Building Mashups for the Enterprise with SABRE

Ziyan Maraikar[1,⋆], Alexander Lazovik[2,⋆⋆], and Farhad Arbab[1]

[1] Centrum voor Wiskunde en Informatica, The Netherlands
{maraikar,farhad}@cwi.nl
[2] INRIA Saclay, Parc Club Orsay Université, France
lazovik@lri.fr

Abstract. The explosive popularity of mashups has given rise to a plethora of web-based tools for rapidly building mashups with minimal programming effort. In turn, this has spurred interest in using these tools to empower end-users to build *situational applications* for business. Situational applications based on Reo (SABRE) is a service composition platform that addresses service heterogeneity as a first-class concern by adopting a mashup's data-centric approach. Built atop the Reo coordination language, SABRE provides tools to combine, filter and transform web services and data sources like RSS and ATOM feeds. Whereas other mashup platforms intermingle data transformation logic and I/O concerns, we aim to clearly separate them by formalising coordination logic within a mashup. Reo's well-defined compositional semantics opens up the possibility of constructing a mashup's core logic from a library of prebuilt connectors. Input/output in SABRE is handled by service stubs generated by combining a syntactic service specification such as WSDL with a constraint automaton specifying service behaviour. These stubs insulate services from misbehaving clients while protecting clients against services that do not conform to their contracts. We believe these are compelling features as mashups graduate from curiosities on the Web to situational applications for the enterprise.

1 Introduction

A recent trend in web applications has been the emergence of so-called *mashups*. Mashups are web applications that literally mash-up or combine disparate web services, RSS and ATOM feeds and other data sources in new and interesting ways. They compose these services in ways usually unanticipated by their original authors. Mashups are thus, ad-hoc by very nature.

More formal approaches to service-oriented computing (SOC), such as WS-BPEL[9], assume the availability of uniform service interfaces and service metadata available in centralised registries. Yet the Utopian promise of uniform service

⋆ Supported by NWO project BRICKS-AFM3.
⋆⋆ This work was carried out during the tenure of an ERCIM "Alain Bensoussan" Fellowship Programme.

A. Bouguettaya, I. Krueger, and T. Margaria (Eds.): ICSOC 2008, LNCS 5364, pp. 70–83, 2008.

interface standards, metadata and universal service registries, in the form of the SOAP, WSDL and UDDI, standards has proven elusive. Instead, prominent web service providers like Google, Yahoo, Amazon and eBay have opted to use lightweight protocols like RSS and ATOM to push data to consumers, while exposing their service offerings as simple REST-style[10] APIs. In turn, each of these service provider APIs have their own syntax and semantics.

A holistic approach to SOC on the Web, must therefore embrace service heterogeneity as a first-class concern, instead of relegating it to a mere implementation detail. Where mashups succeed is in adopting a data-centric approach to service composition. In a sense, they follow the UNIX tradition of gluing arbitrary programmes together using pipes and text processing tools like *awk* and *sed*, to meet any need. We believe this is a useful approach that complements the traditional control-driven coordination and orchestration exemplified by BPEL.

Situational applications based on Reo (SABRE) aims to have the best of both world by marrying formal SOC with the data-driven approach of mashups. SABRE's contributions are threefold. First, it structures mashups by introducing a clear separation of concerns between service coordination and input/output. Secondly, by using the Reo coordination language[2] (described in §4) we are able to graphically define not just data mangling, but also the *semantics* of service composition, that is, the synchronisation constraints we wish to impose between services. Thirdly, SABRE's automatically generated service stubs isolates faults caused either by services that do not conform to their contracts or by misbehaving clients. We put these ideas into practice by providing a rich suite of tools to graphically define a mashups' data-driven logic and service interfaces with precise formal semantics, automatically generate service stubs and a deployment environment running in an Java application container such as Apache Tomcat.

The rest of this paper is organised as follows. In §2 we review the state of the art in mashup platforms and traditional service composition. A discussion of applicability of mashup platforms for service-oriented computing, together with a motivating example appear in §3. In §4 we consider Reo as a means of formalising coordination of services within mashups. In §5 we apply constraint automata as a unifying formalism for Reo as well as a behavioural specification of services. We describe a reference implementation for the SABRE framework and its architectural issues in §6. We conclude in §7, with a summary of the paper and a discussion of our further work.

2 Related Work

2.1 Mashup Platforms

Mashups can, and are, built using traditional web scripting languages like Perl, PHP and JavaScript. Their explosive popularity has however, given rise to a number of "mashup platforms" for building mashups, with minimal programming. This trend parallels the rise of rapid application development (RAD) tools to ease the development of graphical user interfaces in the 90's.

The use of mashups as *situational applications* in enterprise environments is addressed in [11]. They use the phrase "enterprise information mashup fabric" to describe mashup platforms for the enterprise. DAMIA[1] is a concrete realisation of these ideas for building mashups on corporate intranets. Both SABRE's tools and runtime environment have similarities to DAMIA. However, we are more concerned with enabling compositional construction of mashups with well defined semantics. In this respect the SABRE tools share a common purpose with the SENSORIA Development Environment[21] (SDE) for semantic-based development of service-oriented systems. Whereas SDE is a general-purpose SOC tool suite, SABRE focuses on exclusively on tools for data-driven service coordination.

SABRE also borrows ideas from commercial mashup platforms. Google mashup Editor[1](GME) is a textual mashup tool that uses XML based mark-up, combined with HTML and optionally, JavaScript. Each user-interface element in the mashup is defined using a `module` tag that groups an input data source with a template specifying the format of the resulting output. Yahoo Pipes[2], another commercial offering, takes a graphical approach to mashup creation. A variety of data sources can be plumbed through so-called *pipes* that filter and transform data. Predefined pipes are available to connect to data sources like RSS feeds, REST and SOAP web services, perform string, number, and date manipulation, and filter and sort data. Complex pipes may be composed of primitives and invoke external services. Pipes superficially resemble channels in Reo (§4) and SABRE's editor closely resembles the Pipes Editor, but Reo and its tool suite described in §6 actually predates Yahoo Pipes.

Most mashup platforms commingle the core coordination logic of the mashup with external input/output and user interface concerns. Furthermore, none of them consider the semantics of the services being used. Consequently none support true compositional construction of a mashup's logic. For instance, while Yahoo Pipes supports composing pipes, its notion of composition is limited to data transformation operations. Arguably, this notion of composition is sufficient for building the types of simple mashups commonly found on the Web. As mashups migrate into enterprise environments, however, more formal notions of service composition and coordination become highly desirable. SABRE aims to provide the features necessary to support this use case, without unduly burdening the mashup developer.

2.2 Service Coordination

Service coordination refers to managing interactions among different business processes and any atomic services that they may entail. Currently WS-BPEL[9] and WS-CDL[12] are the most widely used languages dealing with orchestration and choreography, respectively. While BPEL is a powerful standard for composition of services, it lacks support for actual coordination of services. Orchestration and choreography, have received considerable attention in the web

[1] `http://googlemashups.com`
[2] `http://pipes.yahoo.com`

services community, and separate standards (e.g., WS-CDL) are being proposed for them. However, orchestration and choreograph are in fact, different facets of coordination. Thus, it is questionable whether such fragmented solutions for various aspects of coordination, which involve incongruent models and standards for choreography and orchestration, can yield a satisfactory SOA. Most efforts up to now have been focused on statically defined coordination (compositions), as in BPEL. To the best of our knowledge the issues involved in dynamic coordination of web services with continuously changing requirements have not been seriously considered. The closest attempts consider automatic or semi-automatic service composition, service discovery, etc. However, all these approaches mainly concentrate on how to compose a service, and do not pay adequate attention to the coordination of existing services.

In SABRE we use the channel-based exogenous coordination language Reo [2] to specify mashup logic. Reo supports a specific notion of composition that enables coordinated composition of individual services as well as composed business processes. It is claimed that BPEL-like languages maintain service independence, but in practice they hard-wire services through the connections that they specify in the process itself. Reo in contrast, allows us to concentrate only on important protocol decisions and define only those restrictions that actually form the domain knowledge, leaving more freedom for process specification, choice of individual services, and their run-time execution. In traditional SOC, it is often difficult and costly to make any modification to the process, due to the complex relationships among its participants. This is a by-product of forcing a process designer to explicitly define all steps in precise execution order in the process specification, resulting from the use of essentially sequential, imperative, and process oriented languages and tools. It is extremely difficult to adapt such over-specified processes to accommodate even minor deviations in implementation. By placing interaction and its coordination at the centre of attention, Reo lifts the level of abstraction for the specification of composed processes.

Several other formalisms have also been developed for composition and coordination of distributed entities e.g., based on Petri nets and π-calculus, and coordination based on mobile channels[19]. However, these general frameworks do not particularly cater to certain issues in service-oriented computing, e.g., non-deterministic nature of services or late binding of service implementations.

3 Mashups Versus Service-Oriented Computing

The upshot of the discussion in §2 is that while mashups and traditional BPEL-style SOC have their strengths, each has many shortcomings that need to be addressed. Since SABRE attempts to bridge the gap between classical SOC and mashups, we begin by trying to identify how SOC best-practises apply to the mashup building process.

Separation of concerns. By virtue of being ad-hoc there is tight coupling between the data mangling logic, the utilised services and feeds, and user

interface elements that render the data in mashups. SOC regards having a clear separation between these concerns as highly desirable.

Composition and coordination. In the context of a mashup, composition usually boils down to ad-hoc "data mangling"; that is, filtering combining and transforming various inputs to generate desired outputs. Furthermore, a mashup's logic should be composable from reusable blocks to facilitate quick, modular construction. No mashup platforms known to us, has any notion of service semantics. Therefore existing mashup platforms cannot support the notion of service coordination. In SOC is control-driven coordination exemplified by BPEL is the norm, but this style does not mesh well with the data-driven nature of mashups.

Service contracts and fault isolation. A service should have both a syntactic specification, using WSDL for example, and a behavioural specification of its semantics. Such precise service contracts shields services from misbehaving clients, while ensuring services actually adhere to their contracts. This helps isolate faults due to misbehaving clients or services. Although there is extensive research on behavioural specification, no industry standards currently exist.

Service binding. There is an inherent trade-off between flexibility in service binding vs. dealing with service heterogeneity. The late-binding approach advocated in SOC assumes uniform service interfaces, and universal registries. Conversely, mashups handle heterogeneity at the cost of being tightly coupled to the services they use. Ideally, we would like a limited form of late-binding at least for standard input source like RSS feeds.

For the remainder of this paper we use the "Sports-fan Dashboard" example to demonstrate mashup development in SABRE, and highlight how we achieve the the first three improvements identified above. Our example mashup shows rele-

Fig. 1. Example sports-fan mashup user interface

vant information based on the fixture schedule of a sports team. The Dashboard uses an RSS or ATOM feed containing a schedule of team fixtures. A fixture calender for Ajax FC for example, is available on Google Calendar[3]. Once the user chooses a match of interest the Sports-fan Dashboard display the following information: (i) a map showing the venue; (ii) news articles about the match; (iii) weather forecast for the day of the match; (iv) option to purchase tickets online, if the weather forecast is "good"[4]. Figure 1 shows a screenshot of the running application.

4 Specifying Mashup Logic with Reo

We formalise the the core logic within a mashup by encoding it in Reo. Reo is *exogenous* in that it imposes a coordination pattern on components, without any knowledge of the internals of the components and without the components having any knowledge of the coordination. This makes Reo ideal for coordinating services from a data-centric perspective. Coordination in Reo is specified by a *connector* consisting of nodes and primitives. Formally, a Reo connector is defined as follows:

Definition 1 (Reo connector). *A* connector $\mathcal{C} = \langle \mathcal{N}, \mathcal{P}, E, node, prim, type \rangle$ *consists of a set* $\mathcal{N}$ *of* nodes*, a set* $\mathcal{P}$ *of* primitives*, a set* E *of* primitive ends *and functions:*

- $node : E \rightarrow \mathcal{N}$*, assigning a node to each primitive end,*
- $prim : E \rightarrow \mathcal{P}$*, assigning a primitive to each primitive end,*
- $type : E \rightarrow \{src, snk\}$*, assigning a type to each primitive end.*

A *node* is where the execution of different primitives is synchronised. Data flow at a node occurs, iff (i) at least one of the attached sink ends provides data and (ii) all attached source ends are able to accept data. A node does a destructive read at one of its sink ends and replicates the data obtained to every one of its source ends.

Typically, Reo primitives are *channels*. Channels can be attached to nodes to compose connectors. The ends of a channel can be either source ends, which accept data or sink ends which produce data. The actual semantics of a channel depends on its type. Reo does not restrict the possible channels used as long as their semantics is provided. Nodes, however, have the fixed semantics defined above, which specifies their routing constraints. Table 1 describes the behaviour of some common Reo channels. The top three channels represent synchronous communication. A channel is called *synchronous* if it delays the success of the appropriate pairs of operations on its two ends such that they can succeed only simultaneously. Note that a Reo connector built from synchronous channels is stateless and its execution is instantaneous in an all-or-nothing fashion. The

[3] `http://www.google.com/calendar/embed?src=jdtvbcrk1vtpn5ec4ptnnfnd6lc0udfprt.calendar.google.com`

[4] Admittedly this was contrived to show the utility of channel composition in Reo.

Table 1. Behaviour of common Reo channels

Sync		Simultaneously accepts data on one end and passes it out the other end
SyncDrain		Simultaneously accepts data on both ends
SyncSpout		Simultaneously produces data on both ends
LossySync		Behaves as a *Sync* if a *take* operation is pending on the output end, otherwise the data is lost
Filter		Passes data matching a filter pattern and loses the rest
FIFO		Buffers a single data item.

FIFO channel enables us to add stateful behaviour to a connector. An extensive discussion of Reo, various channel types and numerous examples can be found in [2]. Formal semantics for Reo has been given using constraint automata[6], connector colouring[7] and structured operational semantics[17].

Once channels are composed into a complex connector, it can be used disregarding its internal details. As an example consider the XOR element shown in Figure 2, built out of five sync channels, two lossy sync channels, and a sync drain. The intuitive behaviour of this connector is that data obtained as input through A is delivered to one of the output nodes F or G. If both F and G is willing to accept data then the merger at node E non-deterministically selects which side of the connector will succeed in passing the data. The sync drain channel B-E and the two C-E, D-E channels ensure that data flows at only one of C and D, and hence F and G.

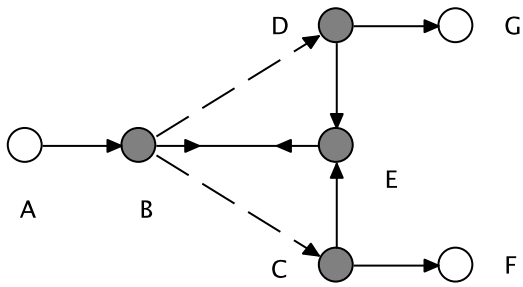

Fig. 2. XOR connector

Services are independent distributed entities that utilise Reo channels and connectors to communicate. The service implementation details remain fully internal to individual elements, while the behaviour of the whole system is coordinated by the Reo circuit. A deeper treatment of using Reo for service coordination is given in [14]. We discuss how we interface services with Reo in SABRE is §5.1.

4.1 Building the Sports-Fan Dashboard in SABRE

Augmenting the Reo tool suite with filter and transformer channels, gives SABRE the data mangling functionality common to other mashup platforms. Filter channels take a filter expression e.g. a data type or regular expression to match against. If a datum matches the filter expression it passes through the channel, otherwise it is dropped . A transformer channel, likewise, accepts a data transformation expressed e.g. as a *sed*-like text replacement or using XPath or XSLT. Each input datum is rewritten according to the transform expression as it passes through. Filters and transformers can also execute user-defined functions. For example, a geo-coding channel that converts place names to latitude and longitude can invoke an external service. Semantically, a filter acts as a specialised lossy sync channel while a transformer acts as a sync.

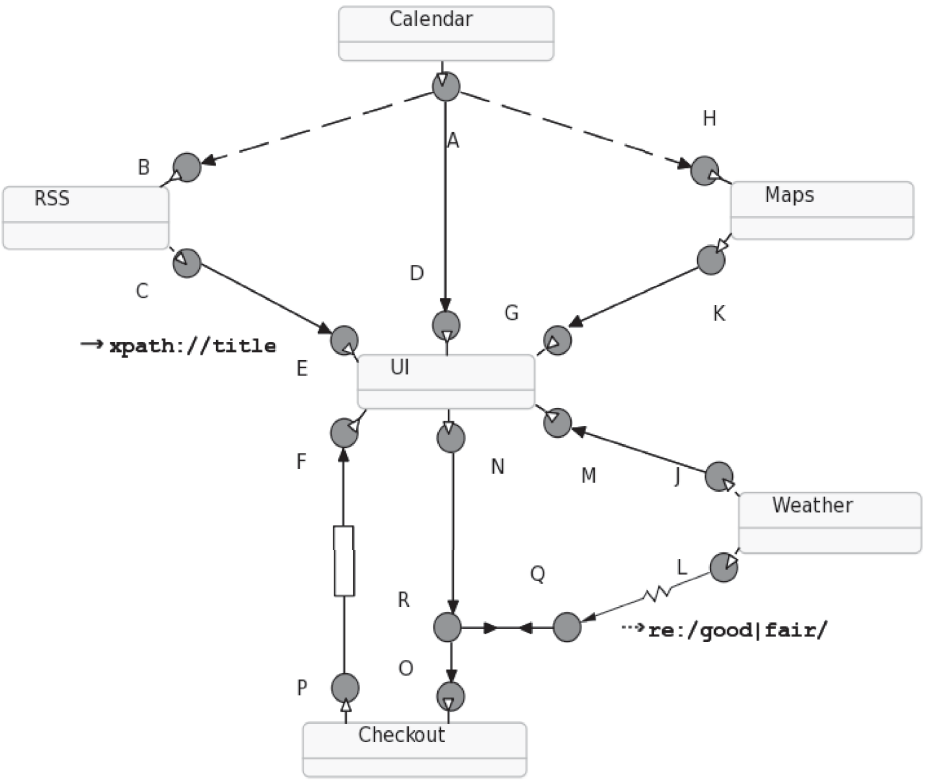

Fig. 3. Reo connector implementing Sports fan mashup's coordination

Figure 3 shows the coordination in our sports fan application defined in Reo. The connector works as follows. Whenever the calendar service has a new event, it is written to node A. From A, the data is passed to node D when the user

interface is ready to accept it. In parallel, data about the new event is also transferred to the RSS and map services, if they are on-line. Otherwise, the lossy channels A-B and A-H just discard the data. The map service provides the location information to the user interface (through the K-G channel) and to the weather service to retrieve the local forecast (K-J channel). The transformer C-E uses an XPath expression to extract titles from the RSS feed for display in the user interface. Whenever the user presses the "Proceed to payment" button, the connector passes the required data through channel N-R. The SyncDrain channel Q-R prevents data flow through the N-R-O pair of channels if the output of the weather service does not "approve" it through the filter channel L-Q, which accepts the data only if the forecast weather forecast is either "good" or "fair". When the ticket reservation is approved (via a Google Checkout test payment account), the result is put into the FIFO channel P-F. The user interface application then receives the current payment status from the buffer.

Using Reo as the basis for coordination offers a number of advantages. A unique feature of SABRE is that connector in Figure 3 completely specifies not just the data mangling logic, but also synchronisation constraints between services. Just as in Yahoo Pipes, SABRE's transformer and filter channels can be composed to effect aggregate data filtering and transformation operations. However, thanks to Reo's *semantic* compositionality, we can also define much more powerful constructs such as the XOR connector in Figure 2. Using library of such predefined connectors, a mashup's coordination logic can be composed with precise formal semantics.

5 Behavioural Specification of Service Using Constraint Automata

Numerous formalisms for behavioural specification have been proposed such as I/O automata[16] and open workflow nets[15]. We use constraint automata[6] to specify the behaviour of services that interact with a SABRE mashup. Constraint automata enables the to use the same formalism as an operational model for the core mashup logic modelled in Reo and for behavioural specification of services we interface with.

Definition 2 (Constraint Automaton[6]). *A constraint automaton (over the data domain Data) is a tuple* $A = (Q, \mathcal{N}, \rightarrow, Q_0)$ *where*

- Q *is a set of states,*
- $\mathcal{N}$ *is a finite set of port names,*
- $DC(\mathcal{N}, Data)$ *the data constraints, are sets of port name - data assignments,*
- $\longrightarrow$ *the transition relation of* A *is a subset of* $Q \times 2^{\mathcal{N}} \times DC \times Q$
- $Q_0 \subseteq Q$ *is the set of initial states.*

For every transition $(q, \mathcal{N}, g, p) \in \longrightarrow$ *we require that: (1)* $\mathcal{N} \neq \emptyset$*, and (2)* $g \in DC(\mathcal{N}, Data)$*.*

A thorough treatment of using constraint automaton as an operational model for Reo connectors can be found in [6]. Intuitively, states represent the configurations of the connector, the transitions the possible one-step behaviour. The meaning of $q \xrightarrow{(N,g)} p$ is that in configuration q the port names $A_i \in N$ have the possibility to perform I/O operations that meet the guard g and that lead from configuration q to p, while the other ports $A_j \in \mathcal{N} \backslash N$ do not perform any I/O-operation. We discuss the use of constraint automata for service specification below.

5.1 Interfacing Web Services with the Sports-Fan Dashboard

Input/output considerations are an integral part of mashup design. We use constraint automata to specify service behaviour in the typical fashion that labelled transition systems are used as formal models for reactive systems. SABRE uses stubs automatically generated from behavioural specifications to bind to services. We extend the method of specifying service behaviour using constraint automata described in [20] to allow for asynchronous service invocation.

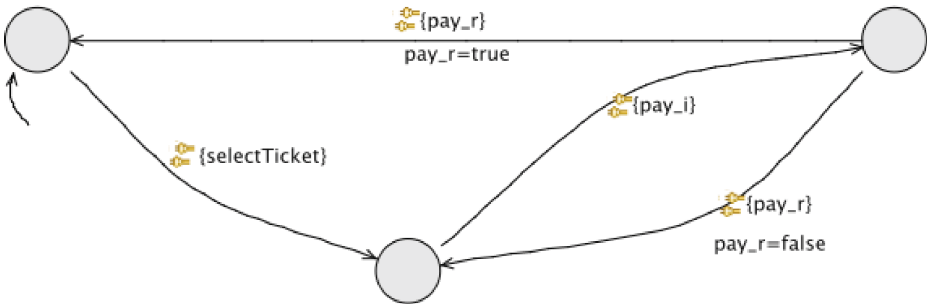

Fig. 4. Constraint automaton specifying the behaviour of the Checkout Service

We use the "Checkout" service in Figure 3 to demonstrate behavioural specification using constraint automata. Suppose this service consists of two operations: `selectTickets` and `pay`. We map each synchronous operations in the service interface to a constraint automaton port with the same name. Asynchronous operations are mapped to two port names suffixed by `_i` and `_r`, corresponding to the operation invocation and return, respectively. In Figure 4, we specify that `selectTickets` is a synchronous operation by placing the single port name on a transition. Following `selectTickets`, a client must invoke the asynchronous `pay` operation that returns a boolean value indicating success or failure. We use constraints to specify data-dependent state changes. For example, based on the return value of the `pay` invocation, we either request payment again, or permit the client to select another ticket to purchase.

6 Implementation

The SABRE implementation consists of a mashup design tool and a runtime environment for mashup execution. The design tool is an enhanced version of the Eclipse Coordination Tools [3] (ECT) — a suite of graphical tools for Reo. Built on the Eclipse Graphical Modelling Framework, ECT consists of graphical editors for Reo connectors, an animator for visualising a connector's semantics, transformation from Reo to constraint automata, and a model checker for constraint automata. The addition of filter and transformer channels brings the data mangling features found in graphical mashup builders like Yahoo Pipes to ECT's Reo editor. We are integrating the Smooks data transformation framework[5] into SABRE, to enable more powerful filter constraints and data transformations to be specified in a declarative fashion.

SABRE's execution environment depicted in Figure 5, consists of a Reo engine and a management interface, hosted in a servlet container such as Tomcat. Reo has several executable implementations available, any of which may be used to run a SABRE mashup. ReoCP is a constraint programming engine that directly executes a Reo circuit based on the colouring semantics for Reo [8]. CASP [3] generates Java code from a constraint automaton representation of a Reo connector. A distributed Reo implementation [18] on Scala Actors is ongoing. Any one of these engines may be plugged into the SABRE runtime via a common interface, depending on the specific deployment needs of the application. For example, a deployment requiring run-time changes to the connector may choose to use ReoCP, while deploying a very large connector is best done using distributed Reo.

The management interface lets a user deploy Reo connectors, and start and stop connector instances via a web browser. Once a connectors is deployed and started, the runtime initialises the Reo engine with the given connector and instantiates service stubs and other server-side components. Stubs for services (ovals on the left) and user interface elements (ovals on the right), depicted in Figure 5, communicate with the engine via synchronous read and write operations on ports (arrows) of the Reo connector being run by the engine. The SABRE runtime also maps ports to URLs, which may be used by remote components to read and write data to ports using HTTP GET and PUT operations respectively.

SABRE generates Java service stub classes from a constraint automaton and a Java interface[6] declaring operations a service provides. Each start state of the automaton is mapped to a thread which listens for reads and writes on the ports of the outgoing transitions of the current state. Once all ports of a transition are active, the thread invokes the corresponding operation(s) with the parameters received on invocation ports and/or writes any return values to return ports.

For user interface creation we envisage a library of common user interface elements like maps and clickable lists. A user interface element may either execute

[5] `http://milyn.codehaus.org/Smooks`

[6] WSDL specification can be easily translated to Java using tools like Apache Axis.

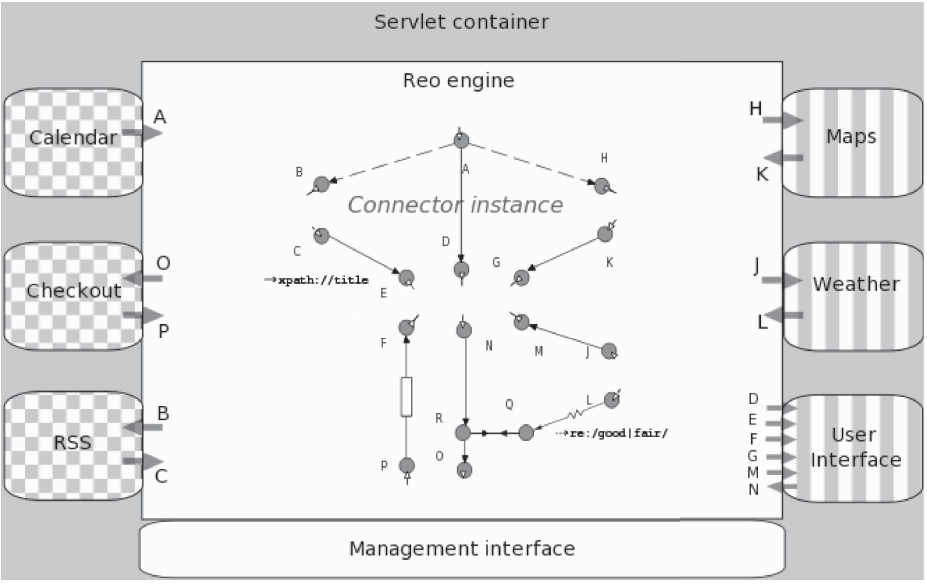

Fig. 5. Sports-fan Dashboard deployed on the SABRE runtime environment

locally on the same server as the mashup, or remotely on the user's browser, such as a component using the Google Maps JavaScript API. Such browser-based UI elements may use JavaScript's `XMLHTTPRequest` object to perform I/O via the URL mapped ports, with the Reo connector executing on the server[7].

7 Conclusion and Future Work

We have presented a framework for rapid composition of heterogeneous data and services on the Web. Rather than approach service coordination from the traditional control flow perspective, we take a data-centric view inspired by mashups. The SABRE framework permits compositional construction of mashups with precise semantics and fault isolation, without sacrificing rapid development and flexibility inherent in mashups. The synchronous semantics of Reo gives us simple transactions that can ensure that a chain of components connected by channels all execute atomically. Compensation is another major concern in specification and implementation of business processes that involve long running transactions. We intend to adapt the schemes used to translate the compensation constructs available in the BPMN standard to provide a compensation mechanism for SABRE [4]. In summary, SABRE improves on the current state of the art in mashup construction platforms by leveraging the strengths of Reo, by using it as the basis for formalising coordination logic in a mashup.

[7] This is the same technique known as *AJAX* in common parlance.

Specifying service behaviour in constraint automata is onerous for casual mashup development. An alternative is to describe a service by a UML sequence diagram and then extract the behavioural specification in the form of a constraint automaton [5]. Ongoing work on Reo also makes it possible dynamically reconfigure connectors based on graph transformations [13]. Dynamic reconfiguration opens up possibilities such as run time service discovery and binding. Finally, we would like to add more prepackaged components along the lines of Yahoo Pipes and streamline SABRE's graphical interface to make it accessible to non-technical end-users.

References

1. Altinel, M., Brown, P., Cline, S., Kartha, R., Louie, E., Markl, V., Mau, L., Ng, Y., Simmen, D., Singh, A.: Damia - a data mashup fabric for intranet applications. In: VLDB, pp. 1370–1373 (2007)
2. Arbab, F.: Reo: a Channel-based Coordination Model for Component Composition. Mathematical Structures in Computer Science 14, 329–366 (2004)
3. Arbab, F., Koehler, C., Maraikar, Z., Moon, Y., Proença, J.: Modeling, testing and executing Reo connectors with the Eclipse Coordination Tools. In: Proceedings of FACS, SCP (to appear, 2008)
4. Arbab, F., Kokash, N., Sun, M.: Towards using Reo for compliance-aware Business Process Modelling. In: Proceedings of ISOLA (to appear, 2008)
5. Arbab, F., Meng, S.: Synthesis of connectors from scenario-based interaction specifications. In: Chaudron, M.R.V., Szyperski, C., Reussner, R. (eds.) CBSE 2008. LNCS, vol. 5282, pp. 114–129. Springer, Heidelberg (2008)
6. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling component connectors in Reo by Constraint Automata. Sci. Comput. Program 61(2), 75–113 (2006)
7. Clarke, D., Costa, D., Arbab, F.: Connector colouring I: Synchronisation and context dependency. Electr. Notes Theor. Comput. Sci. 154(1), 101–119 (2006)
8. Clarke, D., Proença, J., Lazovik, A., Arbab, F.: Deconstructing Reo. In: Proceedings of FOCLASA. ENTCS (to appear, 2008)
9. Curbera, F., Goland, Y., Klein, J., Leymann, F.: Business process execution language for web services. Technical report, IBM (2002), `http://www.ibm.com/developerworks/library/ws-bpel/`
10. Fielding, R.: Architectural styles and the design of network-based software architectures. PhD thesis, Chair-Richard N. Taylor (2000)
11. Jhingran, A.: Enterprise information mashups: Integrating information, simply. In: VLDB, pp. 3–4 (2006)
12. Kavantzas, N., Burdett, D., Ritzinger, G.: Web services choreography description language (WS-CDL) version 1.0. Working draft, W3C (2004), `http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427`
13. Koehler, C., Costa, D., Proenca, J., Arbab, F.: Reconfiguration of Reo connectors triggered by dataflow. In: Proceedings of GT-VMT. Electronic Communications of the EASST, vol. 10 (2008)
14. Lazovik, A., Arbab, F.: Using Reo for service coordination. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 398–403. Springer, Heidelberg (2007)

15. Lohmann, N., Massuthe, P., Wolf, K.: Behavioral constraints for services (chapter Behavioral Constraints for Services). In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 271–287. Springer, Heidelberg (2007)
16. Lynch, N., Tuttle, M.: An introduction to input/output automata. Technical report, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands (1980)
17. Mousavi, M., Sirjani, M., Arbab, F.: Formal semantics and analysis of component connectors in Reo. ENTCS 154(1), 83–99 (2006)
18. J. Proença Towards Distributed Reo. In: CIC workshop (2007), `http://homepages.cwi.nl/~proenca/distributedreo`
19. Scholten, J., Arbab, F., de Boer, F., Bonsangue, M.: A component coordination model based on mobile channels. Fundam. Inform. 73(4), 561–582 (2006)
20. Sun, M., Arbab, F.: Web Services Choreography And Orchestration In Reo And Constraint Automata. In: Proceedings of the 2007 ACM Symposium on Applied Computing (SAC), pp. 346–353. ACM, New York (2007)
21. Wirsing, M., Clark, A., Gilmore, S., Hölzl, M., Knapp, A., Koch, N., Schroeder, A.: Semantic-Based Development of Service-Oriented Systems. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 24–45. Springer, Heidelberg (2006)