A Underlay System for Enhancing Dynamicity within Web Mashups

Heiko Pfeffer
Technische Universität Berlin
Franklinstr. 28/29, 10587 Berlin, Germany
heiko.pfeffer@tu-berlin.de

Abstract

Rich Internet Applications (RIA) and composed Web applications, referred to as Mashups, have become the new generation of Web based applications, aggregating multimedia data such as audio, video and images from multiple providers and combining them to more powerful and value-added applications. In the same time, mobile devices such as smartphones or PDAs are increasingly used to access the Web and Web based applications. Therefore, Web developers nowadays face a huge client heterogeneity, where devices differ in their computing capabilities, screen size, available bandwidth, and mobility pattern.

This paper introduces a service composition model that can underlie modern Web applications and can be executed within a respective runtime at every browser-enabled client. This model allows software develops to create applications by abstracting from concrete services and APIs, which are incorporated dynamically during runtime. Thus, the resulting Underlay System for Web Mashups overcomes device heterogeneity by providing means to execute the application logic in a unified runtime while integrating the concrete service implementations based on the current context of the user and the respective client device.

Keywords: *Service Composition, Mashups, Composed Web Applications, Workflows, Timed Automata, Realtime*

1 Introduction

In 2003, Macromedia coined the term of Rich Internet Applications (RIA), describing a new breed of upcoming Web applications that combines the best of desktop software, the Web, and advanced communication technologies [11]. Here, the aim is to build highly interactive Web applications that feature a broad spectrum of data such as audio, video, images and text in order to enhance the experience of the user. Lawton later introduced an update on the definition of RIA, highlighting the importance of the usage of Web technologies [18]. At the same time, composed Web applications, i.e. applications that were built by the incorporation of content from multiple 3rd party service providers, referred to as Mashups, gained importance. On ProgrammableWeb.com [1], 1318 open APIs and 3985 Mashups were registered in May 2009. The most prominent motives for offering data to communities of users and software developers are the generation of new ideas by independent developers, the search for new revenue streams and the opportunity to draw traffic to the providers' sites and remain or become a dominant player in the Web. Through publicly available APIs the community often develops new features without assistance from the providers, posing the possibility of vertical integration for the providers.

However, the development of such Web Mashups as a composition of 3rd party services is aggravated by differing interface descriptions of 3rd party services and their varying return types that considerably reduce the amount of reusable code and forestall an easy swapping of services with similar functionality. Thus, today's composed Web applications remain tailor-made for specific purposes.

Other domains have already faced similar challenges and responded with respective mechanisms. For instance, architecting distributed applications in open systems with SOA-based software patterns has proven as good practice to achieve robustness and extensibility. Inter-component dependencies are loosened to on-demand selection and purpose-driven interaction at runtime, service consuming and service providing entities stay autonomic; this central concept is often referred to as late binding of services.

However, SOA based applications incorporate a huge protocol stack and are thus a bad fit for modern Web applications based on the REST architectural style [13]. Moreover, SOAs are tailor-made to support large-scale business processes where end-users are not the center of attention as opposed to Web Mashups. Business processes completely lack a presentation layer to display results of computations and to interact with the user. Moreover, they do not provide suitable means to remain responsive to user preferences or context or to support a broad device heterogeneity,

i.e. mobile users accessing Web applications through small portable devices with restricted computing capabilities.

Within the remainder of this work, a Underlay System for Web Mashups is introduced that considerably reduces the problems entailed by a high device heterogeneity by incorporating services based on the current context during runtime. Here, section 2 first gives a general introduction to composed applications, highlighting results from the world of SOA and the Web, respectively. The conceptual part of section 3 discusses the Underlay System itself and builds upon results given in [29], while section 4 highlights an architecture for realizing the proposed approach.

2 State of the Art and Related Work

This section gives a short introduction to the most important concepts relevant within the scope of this work and relates it to similar approaches. While subsection 2.1 focuses on the concept of service composition and workflow languages, subsection 2.2 deals with a respective grounding to the Web architecture, discussing Mashups as a poster child for composed Web applications.

2.1 Service Composition

Service Oriented Architecture (SOA) and Web Services have successfully addressed the problem of Service Composition for business processes by introducing appropriate composition languages [5], enabling the controlled execution of multiple accumulated Web Services. However, those technologies such as UDDI and SOAP strictly rely on fix infrastructures for service discovery and provisioning. First steps towards perpetuating the success of SOA and Web Services to mobile networks have been achieved by providing service frameworks for accessing Web Services from mobile devices such as PDAs and Smartphones [23]. A static service infrastructure for service discovery and provisioning is nevertheless still indispensable. Successful Service Composition methods remain tailor-made for business processes [17].

Within the world of SOA and Web Services, various models for service composition are present. The currently most prominent one is BPEL4WS [3, 8], a combination of IBM's graph-based Web Service Flow Language (WSFL) [19] and Microsoft's block-oriented XLANG [30]. Beside, many other composition languages exist, wherefrom none has made the breakthrough to an holistically accepted standard for Web Service Composition [33]. In order to provide more dynamic service composition approaches featuring a functional-based service discovery and binding, some approaches have replaced the Web Service Description Language (WSDL) [6] by semantic service descriptions such as OWL-S [31]. Especially for mobile devices, more lightweight descriptions have been proposed in order to reduce the computation complexity entailed by semantic reasoning processes [27].

However, dynamic service composition faces the difficulty that new semantic service descriptions have to be generated that can be used for future service discovery and binding. The approach introduced in this paper therefore aims at extending the flexibility of service compositions achieved trough late binding mechanisms to the structure of service composition plans. This proceeding entails the generation of multiple equivalent or similar service compositions with regard to their functionality, which may however differ in their non-functional properties.

2.2 Mashups - Composed Web Applications

Mashups are composed Web applications integrating data from multiple 3rd party providers to provide a value-added functionality and experience to the user. Within this section, we first outline the three basic roles within Mashup based Web applications and then discuss two different Mashups styles that denote possible ways to realize them, building upon [21] and [22] mainly.

2.2.1 Mashup Roles

In general, Mashups architectures feature three elementary roles as illustrated in Figure 1. The Content Provider is a source of data that can be accessed through open APIs over various Web protocols such as REST, RSS or SOAP. The Mashup Site is the new build Web application that requests content and services from various data sources and combines them in order to provide a value-added application to the user. The Client is the interface to the user (presented within a Web-browser). The user can interact with the Mashup through client-side scripting languages such as JavaScript.

Within the following section, we will first focus on the role of content providers, introducing famous and successful services that are exposed through open APIs. Here, special emphasis will be put on the way they expose their APIs, the types of data they return and how their integration can characterize the resulting Mashup. Thereafter, we will discuss basic styles of Mashups, highlighting the different responsibilities of clients and Mashup sites within the respective approaches.

2.2.2 Mashup Styles

In general, it is distinguished between two basic styles for creating and executing Mashups, entailing different responsibilities of the client and the Mashup site in order to execute the actual Mashup. Both styles expose serious advan-

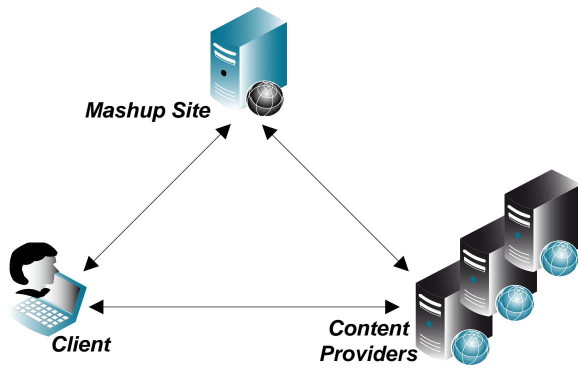

Figure 1. General Roles of a Mashup Web Application.

tages and disadvantages with regard to their performance, load distribution and security, and thus have to be pondered against each other carefully with regard to the requirements of the respective Mashup application.

Server-side Mashups Within server-side Mashups, services and content are integrated at a server, which plays the role of a proxy between the Web application on the client and other Web sites that are integrated into the Mashup. Every request or event originating from the client is forwarded to this proxy server, which then makes the calls on the respective Web sites. Because of this central server role acting as a proxy, server-side Mashups are often referred to as Proxy Mashups. Figure 2 shows the general setting of a server-side Mashup.

Whenever a user generates an event at the client Web browser, e.g. pushes a button or clicks on a map, the event triggers a JavaScript function (#1). This Java Script function makes a HTTP request to the Mashup site (#2). Commonly, this request is an Ajax request; it will later be discussed in greater detail. The request is received by a Web component such as a Servlet or Java Server Page (JSP) on the Mashup site. Based on the received request, the component calls one or multiple methods within a class or multiple classes containing the application logic to make calls on 3rd party APIs (#3). Because of their role of acting as a proxy between the client's request and the request to the actual Mashup servers, these classes are often referred to as Proxy Classes. Note that proxy classes are not restricted in the way they are realized, thus, they can be implemented as plain Java classes or large-scale J2EE components. Thus, the Proxy classes connect to the addressed Mashup servers and request the respective services (#4). In turn, the Mashup servers process the request and respond with the resulting data (#5). The proxy classes receive the response and can process the data before forwarding it to the Web component (#6). This option of processing data on the server side enables most of the advantages of a server-side Mashups. For instance, data can be transformed in a format such as JSON that is easier to process by the client, it can integrate different data sources and only send the integrated piece of data to the client, or data can be buffered or cached in order to increase the performance of succeeding requests. Finally, the Web component sends the response to the client (#7). Here, the view of the page at the client side is updated according to the response. In case the initial request has been an Ajax request in form of an XMLHttpRequest object, this page update is achieved by the callback function within the XMLHttpRequest that manipulates the Document Object Model (DOM) accordingly.

Client-side Mashups Within client-side Mashups, the integration of the single services and data sources is performed at the client instead of using an additional proxy server. Thus, clients connect directly to 3rd party APIs in order to request services; a client-side Mashup is abstractly depicted in Figure 3.

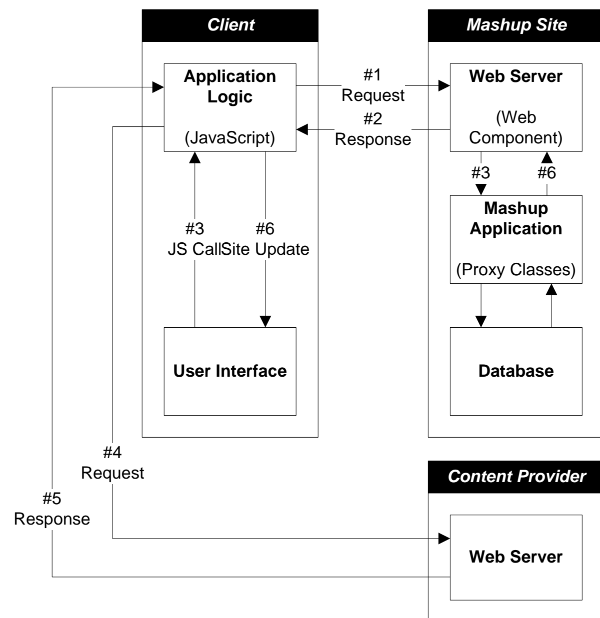

Figure 3. Client-side Mashup.

Initially, the client's browser makes a request to the Mashup site (#1), initiating the server to load the requested Web page into the client (#2). This Web page provides access to JavaScript libraries that enable the client later to directly call services at 3rd party APIs without addressing the Mashup site beforehand. There are three common ways to provide access to JavaScript libraries. First, the Web

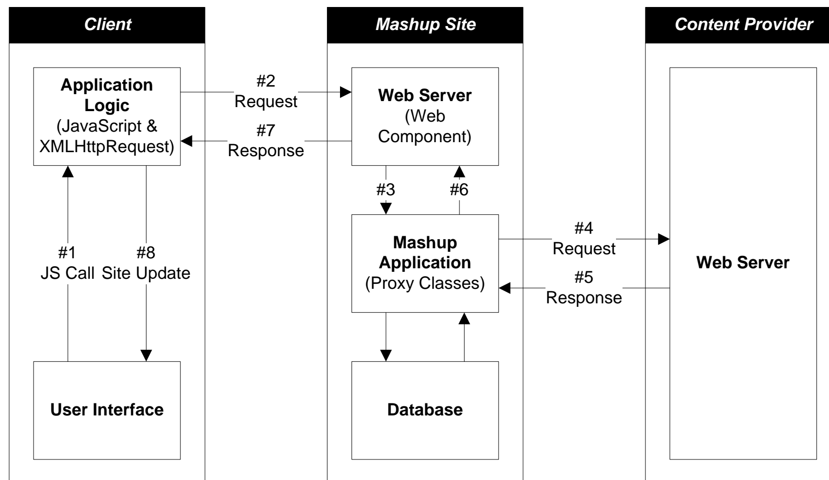

Figure 2. Server-side Mashup.

page may reference the JavaScript library from the respective 3rd party service provider such as Google Maps; here, it is sufficient to reference the library by a valid URL. In case the 3rd party provider does not expose an appropriate JavaScript library, the Mashup developer can provide one on his own and make it available on the Mashup site. Alternatively, other 3rd party libraries may be referenced to ease the Mashup's creation. Specific actions on the Web page trigger the browser to call a function in the JavaScript library integrated within the Web page. This function dynamically creates a <script> tag in the Web site that points to the according 3rd party server (#3). Afterwards, the client initiates an HTTP request based on the ¡script¿ tag including the desired format of the response (#4). As discussed above, the format of the response provided by a service can commonly vary, ranging from XML to YAML or JSON. For client-side Mashups, JSONP (JSON with Padding) is the most common response format since it can be easily evaluated (by the JavaScript function *eval()*) at the client. JSONP extends JSON by the capability of appending the name of a local callback function to the JSON object. Thus, in case the server provides the response in JSONP, a call is made on the callback function with the JSON object as parameter (#6). Finally, the DOM of the page is manipulated by the JavaScript function and the page is updated accordingly.

3 Underlay System for Web Mashups

Within this section, a Underlay System for Web Mashups is specified by introducing a service composition model that can describe and execute composed Web applications, i.e. Mashups, and is -in the same time- responsive to the environmental user context. Special emphasis is put on overcoming the high degree of device heterogeneity we face in today's Web.

3.1 Distinguishing Mashups and Business Processes

Business processes and Web Mashups both constitute composed applications. However, they are created with different requirements. While business processes are tailor-made to describe large-scale business transactions among multiple enterprises, Mashups are rapidly created Web applications for end-users aiming at a high degree of interactivity and graphical presentation, where content from 3rd party providers is combined to add value to the created Web application.

The fact that Mashups are built on top of the Web protocol stack while business processes are designed for SOAs is also reflected in the underlying technologies for remote service invocation and service interworking as shown in Table 1.

Moreover, Table 1 outlines the focus of both technologies. Business processes are defined by service composition languages such as BPEL and provide late binding mechanisms for services that are described by languages such as WSDL and stored in repositories where they can be accessed via UDDI. Thereby, enterprises can modify the implementations of their single services (e.g. upgrade them to a new version) without any need to notify service consumers of their changes since the services are incorporated by their descriptions only and thereby decoupled from the actual service implementation. Mashups on the contrary do not provide any kind of controlled service execution or late binding of services. Instead, Web developers directly integrate the concrete APIs of 3rd party providers into their

Feature	*Business Processes (SOA)*	*Web (Mashups)*
Remote Invocation	WebServices (WSA), XML-RPC	XMLHttpRequest (Ajax), JSON RPC, COMET
Service Interworking	.NET, J2EE	EcmaScript (Java Script)
Service Composition	**Yes**	**No**
Composition Languages	BPEL4WS, WSFL, WSCDL	
Composition Execution Engines	Bexee, Oracle BPEL Process Manager, IBM WBISF	
Late Binding	**Yes**	**No**
Service Description Languages	WSDL, WSDL-S, OWL-S	
Service Discovery	UDDI	
Interaction	**No**	**Yes**
Presentation		HTML, CSS, XML, Streamed audio and video
User Input		HTML Forms, XForms, JS Keyboard/Mouse Event Models

Table 1. Business Processes vs. Mashups

Mashups. In return, Mashups provide a connection to the user by default: Web applications are build by a Web page that consists of HTML, CSS and Java Script and therefore possess a presentation and means to interact with the user by nature. Application logic is only invoked when an event is created by the user that triggers a certain functionality.

The following can be concluded: While business processes enable software developers to create the application logic of a process without dealing with its graphical presentation and possible user interaction, Mashups are created as interactive graphic presentations that react on user input and thereby invoke small parts of application logic that modifies the presentation of the application.

3.2 Requirements for a Mashup Underlay System

The objective of this work is to incorporate the ability of SOA business processes to structure the underlying application logic as a service composition and to dynamically incorporate services or APIs through late binding mechanisms in order to overcome the device heterogeneity of today' Web Mashups. Moreover, this techniques allow for a more dynamic adaptation of Mashups to the users' needs and their environmental context by providing means to include services based on the users' preferences, on the client device and its respective capabilities, and the current performance of the single services.

Given these requirements, the following components are identified as central for the Underlay System for Web Mashups:

- **Workflow** - A workflow graph enables software developers to specify the execution order of the single services and APIs.
- **Dataflow** - A dataflow graph allows developers to connect data retrieved as output of one service to the inputs of other services.
- **Abstract Service Access** - An abstraction layer for services and APIs defines a unified way to access services providing the same resource, i.e. the same object on the presentation and interaction layer. Thus, this layer allows developers to conceptually include a map within their Mashups without specifying the concrete service that is creating this presentation. This procedure does not only free the software developer from programming against multiple APIs, but also allows for a dynamic exchange of the bound services or APIs in case the context of the user changes.
- **Context Awareness** - The Underlay System provides an interface to develop selection mechanisms in case multiple services are available for the same presentation. For instance, in case both Google Maps and Yahoo Maps are identified to be capable of presenting a map to the user, the selection can be made dependent on user preferences or context.
- **Time Awareness** - To overcome device heterogeneity, time is identified as critical context information that can trigger the evaluation of the appropriateness of integrated services. For instance, in case the processing of a service exceeds a predefined time limit, it can be concluded that the processing power of the device is too low and that the service should be replaced. The same technique can be applied to identify services that

are causing too much traffic given the available bandwidth. For instance, one could define that in case the response time of a service exceeds a certain time span, the traffic has to be too high for the currently available bandwidth such that the service has to be replaced with a less demanding one.

Within the next subsections, a service composition model is developed on a theoretical level that builds the core of the Underlay System for Web Mashups and can be executed within a respective runtime both on servers and browser-enabled clients. More insights on the related realization of the Underlay System is given later in section 4.

3.3 Modeling Service Compositions

This section deals with the representation and execution control of service compositions. Therefore, subsection 3.3.1 defines a service model assumed for the software components serving as basis for service compositions provides a general overview of the two main representation paradigms chosen for service compositions. The graph-based representation of the workflow and the dataflow within those compositions is then discussed in subsections 3.3.2 and 3.3.3, respectively. Finally, the interworking of both graphs is exemplarily outlined in subsection 3.3.4.

3.3.1 Service Model and Basic Concepts

Services themselves are considered as atomically executable parts of application logic, whereby the execution of the service is independent from outer computations and data structures. This assumption is in-line with the REST architectural style of the Web, where services are defined as stateless. We rely on IOPE descriptions characterizing a service functionality by its inputs, outputs, preconditions and effects as discussed by Jaeger et al. [16]. Inputs and outputs of a service are distinguishable by a unique identifier referred to as a *port*. Since it is aim at introducing a representation of service composition plans that is independent from the service descriptions they rely on, possible semantically enhanced description of preconditions or effects are not specified. Instead, we assume effect to be present or not, thus, we identify every effect with a boolean indicating whether the effect has already been generated or not. Preconditions can then be pondered against those boolean representations of service effects.

To represent service compositions properly as well as to control their execution, a bipartite graph concept based on a workflow graph controlling the execution of the single components within a service composition and a dataflow graph defining the passage of data between services' output and input ports is introduced. Each transition of the workflow graph corresponds to a service containing I/O parameters and a set of effects it generates during execution as described above.

In case a services precondition is met, the service can be executed, thus, an action representing the service functionality is performed, which consumes the service's input and generates a finite set of effects and a finite set of outputs. Transitions between locations are optionally annotated with guards, which restrict the passage of the transition by requiring the satisfaction of specific preconditions or previously generated outputs. Notably, the output of a service is not necessarily required as input for a service reached by the next transition within the workflow graph; instead, an output may also become relevant after multiple other services have been executed. Therefore, a second graph is defined, specifying the dataflow within a service composition. This dataflow graph shares the set of locations with the workflow graph, but represents the flow of outputs from one service component to the input of another with its transitions. A guard for passing a transition within a workflow graph that would lead to the execution of the next service can thus depend on the presence of a set of effects (expressed through preconditions) and on the availability of all inputs that have to be created as outputs of other services before. Notably, the dataflow graph is not necessarily completely connected, thus may be a set of graphs.

In the following section, the workflow graph of a service composition is formally defined, specifying its guard based transition behavior and its time awareness.

3.3.2 Workflow Graph

The workflow graph is supposed to control the execution order of single services within a service composition. Based on the requirements identified in subsection 3.2, automata theory is proposed as the mathematical basis for service composition representation. In [24], a transformation from UML state machines [10] to timed automata has already proven that automaton theory can be easily accessed by user-friendly modeling languages such as UML, enabling the feasibility of the presented approach. Coevally, automaton theory already provides rapid modifications of the automaton structure by simple operations, because automata are represented as basic digraphs. In order to provide realtime consideration within service compositions, the definition of workflow graphs borrows from timed automata. Timed automata [2, 9] are finite automata [4, 12] extended by a set of real-time valued clocks. In the following, we will refer to the definition provided by Clarke et al. in [7]. Let X be a finite set of real-valued variables standing for clocks. Clock constraints are then defined as follows.

Definition 3.1 (Clock Constraints) *A clock constraint is a conjunctive formula of atomic constraints of the form* $x * n$

or $x - y * n$ for $x, y \in X$, $* \in \{\leq, <, =, >, \geq\}$ and $n \in \mathbb{N}$. $\mathcal{C}(X)$ denotes the set of clock constraints. If φ_1 is in $\mathcal{C}(X)$, then $\varphi_1 \wedge \varphi_2$ is also in $\mathcal{C}(X)$.

A Timed Automaton is then defined as follows.

Definition 3.2 (Timed Automaton) *A timed automaton $\mathcal{A}$ is a 6-tuple $\{\Sigma, S, s_0, X, I, T\}$, where*

1. *Σ is a finite alphabet (standing for actions),*
2. *S is a finite set of locations,*
3. *$s_0 \in S$ are the initial locations (also called starting locations),*
4. *X is a set of clocks,*
5. *$I : S \rightarrow \mathcal{C}(X)$ assigns invariants to locations, i.e., provides a mapping from locations to clock constraints, and*
6. *$T \subseteq S \times \mathcal{C}(X) \times \Sigma \times 2^X \times S$ is the set of transitions.*

Abbreviatory, we will write $s \overset{g,\alpha,\lambda}{\rightarrow} s'$ for $\langle s, g, \alpha, \lambda, s' \rangle$, i.e. the transition leading from location s to s'. The transition is restricted by the constraint g (often called guard); $\lambda \subseteq X$ denotes the set of clocks that is reset during the transition passage.

Notably, many model checker such as UPPAAL [32], operating on timed automata restrict the location invariants to downwards closed ones, i.e. do only allow invariants of the form $x \leq n$ or $x < n$ for $n \in \mathbb{N}$.

Infinite state transition systems (sometimes called infinite state transition graphs) are used as model for a timed automaton $\mathcal{A}$. Deep introductions into the notion of transition systems can be found in [15, 20]. Here, the definition of Clarke et al. [7] will be followed again, specifying an infinite state transition system $\mathcal{T}(\mathcal{A})$ for a given timed automaton $\mathcal{A}$ as a 4-tuple $\mathcal{T}(\mathcal{A}) = \{\Sigma, Q, Q_0, R\}$. A state $q \in Q$ of this transition system is defined as a pair (s, v), where $s \in S$ is a location and $v : x \rightarrow \mathbb{R}^+$ is a clock assignment. The initial states are identified by all initial locations, where all clocks are set to zero, i.e. $Q_0 = \{(s, v) | s \in S_0 \wedge \forall x \in X \, [v(x) = 0]\}$.

The definition of the state transition relation is implied by the following two needs. First, a notion is required to reset a clock to zero, i.e., for $\lambda \in X$, we define $v[\lambda := 0]$ for mapping all clocks in λ to zero. For $d \in \mathbb{R}$, we define $v + d$ as a clock assignment that maps the current value of v to $v(x) + d$ for all clocks $x \in X$. Based on this, two different types of transitions are defined, covering the passage of time and the triggering of actions. Time can pass while the system is in a specific location as long as the according state invariant is not violated. This elapsing of time is referred to as *delayed transition*, specified as $(s, v) \overset{d}{\rightarrow} (s, v + d), d \in \mathbb{R}^+$, subjected to the condition that the invariant $I(s)$ is not violated for every $v + e, 0 \leq e \leq d$. The second type of transitions describes the actual execution of an action and is thus referred to as *action transition*. If there is a transition $\langle s, \alpha, g, \lambda, s' \rangle$ where v satisfies g and $v' = v[\lambda := 0]$, we note $(s, v) \overset{\alpha}{\rightarrow} (s', v')$ for a transition with $\alpha \in \Sigma$. Together, we receive the transition relation $\mathcal{R}$ for $\mathcal{T}(\mathcal{A})$ as $(s, v)\mathcal{R}(s', v')$ (also written as $(s, v) \overset{\alpha}{\Rightarrow} (s', v')$), if there are s'' and v'' so that $(s, v) \overset{d}{\rightarrow} (s'', v'') \overset{\alpha}{\rightarrow} (s', v')$ for some $d \in \mathbb{R}$. Thus, an action α can also be performed without elapsing time, i.e. in case $d = 0$. Moreover α may also stand for the empty action $\tau \in \Sigma$, i.e. a transition can also be passed without entailing the execution of an specific action.

Since actions are supposed to model the behaviour of a service, the execution of an action α is mapped to the consumption of inputs, the creation of outputs and the generations of effects. Inputs and outputs are regarded as typed variables bounded to specific ports, enabling the definition of I/O passage by means of a dataflow graph. The domains of variables thus constitute their primitive data type.

The Workflow Graphs Θ are now modeled as timed automata as defined in Definition 3.2, extended by the ability of actions $\alpha \in \Sigma$ to operate on typed variables and effects. Typed variables are defined as 2-tuples $(x, \Gamma(x))$, where x is a variable and $\Gamma(x)$ its respective domain. For now, only variables x with $x \in R \subset \mathbb{R}$ are considered. Let $\mathcal{V}$ be a finite set of typed variables. Regarding the handling of effects, we define $\mathcal{E}$ as a final set of variables $\gamma^* \in \{0, 1\}$ indicating whether an effect γ has been created (γ^* set to 1) or not (γ^* set to 0). We assume that $\mathcal{V}$ contains at least all variables of $\mathcal{E}$, thus, $\mathcal{E} \subseteq \mathcal{V}$.

Definition 3.3 (Guard Constraints) *A real-value constraint is a propositional logic formula $x * n$, where $x \in \mathcal{V}$ is a typed variable, $n \in \mathbb{R}$ and $* \in \{<, \leq, \geq, >, ==\}$. $\mathcal{C}(R \backslash \mathcal{V})$ denotes the set of real-value constraints containing only variables of $\mathcal{V}$.*

If φ_1 and φ_2 are in $\mathcal{C}(R \backslash \mathcal{V})$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$ and $\neg \varphi_1$ are also in $\mathcal{C}(R \backslash \mathcal{V})$.

Let $\varphi_c \in \mathcal{C}(X)$ be a clock constraint as defined in Definition 3.1, φ_r a real-value constraint.

A guard constraint is a formula $\varphi = \varphi_c | \varphi_r | \varphi_c \wedge \varphi_r$ evaluating to a Boolean. $\mathcal{C}(X \circ R \backslash \mathcal{V})$ denotes the set of all guard constraints.

The workflow graph is now defined as a timed automaton whose actions and guards can also operate on typed variables.

Definition 3.4 (Workflow Graph) *A workflow graph Θ is a timed automaton as defined in Definition 3.2, where*

1. *Σ is a finite alphabet. The alphabet represents the actions which are identified by the single services within the service composition.*

2. *S is a finite set of locations defining the service compositions's current state,*

3. *$s_0 \in S$ are the initial locations defining the initial state of the service composition,*

4. *X is a set of clocks,*

5. *$I : S \rightarrow \mathcal{C}(X)$ assigns invariants to locations, restricting the system in the amount of time it is allowed to remain in the current state, and*

6. *$T \subseteq S \times \mathcal{C}(X \circ R \backslash \mathcal{V}) \times \Sigma \times 2^X \times S$ is the set of transitions, denoting the execution of a service (represented by an action α). The passage of a transition (and thus the execution of a service) thereby depends on whether the according guard is met.*

In order to decide whether an output of a service component is required as an input for another one, a dataflow graph is introduced in the following section.

3.3.3 Dataflow Graph

Dataflow graphs represent the connections of output and input ports. They keep the same locations as the workflow graph introduced in Definition 3.4 and use labeled transitions to express inter-component data passing.

Definition 3.5 (Dataflow Graph Ω) *A dataflow graph Ω is a labeled directed digraph $\Omega = \{N, P, E\}$, where,*

1. *N is a final set of labeled nodes, which is equivalent to the set of locations S held in the according workflow graph Θ,*

2. *P is a set of port mappings represented by 2-tuples $p = (p_1, p_2)$ indicating the passage of the output from port p_1 to the input port p_2, and*

3. *$E \subseteq N \times P \times N$ is a final set of labeled directed transitions.*

Each location represents the data generated by the execution of action α_i; we therefore label a node with $d(\alpha_i)$ indicating the output data from action α_i. If a transition is passed within a workflow graph Θ entailing the execution of action α_i, all outgoing transitions from location $d(\alpha_i)$ within the according dataflow graph Ω are passed. A transition passage $d(\alpha_i) \overset{(p_m,p_n)}{\rightarrow} d(\alpha_j)$ within Ω effectuates that the output at port p_m from action α_i is redirected to input port p_n of action α_j in case action α_i is executed within Θ.

A service composition is specified by its workflow and dataflow graph, i.e., is defined as a 2-tuple $\langle \Theta, \Omega \rangle$.

The following section discusses a small example for service composition representation and control.

3.3.4 Workflow and Dataflow Graph Interworking

The service composition to be represented is abstractly depicted in Figure 4.

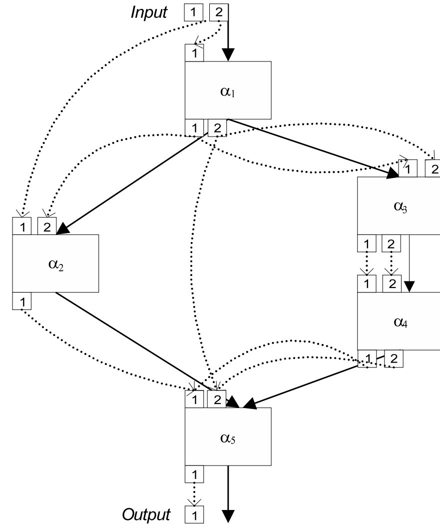

Figure 4. A Possible Service Composition.

The involved service components are represented by rectangles labeled with an action α_i describing their functionality. The order of execution, i.e., the workflow, is depicted by bold arrows while the passage of inputs and outputs is indicated by dashed arrows. They connect the output ports of service components with input port of other components that are drawn as numbered squares.

The example service composition contains five service components (labeled with $\alpha_1, ..., \alpha_5$).

After α_1 has been started, either α_2 or α_3 and α_4 are executed. In case both execution paths are enabled, i.e. all premises in terms of I/O and effect availability are met, a path is chosen nondeterministically according to the notion of transition systems.

While service executions in common service composition representations such as in Figure 4 are represented as nodes within graphs, the one introduced in this paper models service executions as actions performed during transitions between locations. The workflow graph deduced from the example service composition is illustrated in Figure 5 (a).

Guards restricting the transition from a location s to a location s' operate on the generated effects (through pre-

conditions) and a final set of inputs and outputs. Outputs are not stored but redirected to input ports by means of the according dataflow graph. The dataflow graph depicted in Figure 5 (b) is supposed to belong to the workflow graph illustrated in Figure 5 (a). For instance, the execution of α_1 implies the redirection of the output from the first output port of α_1 to the second input port of α_2 and the first input port of α_3. Moreover, the output from the second output port of α_1 is forwarded to the second input port of α_3 and to the second input port of α_5.

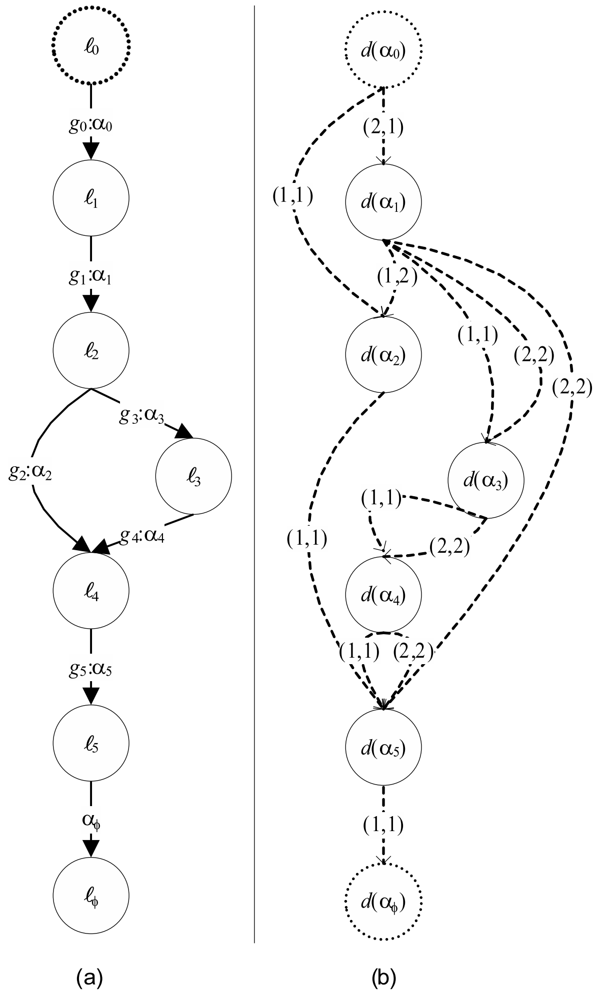

Figure 5. (a) An according workflow graph. (b) A dataflow graph representing the I/O Passing.

3.3.5 Enabling Parallel Execution

Because of their relation to automata and their given transition relation introduced in section 3.3.2, workflow graphs already support a wide set of control structures required to guide the execution order of services; the three most important ones are abstractly depicted in Figure 6. First, nondeterministic decisions between multiple possible service invocations are modeled with a finite set of outgoing transitions labeled with the same guard g. In case the guard is met, all transitions are enabled and the transition relation discussed in section 3.3.2 selects a transition nondeterministically. Decisions such as *if-else* or *switch* statements are represented by multiple outgoing transitions labeled with different guards $g_1, ..., g_k$. Here, deadlocks can occur in case all guards are evaluated as *false* under a given set of constraints. Moreover, multiple guards may be *true*, entailing a non-deterministic decision making between the enabled transitions. Last, looping can be modeled with transitions pointing back to the origin state as depicted in Figure 6 (c).

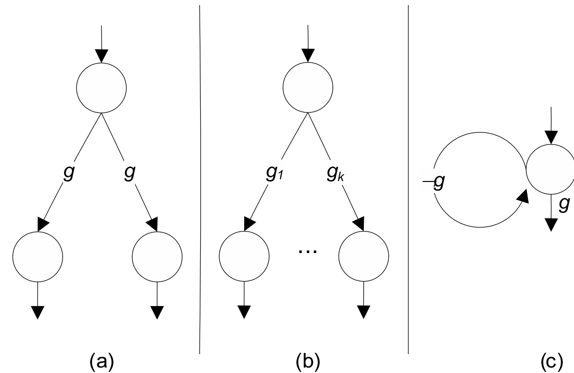

Figure 6. Basic control structures of automata semantics. (a) Nondeterministic Selection. (b) Choice. (c) Looping.

However, workflow graphs do not provide all means to model parallel execution of services. By passing a transition, the according service is executed. Note that the introduced model does not assume that the workflow graph remains in the location reached by the transition passage until the service's execution is finished. Instead, the next transition is directly passed (in case at least one transition's guards are met), entailing the execution of the next service. Thereby, service execution becomes parallel. If both services have been started on the same device, the local scheduling algorithm ensures their pseudo-parallel execution. In case they are executed on different nodes and therefore multiple processors, they are running in parallel. However, this approach fails in case the biggest subset of a given set of services should be executed in parallel.

Assume two services α_1 and α_2 are selected for parallel execution. In case the services are initiated in the order

$\alpha_1 \rightarrow \alpha_2$, it may be possible that α_1 cannot be started since an input is missing. The blocking of the transition would entail that also service α_2 is not started, although its guard may be met. Therefore, a control structure for parallel execution of services is required. The left-hand side of Figure 7 depicts an abstract workflow reflecting the previously described situation of two services α_1 and α_2 that should be executed in parallel.

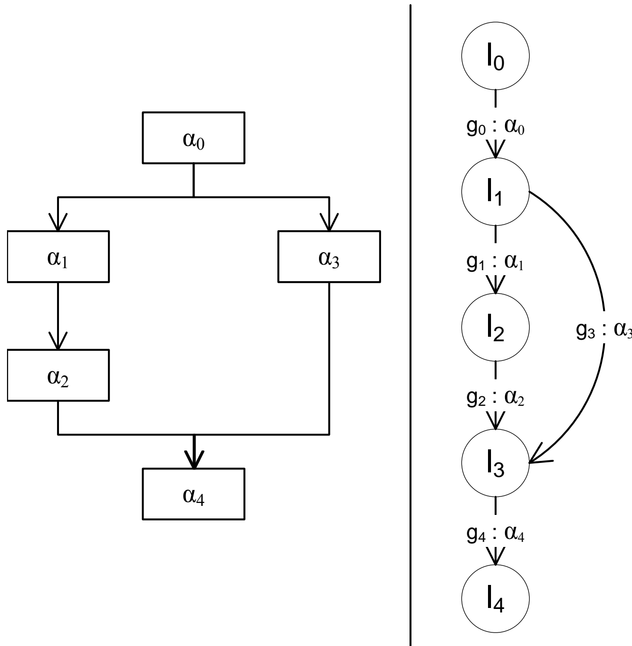

Figure 7. Timed automaton structure ensuring parallel execution.

The right-hand side of the same Figure depicts the timed automaton that corresponds to the abstract service composition. Here, the execution of α_2 would be forestalled in case guard g_1 is not met although g_2 may be fulfilled. In order to unblock this behavior, an additional idle state is automatically inserted during the generation of the workflow graph that is entered after every execution of a service that may be executed in parallel to other services. The accordingly modified workflow graph is illustrated in Figure 8.

Because of this modification, the timed automaton moves into the idle state as soon as the guard of the next service is not met, such that the invocation of another service can be initiated whose guards is fulfilled. The utilization of semaphores ensures that every service is only executed once.

Thus, by inserting a special control structure that can be generated for a given finite set of services, workflow graphs possess the same expressiveness as other workflow languages such as BPEL, while relying on sound formal

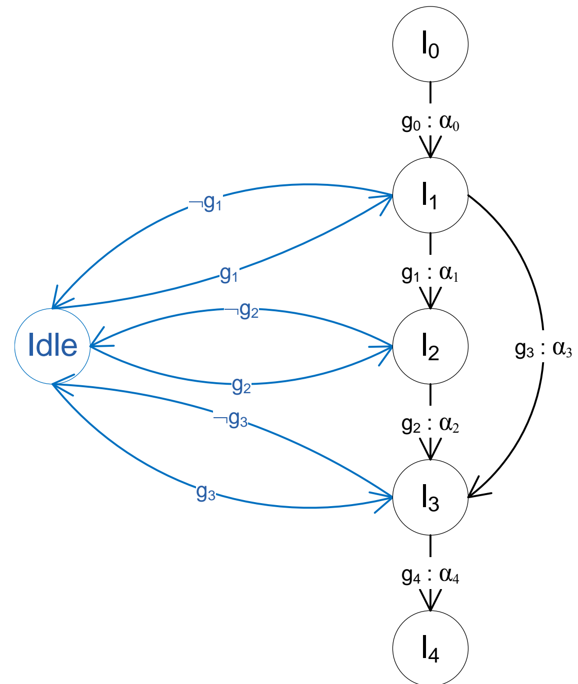

Figure 8. Timed automaton structure ensuring parallel execution.

model featuring realtime consideration and flexibility.

4 An Architecture featuring a Underlay System for Web Mashups

Within this section, the previously introduced formal method for the representation of workflow-based service compositions within a Underlay System for Web Mashups is realized as an architecture enabling the utilization of workflows for the representation of Web applications and features the late binding of 3rd party APIs. This architecture is implemented according to the REST architectural style, the architectural style of the Web, that was proposed by Fielding within his PhD thesis and is known for its related realization within the Web, HTTP [13].

Figure 9 gives an high-level overview of the proposed architecture.

Here, the Mashup site, i.e. the Mashup proxy, is extended by 4 key components. First, a workflow engine for the previously introduced model based on timed automata enables the structured execution of service compositions. In addition to version for servers written in Java, the runtime is also implemented in pure Java Script that can be transferred to any browser enabled client during initialization such that the workflows can be executed on the client side and in-

corporate local services residing on the client. A detailed description of the implementation was presented in [25]. The single services of the service composition are given as abstract triples <*res, act, attr[]*>. Here, *res* describes the resource that should result from the respective service execution, *act* defined an action that should be performed on the resource *res*, and *attr[]* denotes a set of attributes that are required for the execution of the specified action. Thus, the triple <*map, center, lat=23432, long=92834*>

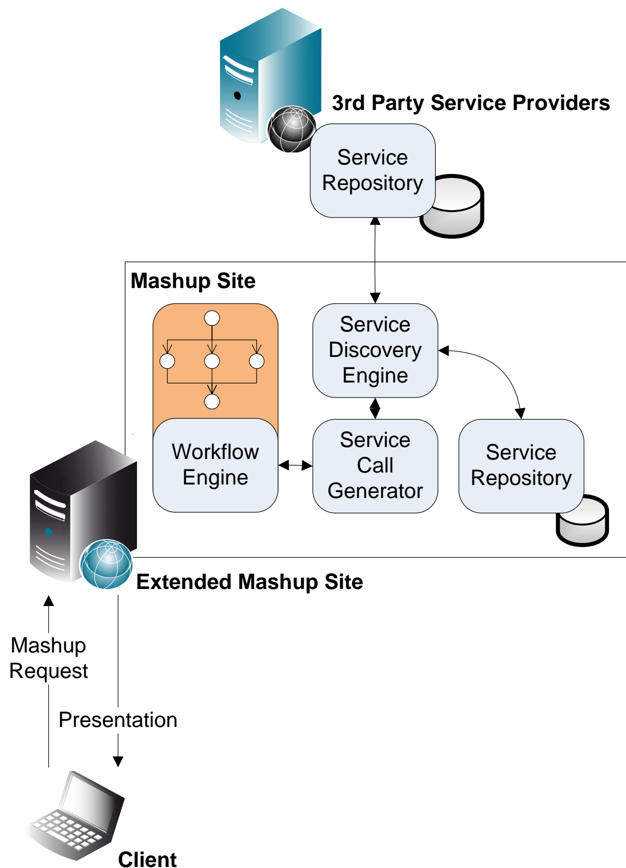

Figure 9. Server-side Service Composition Engine for Composed Web Applications: Architecture.

denotes that a map is requested that is centered on the location given by the longitude *long* and latitude *lat*. In general, a location is not known by its longitude and latitude, but by its name. Therefore, another service may be requested by the triple <*location, getCoordinates, location=Berlin*> that generates the longitude and latitude values for the city "Berlin". The output of this service can thus be used as input for the next service. Of course, the attribute "Berlin" can also be given during runtime by a user input.

This proceeding interprets the workflow model in a REST conform way: The developer requests a resource and is returned a representation of the resource that depends on the given action and set of attributes. For instance, while the triple <*location, getCoordinates, location=Berlin*> returns the location of a given city by its longitude and latitude coordinates, the triple <*location, getStreetName, location=Peter's Pub*> may return a String contain the street name of a bar called "Peter's Pub".

To enable such an abstraction from actual service implementations, the proposed architecture features a late binding engine. Here, 3rd party provides can describe their exposed APIs featuring REST interfaces with the Web Application Description Language (WADL) [14]. (The integration of other services such as classic SOAP Web Services described by WSDL [6] are also supported but are not described in greater detail here.) The late binding engine is capable of matching the previously introduced triples denoting abstract descriptions of a requested resource to these WADL descriptions and return a set of WADL files that describe services which can generate the requested resource. For instance, a triple <*location, getCoordinates, location=Berlin*> may be matched by Google Maps, Yahoo Maps, and suchen.de. One of these WADL based descriptions can now be passed to the Service Call Generator where a REST call for the chosen service is automatically created and handed back to the workflow engine where the service can be invoked.

This proceeding enables more responsiveness and dynamicity of Web Mashups on multiple levels. First, service interfaces are encapsulated behind abstract descriptions which reduce the amount of coding for the software developer. Thus, instead of dealing with the concrete API of Google Maps, software developers can interface with the abstract notion of triples, which is conform to the REST architectural style of the Web. Thereby, the integration of multiple APIs providing a similar functionality can be avoided. For example, a developer that has always used Google Maps to integrate a map into his or her Web application may decide to use Yahoo Maps instead. In this case, the developer would have to incorporate a complete new API into his or her Mashup, although both services provide the same functionality.

However, the dynamic binding of services does not only reduce the development time for Mashups, but also enables the consideration of device capabilities, user profiles, context information, and Quality of Service (QoS) parameters of services. Thus, in case the late binding engine returns multiple WADL files, i.e., has discovered multiple services or APIs that can provide the requested resource, the service may be explicitly selected that matches the current circumstances best. For instance, in case a user has defined in his profile that he prefers services from Google because of the

familiar user interface, Google Maps may be bound instead of Yahoo Maps. On the other hand, a monitoring entity may have noticed that the response time of Yahoo has been considerably shorter than from Google, leading to the incorporation of Yahoo Maps instead of Google Maps; this incorporation of non-functional properties if services has been introduced in [26]. It also enables the recovery of services that expose a weak performance during runtime, i.e. in case a service is not responding or responding too slow, it may be dynamically replaced during runtime by another service.

In addition to the flexibility provided by the dynamic integration of single service implementations within the service composition, the workflow graph itself may be adapted in its structure to as a response to environmental changes; this adaption has already been introduced in [28].

Thus, the architecture proposed in this work provides means to overcome the problem of heterogeneous clients by providing means to bind and replace services dynamically before and during runtime based on non-functional properties of services, context information, device capabilities or user profiles.

5 Conclusion and future Prospect

Within this paper, a Underlay System for Web Mashups was introduced that combined the advantages of SOA business processes and the rich presentation and interaction capabilities of today's Mashups to overcome the challenge of device heterogeneity and context dependence in modern Web applications. Therefore, a formal model for service compositions based on a bipartite graph concept has been introduced that consists of a workflow graph defined as a timed automaton over an extended finite set of typed variables and a dataflow graph specifying the passage of data between the single services. By introducing a special structure of a timed automaton, the semantics of classic automata were extended to support parallel execution of services beside their given ability to express interleaving, decisions and looping. The model has then been used to build the basis for a new type of rich Web Mashups that are responsive the the users' preferences and device context by supporting late binding of 3rd party services during runtime.

Within future work, algorithms for the automatic creation of service compositions are developed. Here, the descriptions for 3rd party services are extended by lightweight semantics to support the automatic creation of workflows and binding of services based on a request given by either a user or another service. Moreover, a GUI is developed that enables the manual creation of service compositions in an intuitive way; this representation is then transformed into corresponding workflow and dataflow graphs automatically.

References

[1] www.programmableweb.com.

[2] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proceedings of of the 5th Annual Symposium on Logic in Computer Science*, pages 414–425. IEEE Computer Society Press, 1990.

[3] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services, Version 1.1, May 2003. [Online]. Available: http://www.ibm.com/developerworks/library/specification/ws-bpel/. [Accessed: November, 2007].

[4] J. Berstel. *Transductions and Context-free Languages*. B.G. Teubner, Stuttgart, 1979.

[5] A. Bucchiarone and S. Gnesi. A Survey on Services Composition Languages and Models. In A. Bertolino and A. Polini, editors, *in Proceedings of International Workshop on Web Services Modeling and Testing (WS-MaTe2006)*, pages 51–63, Palermo, Sicily, ITALY, June 9th 2006.

[6] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. In *World Wide Web Consortium (W3C) recommendation*, January 2006. [Online]. Available: http://www.w3c.org/TR/wsdl20/. [Accessed: Mar. 24, 2006].

[7] E. M. J. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts and London, England, 1999.

[8] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business Process Execution Language for Web Services (Version 1.0), July 2002.

[9] D. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, number 407 in LNCS, pages 197–212. Springer, 1989.

[10] L. Doldi. *UML2 illstrated, Developing Real-Time & Communication Systems*. TMSO, 2003.

[11] J. Duhl. Rich Internet Applications. *Whitepaper (sponsored by Macromedia and Intel)*, page Online: http://www.adobe.com/resources/ business/rich_internet_apps/whitepapers.html, 2003.

[12] S. Eilenberg. *Automata, Languages, and Machines*, volume Volume A. Academic Press, New York, 1974.

[13] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[14] M. J. Hadley. Web application description language (wadl). Technical report, Sun Microsystems Inc., November 2006.

[15] M. Huth and M. Ryan. *Logic in Computer Science - Modeling and reasoning about systems*. Cambridge University Press, 2004.

[16] M. Jaeger, L. Engel, and K. Geihs. A Methodology for Developing OWL-S Descriptions. In *First International Conference on Interoperability of Enterprise Software and Applications Workshop on Web Services and Interoperability (INTEROP-ESA '05)*. Springer, 2005.

[17] F. Lautenbacher and B. Bauer. A Survey on Workflow Annotation & Composition Approaches. In *Proceedings of the*

Workshop on Semantic Business Process and Product Lifecycle Management (SemBPM) in the context of the European Semantic Web Conference (ESWC), pages 12–23, Insbruck, Austria, 7th June 2007.

[18] G. Lawton. New ways to build rich internet applications. *Computer*, 41(8):10–12, Aug. 2008.

[19] F. Leymann. Web Service Flow Language (WSFL 1.0). In *IBM*, May 2001.

[20] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[21] E. Ort, S. Brydon, and M. Basler. Mashup styles, part 1: Server-side mashups. Technical report, Sun Developer Network (SDN), May 2007.

[22] E. Ort, S. Brydon, and M. Basler. Mashup styles, part 2: Client-side mashups. Technical report, Sun Developer Network (SDN), August 2007.

[23] C. E. Ortitz. Introduction to J2ME Web Services, April 2005. [Online]. Available: http://developers.sun.com/mobility/apis/articles/wsa. [Accessed: October, 2007].

[24] H. Pfeffer. UPPAAL Model Checking as Performance Evaluation Technique. Master's thesis, Rheinische Friedrich-Whilhelms-Universität Bonn, 2005.

[25] H. Pfeffer, L. Bassbouss, and S. Steglich. Structured service composition execution for mobile web applications. In *Proceedings of the 12th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS 2008)*, volume ISBN: 978-0-7695-3377-3, pages 112–118, Kunming, China, 2008. IEEE Computer Society Press.

[26] H. Pfeffer, S. Kr"ussel, and S. Steglich. Fuzzy Service Composition Evaluation In Distributed Environments. In *In Proceedings of I-CENTRIC 2008*, 2008.

[27] H. Pfeffer, D. Linner, C. Jacob, and S. Steglich. Towards Light-weight Semantic Descriptions for Decentralized Service-oriented Systems. In *Proceedings of the 1st IEEE International Conference on Semantic Computing (ICSC 2007)*, volume CD-ROM, Irvine, California, USA, 17-19 September 2007. PLJ+07.

[28] H. Pfeffer, D. Linner, and S. Steglich. Dynamic adaptation of workflow based service compositions. In *ICIC '08: Proceedings of the 4th international conference on Intelligent Computing*, number ISBN: 978-3-540-87440-9, pages 763–774, Berlin, Heidelberg, 2008. Springer-Verlag.

[29] H. Pfeffer, D. Linner, and S. Steglich. Modeling and controlling dynamic service compositions. *Computing in the Global Information Technology, 2008. ICCGI '08. The Third International Multi-Conference on*, pages 210–216, 27 2008-Aug. 1 2008.

[30] S. Thatte. XLANG - Web Services for Business Process Design, 2001.

[31] The OWL Services Coalition. OWL-S: Semantic Markup for Web Services, November 2004. [Online]. Available: http://www.daml.org/services/owls/1.1/. [Accessed: February 26, 2007].

[32] U. University and A. University. UPPAAL 4.0.6. http://www.uppaal.com, December 2007.

[33] W. van der Aalst. Don't go with the flow: Web services composition standards exposed. *IEEE Intelligent Systems*, 2003.