeholders; some templates are readily available, own ones can easily be uploaded) and placing the composition's UI components.
3. After checking a preview of the application in the editor, she stores the UCL composition in the *online registry*, and the application appears in the registry browser.

Once the new composite application has been defined, it can be executed either through the registry browser or via a dedicated URI. As the application is started, the *runtime environment* parses the UCL file, loads the layout, and instantiates UI components using the constructor parameters specified in the UCL file. During the execution of the application, the runtime environment logs the occurrence of events and operation calls. Authorized users can then monitor and analyze executions of compositions through an interface that allows the graphical exploration of the events. We discuss neither the monitoring interface nor the authorization model as they do not correspond to significant innovations or contributions of the paper. The authorization model is essentially role-based, while the monitoring and analysis is (in the present version) limited to a graphical process-oriented GUI for monitoring each instance and a reporting infrastructure to view statistics on executions (e.g., average lifetime, statistics on the duration on each operation, detection of outliers).

7 Conclusion

In this paper, we have considered a novel approach to UI and service composition on the Web, i.e., *universal composition*. This composition approach is the foundation of the mashArt project, which aims at enabling even non-professional programmers (or Web users) to perform complex UI, application, and data integration tasks online and in a hosted fashion (integration as a service). Accessibility and ease of use of the

composition instruments is facilitated by the simple composition logic and implemented by the intuitive graphical editor and the hosted execution environment. The platform comes with an online registry for components and compositions and will provide tools for monitoring and analysis of hosted compositions.

The key findings of our work are: (i) state and events/operations are the main abstractions we need for universal integration; (ii) it is possible to provide a simple yet universal composition model by combining synchronization constructs with flow-based ones; (iii) essential to simplicity is the separation of what is simple and exposed to the composer from what is complex and exposed to professional programmers (creating reusable components); (iv) universal composition requires a division of client-side and server-side composition logic for scalability and usability purposes.

Acknowledgments. We thank Maristella Matera, Jin Yu and Regis Saint-Paul for their contribution to the Mixup framework.

8 References

[1] J. Yu, et al., Understanding Mashup Development and its Differences with Traditional Integration, *Internet Computing*, vol. 12, no. 5, 2008, pp. 44-52.

[2] OASIS. Web Services for Remote Portlets, August 2003. [Online]. www.oasis-open.org/committees/wsrp

[3] J. Yu, et al., A Framework for Rapid Integration of Presentation Components, *WWW'07*, 2007, pp. 923-932.

[4] G. Alonso, F. Casati, H. Kuno, V. Machiraju, *Web Services: Concepts, Architectures and Applications*. Springer, 2003.

[5] S. Dustdar, W. Schreiner, A survey on web services composition, *Int. J. Web Grid Services*, vol. 1, no. 1, pp. 1-30, 2005.

[6] OASIS. Web Services Business Process Execution Language Version 2.0, April 2007. [Online]. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

[7] C. Pautasso, BPEL for REST, *BPM'08*, Milano, 2008.

[8] T. van Lessen, et al. A Management Framework for WS-BPEL, *ECoWS'08*, Dublin, 2008.

[9] F. Curbera, et al. Bite: Workflow Composition for the Web, *ICSOC'07*, Vienna, 2007, pp. 94-106.

[10] E. M. Maximilien, et al. An Online Platform for Web APIs and Service Mashups, *Internet Computing*, vol. 12, no. 5, pp. 32-43, Sep. 2008.

[11] D. Braga, et al. Optimization of Multi-Domain Queries on the Web, in *VLDB'08*, Auckland, 2008, pp. 562-573.

[12] F. Daniel, et al. Understanding UI Integration - A Survey of Problems, Technologies, and Opportunities, *IEEE Internet Computing*, pp. 59-66, May 2007.

[13] Microsoft Corporation. Smart Client - Composite UI Application Block, December 2005. [Online]. http://msdn.microsoft.com/en-us/library/aa480450.aspx

[14] The Eclipse Foundation. Rich Client Platform, October 2008. [Online]. http://wiki.eclipse.org/index.php/RCP

[15] Sun Microsystems. JSR-000168 Portlet Specification, October 2003. [Online]. http://jcp.org/aboutJava/communityprocess/final/jsr168/

[16] R. Acerbis, et al. Web Applications Design and Development with WebML and WebRatio 5.0. *TOOLS* (46) 2008, pp. 392-411.

[17] J. Gómez, et al. Tool Support for Model-Driven Development of Web Applications, *WISE'05*, pp. 721-730.