This is a post-peer-review, pre-copyedit version of the following article: LAngel Lagares Lemos, Florian Daniel, and Boualem Benatallah. 2015. Web service composition: A survey of techniques and tools. ACM Comput. Surv. 48, 3, Article 33 (December 2015), 41 pages.
The final authenticated version is available online at: http://dx.doi.org/10.1145/2831270

Web Service Composition: A Survey of Techniques and Tools

Angel Lagares Lemos (University of New South Wales), Florian Daniel (University of Trento), Boualem Benatallah (University of New South Wales)

Web services are a consolidated reality of the modern Web with tremendous, increasing impact on everyday computing tasks. They turned the Web into the largest, most accepted, and most vivid distributed computing platform ever. Yet, the use and integration of Web services into composite services or applications, which is a highly sensible and conceptually non-trivial task, is still not unleashing its full magnitude of power. A consolidated analysis framework that advances the fundamental understanding of Web service composi-tion building blocks in terms of concepts, models, languages, productivity support techniques, and tools is required. This framework is necessary to enable effective exploration, understanding, assessing, comparing, and selecting service composition models, languages, techniques, platforms, and tools. This article estab-lishes such a framework and reviews the state of the art in service composition from an unprecedented, holistic perspective.

Categories and Subject Descriptors: H.3.5 [Online Information Services]: Web-based Services
General Terms: Methods, Techniques, Tools
Additional Key Words and Phrases: Web services, service composition, service oriented computing

1. INTRODUCTION

Service composition encompasses all those processes that create added-value services, called *composite* or *aggregated* services, from existing services. The need to integrate services is recognized in both the enterprise and the consumer arenas [Bai et al. 2009]. A Gartner report stated that "through 2014, the act of composition will be a stronger op-portunity to deliver value from software than the act of development" [Hill et al. 2010]. More recently the focus has shifted to more specific types of compositions: the "Top 10 Strategic Technology Trends for 2014" Gartner report [Gartner 2013] confirms the significance of composition of cloud services, and Forrester in its "2015 Predictions for Application Development and Delivery Professionals" [Facemire et al. 2014] highlights composition for mobile apps (i.e., assembling front-end components) as a cornerstone to enhance development productivity, leveraging Web components and platforms such as HTML5 and Google Polymer, a library for creating these components.

Authors' addresses: A. Lagares Lemos and B. Benatallah, School of Computer Science and Engineering (CSE), University of New South Wales (UNSW), Sydney, NSW 2052, Australia; emails: alagares@gmail.com, boualem@cse.unsw.edu.au; F. Daniel, Department of Information Engineering and Computer Science (DISI), University of Trento, 38123 Povo (TN), Italy; email: daniel@disi.unitn.it.

This work is supported by SDA Project, Smart Services CRC, Australia, and the Faculty of Engineering at UNSW Australia.

Received June 2014; revised May 2015; accepted September 2015

Service composition is a very active area of research and more than that, in many cases there is research from the fields of workflows and software components that can be leveraged [Dustdar and Schreiner 2005]. By bringing together Web services and semantic Web technologies, the semantic Web services paradigm promises to make a step forward in improving Web services composition through rich and machine understandable representations of service properties and capabilities, as well as reasoning mechanisms to select and aggregate services [McIlraith et al. 2001].

Several variations of service description models, interaction protocols, and composition languages and techniques exist. Overall, on one hand mainstream composition techniques focus mostly on application and data services. Composition logic typically relies on procedural flow or scripting languages [Lau and Rana 2010]. The interaction with component services is performed through low-level APIs and protocols. Interestingly, current service composition environments rarely provide productivity support tools similarly to those provided by modern Integrated Development Environments (IDEs). For instance, IDEs support developers by providing code search, reuse, debugging, generation, and so on. Like traditional programmers, service composers would benefit from environments that bring together productivity techniques, such as services discovery, reuse, code generation, documentation, integration with service composition environments. Such environments may have different requirements in terms of component models, interaction protocols, and orchestration constructs; for example, cloud services orchestration requires coordination of hardware and software resources across various layers. In addition to traditional control flow and dataflow constructs, there is also a need for abstractions to support consistency across physical and logical layers, exception handling, and flexible, efficient coordination, selection, and scheduling of resources.

In this article, we propose an articulated framework for analyzing and comparing Web service composition approaches. Previous work mostly focused on specific aspects of services engineering, including services discovery and quality of services (e.g., Garofalakis et al. [2004] and Mani and Nagarajan [2005]). Existing surveys addressing Web services composition are mostly fragmented and lack a holistic view of the problem. These surveys investigated specific composition aspects, such as usability, adaptability and efficiency of the service composition mechanisms [Brønsted et al. 2007], dynamic service composition (consisting of composing an application autonomously when a user queries for an application at runtime) [Eid et al. 2008], process-based service composition languages [Van der Aalst et al. 2003], languages and models [ter Beek et al. 2006; Milanovic and Malek 2004], Web mashups [Grammel and Storey 2010; Benslimane et al. 2008], scientific workflows [Barker and Van Hemert 2008], life cycle [Pessoa et al. 2008], QoS of the composition [Strunk 2010], service description and interaction languages and protocols [Dustdar and Schreiner 2005], and semantic service composition [Bartalos and Bieliková 2011]. Clearly, previous efforts by research and practitioner communities have produced promising results that are certainly useful. However, a more consolidated and holistic analysis framework that advances the fundamental understanding of Web service composition building blocks in terms of concepts, models, languages, productivity support techniques, and tools is required. This framework is necessary to enable effective exploration, understanding, assessing, comparing, and selecting service composition models, languages, techniques, platforms, and tools.

The work presented in this article aims to create this understanding and analysis framework. It provides an extensive survey of the main issues and solutions in Web service composition. After introducing the necessary background on the core composition concerns (next section), we propose our framework for analyzing and comparing service composition solutions (Section 3). The framework proposes a set of dimensions (i.e., language, knowledge reuse, automation, tool support, execution platform and

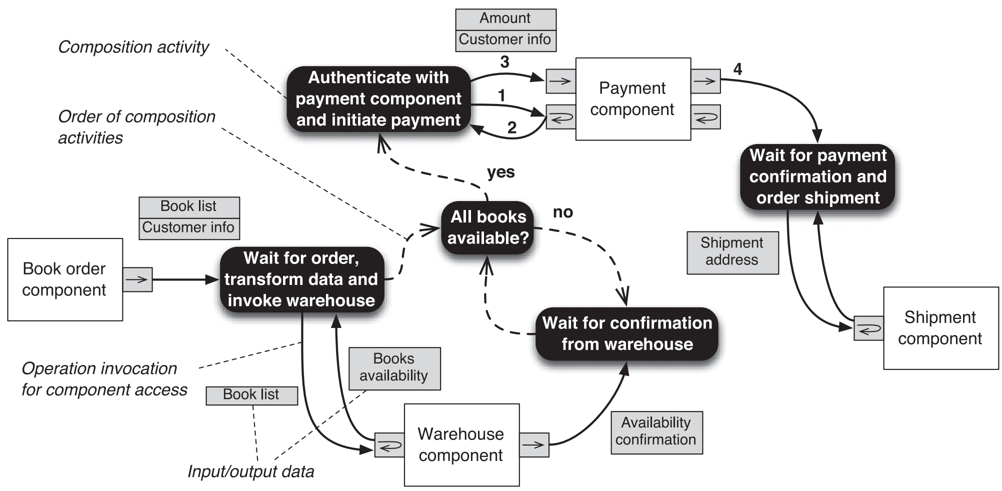

Fig. 1. A composite application for the processing of book orders. In black and bold the activities of the composition logic gluing together components.

target user), which we discuss in Sections 4–9. We then apply the framework to a selection of prominent service composition technologies and tools (Section 10) and provide our conclusion and outlook in the last section.

2. COMPOSITION-BASED SOFTWARE DEVELOPMENT

The basic ingredient of any composite application are the software components (we use the terms *software component* and *component* interchangeably). These encapsulate *functionality*, *data*, and/or a *user interface* (UI), which can be reused, and provide a set of *operations*, which allow one to programmatically interact with the encapsulated functionality and/or UI. Examples of typical components we refer to are SOAP and RESTful Web services.

2.1. Composition Concerns

Figure 1 illustrates the internals of a book order application leveraging on four components. It allows us to describe the core concerns that must be addressed:

—***Component access***: Using different components requires mastering the different component models and supporting their access mechanisms. That is, it may be necessary to invoke operations, to send notifications, and to intercept events.

—***Conversation management***: A component may also be accompanied by a business protocol [Alonso et al. 2004a], which is a specification that tells in which order the operations of the component must be enacted, so as to establish a correct conversation with the component. For instance, the Payment component in Figure 1 requires its clients first to authenticate with the component (steps 1 and 2), then to initiate the payment process (3), and then to wait for a completion event (4).

—***Control flow***: Composition activities can generally not be executed randomly; for example, it does not make sense to invoke the Warehouse component in absence of the ordered list of books. It is necessary to order composition activities, in order to achieve a given objective. For example, after receiving an order the composition in Figure 1 invokes immediately the Warehouse component, forming a so-called *sequence* of invocations. Ordering activities may also require taking *decisions* on which activity to perform next. For example, the “All books available?” check either starts the payment procedure or waits for the warehouse to confirm the availability of all books, depending on the availability of the ordered books.

—***Dataflow***: In a composition, the input of one component is typically produced by another component as output (if it is not provided by the composition logic itself). For instance, the Warehouse component in Figure 1 requires a book list as input, which is provided by the Book order component as output. Specifying how inputs derive from outputs defines a so-called *dataflow* among components.

—***Data transformation***: Most of the times, outputs and inputs do not immediately match, e.g., in terms of data formats or size. Propagating data from one component to another may therefore require an intermediate *data transformation* step (e.g., reformatting, splitting or merging), so as to make components compatible. For instance, the book list for the Warehouse component is a subset of the data produced by the Book order component.

Addressing the above concerns for all components to be integrated produces a specification of how the composite application should behave. Next to these composition features, developing a composite application may also require dealing with a set of *cross-cutting concerns*, such as security and service level agreements.

2.2. Orchestration vs. Choreography

The composition example illustrated in Figure 1 specifically focuses on the internals of the composite application and on the coordination of composition actions and components. It does so from a *centralized perspective*, expressing how the composition has to act, in order to integrate components. The result is an *executable design* of the composite application called an *orchestration* [Peltz 2003].

Composite applications may be designed also by taking a *distributed perspective*, in which the providers of the components participating in a composition jointly agree on a common communication protocol for their components. The result is a *contract* (the protocol) between the co-operating partners, which is not executable and must be implemented inside of each component individually. This contract is commonly known as ***choreography*** [Peltz 2003]. Choreographies are particularly useful in those situations in which multiple parties have to collaborate, but none of them wants to take the responsibility of running a centralized orchestration (e.g., in B2B transactions).

3. WEB SERVICE COMPOSITION TAXONOMY

The focus of this survey is on the *composition of reusable software services*, specifically *orchestrations*. Also, despite the similarity with programming, we do not study here the suitability of generic programming languages for developing composite applications. Our center of attention is on all those approaches, languages and tools that propose *composition-specific* development abstractions and solutions.

The service composition space delimited by these assumptions is still vast. To facilitate a focused analysis, we developed the taxonomy illustrated in Figure 2, which proposes a *holistic view* on the problem of orchestration-based composition and aims to highlight the key aspects and options one has to face when composing software. The taxonomy is based on our own experience in Web services composition (e.g., Lagares Lemos et al. [2013] and Benatallah et al. [2003]), business process management (e.g., Benatallah et al. [2004] and Daniel et al. [2009b]) and mashups (e.g., Daniel et al. [2007]), as well as extensive literature review in related areas, discussions with colleagues, experimentation with various systems and prototypes, which allowed us to identify common building blocks for the different variations in services composition systems.

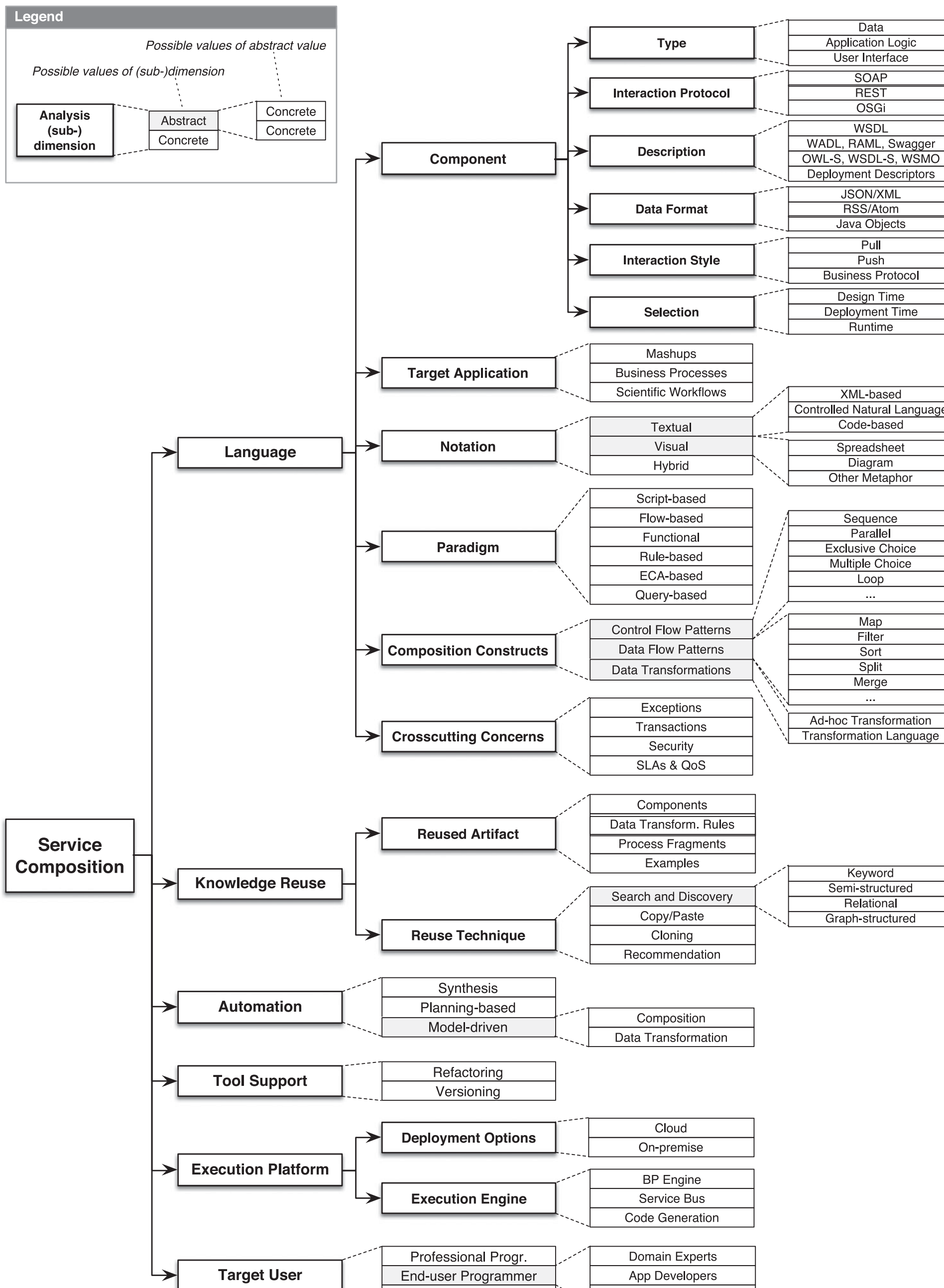

Fig. 2. Web service composition taxonomy.

In the previous section, we discussed *what* developing a composition means. Our taxonomy looks into *how* compositions can be developed, independently of technologies or target composites, and into how the composition process can be *assisted*, so as to ease development. Part of this problem requires also understanding *who* actually composes software and which composition techniques or practices are suitable to which profile of developer. Accordingly, we split the composition problem into six *dimensions*, which in turn may be split into *sub-dimensions*:

—***Composition language***: This is the core dimension of our analysis. The language decides how composition occurs, which composition activities are supported, and how. Recalling Figure 1, the composition language is the formalism that allows one to express the composition logic and to execute the composite application. Given its crucial role in the composition process, we further split the language dimension into sub-dimensions and look at the different *components* and *target applications* a language may support, the specific *notation* and *paradigm* is adopts, as well as at *composition constructs* and *cross-cutting concerns*.
—***Knowledge reuse***: Acknowledging the reuse-based nature of composition (the practice is based on the reuse of existing software components), we then analyze more closely which composition *artifacts* exist that can be reused. The development of a composite application may be non-trivial, and composition is not yet an as widespread practice as one might think. Effective reuse—and the respective *reuse techniques*—assumes, therefore, a special role in the context of composition.
—***Automation***: Thanks to the availability of dedicated composition languages, which are characterized by a relatively limited set of composition constructs, and of reusable components, which encapsulate most of the complexity of a composite application, the decision space of composition is generally much smaller compared, for example, to software development with generic programming languages. This observation has inspired a set of automated composition approaches and model-driven composition practices, whose roles and value is important to understand.
—***Tool support***: The previous three dimensions specifically analyze how a composite application can be developed. Ideally, but not mandatorily, this process is supported by a suitable development tool or integrated development environment (IDE), which assists the developer throughout the whole development process with application-independent infrastructure or functionalities.
—***Execution platform***: One of the key aspects of composition is how ready composite applications are deployed and executed. Deployment can be supported through different *options*, easing or not the work of the developer (e.g., think at cloud computing and the as-a-service paradigm). The execution can then be supported by different *execution engines* characterized based their architectural design and the composition languages that they support.
—***Target users***: Since composition approaches aim to simplify the development of applications compared to generic programming, speculations on the skills necessary to successfully compose an application have soon emerged. With this dimension, we aim to understand which kinds of target users (ranging from professional developers to end users) there are and which composition practices they are able to master.

We use the taxonomy in Figure 2 to structure the remainder of this article.

4. SERVICE COMPOSITION LANGUAGES

4.1. Components

We catalog six sub-dimensions, namely type, description, data format, interaction protocol, interaction style and selection, to characterize components. The type tells what kind of functionality is provided, the description how it is advertised, the data format indicates the language used for the representation of exchanged data, the interaction protocol defines how services communicate, the interaction style expresses how communications are initiated, and the selection tells when services are chosen for composition.

4.1.1. Type. We identify three types of component depending on whether the component acts as data source, provides access to application logic, or has a GUI.

—*Data* components consist of services that enclose a Web source in a given format (e.g., RDF+XML, RSS) with a defined data structure and (possibly) relationships between the data elements. These components can be composed to form new services. Mashup tools typically give the user the ability to compose data services (e.g., Yahoo! Pipes supports the visual aggregation of different data sources) [Wang et al. 2009].

—*Application logic* components provide business functionality to other applications [Srivastava and Koehler 2003], such as checking stock availability in a storehouse, processing an online payment, or requesting the shipping of the goods. So-called Business Process Management (BPM) suites provide the necessary functionality to integrate application logic components to realizing given business goals [Papazoglou 2003]. Examples of such suites are Pega's BPM,[1] Bonita BPM,[2] and Oracle BPM.[3]

—*User interface* components include content extracted from Web pages and UI widgets, such as W3C widgets [Caceres 2012], Java portlets [Abdelnur and Hepper 2003], or proprietary formats [Yu et al. 2007]. Typical UI widgets are login widgets, map widgets, and search widgets. Composition of UI widgets typically occurs inside widget containers or engines, which provide for the instantiation and rendering of widgets and support basic infrastructure services (e.g., user management, persistent storage, URL forwarding). Liferay[4] is a portal for Java portlet integration; Apache Rave[5] is an engine for the integration of W3C and OpenSocial[6] widgets, while Daniel et al. [2009a] provide for the integration of the UI widgets proposed in Yu et al. [2007].

4.1.2. Interaction Protocol. SOAP and REST are the archetypical protocols in Web services, and we also include OSGi as an emerging protocol that provides an alternative to the common SOA approach.

—*SOAP* (formerly Simple Object Access Protocol) [Box et al. 2000] is a simple XML-based communication protocol that permits the exchange of information via HTTP and RPC. A SOAP service is operations-based, in that it exposes actions (the operations) performed by the Web service. Regarding the message format, SOAP defines a standard message format for communication, describing how information should be packaged into an XML document. Languages that support the composition of SOAP-based Web services include BPEL and WS-CDL [Kavantzas et al. 2005].

—*REST* [Fielding 2000] (Representational State Transfer) is an architectural pattern that exposes data and functionality through resources accessed via dedicated URLs over HTTP. REST services feature a request-response pattern, where the HTTP methods *Post*, *Get*, *Put,* and *Delete* on a given resource are mapped to the respective CRUD operations *Create*, *Read*, *Update,* and *Delete*. Service responses contain the representation of the requested resource presented in CSV, JSON, XML, or similar

[1]www.pega.com/bpm-suite.
[2]http://www.bonitasoft.com/.
[3]http://www.oracle.com/us/technologies/bpm/.
[4]http://www.liferay.com.
[5]http://rave.apache.org.
[6]http://opensocial.org.

.

formats. RESTful services composition has been realized in languages like JOpera [Pautasso 2009a], which features a visual composition language, as well as in mashup tools like SABRE [Maraikar et al. 2008], which enables the integration of RESTful data sources. An attempt to enable the composition of RESTful services in BPEL is the BPEL extension "BPEL for REST" [Pautasso 2009b].

—*OSGi* [Alliance 2014] (Open Services Gateway Initiative) is a specification that defines a common and open architecture to develop, deploy and manage services inside a Java Virtual Machine (JVM). It features a light service model that enables the publication, binding, and association of services through a service registry. A service is a normal Java object defined semantically by its service interface, which is usually a Java interface. The data format is thus Java objects, and the examples of languages, platforms, or tools for the composition of these kind of services are still exiguous; examples are SOA platforms based on Service Component Architecture (SCA) [Edwards 2011] such as Apache Tuscany,[7] Fabric3,[8] and Paremus.[9]

4.1.3. Description. We group the different ways of describing components into four different lines: SOAP (WSDL, SSDL), REST (WADL, RAML, Swagger), Semantic WS (OWL-S, WSDL-S, WSMO), and deployment descriptors.

—*WSDL* (Web Service Description Language) [Christensen et al. 2001] is an XML-based specification for Web service description. It describes the operations that make up a service, the messages exchanged by each operation, the parts that form each message, and the protocol bindings.

—*WADL/RAML/Swagger:* WADL (Web Application Description Language) [Hadley 2006] is an XML-based description of HTTP-based applications (typically REST). WADL describes a service as a set of resources and their relationships. RAML[10] (RESTful API Modeling Language) is a YAML-based language for describing RESTful APIs. Swagger[11] is a specification for describing RESTful Web services featuring an "interactive" documentation that enables the production, consumption and visualization of the REST APIs.

—*OWL-S/WSDL-S/WSMO:* OWL-S (Ontology Web Language for Services, formerly DAML-S) [Martin et al. 2004] is an ontology that provides a standard vocabulary to semantically describe services. It includes preconditions and (conditional) effects in the description of the Web services, as well as enriched semantic representations of Web service inputs and outputs [Martin et al. 2005]. Similar technologies are WSDL-S (Web Service Semantics) [Akkiraju et al. 2005] and WSMO (Web Service Modeling Ontology) [De Bruijn et al. 2005]. All these languages aim to enhance the discovery, interoperability and composition of so-called Semantic Web Services.

—*Deployment descriptors:* For UI components that require local installation, descriptors serve a twofold purpose: they both describe the interface of components and, at the same time, also serve as deployment configurators. W3C widgets [Caceres 2012] contain a respective config.xml file, Java portlets [Abdelnur and Hepper 2003] a portlet.xml file. Portlets accessible remotely via WRSP [Thompson 2008] are based on common Web services and, hence, described using WSDL. Similarly, OSGi service bundles [Alliance 2014] (JAR files that pack Java classes and resources) contain an XML file with a deployment description, while OSGi services with remote access also expose a corresponding WSDL description.

[7]http://tuscany.apache.org/.
[8]http://www.fabric3.org.
[9]https://paremus.com/.
[10]http://raml.org/.
[11]http://swagger.io/.

4.1.4. Data Format. We split the different message exchange formats used in Web service composition in three categories:

—*JSON* (JavaScript Object Notation) is a lightweight data exchange format [Crockford 2006]. It is text-based and human-readable. It was designed to represent structured data in the scripting language JavaScript, but today JSON is language-independent and all major programming languages provide JSON parsers. JSON is simpler and less verbose than *XML* (XML element has a name, and content is enclosed between pairs of matching tags) [Bray et al. 1997]. However, XML has richer semantics, for example, XML supports nodes of different kinds and different data types [Boyer et al. 2011]. The majority of the Web service composition environments that support RESTful services support JSON. For example, Drupal [Purer 2011], a Web framework that permits the integration of APIs, provides a library to support JSON as a payload format.
—*RSS/Atom* are both XML-based syndication formats for the exchange of Web feeds. RSS [RSS Advisory Board 2009] and Atom [Nottingham and Sayre 2005] are used to publish Web content that is regularly updated (e.g., blog posts and online newspapers). A large number of mashup tools, recognized as one of the most relevant Web 2.0 technologies, support the composition of RSS/Atom feeds [Beletski 2008]. Examples are Damia [Simmen et al. 2008] or Yahoo! Pipes.
—*Java objects* are instances of Java classes. OSGi, for instance, uses Java objects as data exchange format in composition [Gruber et al. 2005]. Only few approach focus on composition based on Java objects. Worth mentioning are Lee et al. [2014], a framework for composing SOAP, Non-SOAP (e.g., OSGi) and Non-Web Services [Diaz Redondo et al. 2007], and a BPEL-style solution to compose OSGi services.

4.1.5. Interaction Style. The mechanism supported by a component to communicate with its clients determines the component's interaction style.

—*Pull:* This style is used when the client explicitly invokes a Web service following a request-response pattern. The communication is thus started by the client, and the component cannot communicate with the client if there is no specific request. The pull strategy is supported by most composition languages, including BPEL, which has dedicated invoke and receive activities, and mashup tools, such as Yahoo! Pipes.
—*Push:* This style allows the Web service to communicate with a client even without explicit requests, provided the client has registered/subscribed with the Web service. The registration is necessary to inform the component about the client's communication endpoint and about the events of interest (e.g., on change, on deletion). This design pattern is known as *publish-subscribe*. The push strategy is infrequent in Web service composition languages with REST-based APIs, since the REST protocol requires the client to initiate the communication [Bozdag et al. 2007]. BPEL supports push interactions with the *receive* activity that can wait (listen) for incoming messages and *event handlers* that can react to generic events triggered by partners.
—*Business protocols:* In addition to push or pull patterns, a business protocol specifies order constraints for invocation sequences [Motahari Nezhad et al. 2007]. That is, by means of a protocol a service provider establishes the rules of the conversation between a service and its clients. For instance, it is typically necessary to add items to a shopping cart *before* the order can be checked out or payed. For a service to be able to properly manage multiple parallel conversations, it is necessary to implement so-called *message correlation rules*, which allow the service to correlate incoming messages with the respective conversation instances they refer to. For example, in BPEL those rules are called *correlation sets* and are a set of properties that uniquely

identify a conversation [Daniel and Pernici 2006]. Web services that implement a business protocol are known as *stateful*; otherwise, they are known as *stateless*.

4.1.6. Selection. This is the process of searching for and identifying concrete Web services to be used in a composition, given a composition's requirements. Component selection may occur at three different stages of the service composition life cycle: design time, deployment time, and runtime. If services are bound to the process at design time, the approach is called *static composition*; if they are are bound at deployment time or runtime, the approach is called *dynamic composition*.

—*Design time* is the phase during which the developer builds the composite service. The developer chooses the services once for all and, unless the composition is modified, the selected Web services are permanently bound to the service composition. A representative of component selection at design time is SECE [Beltran et al. 2012], a platform for context-aware service composition based on user-defined rules; in SECE, a user builds a composition by selecting actions from a predefined set. The actions represent predefined interactions with concrete Web services.
—*Deployment time* is the phase during which a composition is installed in the runtime environment for execution. When the service composition platform enables the selection of atomic services at deployment time, usually the composition developed at design time is stored as a template that can be re-configured every time the composition is deployed. For instance, this occurs in a Service Creation Environment (SCE) [Braem et al. 2006], which provides service composition templates that are defined as abstract descriptions of reusable compositions containing placeholders for services. At deployment time, a code generator binds service placeholders to concrete services.
—*Runtime* is the phase during which the service composition is executed. This kind of selection, in the majority of the platforms or tools that implement this type of selection, is based on algorithms that bind abstract services to concrete ones based on Quality of Service (QoS) attributes such as price, execution duration, security, availability, and reliability. For example, Zeng et al. [2003] propose a global planning approach to select services based on quality constraints and preferences. The METEOR-S prototype [Verma et al. 2005] supports all the three types of selection.

4.2. Target Application

The types of systems and the domains service composition languages cater for are:

—*Mashups* are Web applications that aggregate existing Web resources, including data, presentation and application functionality. Data and presentation typically come in the form of standard data interchange formats such as XML and JSON, syndication formats such as RSS or Atom feeds, or as HTML, ShockWave Flash (SWF), or other graphical elements such as widgets. Application functionality is usually provided via open, REST-inspired APIs developed in languages such as Ruby or JavaScript. The most popular mashup tool today is Yahoo! Pipes [Chen et al. 2012], which aggregates data sources. IBM InfoSphere MashupHub[12] focuses on Enterprise Content Management (ECM) and the integration of heterogeneous data sources (e.g., Web services, databases, local files). JackBe Presto[13] is a mashup platform for enterprise mashups with a graphical editor based on the wiring paradigm. Enterprise Mashup Markup Language (EMML) [Hinchcliffe and Benson 2011] is a domain-specific XML dialect for mashups.

[12]http://www-03.ibm.com/software/products/us/en/mashuphub.
[13]http://jackbe.com/products/presto.

—*Business processes:* A business process is a set of activities, functional roles, and relationships that describe a function of the business that accomplishes a business goal [Hollingsworth 1995]. Business Process Management Systems (BPMSs) support the design, execution, and management of business processes [Van Der Aalst et al. 2003], and represent a powerful instrument for business-to-business (B2B) integration. The *de facto* standard process modeling language is BPMN [Group et al. 2004], whose version 2.0 also has a 1:1 mapping to BPEL. Another relevant, high-level business process language is Yet Another Workflow Language (YAWL) [Van Der Aalst and Ter Hofstede 2005]. The founding efforts on using Web services to cope with B2B integration include eFlow [Casati et al. 2000] and DySCo [Piccinelli et al. 2003].

—*Scientific workflows:* Scientific Workflow Management Systems (SWfMSs) rely on domain-specific languages to enable scientists to effectively define, reuse, execute, and monitor experiments and data analysis through the acts of composition and orchestration. SWfMSs are mostly data-centric, which also implies that a majority of SWfMSs opt for a dataflow-oriented composition approach with implicit control flow. Triana [Taylor et al. 2007], Taverna [Hull et al. 2006], and Kepler [Ludäscher et al. 2006] are representative composition-based SWfMSs.

4.3. Notation

A composition language notation consists of the system of marks, signs, icons, or characters that represent services, dataflow and control flow operators (vocabulary), the rules for their combination (grammar), and their meaning (semantics). We have identified three classes of notations, that is, textual, visual, and hybrid (a combination of both textual and visual). Textual and visual notations differ in the number of dimensions they use; textual languages use one dimension to express a composite service (e.g., a sequence of characters), and visual languages use more than one (e.g., including spatial placement and graphical elements) [Burnett 1999].

4.3.1. Textual. We split the textual notations into three categories.

—*XML-based:* XML [Bray et al. 1997] is a markup language designed for the representation of structured data in a machine and human readable format. It is commonly used for data exchange over the Internet, and several widely known standardized languages are based on it (e.g., XSLT [Clark et al. 1999] and XQuery [Boag et al. 2002]). XML has been used largely to specify Web service composition languages. For example, DAML-S [Ankolekar et al. 2002], WSFL [Leymann 2001], BPEL [Andrews et al. 2003], and WSCI [Arkin et al. 2002] are all XML-based.

—*Code-based* notation entails the use of computer languages that are not XML-based, they may be proprietary or based on currently existing textual computer languages. For example, ql.io, a data mashup tool developed by eBay, permits the integration of HTTP APIs by using a domain-specific language based on SQL and JSON.

—*Controlled natural languages:* These are languages whose vocabulary and grammar are subsets of those of a natural language and are thus easier to learn and use than programming languages [Wyner et al. 2010]. The "control" consists in the selection of the supported vocabulary and grammar. For example, CoScripter [Leshed et al. 2008] allows the user to describe Web-based processes using instructions such as "go to," Aghaee and Pautasso [2012] propose EnglishMash for mashup development with live preview, and Cremene et al. [2009] propose a language based on templates.

4.3.2. Visual. Visual Programming Languages (VPL) offer abstractions that hide technological details via visual symbols and graphical notations [Chignell et al. 2010]. The aim is to effectively represent information and to ease understanding. Our analysis

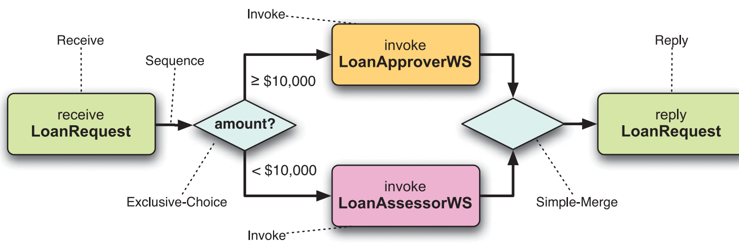

Fig. 3. Representation of a BPEL process with control flow annotations.

has identified four types of visual representations, three of them represent individual notations, the other one includes any combination of them.

—*Spreadsheet-based* approaches usually target end-user programmers [Lieberman et al. 2006]. Examples of tabular notations in service composition are [Obrenović and Gašević 2008] and Husky [Skrobo 2007]; the latter proposes a language where cells encapsulate basic programming elements (e.g., a service invocation), and the control flow is derived from the locations of the cells, where two events are sequential if they are located in two adjacent cells (left to right).
—*Diagram-based* notations consist of symbols and connectors, where the symbols are usually geometric shapes that represent an artifact of the composition, and connectors are wires or arrows that represent the control flow or dataflow or a relationship between the artifacts. For instance, JOpera [Pautasso and Alonso 2005] uses flow graphs for both dataflow and control flow specification (Figure 3 shows a control flow diagram). Mashup tools that apply flow diagrams are JackBe Presto and Yahoo! Pipes, but diagrammatic representations of flow diagrams can be found in scientific workflows as well. For instance, Kepler [Ludäscher et al. 2006] uses icons to represent discrete computational components and arrows to represent dataflows. Other diagram types used for service composition are state charts [Zeng et al. 2003], Petri nets [Hamadi and Benatallah 2003], and UML activity diagrams [Skogan et al. 2004].
—*Other metaphors* include visual notations that use other representations of real objects or situations easily recognizable by users. Used metaphors include, for example, storyboards [Soriano et al. 2008] and jigsaws [Danado and Paternò 2012].

4.3.3. Hybrid. Hybrid notations are a combination of the previous notations. For instance, Vegemite [Lin et al. 2009] is a mashup tool that extends CoScripter with a spreadsheet-like environment called *VegeTable*. The result is a script-based, textual and a spreadsheet-based, visual notation.

4.4. Paradigm

A language paradigm is an approach of programming based on a coherent set of principles and practices, which determine the suitability of the language for solving certain types of problems [Van Roy 2009]. The service composition tools proposed so far cover a variety of paradigms, which we categorize into six classes.

4.4.1. Script-Based. Scripting languages (e.g., Perl [Wall et al. 2000]) are interpreted (not compiled) and typically serve well-defined, domain-specific purposes [Loukissas 2003]. Scripting languages are often of low complexity and, therefore, potentially suitable also to casual users (who, however, still have to learn a programming language). This paradigm has been applied to service composition in different areas, for example, in the areas of mashups [Sabbouh et al. 2007] and workflows, where scripting languages like Swift [Wilde et al. 2011] enable the specification of scientific workflows.

4.4.2. Flow-Based. This paradigm specifies programs as networks of processes that are connected in a directed acyclic graph. This approach is commonly used in service composition, as the processes in these kinds of languages are "black boxes," for example, the Web services, which provide a functionality to remote clients without exposing their internals. The common constructs in the case of service composition are control flow and dataflow connectors. Yahoo! Pipes is an example of a dataflow-based approach, whose graphical modeling language has connectors that define the data passing between processes. Also BPEL is flow-based, however, with a focus on control flow.

4.4.3. Functional. Functional languages are based on the construct of mathematical functions. Programs are defined as functions, whose evaluation represents the output of the program. Functions are side-effect-free and stateless in that the result of a function depends exclusively on its inputs, reason why this paradigm is considered purely data-oriented. The paradigm is highly exploited in scientific workflows, where the parallelization of computations is important, a feature that is facilitated by Web services modeled as functions without ordering constraints. The representation of services as functions is valid for services that are stateless and side-effect–free [Tan et al. 2010]. A functional language is, for example, used in the mashup tool MashMaker [Ennals and Gay 2007]. It is, however, worth noting that functional programming in service composition does typically not comply completely with the pure functional programming paradigm [Hudak 1989], as pure functional programming has some practical limitations (e.g., it does not permit I/O operations).

4.4.4. Rule-Based. Rule-based programming systems consist of facts, rules, and control strategies. Facts and rules are the knowledge of the system. Facts are the information (i.e., statements and relationships), and rules are condition-action expressions transforming facts. Control strategies resolve conflicts if conflicting rules are triggered. Rule-based systems are highly modular: they can be broken down into parts, solved separately, and integrated afterward [Mohan 2000]. For example, Orriëns et al. [2003b] use business rules specific to service composition life-cycle phases (e.g., data rules, constraint rules, exception rules) to drive a service composition process; SWORD [Ponnekanti and Fox 2002], a developer toolkit for Web service composition, represents services as rules that transform inputs (condition) into outputs (action) and is able to determine whether a desired composite service, specified by means of facts and rules, can be realized using a set of given services or not.

4.4.5. ECA-Based. Event-driven systems have been used to support interactions in several classes of loosely coupled and dynamic applications [Russell et al. 2006]. More specifically, Event Condition Action (ECA) rules have been used to encode the logic of composite services in function of runtime events instead of facts in a knowledge base. ECA rules are especially attractive to support the customization of composite services, as rules can easily be added, modified, or removed to reflect new requirements. Although apparently similar to rule-based approaches, ECA-based compositions are not easily amenable to systematic reasoning, such as property verification, as they lack the necessary knowledge base.

4.4.6. Query-Based. Query languages are designed for the processing of data (i.e., retrieve, insert, modify, delete) at a high level of abstraction. The quintessential, query-based language is the Structured Query Language (SQL) [Chamberlin and Boyce 1974]. Query languages are categorized depending of the artifact they query for; for instance, SQL and OQL [Alashqur et al. 1989] are *database* query languages, XSLT and XQuery are *XML* query languages, and SPARQL is a *graph* query language. Various are the examples of query languages applied to the composition of services: XQSE [Borkar et al. 2008], based on XQuery, integrates data services in the AquaLogic platform [Carey 2006]; XL [Florescu et al. 2002] integrates Web services into XML documents; and ql.io[14] is an SQL-based language to fetch data from services.

4.5. Composition Constructs

Composition constructs are the building blocks that enable the aggregation of Web services. The constructs supported by a language are an objective measure of the expressiveness of the language. There are two essential types of composition constructs in service composition, that is, *control flow* constructs (for process-oriented compositions) and *dataflow* constructs (for data-oriented compositions) [Fensel and Bussler 2002]. Other types of constructs we consider are *data transformations*.

4.5.1. Control Flow Patterns. The specification of a control flow is based on control flow constructs that represent communications with atomic services and specify the execution order of communications [Tran et al. 2008]. Communication primitives typically define a single interaction between a process and an atomic Web service. For example, communication primitives of BPEL are *invoke*, *receive*, *wait*, and *reply*. The order of execution of services is determined by control flow constructs that allow the implementation of basic and advanced control flow patterns. This dichotomy has been consistently applied in the literature (e.g., van Der Aalst et al. [2003]), and there is a wide consensus about which category each control flow pattern belongs to: *basic* control flow patterns are sequence, parallel split, synchronization, exclusive choice, and simple merge; *advanced* control flow patterns include multi-choice, loops, and similar.

Figure 3 represents a BPEL process that uses six control flow constructs: (i) *sequences* partially order activities, (ii) *receive* indicates that the process expects an incoming message (a loan request), (iii) *exclusive-choice* represents a conditional branch where only one path is executed (the path taken depends on the amount of the loan requested), (iv) *invoke* calls services (this construct is used to invoke the two Web services "LoanApproverWS" and "LoanAssesorWS"), (v) *simple-merge* joins alternative branches into a single branch, and (vi) *reply* sends a message in response to a received message (it sends the reply to the actual loan request).

4.5.2. Dataflow Patterns. A dataflow defines how data are passed among Web services in terms of the actions performed on the output of a service that is transferred as input to another service [Rahm and Bernstein 2001]. Every composition language, either by graphically wiring outputs to inputs (e.g., in Yahoo! Pipes) or via textual expressions (e.g., in BPEL), supports the specification of dataflow. Dataflow constructs commonly perform actions such as copying data (mapping), organizing data based on certain criteria (sorting), and combining data (merging).

There exist two basic data passing paradigms: blackboard and explicit dataflows [Alonso et al. 2004a]. The *blackboard* paradigm stores data centrally in shared variables that are used as sources and targets by Web service activities. Several service composition languages follow this paradigm, among them BPEL. The *explicit dataflow* paradigm makes dataflows an integral part of the composition model by means of dataflow connectors. A dataflow connector describes how data is manipulated and routed to or from Web services.

4.5.3. Data Transformation Capabilities. The instructions specified in the dataflow may include data manipulations. In order to ensure the data exchange between

[14]http://ql.io.

heterogeneous Web services, that is, with mismatching output and input data formats, suitable data transformation constructs may be needed. Typically, transformations take valid data under one schema and convert them to valid data under another schema [Pessoa et al. 2008]. The data transformation capabilities we identified are either *ad-hoc transformations* or based on dedicated *transformation languages*.

—*Ad-hoc transformations* are based on dedicated data transformation constructs provided by the composition language. For example, BioFlow [Jamil et al. 2010], a declarative language for scientific workflows, extends SQL with specific statements for the transformation/integration of XML data. Asavametha et al. [2011] propose the use of Topes [Scaffidi et al. 2008], a transformation language for strings able to reformat strings passed between services. Active XML [Abiteboul et al. 2004] is a framework for the integration of data by means of service calls embedded in XML documents.
—*Transformation languages* require the service composition language to embed data transformation languages or even generic programming languages, which are able to perform data transformations. Typically, composition languages that support the use of transformation languages support the invocation of XPath functions or external XSLT stylesheets (e.g., JOpera, BPEL). Examples of composition languages that allow the use of code snippets in generic programming languages include workflow languages such as Taverna [Hull et al. 2006] (Java) and Kepler (Java and Perl).

4.6. Crosscutting Concerns

Service composition comes with a number of crosscutting concerns that can be supported by the infrastructure and do not depend on any individual composition. In the following, we overview the most important crosscutting concerns in service composition, that is, exceptions, transactions, security, and Service Level Agreements (SLAs).

4.6.1. Exceptions. Exceptions are anomalous behaviors that occur during the enactment of the process defined by a composite service. They are caused by unresponsive Web services, unavailable services, unexpected messages from a Web service, and similar; Chan et al. [2009] provide an extensive analysis of causes for faults in service composition. Exception handling permits the composite service to detect failures and to take corrective actions [Alonso et al. 2004b; Gutierrez-Garcia and Ramos-Corchado 2011]. For instance, BPEL fault handlers permit one to catch faults, throw events, and compensate faults. JOpera [Pautasso and Alonso 2005] supports exception handling via visual constructs added to the control flow graph. In particular, the behavior of the process in case of exception is defined by adding connectors to a task (e.g., a Web service invocation) that are fired in case of failure of the task. In AO4BPEL [Charfi and Mezini 2004] and Dynamo [Baresi et al. 2007], Aspect-Oriented Programming (AOP) is proposed to supervise and handle exceptions in BPEL processes. An approach to handle exceptions via an extended Petri net model is explained in Hamadi et al. [2008].

4.6.2. Transactions. A transaction is a group of Web service interactions that achieve a logic (sub-)goal within a service composition only if all interactions complete successfully [Bernstein and Newcomer 2009]. For example, a stock broker application may be composed of one Web service that withdraws money from the customer's bank account and one that deposits the money in the broker's bank account. The two actions must be grouped into a transaction, since both services must succeed for the bank transfer to be correct. If an error occurs in a transactions, the actions of the transactions that have already been performed must be compensated, that is, rolled back until the status right before the transaction started. BPEL supports compensation handling via compensation actions and is typically used in conjunction with WS-Transaction [Cabrera et al. 2002] and WS-Coordination [Cabrera et al. 2004], which empower BPEL with distributed coordination capabilities. Also BPMN defines sub-processes that can be associated with compensation events.

4.6.3. Security. The use of Web services implies the crossing of trust boundaries and the involvement of software of uncertain reliability, which asks for the mitigation of risks. The security mechanisms that aim to mitigate risks are applied at four different levels in an SOA, namely user, message, service, and transport.

At the *user* level, the goal is to verify the users' identity and to control access to resources (i.e., services, operations), which is achieved via two techniques: authentication and authorization. *Authentication* assures the truthfulness of a user's identity. For instance, OpenID [Recordon and Reed 2006] is an open, decentralized, single sign-on standard for user authentication. *Authorization* is the process of granting the authenticated user access rights to read or write requested resources. Common standards to provide access control in SOAs are WS-Security [Nadalin et al. 2004] for SOAP Web services and OAuth [Hardt 2012] for REST APIs [Prehofer et al. 2010]. A standard that includes identification, verification, and access control is the Security Assertions Markup Language (SAML) [Ragouzis et al. 2008].

At the *message* level, the goal is to assure confidentiality and integrity. In order to assure *confidentiality*, the message must be encrypted. In the case of XML-formatted messages, the W3C proposes for instance the use of XML-Encryption [Imamura et al. 2002]. *Data integrity* can be achieved by adding data integrity fields, such as checksums [Finkenzeller 2003] or by using the XML-Signature specification [Imamura et al. 2002]. The WS-Security standard provides data confidentiality and integrity, as it includes the XML-Encryption and XML-Signature specifications.

At the *service* level, the objective is to ensure the availability and correct functioning of a service. Such must be protected from threads and attacks that may affect the service itself or one of the systems or resources required for its functioning. The majority of attacks belong to the category Denial of Service (DoS) [Needham 1994] and are prevented via mechanisms like intrusion detection, XML filtering controls, and specialized XML gateways/firewalls [Tipnis and Lomelli 2009]. WS-Security Policy, based on WS-Policy [Vedamuthu et al. 2007], permits the specification of security policies (i.e., requirements and capabilities) by the service provider, which is also of help in the prevention of attacks. Specific attacks to service compositions in BPEL are typically fended by detecting semantically invalid requests (attack messages) or by using firewalls. A detailed list of Web service attacks including service composition attacks can be found in Jensen et al. [2007].

At the *transport* level, the goal is to guarantee a seamless and reliable communication between parties. The protocols and specifications at this level provide mechanisms that cope not only with transport threads but also with service, message, and identity risks as well. One of the most popular approaches in this respect is the Transport Layer Security protocol [Dierks 2008] (TLS, formerly known as SSL), a cryptographic protocol used to secure connections over the Internet that provides privacy, authentication, and reliability. It is also worth mentioning Hypertext Transfer Protocol Secure (HTTPS), which is a protocol at the application layer that makes use of SSL/TLS to transfer sensitive data. The use of HTTPS is very common in RESTful APIs to secure the communication and to prevent eavesdropping attacks (e.g., man-in-the-middle).

4.6.4. SLAs and QoS. SLAs [Lamanna et al. 2003] are contractual obligations that describe the mutual responsibilities between a service consumer and a service provider. The core of the contract is the service guarantees. It includes functional and non-functional aspects. The *functional* aspects define what the service is expected to deliver (e.g., operations and outcomes). The *non-functional* aspects define QoS guarantees (e.g., regarding availability, price, response time, or throughput) [Kritikos et al. 2013]. An SLA usually also contains clauses that define business rules, such as restrictions, penalties, resolution of disputes, and payments [Yan et al. 2007]. A standard protocol for the specification of SLAs is WS-Agreement [Andrieux et al. 2004]. Yan et al. [2007] use agents to negotiate QoS constraints with providers to dynamically select services in a composition. Canfora et al. [2005] present a QoS-aware service composition approach based on genetic algorithms.

5. KNOWLEDGE REUSE

Composition-based productivity is not only achieved through suitable composition models and languages, but also through reuse. In fact, reuse may lower development times and increase software quality. In this respect, the reused artifact (what) and the adopted reuse technique (how) are of particular importance.

5.1. Reused Artifact

An artifact is a logical entity of a service composition, for example, a single element like a *component* or a *data transformation rule* or multiple related elements, in which case we distinguish so-called *process fragments* from complete *examples* of ready processes.

5.1.1. Components. Components enable the reuse of data, application logic, or UIs. The reuse of Web services includes the reuse of both atomic and composite Web services, as the later are the same as the former from the user's point of view from the moment they are encapsulated and published. Service reusability is one of the founding principles of SOA, and the reuse of services is the essence of service composition.

5.1.2. Data Transformation Rules. Data integration in service composition is one of the most time consuming and tedious tasks. The reuse of rules defined for the transformation of data may thus significantly decrease the effort of this task. Data transformation rules are expressions that permit the manipulation of the data. They comprise two main elements: transformation functions and data sources. An example of transformation rules in service composition can be found in Thöne et al. [2003], where the authors propose a UML-based service composition language called *UML-WSC* that features graphical constructs to represent data sources and transformation rules (mappings).

5.1.3. Process Fragments. Process fragments are coarse-grained units of composition logic [Schumm et al. 2012]. Their reuse requires that the fragments are self-contained and coherent; hence, they must be meaningful and have a clear functionality even when they are not integrated in any service composition [Markovic and Pereira 2008].

A widely used representation of process fragments consists of modeling them in an abstract manner, aiming to cover a wide range of processes where the fragments can be reused. This representation has been realized mainly through the application of *templates*, where a template is a specification of a service composition fragment with abstract placeholders expressed in a concrete language. When a template is used to model a composite service placeholders must be manually or automatically concretized, in order to configure the template to be included in the service composition; this process is called *template instantiation* [Volpano and Kieburtz 1985]. An illustrative example of reusing service composition fragments based on templates is Geebelen et al. [2008], a framework for the design of BPEL processes that includes a library of templates that can be integrated in a composition. Process/composition fragments have also been used in the context of mashups: MatchUp [Greenshpan et al. 2009] supports the autocompletion of mashups via so-called *mashlets* (e.g., data source components) or *glue patterns* (pieces of integration logic for mashlets). Roy Chowdhury et al. [2011] propose the reuse of more complex fragments, all equipped with suitable data mappings and component gluing logics. VisComplete [Koop 2008], a system for developing visualization pipelines, proposes the use of so-called *partial completions*, that is, sets of structural changes that complete a given partial visualization pipeline, so as to reflect the structure of pipelines contained in a collection of existing pipelines.

5.1.4. Examples. The reuse of examples in Web service composition consists in using previously designed compositions and adjusting them to new requirements. The main differences with the previous artifacts is the concreteness and completeness of the examples: while all artifacts mentioned above were based on reusing parts of processes (e.g., components or fragments) or on reusing abstract artifacts such as templates, examples are concrete and complete processes. They are full-fledged solutions to specific problems and are not generalized or abstracted. Their reuse requires therefore to manually adapt (edit) the example to the new requirements. For instance, Yahoo! Pipes enables reuse of examples by means of cloning (see Section 5.2.3).

5.2. Reuse Technique

Given an artifact to be reused, it is important to understand *how* it can be reused in practice. In this respect, our study identified a varied set of techniques (in Appendix A we discuss how developers can help each other to reuse knowledge).

5.2.1. Search and Discovery. This is the activity performed by the developer when he expresses requirements or constraints the target artifact should satisfy as a query.

—*Keyword search* looks for artifacts that present a specific term (the keyword) in any of the attributes of the artifact (e.g., description, name, tags) [Rajasekaran et al. 2005]. Keyword search has been largely applied to discover Web services [Bachlechner et al. 2006]. For instance, UDDI [Bellwood et al. 2002], the standard for publishing and finding Web service descriptions, or myExperiment [Goble et al. 2010], a repository of scientific workflows, support keyword search.
—*Semi-structured search* makes also use of matchmaking based on subsumption and equivalence relationships. Matchmaking techniques have been applied to semantic services, most of them based on the pioneering work of Paolucci et al. [2002]. In service composition, for instance, IPM-PQL [Choi et al. 2007] is a semi-structured XML-based process query language that allows the user to query a registry using context (e.g., actors, resources), structure (e.g., activities), and/or classifications (categories).
—*Relational query* languages are specialized in the extraction of information from relational databases, among them SQL stands out as the de facto standard [Leavitt 2010]. An example of Web service registry that supports relational search via a subset of SQL is ebXML Registry Services [ebXML Registry Technical Committee et al. 2002]. With a completely different purpose, a relational query language is used in Yahoo Query Language (YQL).[15] Through its Web service interface, it permits access and manipulation of data from the Internet (e.g., Yahoo! Pipes models) by using SQL-like commands.
—*Graph-structured search* supports querying graph-structured data (e.g., RDF) using graph query languages. For instance, SPARQL is used in registries like iServe [Pedrinaci et al. 2010], an open repository that exposes service descriptions as Linked Data [Bizer et al. 2009]. Examples of visual graph query languages in service composition are BPMN-Q [Awad 2007], designed for querying BPMN diagrams, and BP-QL [Beeri et al. 2008], focused on the elements and structure of BPEL descriptions.

[15] developer.yahoo.com/yql/.

5.2.2. Copy/Paste. It is the ability of selecting a service composition artifact from one location (e.g., a repository, a service composition tool) and inserting a copy of it into another location. Copy/paste is supported by all the service composition tools that adopt textual notations, whereas for visual notations, it is more complex to implement. Intalio—BPMS Process Designer,[16] for instance, supports copy/paste of BPMN diagrams (tasks, events, gateways), including their characteristics and dataflow specifications.

5.2.3. Cloning. This is the act of creating a replica of an existing service composition (use as example), so as to adapt it and extend it with new constructs to meet new needs. The implementation of this technique is generally simple and available in the great majority of service composition tools that support saving/opening compositions and sharing them with the community. Cloning is, for example, highly used in Yahoo! Pipes. Stolee et al. [2011] specifically study this practice, examined approximately one third of the pipes in the Yahoo! Pipes repository, and found that over 54% of the pipes had been cloned at least once.

5.2.4. Recommendation. Recommending artifacts means pro-actively suggesting artifacts that may facilitate the composition process. Recommendations typically come from so-called recommender systems, which suggest information of likely interest to a user based on profiles, usage histories, and usage context [Resnick and Varian 1997]. Manikrao and Prabhakar [2005], for example, suggest Web services based on users' ratings of Web services. In the context of mashups, the MatchUp project [Greenshpan et al. 2009] recommends mashup fragments based on partial matching of mashup structures. A similar approach is proposed by Roy Chowdhury et al. [2011], who recommend composition patterns in a visual mashup development environment by performing on-the-fly similarity search over a knowledge base of reusable patterns. Mashup Advisor [Elmeleegy et al. 2008] uses artificial intelligence to provide recommendations in IBM Lotus Mashup Maker. VisComplete [Koop 2008] uses graph similarity and data mining to provide support for completing visualization pipelines based on information obtained from a repository of existing pipelines represented as graphs.

6. AUTOMATION

Service composition is a complex task that has inspired a variety of automation techniques trying to overcome some of the complexity. We identified three automation techniques of major importance: synthesis, planning, and model-driven development.

6.1. Synthesis

The synthesis of Web service compositions interprets Web services as *state transition systems* modeling the services' *business protocol*, taking into account that services are generally *non-deterministic* (the occurrence of transitions cannot be foreseen in advance) and *stateful* (the occurrence of a given transition depends on past transitions) [Fiadeiro et al. 2007]. The goal of synthesis is to identify an *orchestrator* that integrates all necessary services to mimic a given target behavior expressed again as a state transition system [Lämmermann 2002]. The problem is complicated and may require the use of complex control flow and dataflow dependencies [Marconi and Pistore 2009].

The most widely used approach to service composition synthesis is the so-called *Roman Model* [Calvanese et al. 2008]. Service composition in this approach is achieved by synthesizing an orchestrator (an implementation of service orchestration logic) that realizes the target service using fragments of the available services. The techniques proposed to synthesize the orchestrator are diverse. The first approaches reduced

[16] http://www.intalio.com/products/bpms.

the problem to Proportional Dynamic Logic (PDL) [Berardi et al. 2005], more recent approaches applied Linear Time Logic (LTL) [Piterman et al. 2006] and simulation [Berardi et al. 2008]. Pistore et al. [2005] use symbolic model checking to generate an executable BPEL process, starting from an abstract description of component services in BPEL and a set of requirements and constraints of the target service composition expressed in Eagle [Lago et al. 2002].

6.2. Planning

Planning, which is a branch of Artificial Intelligence (AI) [Rao et al. 2006] uses semantic Web services with machine-understandable descriptions of service properties and capabilities and reasoning mechanisms to select and aggregate services [McIlraith et al. 2001]. Semantic Web services are rich and machine-understandable descriptions of service properties and capabilities [McIlraith et al. 2001]. Semantic Web services consist of a formal invocation, pre- and post-conditions, and semantic input/output descriptions, which enable automatic composition. Examples of planning-based composition techniques are Hierarchical Task Networks (HTNs) [Erol et al. 1994], situation calculus [Levesque et al. 1998], rule-based reasoning [Buchanan and Shortliffe 1984], and the Planning Domain Definition Language (PDDL) [McDermott et al. 1998].

SHOP2 [Sirin et al. 2004] represents services as actions and applies task decomposition in HTN planning to DAML-S based services. McIlraith and Son [2002] leverage on Golog (a high-level logic programming language for dynamic domains, with control constructs, support for non-deterministic choices and user constraints to enable automatic composition) extensions [Levesque et al. 1997] to support planning-based composition. A rule-based planning approach is proposed in Medjahed et al. [2003], and a rule-based planner is adopted in SWORD [Ponnekanti and Fox 2002] (see Section 4.4.4). Redavid et al. [2008] use Semantic Web Rule Language (SWRL) rules to generate candidate service compositions, starting from a given target service (the goal). Cugola et al. [2012] propose a declarative, logic-like language to model service orchestrations called DSOL associated with an ad-hoc engine that uses planning to support self-adaptive service compositions at runtime. Kubczak et al. [2009] propose the synthesis of mashups using a planning-based approach.

6.3. Model-Driven Development

Model-driven development is practice that aims to alleviate the developer from low-level coding and to reason at a high level of abstraction, typically by drawing a model of the target application. Coding is automated (at least partially) by generating code implementing the functionalities expressed in the model. Automation thus comes in the form of reused schematic and recurrent code fragments, which, in the context of service composition, either express composition logic or data transformations.

6.3.1. Composition Logic. Model-driven approaches for service composition provide logical constructs, such as services, service invocations, dataflows, control flows, and similar, that are technology agnostic. Models can then be translated into executable service composition languages without human intervention [De Castro et al. 2006]. Orriëns et al. [2003a], for example, use UML to describe compositions and the OCL to define business rules. Similarly, a large body of research has adopted UML for the design of Web service compositions [Gardner 2003; Sheng and Benatallah 2005; Caceres et al. 2003; Skogan et al. 2004; Mayer et al. 2008]. However, UML it is not the only option. For example, Baina et al. [2004] explain the generation of BPEL skeletons from two models: a state machine model for the specification of Web service conversations and a state chart for the composition. Manolescu et al. [2005] leverage on the Web Modeling Language (WebML [Ceri et al. 2002]) for the model-driven development of Web applications and Web service orchestrations. The recent trend is to use BPMN for service composition [Group et al. 2004].

6.3.2. Data Transformation. Specifying data transformation rules (see Section 4.5.3) manually can be a non-trivial and tedious task. Model-driven engineering permits the automation of transformation rules implementation. The use of models permits to raise the level of abstraction enabling a better management of complex tasks; and, in addition, for data transformation, permits to encompass different schema types (e.g., XML, relational) under the same representation. For instance, Jouault et al. [2008] provide a modeling notation to express semantic correspondences between model elements and an automated data mapping rules generation process based on model transformation languages. Bernstein and Melnik [2007] propose a model management system that defines operations for the manipulation of models and operations that describe data mappings between source and target schemas. Avazpour et al. [2013], instead, propose a tool that automatically generates mappings between source and target models starting from examples of existing mappings.

7. TOOL SUPPORT

Other software tools may provide developers with productivity support, such as versioning, debugging, testing, and refactoring. However, we have found that only refactoring and versioning have been addressed so far.

7.1. Refactoring

Refactoring aims to systematically improve how a service composition has been implemented without altering its functionality [Fowler 1999]. Service-based systems, specially when they are business oriented, require continuous changes. A good design of services is essential to ensure that they can be easily updated. However, the design of a service frequently suffers modifications, and refactoring techniques may make sure the design stays good as development goes on [Krogdahl et al. 2005]. Visual refactoring is one of the features provided by JOpera [Pautasso 2005]. This technique supports refactoring operations such as renaming, extraction, and inlining of composition fragments, and synchronization of service interface modifications. Stolee and Elbaum [2011] found out that 81% of the mashups developed with Yahoo! Pipes contained deficiencies and proposed an automatic identification and resolution of such deficiencies by the application of refactoring techniques consisting of customized graph transformations.

7.2. Versioning

Service composition systems should integrate versioning techniques, such as version control tools, to track the evolution of the services and to guarantee that the different versions of published services can be used by third parties unproblematically [Gold et al. 2004]. An approach to versioning in SOA is presented Leitner et al. [2008], which studies the possible changes of WSDL services and propose an approach to versioning them using service version graphs and selection strategies. More relevant to service composition is the work in Joeris and Herzog [1999], which addresses the problem of versioning for workflows by separating task definitions into interface and process definitions.

8. EXECUTION PLATFORM

The execution platform is where service compositions are deployed and run. In this section, we explain the respective deployment and execution options.

8.1. Deployment Options

Deployment is the process of making a finished Web service composition operational and available for execution. We identify two core approaches: cloud versus on-premise.

8.1.1. Cloud. The cloud model is composed of three service models: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). SaaS provides the consumer with online access to applications (e.g., Google Docs); PaaS typically provides access to environments for application development, deployment, and execution (e.g., a Web server for Web applications, including the necessary software development platforms, operating systems, Web services, databases); IaaS provides the consumer the capability to manage on his own infrastructure elements (e.g., compute, storage, network resources), and to install own operating systems and applications [Mell and Grance 2011]. IaaS could, therefore, be used for the deployment of services and composition, but it does not provide any specific support for this; PaaS has instead been applied in service composition for the hosted creation, deployment, and execution of compositions. Development instruments may have different levels of integration with the cloud: they may be fully integrated, meaning that everything occurs within the cloud environment, or they may be partially integrated, which is the case of service composition tools that provide a desktop application for the creation of the service composition and an execution environment that instead runs in the cloud.

Example PaaS offers are those of BPM vendors such as Apache Stratos[17] and IBM BPM on Cloud,[18] those of mashup tools such as Yahoo! Pipes and JackBe Presto Cloud and those of SWfMSs such as Pegasus [Deelman et al. 2005] and Tavaxy [Abouelhoda et al. 2012], an integration of Taverna and Galaxy [Goecks et al. 2010] with cloud support. The typical concern with cloud deployment is the loss of control by the developer over aspects like security and availability. Considering the distributed nature of service compositions and that different service providers may deploy on different clouds with different and possibly diverging security policies, security is indeed a problem [Wei and Blake 2010], but economic [Tak et al. 2011], environmental [Berl et al. 2010], and regulatory factors [Jaeger et al. 2008] must also be considered.

8.1.2. On-Premise. Compositions deployed on-premise are made operational on the platforms and infrastructures hosted in-house. Security, availability, and the overall management of hardware and software are under the responsibility of the composer. This option is still considered more secure, as there is full control over services, systems, and data. An example of service composition system that allows the deployment on-premise is Intalio BPMS, which, among other runtime components, makes use of the Apache ODE BPEL engine. The main concerns related to on-premise deployment are scalability and flexibility of operation [Wang et al. 2010]. Implementing own, large-scale service composition infrastructures can be complex and costly, which makes this option less suitable for SMEs or individuals, which therefore usually try to alleviate their requirements or to move to the cloud [Kim 2009].

8.2. Execution Engine

There are three major services composition execution approaches: process execution engine, service bus, and code generation.

8.2.1. Business Process Engine. A business process engine is a centralized controller for the instantiation, monitoring, analysis and management of processes [Chang 2006]. They are designed to balance the use of system resources in function of the duration of

[17] http://stratos.apache.org/.
[18] https://www.bpm.ibmcloud.com.

the processes: for *short-running* processes, the engine keeps the system resources active, and process state is maintained in quick-access memory for optimal performance; for *long-running* processes, the engine releases system resources and stores process state in permanent memory (e.g., a database). Well-known business process engines for service composition are OW2 Orchestra Engine[19] for BPEL and the open-source BPM suite jBPM[20] for BPMN 2.0 processes.

8.2.2. Service Bus. A service bus or enterprise service bus (ESB) is an software infrastructure that provides connectivity for the exchange of messages between services. The essential feature of a service bus is mediation, which enables interconnectivity between heterogeneous services, regardless of data formats or transport protocols. But a service bus is also responsible for collecting, processing (i.e., translating and/or transforming), routing, and delivering messages. ESBs are typically based on common standards: Java Message Service (JMS) for messaging, XSLT for transformations, and Java Connector Architecture (JCA) and SOAP for the connectivity [Chang 2006]. An example of open-source ESB is OpenESB[21], based on the Java Business Integration (JBI) specification. Another example is Oracle Service Bus,[22] which also provides features such as security, load balancing, and monitoring.

8.2.3. Code Generation. The composition of Web services can, however, also be done using generic programming languages that are not specifically tailored to composition (e.g., Java, PHP). Executing long-running processes with a generic languages can be a cumbersome task, as their execution is generally not optimized for the orchestration of services with long-lasting processing times and the management of process state. The risk of failures during runtime and, hence, the interruption of a running process, is high. Nevertheless, for compositions with low complexity, the use of generic programming languages is feasible, also thanks to frameworks that specifically support the interaction with Web services. Well-known examples of such frameworks are Apache Axis2[23] for Java and C, gSOAP[24] for C and C++, and WSO2 WSF/PHP[25] for PHP.

9. TARGET USERS

The last dimension of our analysis framework aims to understand the nature of the users effectively engaged in the practice of composition. Traditionally, software engineering distinguished between *programmers* (or *developers*) and *end-users*—the former developing software, the latter using it. Over the last years, however, these roles have increasingly blurred, and today this distinction is no longer as clear as one might think. In the course of our analysis, we identified the following three types of users:

—*Professional programmers*: Professional programmers have all the necessary composition skills to develop also very sophisticated composites, possibly including solutions to crosscutting concerns like transactions or security. They have sufficient knowledge of the necessary language notations and composition paradigms and, if not, know how to acquire such autonomously. One key skill of professional programmers is the ability to develop new APIs or components for reuse by others, not only composite applications. Programmers developing RESTful Web services, SOAP Web

[19]http://orchestra.ow2.org/.
[20]www.jboss.org/jbpm.
[21]http://www.open-esb.net.
[22]http://www.oracle.com/technetwork/middleware/service-bus.
[23]http://axis.apache.org/axis2/java/core.
[24]http://www.cs.fsu.edu/~engelen/soap.html.
[25]http://wso2.com/products/web-services-framework/php.

services, UI widgets, and so on fall into this category, as do programmers using composition languages like BPEL or directly Java or C#. They not only develop simple composites but also complex, mission-critical B2B integrations.

—*End-user programmers*: These are programmers that are able to develop their own "computations" in the form of composite applications that serve some limited, typically personal, purpose. They know about components, software reuse, and are familiar with some composition paradigm that allows them to compose components. Depending on their work and interests, they may master some notation (e.g., BPMN) for composition. We distinguish three sub-classes of end-user programmers:

  —*Domain experts* are people who compose software artifacts in the context of specific, limited domains they are familiar with. Examples of domain experts are secretaries or accountants who use spreadsheets to automate bookkeeping tasks, scientists that develop scientific workflows, for example, to analyzed human genomes, business process modelers who model processes that involve both human actors and automated computing tasks, and the like. An example of SWfMS suitable for domain experts is Taverna [Hull et al. 2006].

  —*App developers* are people who develop client-side configurations and scripts for component-based applications, leveraging on backend-as-a-service offers. They do not have advanced programming skills, but they are able to glue together APIs and services. Examples of app developers are people who configure content management systems (e.g., WordPress) or who develop simple Web or mobile applications starting from easy-to-use programming frameworks such as Ruby on Rails or Django.

  —*DevOps* are advanced system administrators and/or IT managers who engage in programming, so as to automate and optimize their everyday IT operations tasks. They mediate between the needs of developers (e.g., fast new software changes) and those of IT operators (e.g., stability). More and more, DevOps complements agile software development, which, for instance, asks for the development of effective command line scripts or the design of automated system migration workflows. A platform that fits the characteristics of DevOps for the composition of services is BlueMix,[26] which enables them to build service-based cloud applications.

—*End-user app remixers* are people who do not have any notion of software development or composition. They are even unaware of APIs, Web services, and UI widgets (i.e., of components). Yet, they are familiar with the Web and applications in general. Thus, they think in terms of applications (the concepts of service and application are blurred) and of simple rules to integrate them. Most notably, if-this-then-that (ifttt[27]) caters for the needs of these users: it allows them to write simple ECA rules, such as "if a new photo about me is uploaded to Instagram, add it to Dropbox." The necessary application wrappers or API/service invocations are taken over by ifttt.

This taxonomy of target users is independent of the intent behind the users' development efforts and not meant to be exhaustive. Over time, new types of domain experts or other sub-classes of end-user programmers may emerge; however, we expect the three top-level types to keep their validity for long.

10. APPLYING THE TAXONOMY: EVALUATION OF SERVICE COMPOSITION APPROACHES

In this section, we discuss and compare the different state-of-the-art approaches in service composition, by classifying and characterizing them along the dimensions of the presented taxonomy. The approaches analyzed include major research prototypes

[26]http://www.ibm.com/cloud-computing/bluemix/.
[27]https://ifttt.com/.

and industrial systems, as well as service composition methods and techniques. The selection of the approaches was a three-step process that involved:

—A *preliminary selection* of candidate contributions from *leading, peer-reviewed research conferences* (main and demo tracks) and *journals* relevant to the domain from the year 2000 onwards, including the following conferences: BPM, CAiSE, CKIM, EDBT, ER, ICDE, ICSE, ICSOC, ICWS, ISWC, VLDB, WISE, WWW; and journals: IS, IEEE Internet Computing, TKDE, TSC, TOSEM, TWEB, VLDBJ.
—A *further selection* of systems derived from the *authors' knowledge and informal conversations* with academic colleagues and industry experts.
—A major *refinement* of the selected systems based upon continuous discussion between the authors under the basis of the *criteria* established for the selection: relevance, significance, impact, and originality of the approach.

At the end of this process, 12 platforms were selected: eFlow [Casati et al. 2000], FormSys [Service Oriented Computing Group 2010], Intalio BPMS,[28] JOpera [Pautasso and Alonso 2005], Self-Serv [Benatallah et al. 2003], SHOP2 [Nau et al. 2003], Sword [Ponnekanti and Fox 2002], Taverna [Hull et al. 2006], XL [Florescu et al. 2003], Yahoo! Pipes,[29] YAWL [Van Der Aalst and Ter Hofstede 2005], and jBPM.[30]

We split the taxonomy into three parts for a more concise and comprehensive analysis. First, we analyze the language aspects and the target user; second, the knowledge reuse and automation; and third, the tool support and execution platform. For each part, we analyze the 12 platforms selected using the analysis framework proposed. The same three parts have been used to split Appendix B, where we list all the approaches presented in this survey organized by characteristics.

10.1. Language and Target User

Language and target user are two different yet closely related dimensions. The design of the language greatly determines the target users of the system [Lieberman et al. 2006]. Therefore, to analyze the different languages and how their design affects the target user in service composition platforms, we performed the analysis on the selected tools using these two dimensions together. Table I maps the selected platforms onto the taxonomy of language aspects described in Section 4.

From the analysis of the language and target user dimensions, we observe that:

—Nine out of 12 languages leverage on *flow-based paradigms* and *visual notations*, although there are a variety of other notations and paradigms providing for the same or similar composition features (some of which are referenced in this work). This underlines the empirical affirmation and suitability of visual abstractions to represent composition concerns and express composition logic.
—The languages that target professional programmers support a much richer set of *control flow constructs* than platforms targeting end users, which typically support only sequence and exclusive choice constructs. While this finding is rather expected, it also manifests an important need for research on intuitive, effective abstractions and diagramming languages that can be mastered not only by professionals.
—*Crosscutting concerns* are not addressed or addressed only vaguely in research platforms. Commercial platforms, instead, do cover crosscutting concerns, as their use in commercial production environments simply demands for solutions addressing issues such as security or quality.

[28] http://www.intalio.com/products/bpms/overview/.
[29] http://pipes.yahoo.com/.
[30] http://www.jbpm.org/.

Table I. The *Language* and *Target User* Dimensions of the Selected Platforms

			Language				
				Composition Constructs			
	Component	Target App.	Notation and Paradigm	Control Flow	Dataflow and Data Transf.	Crosscutting Concerns	Target User
eFlow	Composes application logic. XML data format. Business protocol interaction. Selection at runtime	Business processes	Visual notation, flow diagram. Flow-based paradigm	Simple control-flow patterns (sequence and parallel)	Dataflow defined by mapping functions. Data transformation capabilities not defined	Security rules (authorization)	Professional programmers (Service developers)
FormSys	Composes application logic. Proprietary service description on top of WSDL. XML format. SOAP interaction protocol. Push interaction. Selection at design time	Business processes	Visual, based on forms. Flow-based	Simple control-flow patterns (sequence and exclusive choice)	Complex dataflow constructs, and ad-hoc transformation capabilities	Not addressed	End-user programmers (Domain experts)
Intalio BPMS	Composes application logic. JSON and XML formats. SOAP and REST protocols. Business protocol interaction. Design time selection	Business processes	Visual notation, flow diagrams (BPMN modeler). Flow-based paradigm	Complex control-flow constructs	Data mapper with predefined functions. Allows the use of transformation languages	Exceptions, transactions, authentication	Professional programmers
Jopera	Composes application logic and data. RSS, JSON and XML formats. SOAP and REST protocols. Business protocol interaction. Design time selection	Business processes	Visual notation, flow diagrams. Flow-based paradigm	Complex control-flow constructs	No specific dataflow constructs. Data transformation via transformation languages	Exceptions	Professional programmers

(Continued)

Table I. Continued

			Language				
	Component	Target App.	Notation and Paradigm	Composition Constructs: Control Flow	Composition Constructs: Dataflow and Data Transf.	Crosscutting Concerns	Target User
Self-Serv	Composes application logic. XML format. SOAP protocol. Business protocol interaction. Runtime selection	Business processes	Visual notation, state charts. Flow-based paradigm	Complex control flow constructs (sequence, choice, repeat, and parallel)	Dataflow through variable assignments and arithmetic expressions	Not addressed	Professional programmers
SHOP2	Composes application logic. OWL-S description. XML format. SOAP protocol. Pull interaction. Runtime selection	Business processes	Textual notation	Simple control flow patterns (sequence and exclusive choice)	No specific dataflow constructs	Not addressed	Professional programmers
Sword	Composes application logic. Proprietary services defined in an entity relationship based model: Protocol not specified. Pull interaction. Semi-automatic selection at design time	Business processes	Textual notation. Rule-based paradigm	No specific control-flow constructs	No specific dataflow constructs	Not addressed	Professional programmers
Taverna	Composes data. Java, JSON, and XML formats. Supports SOAP and REST protocol. Push interactions. Design time selection	Scientific workflows	Visual notation, flow diagrams. Flow-based paradigm	Simple control flow (sequence and exclusive choice)	Dataflow for data exchange, prebuilt processors for data processing. Data transformation via transformation languages	Exceptions (retry and alternative task)	End-user programmers (Domain experts)

(Continued)

Table I. Continued

			Language				
		Target	Notation and	Composition Constructs		Crosscutting	
	Component	App.	Paradigm	Control Flow	Dataflow and Data Transf.	Concerns	Target User
XL	Composes application logic and data. XML format. Supports SOAP protocol. Business protocol interactions. Design time selection	Business processes	Textual XML-based notation. Query-based paradigm	Complex control flow constructs (sequence, choice, loops, and parallel)	Dataflow constructs for dataflow dependencies. Data transformation via transformation languages	Exceptions	Professional programmers
Yahoo! Pipes	Composes data. RSS and JSON formats. REST interaction protocol. Pull interactions. Design time selection	Mashups	Visual notation, flow diagrams. Flow-based paradigm	No control flow constructs	Complex dataflow constructs provided by pre-defined modules	Not addressed	End-user programmers
YAWL	Composes application logic & data. XML format. Proprietary interaction protocol HTTP-based. Business protocol interactions. Design time selection	Business processes	Visual notation, Petri-nets. Flow-based paradigm	Complex control flow constructs	Dataflow and data transformation based on XML-transformation languages	Not addressed	Professional programmers
jBPM	Composes application logic. XML, JSON and Java objects. SOAP, REST and OSGi protocols. Business protocol interactions. Design time selection	Business processes	Visual notation, BPMN. Flow-based paradigm	Complex control flow constructs	Dataflow based on a data mapper. Data transformation via scripts	Exceptions, transactions, task confidenciality	Professional programmers

—Service composition is still a prerogative of *professional programmers*. Despite the adoption of intuitive visual abstractions, composing services still requires specialized development expertise and software engineering knowledge, both skills end users lack. Evidently, only few approaches succeed in simplifying the actual composition problem (e.g., data passing, business protocols, correlation, and similar), which thus remains complex.

We identify the following issues as future directions for composition languages:

—*End user service composition.* A commonly overlooked limitation of current systems is that they do not make composition languages accessible to end users (also called knowledge workers). Even sophisticated professional programmers and system administrators are regularly forced to resort to understanding different low-level service APIs, and procedural programming constructs, to create and maintain composite services. End users often need to access, manipulate, integrate, and analyze data from various sources and should, like professional programmers, also be able to benefit from the power of the service-oriented programming paradigm. We believe that service composition languages should enable end users to easily and declaratively specify some simple yet powerful composition scripts, for example, visual language that allow data analysts to drag and drop pre-built data access and analyze services, compose them using sequence and conditional flows [Weber et al. 2013].

—*Federated cloud resources orchestration.* Web services are now the glue of cloud services, and their interactions are binding resources and operations, providing an abstraction layer that shifts the focus from infrastructure and operations to available cloud services and application deployment. Overall, existing cloud services orchestration techniques typically rely on procedural programming in general-purpose or scripting languages [Papazoglou and van den Heuvel 2011]. They follow a bottom-up (or pull) provider-centric approach in which consumers are forced to create and manage complex cloud resource configurations using low-level and heterogeneous APIs. This leads to an inflexible and costly environment, which adds considerable complexity, demands extensive programming effort, requires multiple and continuous patches, and perpetuates closed cloud solutions. These difficulties have led to early solutions focused on providing unified interfaces over heterogeneous cloud provider APIs (e.g. Apache Deltacloud,[31] Apache Libcloud,[32] jclouds,[33] OpenStack[34]). Nevertheless, federated cloud services should be dynamically orchestrated in accordance with high-level policies specified by administrators on behalf of cloud resource consumers. Existing service composition techniques (e.g., the Web Service Business Process Execution Language (BPEL) and Business Process Modeling Notation (BPMN)) focus primarily on the application layer. However, orchestrating cloud resources requires rich abstractions to reason about application resource requirements and constraints, support exception handling, flexible and efficient scheduling of resources. The extension of service composition and orchestration techniques to provide effective federated cloud resource orchestration coping with large-scale heterogeneous cloud environments will become increasingly important.

10.2. Knowledge Reuse and Automation

Table II maps the selected platforms onto the taxonomy of knowledge reuse and automation aspects described in Sections 5 and 6.

[31]http://deltacloud.apache.org.
[32]http://libcloud.apache.org.
[33]www.jclouds.org.
[34]http://www.openstack.org.

Table II. The *Knowledge Reuse* and *Automation* Dimensions of the Selected Platforms

	Knowledge Reuse		
	Reused Artifact	Reuse Technique	Automation
eFlow	Components, examples	Not addressed	Dynamic binding of nodes with concrete services
FormSys	Components, examples, and data transformation rules	Keyword search. Wiki of data transformation functions	Not addressed
Intalio BPMS	Components, data transformation rules, and examples	Keyword search, copy/paste, forum	Not addressed
JOpera	Components and examples	Keyword search, copy/paste, forum	Model-driven composition
Self-Serv	Components, examples	Keyword search. UDDI Registry	Service containers are bound with concrete services at run time
SHOP2	Components	Not specified	Semantic-based composition using HTN planning
Sword	Components	Not specified	Semantic-based composition using a rule-based planner
Taverna	Components, examples, and fragments through their encapsulation as components	Keyword search, copy/paste, repository and forum	Not addressed
XL	Components	Not specified	Not addressed
Yahoo! Pipes	Components and examples	Keyword search, cloning, repository and blog	Not addressed
YAWL	Components, examples	Repository	Not addressed
jBPM	Components, data transformation rules, and examples	Keyword search, copy/paste, forum	Not addressed

From the characteristics of the platforms selected regarding the knowledge reuse and automation dimensions, we identify the following points:

—Surprisingly, the value of *reuse* is still largely underestimated. Most platforms only focus on the core artifact (i.e., components), and only very few provide support for more complex ones, such as data transformation rules or fragments/patterns, which instead would represent a significant help to developers. Only on Taverna provides for the reuse of fragments.
—*Keyword search* is the most prominent search method supported. The reason for this is very likely twofold. On the one hand, the platforms support reuse only of artifacts that do not require the application of complex techniques, such as components and examples; on the other hand, keyword search is simply easy to implement.
—*Model-driven development* is the most prominently adopted automation technique, in line with the observation that most platforms adopt visual modeling languages. Support for more advanced automation is approached only by few platforms and only partially addressed in some others. This is partly due to the complexity of providing effective, user-friendly automation approaches (e.g., for the reuse of fragments), and it is also partly due to the fact that some approaches never matured from research prototypes into commercial products (e.g., semantics-based composition).

We see the evolution of the work in knowledge reuse following a prevailing direction, that of *Composition Knowledge Graphs* (CKGs). Conceptually, this is similar to work already done in query languages in databases, leading to a unified representation, manipulation, and reuse of composition knowledge and thereby enabling simplified

Table III. The *Tool Support* and *Execution Platform* Dimensions of the Selected Platforms

	Tool Support	Execution Platform: Deployment Options	Execution Platform: Execution Engine
eFlow	N/A	On premisses	Business process engine
FormSys	Manuals	On premisses	Business process engine
Intalio BPMS	Versioning, manuals, and tutorials	On premisses and on cloud	Business process engine
Jopera	Refactoring, versioning, manuals, tutorials	On premisses	Business process engine
Self-Serv	N/A	On premisses	Business process engine
SHOP2	N/A	On premisses	Business process engine
Sword	N/A	On premisses	Business process engine
Taverna	Versioning, manuals, tutorials, and FAQ	On premisses and on cloud	Business process engine
XL	N/A	N/A	N/A
Yahoo! Pipes	Manuals, tutorials, and FAQ	On cloud deployment	Code generation
YAWL	Manuals, tutorials, and FAQ	On premisses	Business process engine
jBPM	Manuals, tutorials, FAQ, and refactoring through Eclipse tools	On premisses and on cloud	Business process engine

and productive service-enabled composition and customization. Central to this is the concept of CKG, where common services integration related low-level logic can be abstracted, organized, incrementally shared, and thereby re-used by developers. The type of knowledge captured could be organized according to various dimensions including: APIs, Resources, Events, and Tasks. By identifying entities (i.e. types/attributes, relationships for each dimension, and their specialization), novel foundations will be introduced to accumulate current dispersed composition knowledge in a structured framework.

10.3. Tool Support and Execution Platform

Table III maps the selected platforms onto the taxonomy of tool support and execution platform aspects described in Sections 7 and 8, respectively.

From the data presented in Table III, we note the following implications:

—The *tool support* regarding software engineering activities for service composition is still quite poor. This is remarkable, as the development of composite services is not a new discipline; therefore, it would be expected that software engineering techniques had been applied broadly to it to support developers in their work.

—We note a different trend regarding the deployment options: although cloud computing is a relatively new field, the *deployment on cloud* is a feature that has been rapidly adopted by service composition platforms, especially by those that appeared recently and, of course, are still active.

—The preferred choice for the execution of composite services are *business process engines*. These are the powerful instruments that provide much more than just the execution of composite services. Typical built-in features comprise support for multiple composite services in parallel, administration dashboards, runtime monitoring, progression logging, and similar. These benefits by large outweigh the efficiency advantage of compiled compositions.

We envision two different but complementary future directions: *high-level abstractions* and *composition middleware intelligence*. Although the proliferation of assembling applications from cloud-based APIs will increase our ability to increase development productivity, there are significant shortfalls in seamlessly integrating composition languages and tools with scalable data processing platforms such as Hadoop to scale the provisioning of data-intensive services (e.g., process analytics pipelines). The composition layer should contain the intelligence responsible for specifying service interactions, while the data processing layer should contain the intelligence responsible for dataflow and processing leveraging platforms such as Hadoop. This will enable developers to specify application requirements and constraints using high-level and composition-aware abstractions. Composition middleware will automatically translate these abstractions into the efficient and platform-aware execution scripts.

11. CONCLUSION AND OUTLOOK

Web services and Web service composition are a powerful technology that has the potential to transform applications, hardware, and software resources into standardized, reusable, and dynamically integrated software components. In this comprehensive survey, we studied a great variety of service composition languages, techniques, and tools. We proposed a taxonomy that consists of the dimensions that characterize and compare service composition approaches. We provided a systematic analysis of the most representative service composition approaches by evaluating and classifying them against the proposed taxonomy. While Web services are now firmly recognized as engines of online, service-enabled business transformation and major advancements in composition technology have been made, there are still crucial gaps in the service composition endeavor.

We conclude this survey by identifying two additional key open research issues in service composition technology: social/crowd computing support and engineering of composite services.

—*Social/crowd computing support.* Increasingly, composite applications also leverage on human computations [Quinn and Bederson 2011], next to machine computations made available through Web services. For instance, social networks or crowdsourcing enable access to vast user bases, which can be leveraged on for performing tasks that the machine is not able to perform as good as humans do (e.g., raking a set of photos) or that it cannot do at all (e.g., providing an opinion on a given topic). Yet, these tasks are more and more integrated into modern applications and represent a real resource for innovative businesses. Integrating them, however, is not yet as straightforward as it could be. Web services technology, as of today, exclusively focuses on machine computations only and does not take into account the specific needs that emerge when humans are involved in applications. Hence, much more needs to be done in order to conciliate the needs of humans with those of machines and to enable a seamless integration of both worlds. For instance, human computations are characterized by non-determinism (two different runs of a task or process generally lead to different results), high uncertainty (crowd workers, for instance, oftentimes cheat), the need for suitable quality control mechanisms beyond common service level agreements (e.g., assigning a given task to multiple workers may allow one to reduce the effect of noise or cheating), and, finally, the need for extended coordination approaches able to bring together human and machine computations. All these aspects may require a re-thinking and extension of today's Web services abstractions and technology.

—*Engineering composite services.* Current systems are rarely transparent and adaptive. Designers and developers deal with heterogeneous and autonomous components, with different characteristics describing various and complex dimensions (e.g, functional properties, QoS, policies, resource requirements), often using several semantically unrelated notations. This leads to fragmentation of modeling, analysis, and reasoning, and consequently breaks the maxim of sound, continuous, incremental, and end-to-end design and engineering. Versioning, refactoring, and limited reuse are the only software engineering activities that have more than a marginal presence in service composition platforms. As we reported earlier, we did not find significant evidence of other software engineering activities applied to service composition. There is a need for further research into unified methods, models, and tools to design, test, effectively reuse and build consistent, highly available, robust, reusable, customizable, and composable services, building upon lessons from other disciplines, and focusing on the unique challenges in service composition design and engineering [Sifakis 2011]. This is essential for faster delivery and the sound, scalable engineering of service-based systems. It confers the advantages of productivity, continuity, adaptivity, and correctness.

REFERENCES

Alejandro Abdelnur and Stefan Hepper. 2003. *Java Portlet Specification, Version 1.0*. Technical Report JSR 168. Sun Microsystems, Inc.

Serge Abiteboul, Omar Benjelloun, Ioana Manolescu, Tova Milo, and Roger Weber. 2004. Active XML: A data-centric perspective on Web services. In *Web Dynamics*. Springer, 275–299.

Mohamed Abouelhoda, Shadi A. Issa, and Moustafa Ghanem. 2012. Tavaxy: Integrating Taverna and Galaxy workflows with cloud computing support. *BMC Bioinformatics* 13, 1 (2012), 77.

Saeed Aghaee and Cesare Pautasso. 2012. EnglishMash: Usability design for a natural mashup Composition environment. In *ICWE Workshops*. 109–120.

Rama Akkiraju, Joel Farrell, John A. Miller, Meenakshi Nagarajan, Amit Sheth, and Kunal Verma. 2005. *Web Service Semantics - WSDL-S*. W3c member submission. Technical Note, Version 1.0. Retrieved from http://lsdis.cs.uga.edu/library/download/WSDL-S-V1.pdf.

A. M. Alashqur, Stanley Y. W. Su, and Herman Lam. 1989. OQL: A query language for manipulating object-oriented databases. In *VLDB*. 433–442.

Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. 2004a. *Web Services: Concepts, Architectures and Application*. Springer-Verlag.

Gustavo Alonso, Cesare Pautasso, and Biörn Biörnstad. 2004b. *CS Adaptability Container*. Technical Report. Information Society Technology.

T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, et al. 2003. Business Pocess Execution Language for Web Services (BPEL4WS). Specification, Version 1.1. BEA Systems, IBM Corp., Microsoft Corp., SAP AG, Siebel Systems. Retrieved from https://msdn.microsoft.com/en-us/library/ee251594%28v=bts.10%29.aspx.

Alain Andrieux, Karl Czajkowski, Asit Dan, Kate Keahey, Heiko Ludwig, Toshiyuki Nakata, Jim Pruyne, John Rofrano, Steve Tuecke, and Ming Xu. 2004. Web services agreement specification (WS-Agreement). In *Global Grid Forum*, Vol. 2.

Anupriya Ankolekar, Mark Burstein, Jerry R. Hobbs, Ora Lassila, David Martin, Drew McDermott, Sheila A. McIlraith, Srini Narayanan, Massimo Paolucci, Terry Payne, et al. 2002. DAML-S: Web service description for the semantic web. In *The Semantic Web (ISWC'02)*. Springer, 348–363.

Assaf Arkin, Sid Askary, Scott Fordin, Wolfgang Jekeli, Kohsuke Kawaguchi, David Orchard, Stefano Pogliani, Karsten Riemer, et al. 2002. Web service choreography interface (WSCI) 1.0. Standards proposal by BEA Systems, Intalio, SAP, and Sun Microsystems. Retrieved from http://www.w3.org/TR/wsci-link.

Atipol Asavametha, Prashanth Ayyavu, and Christopher Scaffidi. 2011. No application is an island: Using topes to transform strings during data transfer. In *ICISA*. 1–10.

Iman Avazpour, John Grundy, and Lars Grunske. 2013. Tool support for automatic model transformation specification using concrete visualisations. In *ASE*. 718–721.

Ahmed Awad. 2007. BPMN-Q: A language to query business processes. In *EMISA*, Vol. 119. 115–128.

Daniel Bachlechner, Katharina Siorpaes, Dieter Fensel, and Ioan Toma. 2006. Web service discovery-a reality check. In *ESWC*, Vol. 308.

Jianbo Bai, Hong Xiao, Xianghua Yang, and Guofang Zhang. 2009. Study on integration technologies of building automation systems based on web services. In *CCCM 2009*, Vol. 4. 262–266.

Karim Baina, Boualem Benatallah, Fabio Casati, and Farouk Toumani. 2004. Model-driven web service development. In *CAISE*. 290–306.

Luciano Baresi, Sam Guinea, and Liliana Pasquale. 2007. Self-healing BPEL processes with Dynamo and the JBoss rule engine. In *ESSPE Workshop*. 11–20.

Adam Barker and Jano Van Hemert. 2008. Scientific workflow: a survey and research directions. In *Parallel Processing and Applied Mathematics*. Springer, 746–753.

Peter Bartalos and Mária Bieliková. 2011. Automatic dynamic web service composition: A survey and problem formalization. *Computing and Informatics* 30, 4, 793–827.

Catriel Beeri, Anat Eyal, Simon Kamenkovich, and Tova Milo. 2008. Querying business processes with BP-QL. *Information Systems* 33, 6 (2008), 477–507.

Oleg Beletski. 2008. End user mashup programming environments. In *T-111.5550 Seminar on Multimedia*.

T. Bellwood, L. Clément, D. Ehnebuske, A. Hately, M. Hondo, Y. L. Husband, K. Januszewski, S. Lee, B. McKee, et al. 2002. The Universal Description, Discovery and Integration (UDDI) Specification. Oasis, 5 (2002), 16–18. Retrieved from http://uddi.org/pubs/uddi-v3.00-published-20020719.htm.

Victoria Beltran, Knarig Arabshian, and Henning Schulzrinne. 2012. Ontology-based user-defined rules and context-aware service composition system. In *The Semantic Web: ESWC 2011 Workshops*. Springer, 139–155.

Boualem Benatallah, Fabio Casati, and Farouk Toumani. 2004. Web service conversation modeling: A cornerstone for e-business automation. *IEEE Internet Computing* 8, 1 (2004), 46–54.

Boualem Benatallah, Quan Z. Sheng, and Marlon Dumas. 2003. The self-serv environment for web services composition. *IEEE Internet Computing* 7, 1 (2003), 40–48.

Djamal Benslimane, Schahram Dustdar, and Amit Sheth. 2008. Services mashups: The new generation of web applications. *IEEE Internet Computing* 12, 5 (2008), 13–15.

Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Richard Hull, and Massimo Mecella. 2005. Automatic composition of transition-based semantic web services with messaging. In *VLDB*. 613–624.

Daniela Berardi, Fahima Cheikh, Giuseppe De Giacomo, and Fabio Patrizi. 2008. Automatic service composition via simulation. *International Journal of Foundations of Computer Science* 19, 2 (2008), 429–451.

Andreas Berl, Erol Gelenbe, Marco Di Girolamo, Giovanni Giuliani, Hermann De Meer, Minh Quan Dang, and Kostas Pentikousis. 2010. Energy-efficient cloud computing. *Computer Journal* 53, 7 (2010), 1045–1051.

Philip A. Bernstein and Sergey Melnik. 2007. Model management 2.0: Manipulating richer mappings. In *SIGMOD*. 1–12.

Philip A. Bernstein and Eric Newcomer. 2009. *Principles of Transaction Processing*. Morgan Kaufmann.

Christian Bizer, Tom Heath, and Tim Berners-Lee. 2009. Linked data-the story so far. *International Journal on Semantic Web and Information Systems (IJSWIS)* 5, 3 (2009), 1–22.

S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. 2002. XQuery 1.0: An XML query language. World Wide Web Consortium. *W3C Working Draft* 15 (2002). Retrieved from http://www.w3.org/TR/xquery.

Vinayak Borkar, Michael Carey, Daniel Engovatov, Dmitry Lychagin, and others. 2008. XQSE: An XQuery scripting extension for the AquaLogic data services platform. In *ICDE*. 1229–1238.

Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. 2000. Simple Object Access Protocol (SOAP) 1.1. World Wide Web Consortium note (2000). Retrieved from http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

John Boyer, Sandy Gao, Susan Malaika, Michael Maximilien, Rich Salz, and Jerome Simeon. 2011. Experiences with JSON and XML transformations. In *W3C Workshop on Data and Services Integration*.

Engin Bozdag, Ali Mesbah, and Arie Van Deursen. 2007. A comparison of push and pull techniques for ajax. In *9th IEEE International Workshop onWeb Site Evolution (WSE'07)*. IEEE, 15–22.

Mathieu Braem, Niels Joncheere, Wim Vanderperren, Ragnhild Van Der Straeten, and Viviane Jonckers. 2006. Guiding service composition in a visual service creation environment. In *4th European Conference on Web Services (ECOWS'06)*. IEEE, 13–22.

Tim Bray, Jean Paoli, C. Michael Sperberg-McQueen, Eve Maler, and François Yergeau. 1997. Extensible markup language (XML). *World Wide Web Journal* 2, 4 (1997), 27–66.

Jeppe Brønsted, Klaus Marius Hansen, and Mads Ingstrup. 2007. A survey of service composition mechanisms in ubiquitous computing. In *UbiComp Workshops*. 87–92.

Bruce G. Buchanan and Edward H. Shortliffe. 1984. *Rule Based Expert Systems: The Mycin Experiments of the Stanford Heuristic Programming Project*. Addison-Wesley.

Margaret M. Burnett. 1999. Visual programming. In *Wiley Encyclopedia of Electrical and Electronics Engineering*.

Felipe Cabrera, George Copeland, Bill Cox, Tom Freund, Johannes Klein, Tony Storey, and Satish Thatte. 2002. Web services transaction (WS-transaction). BEA, IBM and Microsoft. Technical Report (2002). Retrieved from http://xml.coverpages.org/WS-Transaction2002.pdf.

Felipe Cabrera, George Copeland, Tom Freund, Johannes Klein, David Langworthy, David Orchard, John Shewchuk, and Tony Storey. 2004. Web services coordination (WS-Coordination). Specification by BEA, IBM, and Microsoft (2004). Retrieved from http://xml.coverpages.org/WS-Coordination200411.pdf.

Marcos Caceres. 2012. Packaged Web Apps (Widgets) - Packaging and XML Configuration (2nd ed.). *W3C Recommendation.* Retrieved from http://www.w3.org/TR/widgets/.

Paloma Caceres, Esperanza Marcos, and Belen Vela. 2003. A MDA-based approach for web information system development. In *Workshop in Software Model Engineering*.

Steven P. Callahan, Juliana Freire, Emanuele Santos, Carlos E. Scheidegger, Claudio T. Silva, and Huy T. Vo. 2006. VisTrails: Visualization meets data management. In *SIGMOD*. 745–747.

Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, Massimo Mecella, and Fabio Patrizi. 2008. Automatic service composition and synthesis: The Roman model. *IEEE Data Engineering Bulletin* 31, 3 (2008), 18–22.

Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. 2005. An approach for QoS-aware service composition based on genetic algorithms. In *GECCO*. 1069–1075.

Michael Carey. 2006. Data delivery in a service-oriented world: The BEA aquaLogic data services platform. In *SIGMOD*. 695–705.

Fabio Casati, Ski Ilnicki, LiJie Jin, Vasudev Krishnamoorthy, and Ming-Chien Shan. 2000. Adaptive and dynamic service composition in eFlow. In *CAISE*. 13–31.

Stefano Ceri, Piero Fraternali, Aldo Bongio, Marco Brambilla, Sara Comai, and Maristella Matera. 2002. *Designing Data-Intensive Web Applications*. Morgan Kauffmann.

Donald D. Chamberlin and Raymond F. Boyce. 1974. SEQUEL: A structured English query language. In *SIGFIDET (now SIGMOD) Workshops*. 249–264.

K. S. May Chan, Judith Bishop, Johan Steyn, Luciano Baresi, and Sam Guinea. 2009. A fault taxonomy for web service composition. In *ICSOC Workshops*. 363–375.

James F. Chang. 2006. *Business Process Management Systems: Strategy and Implementation*. Auerbach Publications.

Anis Charfi and Mira Mezini. 2004. Aspect-oriented web service composition with AO4BPEL. In *Web Services*. Springer, 168–182.

Xi Chen, Angel Lagares Lemos, Moshe Chai Barukh, and Boualem Benatallah. 2012. Service graph base: A unified graph-based platform for representing and manipulating service artifacts. In *SOCA*. 1–8.

Mark Chignell, James Cordy, Joanna Ng, and Yelena Yesha. 2010. *The Smart Internet: Current Research and Future Applications*. Vol. 6400. Springer.

Injun Choi, Kwangmyeong Kim, and Mookyung Jang. 2007. An XML-based process repository and process query language for integrated process management. *Knowledge and Process Management* 14, 4 (2007), 303–316.

Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, et al. 2001. Web services description language (WSDL) 1.1. W3C Note. (March 2001). Retrieved from http://www.w3.org/TR/wsdl.

J. Clark and others. 1999. XSL transformations (XSLT) version 1.0. *W3C Recommendation* (1999). Retrieved from http://www.w3.org/TR/xslt.

Marcel Cremene, J.-Y. Tigli, Stéphane Lavirotte, F.-C. Pop, Michel Riveill, and Gaëtan Rey. 2009. Service composition based on natural language requests. In *SCC*. 486–489.

Douglas Crockford. 2006. The application/JSON media type for JavaScript Object Notation (JSON), 2006. *IETF Tools.* Retrieved from http://www.ietf.org/rfc/rfc4627.txt.

Gianpaolo Cugola, Carlo Ghezzi, and Leandro Sales Pinto. 2012. DSOL: A declarative approach to self-adaptive service orchestrations. *Computing* 94, 7 (2012), 579–617.

Jose Danado and Fabio Paternò. 2012. Puzzle: A visual-based environment for end user development in touch-based mobile phones. In *Human-Centered Software Engineering*. Springer, 199–216.

F. Daniel, F. Casati, B. Benatallah, and M. C. Shan. 2009a. Hosted universal composition: Models, languages and infrastructure in mashArt. In *Conceptual Modeling – ER*. 428–443.

Florian Daniel, Fabio Casati, Vincenzo D'Andrea, Emmanuel Mulo, Uwe Zdun, Schahram Dustdar, Steve Strauch, David Schumm, Frank Leymann, Samir Sebahi, et al. 2009b. Business compliance governance in service-oriented architectures. In *AINA*. 113–120.

Florian Daniel, Maristella Matera, Jin Yu, Boualem Benatallah, Regis Saint-Paul, and Fabio Casati. 2007. Understanding UI integration: A survey of problems, technologies, and opportunities. *IEEE Internet Computing* 11, 3 (2007), 59–66.

Florian Daniel and Barbara Pernici. 2006. Insights into web service orchestration and choreography. *International Journal of E-Business Research (IJEBR)* 2, 1 (2006), 58–77.

Jos De Bruijn, Christoph Bussler, John Domingue, Dieter Fensel, Martin Hepp, M. Kifer, B. König-Ries, J. Kopecky, R. Lara, E. Oren, et al. 2005. Web service modeling ontology (WSMO). W3C Member Submission (2005). Retrieved from http://www.w3.org/Submission/WSMO/.

Valeria De Castro, Esperanza Marcos, and Marcos Lopez Sanz. 2006. A model driven method for service composition modelling: a case study. *International Journal of Web Engineering and Technology* 2, 4 (2006), 335–353.

Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, et al. 2005. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming* 13, 3 (2005), 219–237.

R. P. Diaz Redondo, A. Fernández Vilas, M. Ramos Cabrer, and J. J. Pazos Arias. 2007. Enhancing residential gateways: OSGi service composition. *IEEE Transaction on Consumer Electronics* 53, 1 (2007), 87–95.

Tim Dierks. 2008. *The Transport Layer Security (TLS) Protocol Version 1.2*. *Internet Engineering Task Force.* (2008).

Schahram Dustdar and Wolfgang Schreiner. 2005. A survey on web services composition. *International Journal of Web and Grid Services* 1, 1 (2005), 1–30.

ebXML Registry Technical Committee. 2002. *ebXML Registry Services Specification v2. 0*. Technical Report. OASIS. Retrieved from http://www.oasis-open.org/committees/regrep/documents/2.0/specs/ebrs.pdf.

M. Edwards. 2011. Service Component Architecture (SCA). Retrieved from http://www.oasis-opencsa.org/sca.

Mohamad Eid, Atif Alamri, and Abdulmotaleb El Saddik. 2008. A reference model for dynamic web service composition systems. *International Journal of Web and Grid Services* 4, 2 (2008), 149–168.

Hazem Elmeleegy, Anca Ivan, Rama Akkiraju, and Richard Goodwin. 2008. Mashup advisor: A recommendation tool for mashup development. In *ICWS*. 337–344.

Rob Ennals and David Gay. 2007. User-friendly functional programming for web mashups. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 223–234.

Kutluhan Erol, James Hendler, and Dana S. Nau. 1994. HTN planning: Complexity and expressivity. In *AAAI*, Vol. 94. 1123–1128.

M. Facemire, J. S. Hammond, C. Mines, and E. Wheeler. 2014. *Predictions 2015: Mobile Development Goes Composable, Contextual, and Cross-Touchpoint*. Technical Report. Forrester Inc.

Dieter Fensel and Christoph Bussler. 2002. The web service modeling framework WSMF. *Electronic Commerce Research and Applications* 1, 2 (2002), 113–137.

José Luiz Fiadeiro, Antónia Lopes, and Laura Bocchi. 2007. Algebraic semantics of service component modules. In *Recent Trends in Algebraic Development Techniques*. Springer, 37–55.

Roy Thomas Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. Dissertation. University of California, Irvine.

Klaus Finkenzeller. 2003. *Data Integrity*. Wiley Online Library.

Daniela Florescu, Andreas Grünhagen, and Donald Kossmann. 2003. XL: An XML programming language for web service specification and composition. *Computer Networks* 42, 5 (2003), 641–660.

Daniela Florescu, Andreas Grünhagen, Donald Kossmann, and Steffen Rost. 2002. XL: A platform for Web Services. In *SIGMOD*. 625–625.

Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.

Tracy Gardner. 2003. UML Modelling of Automated Business Processes with a Mapping to BPEL4WS. In *ECOOP* (2003), 30–34.

John Garofalakis, Yannis Panagis, Evangelos Sakkopoulos, and Athanasios Tsakalidis. 2004. Web service discovery mechanisms: Looking for a needle in a haystack. In *International Workshop on Web Engineering*.

Gartner. 2013. Top 10 Strategic Technology Trends for 2014. Retrieved from http://www.gartner.com/newsroom/id/2603623.

Kristof Geebelen, Sam Michiels, and Wouter Joosen. 2008. Dynamic reconfiguration using template based web service composition. In *MW4SOC Workshop*. 49–54.

Carole A. Goble, Jiten Bhagat, Sergejs Aleksejevs, Don Cruickshank, Danius Michaelides, David Newman, et al. 2010. myExperiment: A repository and social network for the sharing of bioinformatics workflows. *Nucleic Acids Sesearch* 38, suppl 2 (2010), W677–W682.

Jeremy Goecks, Anton Nekrutenko, James Taylor, T. Galaxy Team, et al. 2010. Galaxy: A comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biology* 11, 8 (2010), R86.

Nicolas Gold, Andrew Mohan, Claire Knight, and Malcolm Munro. 2004. Understanding service-oriented software. *IEEE Software* 21, 2 (2004), 71–77.

Lars Grammel and Margaret-Anne Storey. 2010. A survey of mashup development environments. In *The Smart Internet*. Springer, 137–151.

Ohad Greenshpan, Tova Milo, and Neoklis Polyzotis. 2009. Autocompletion for Mashups. *Proceedings of the VLDB Endowment* 2, 1 (Aug. 2009), 538–549.

Roy Grønmo and Ida Solheim. 2004. Towards modeling web service composition in UML. *WSMAI* 4 (2004), 72–86.

BPMI Notation Working Group. 2004. Business Process Modeling Notation (BPMN) Version 1.0. Retrieved from http://www.omg.org/bpmn/Documents/BPMN_V1-0_May_3_2004.pdf.

D. Gruber, B. J. Hargrave, Jeff McAffer, Pascal Rapicault, and Thomas Watson. 2005. The Eclipse 3.0 platform: Adopting OSGi technology. *IBM Systems Journal* 44, 2 (2005), 289–299.

J. Octavio Gutierrez-Garcia and Felix F. Ramos-Corchado. 2011. Exception handling in pervasive service composition using normative agents. *Journal of Web Engineering* 10, 3 (2011), 175–196.

Marc J. Hadley. 2006. Web application description language (WADL). Technical Report TR-2006-153. Sun Microsystems. Retrieved from https://wadl.java.net/wadl20061109.pdf.

Rachid Hamadi and Boualem Benatallah. 2003. A Petri net-based model for web service composition. In *ADC*, Vol. 17. 191–200.

Rachid Hamadi, Boualem Benatallah, and Brahim Medjahed. 2008. Self-adapting recovery nets for policy-driven exception handling in business processes. *Distributed and Parallel Databases* 23, 1 (2008), 1–44.

Dick Hardt. 2012. *The OAuth 2.0 Authorization Framework*. Technical Report. RFC 6749, IETF.

Ramy Ragab Hassen, Lhouari Nourine, and Farouk Toumani. 2008. Protocol-based web service composition. In *ICSOC*. 38–53.

J. B. Hill, B. J. Lheureux, E. Olding, D. C. Plummer, B. Rosser, and J. Sinur. 2010. *Predicts 2010: Business Process Management Will Expand Beyond Traditional Boundaries*. Technical Report. Gartner.

Dion Hinchcliffe and Jim Benson. 2011. *EMML Changes Everything: Profitability, Predictability, & Performance through Enterprise Mashups*. Technical Report. Open Mashup Alliance.

David Hollingsworth. 1995. The workflow reference model. Specification. Document Number TC00-1003 Version 1.1. The Workflow Management Coalition. Retrieved from ftp://www.ufv.br/dpi/mestrado/Wkflow-BPM/The%20Workflow%20Reference%20Model.pdf.

Paul Hudak. 1989. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR)* 21, 3 (1989), 359–411.

Duncan Hull, Katy Wolstencroft, Robert Stevens, et al. 2006. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research* 34 (2006), 729–732.

Takeshi Imamura, Blair Dillaway, Edi Simon, et al. 2002. XML-encryption syntax and processing. W3C Recommendation. *World Wide Web Consortium (W3C)* 10 (2002).

Paul T. Jaeger, Jimmy Lin, and Justin M. Grimes. 2008. Cloud computing and information policy: Computing in a policy cloud? *Journal of Information Technology & Politics* 5, 3 (2008), 269–283.

Hasan Jamil, Aminul Islam, and Shahriyar Hossain. 2010. A declarative language and toolkit for scientific workflow implementation and execution. *International Journal of Business Process Integration and Management* 5, 1 (2010), 3–17.

Meiko Jensen, Nils Gruschka, Ralph Herkenhoner, and Norbert Luttenberger. 2007. Soa and web services: New technologies, new standards-new attacks. In *ECOWS*. 35–44.

Gregor Joeris and Otthein Herzog. 1999. *Managing Evolving Workflow Specifications with Schema Versioning and Migration Rules*. Technical Report. University of Bremen TZI.

Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. 2008. ATL: A model transformation tool. *Science of Computer Programming* 72, 1 (2008), 31–39.

N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, and C. Barreto. 2005. Web services choreography description (WS-CDL) version 1.0. W3C Candidate Recommendation. Retrieved from http://www.w3.org/TR/ws-cdl-10/.

Won Kim. 2009. Cloud computing: Today and tomorrow. *Journal of Object Technology* 8, 1 (2009), 65–72.

David Koop. 2008. VisComplete: Automating suggestions for visualization pipelines. *IEEE Transactions on Visualization and Computer Graphics* 14 (2008), 1691–1698.

Kyriakos Kritikos, Barbara Pernici, Pierluigi Plebani, Cinzia Cappiello, Marco Comuzzi, Salima Benrernou, Ivona Brandic, Attila Kertész, Michael Parkin, and Manuel Carro. 2013. A survey on service quality description. *ACM Computing Surveys (CSUR)* 46, 1 (2013), 1.

Pal Krogdahl, Gottfried Luef, and Christoph Steindl. 2005. Service-oriented agility: An initial analysis for the use of Agile methods for SOA development. In *ICSOC*, Vol. 2. 93–100.

C. Kubczak, T. Margaria, and B. Steffen. 2009. Mashup development for everybody: a planning-based approach. In *Workshop on Service Matchmaking & Resource Retrieval in the Semantic Web (CEUR-WS).*

Angel Lagares Lemos, Moshe Chai Barukh, and Boualem Benatallah. 2013. DataSheets: A spreadsheet-based data-flow language. In *ICSOC*. 616–623.

Ugo Dal Lago, Marco Pistore, and Paolo Traverso. 2002. Planning with a language for extended goals. In *AAAI/IAAI*. 447–454.

D. Davide Lamanna, James Skene, and Wolfgang Emmerich. 2003. SLAng: A language for defining service level agreements. In *FTDCS*. 100–1006.

Sven Lämmermann. 2002. *Runtime Service Composition via Logic-based Program Synthesis*. Ph.D. Dissertation. KTH.

Kung-Kiu Lau and Tauseef Rana. 2010. A taxonomy of software composition mechanisms. In *SEAA*. 102–110.

Neal Leavitt. 2010. Will NoSQL databases live up to their promise? *Computer* 43, 2 (2010), 12–14.

Jonathan Lee, Shin-Jie Lee, and Ping-Feng Wang. 2014. A Framework for Composing SOAP, Non-SOAP and Non-Web Services. *IEEE Transactions on Services Computing* 8, 2 (2014), 240–250.

Philipp Leitner, Anton Michlmayr, Florian Rosenberg, and Schahram Dustdar. 2008. End-to-end versioning support for web services. In *SCC*, Vol. 1. 59–66.

Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. 2008. CoScripter: Automating & sharing how-to knowledge in the enterprise. In *CHI*. 1719–1728.

Hector Levesque, Fiora Pirri, and Ray Reiter. 1998. Foundations for the situation calculus. *Linköping Electronic Articles in Computer and Information Science* 3, 18 (1998), 1–18. Retrieved from http://www.ep.liu.se/ea/cis/1998/018/cis98018.pdf.

Hector J. Levesque, Raymond Reiter, Yves Lesperance, Fangzhen Lin, and Richard B. Scherl. 1997. GOLOG: A logic programming language for dynamic domains. *The Journal of Logic Programming* 31, 1 (1997), 59–83.

Frank Leymann. 2001. Web services flow language (WSFL 1.0). (2001). Whitepaper. IBM Software Group. 621–630. Retrieved from http://xml.coverpages.org/WSFL-Guide-200110.pdf.

Henry Lieberman, Fabio Paternò, Markus Klann, and Volker Wulf. 2006. End-user development: An emerging paradigm. In *End User Development*. Springer, 1–8.

James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A. Lau. 2009. End-user programming of mashups with vegemite. In *IUI*. 97–106.

Yanni Loukissas. 2003. *Rulebuilding: Exploring Design Worlds through End-User Programming*. Ph.D. Dissertation. Massachusetts Institute of Technology.

Bertram Ludäscher, Ilkay Altintas, Chad Berkley, et al. 2006. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience* 18, 10 (2006), 1039–1065.

Anbazhagan Mani and Arun Nagarajan. 2005. Understanding quality of service for Web services. (2005).

Umardand Shripad Manikrao and T. V. Prabhakar. 2005. Dynamic selection of web services with recommendation system. In *NWeSP*.

Ioana Manolescu, Marco Brambilla, Stefano Ceri, Sara Comai, and Piero Fraternali. 2005. Model-driven design and deployment of service-enabled web applications. *ACM TOIT* 5, 3 (Aug. 2005), 439–479.

Ziyan Maraikar, Alexander Lazovik, and Farhad Arbab. 2008. Building mashups for the enterprise with SABRE. In *ICSOC*. 70–83.

Annapaola Marconi and Marco Pistore. 2009. Synthesis and composition of web services. In *Formal Methods for Web Services*. Springer, 89–157.

Ivan Markovic and Alessandro Costa Pereira. 2008. Towards a formal framework for reuse in business process modeling. In *BPM Workshops*. 484–495.

David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, et al. 2004. OWL-S: Semantic markup for web services. *W3C Member Submission* 22 (2004), 2007–04.

David Martin, Massimo Paolucci, Sheila McIlraith, Mark Burstein, Drew McDermott, Deborah McGuinness, Bijan Parsia, Terry Payne, Marta Sabou, et al. 2005. Bringing semantics to web services: The OWL-S approach. In *Semantic Web Services and Web Process Composition*. Springer, 26–42.

Philip Mayer, Andreas Schroeder, and Nora Koch. 2008. MDD4SOA: Model-driven service orchestration. In *EDOC*. 203–212.

Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. 1998. *PDDL: The Planning Domain Definition Language*. Cvc tr-98-003/dcs tr-1165, Yale Center for Computational Vision and Control.

Sheila McIlraith and Tran Cao Son. 2002. Adapting Golog for composition of semantic web services. *KR* 2 (2002), 482–493.

Sheila A. McIlraith, Tran Cao Son, and Honglei Zeng. 2001. Semantic web services. *IEEE Intelligent Systems* 16, 2 (2001), 46–53.

Brahim Medjahed, Athman Bouguettaya, and Ahmed K. Elmagarmid. 2003. Composing web services on the semantic web. *The VLDB Journal* 12, 4 (2003), 333–351.

Peter Mell and Timothy Grance. 2011. The NIST definition of cloud computing (draft). *NIST Special Publication* 800, 145 (2011), 7.

Nikola Milanovic and Miroslaw Malek. 2004. Current solutions for web service composition. *IEEE Internet Computing* 8, 6 (2004), 51–59.

Chilukuri Krishna Mohan. 2000. Rule based programming In *Frontiers of Expert Systems*. Springer, 99–131.

Hamid Reza Motahari Nezhad, Boualem Benatallah, Axel Martens, Francisco Curbera, and Fabio Casati. 2007. Semi-automated adaptation of service interactions. In *WWW*. 993–1002.

Anthony Nadalin, Chris Kaler, Ronald Monzillo, and Phillip Hallam-Baker. 2004. Web services security: SOAP message security 1.0 (WS-Security 2004). *Oasis Standard* 200401 (2004), 1–20010502.

Dana S. Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. 2003. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research (JAIR)* 20 (2003), 379–404.

Roger M. Needham. 1994. Denial of service: An example. *Communications of the ACM* 37, 11 (1994), 42–46.

Mark Nottingham and Robert Sayre. 2005. *The Atom Syndication Format (RFC 4287)*. Technical Report. IETF Working Group.

Željko Obrenović and Dragan Gašević. 2008. End-user service computing: Spreadsheets as a service composition tool. In *ITSC*.

Bart Orriëns, Jian Yang, and Mike Papazoglou. 2003a. Model driven service composition. In *ICSOC*. 75–90.

Bart Orriëns, Jian Yang, and Mike P. Papazoglou. 2003b. A framework for business rule driven service composition. In *Technologies for E-Services*. Springer, 14–27.

OSGi Alliance. 2014. OSGi Service Platform (OSGi Core) - Release 6. (June 2014). Retrieved from https://osgi.org/download/r6/osgi.core-6.0.0.pdf.

Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia Sycara. 2002. Semantic matching of web services capabilities. In *The Semantic Web–ISWC 2002*. Springer, 333–347.

Michael P. Papazoglou. 2003. Web services and business transactions. *World Wide Web* 6, 1 (2003), 49–91.

Michael P. Papazoglou and Willem-Jan van den Heuvel. 2011. Blueprinting the cloud. *IEEE Internet Computing* 15, 6 (2011), 74–79.

Cesare Pautasso. 2005. JOpera: An agile environment for web service composition with visual unit testing and refactoring. In *IEEE Symposium on Visual Languages and Human-Centric Computing*. 311–313.

Cesare Pautasso. 2009a. Composing restful services with JOpera. In *Software Composition*. Springer, 142–159.

Cesare Pautasso. 2009b. RESTful Web service composition with BPEL for REST. *Data & Knowledge Engineering* 68, 9 (2009), 851–866.

Cesare Pautasso and Gustavo Alonso. 2005. The JOpera visual composition language. *Journal of Visual Languages & Computing* 16, 1 (2005), 119–152.

Carlos Pedrinaci, Dong Liu, Maria Maleshkova, David Lambert, Jacek Kopecky, and John Domingue. 2010. iServe: A linked services publishing platform. In *CEUR Workshop Proceedings*, Vol. 596.

Chris Peltz. 2003. Web services orchestration and choreography. *Computer* 36, 10 (2003), 46–52.

Rodrigo Mantovaneli Pessoa, Eduardo Silva, Marten van Sinderen, Dick A. C. Quartel, and Luís Ferreira Pires. 2008. Enterprise interoperability with SOA: Survey of service composition approaches. In *EDOC Workshops*. 238–251.

Giacomo Piccinelli, Anthony Finkelstein, and Scott Lane Williams. 2003. Service-oriented workflow: The DySCo framework. In *Euromicro Conference*. IEEE, 291–297.

Marco Pistore, Paolo Traverso, Piergiorgio Bertoli, and Annapaola Marconi. 2005. Automated synthesis of composite BPEL4WS web services. In *ICWS*. 293–301.

Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. 2006. Synthesis of reactive (1) designs. In *Verification, Model Checking, and Abstract Interpretation*. Springer, 364–380.

Shankar R. Ponnekanti and Armando Fox. 2002. Sword: A developer toolkit for web service composition. In *WWW*. 7–11.

Christian Prehofer, Jilles van Gurp, Vlad Stirbu, Sailesh Satish, Sasu Tarkoma, Cristiano di Flora, and Pasi P. Liimatainen. 2010. Practical web-based smart spaces. *IEEE Pervasive Computing* 9, 3 (2010), 72–80.

Klaus Purer. 2011. *Web Service Composition in Drupal*. Master's thesis. Vienna University.

Alexander J. Quinn and Benjamin B. Bederson. 2011. Human computation: A survey and taxonomy of a growing field. In *CHI*. 1403–1412.

Nick Ragouzis, John Hughes, Rob Philpott, Eve Maler, Paul Madsen, and Tom Scavo. 2008. Security assertion markup language (SAML) v2. 0 technical overview. *OASIS Committee Draft* 2 (2008). Oasis Security Services. Retrieved from http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-tech-overview-2.0.html.

Erhard Rahm and Philip A. Bernstein. 2001. A survey of approaches to automatic schema matching. *VLDB Journal* 10, 4 (2001), 334–350.

Preeda Rajasekaran, John Miller, Kunal Verma, and Amit Sheth. 2005. Enhancing web services description and discovery to facilitate composition. In *SWSWPC Workshop*. 55–68.

Jinghai Rao, Peep Küngas, and Mihhail Matskin. 2006. Composition of semantic web services using linear logic theorem proving. *Information Systems* 31, 4 (2006), 340–360.

David Recordon and Drummond Reed. 2006. OpenID 2.0: A platform for user-centric identity management. In *DIM*. 11–16.

Domenico Redavid, Luigi Iannone, Terry Payne, and Giovanni Semeraro. 2008. OWL-S Atomic services composition with SWRL rules. In *Foundations of Intelligent Systems*. Springer, 605–611.

Paul Resnick and Hal R. Varian. 1997. Recommender systems. *Communications of the ACM* 40, 3 (1997), 56–58.

A. Ro, L. Xia, H. Y. Paik, and C. Chon. 2008. Bill organiser portal: A case study on end-user composition. In *WISE 2008 Workshops*. Springer, 152–161.

Florian Rosenberg, Philipp Leitner, Anton Michlmayr, Predrag Celikovic, and Schahram Dustdar. 2009. Towards composition as a service-a quality of service driven approach. In *ICDE*. 1733–1740.

Soudip Roy Chowdhury, Florian Daniel, and Fabio Casati. 2011. Efficient, interactive recommendation of mashup composition knowledge. In *ICSOC*. 374–388.

RSS Advisory Board. 2009. RSS 2.0 Specification. Retrieved from http://www.rssboard.org/rss-specification.

Nick Russell, Wil M. P. van der Aalst, Natalya Mulyar, et al. 2006. *Workflow Control-Flow Patterns: A Revised View*. BPM Center Report BPM-06-22.

Marwan Sabbouh, Jeff Higginson, Salim Semy, and Danny Gagne. 2007. Web mashup scripting language. In *WWW*. 1305–1306.

Christopher Scaffidi, Brad Myers, and Mary Shaw. 2008. Topes: Reusable abstractions for validating data. In *ICSE*. 1–10.

David Schumm, Dimitrios Dentsas, Michael Hahn, Dimka Karastoyanova, Frank Leymann, and Mirko Sonntag. 2012. Web service composition reuse through shared process fragment libraries. In *ICWE*. 498–501.

David Schumm, Dimka Karastoyanova, Frank Leymann, and Steve Strauch. 2011. Fragmento: Advanced process fragment library. In *Information Systems Development*. Springer, 659–670.

University New South Wales Service Oriented Computing Group. 2010. FormSys Project. Retrieved from http://www.cse.unsw.edu.au/~FormSys/FormSys/index.htm.

Quan Z. Sheng and Boualem Benatallah. 2005. ContextUML: A UML-based modeling language for model-driven development of context-aware web services. In *ICMB 2005*. 206–212.

Joseph Sifakis. 2011. A vision for computer science—the system perspective. *Central European Journal of Computer Science* 1, 1 (2011), 108–116.

David E. Simmen, Mehmet Altinel, Volker Markl, Sriram Padmanabhan, and Ashutosh Singh. 2008. Damia: Data mashups for intranet applications. In *SIGMOD*. 1171–1182.

Evren Sirin, Bijan Parsia, Dan Wu, James Hendler, and Dana Nau. 2004. HTN planning for web service composition using SHOP2. *Web Semantics: Science, Services and Agents on the World Wide Web* 1, 4 (2004), 377–396.

David Skogan, Roy Grønmo, and Ida Solheim. 2004. Web service composition in UML. In *EDOC*. 47–57.

Daniel Skrobo. 2007. *HUSKY: A Spreadsheet for End-User Service Composition*. Ph.D. Dissertation. University of Zagreb.

Javier Soriano, David Lizcano, Juan José Hierro, Marcos Reyes, Christoph Schroth, and Till Janner. 2008. Enhancing user-service interaction through a global user-centric approach to SOA. In *ICNS*. 194–203.

Joel Spolsky. 2008. Stackoverflow.com (Joel on Software). (April 2008). http://www.joelonsoftware.com/items/2008/04/16.html.

B. Srivastava and J. Koehler. 2003. Web service composition-current solutions and open problems. In *ICAPS 2003 Workshop on Planning for Web Services*, Vol. 35. Citeseer.

Kathryn T. Stolee and Sebastian Elbaum. 2011. Refactoring pipe-like mashups for end-user programmers. In *ICSE*. 81–90.

Kathryn T. Stolee, Sebastian Elbaum, and Anita Sarma. 2011. End-user programmers and their communities: An artifact-based analysis. In *ESEM*. 147–156.

Anja Strunk. 2010. QoS-aware service composition: A survey. In *ECOWS*. 67–74.

Byung Chul Tak, Bhuvan Urgaonkar, and Anand Sivasubramaniam. 2011. To move or not to move: The economics of cloud computing. In *USENIX*. 5–5.

Wei Tan, Paolo Missier, Ian Foster, Ravi Madduri, David De Roure, and Carole Goble. 2010. A comparison of using Taverna and BPEL in building scientific workflows: The case of caGrid. *Concurrency and Computation: Practice and Experience* 22, 9 (2010), 1098–1117.

Ian Taylor, Matthew Shields, Ian Wang, and Andrew Harrison. 2007. The triana workflow environment: Architecture and applications. In *Workflows for e-Science*. Springer, 320–339.

Maurice ter Beek, Antonio Bucchiarone, and Stefania Gnesi. 2006. *A Survey on Service Composition Approaches: From Industrial Standards to Formal Methods*. Technical Report. 2006-TR-15.

Rich Thompson. 2008. Web Services for Remote Portlets Specification v2. 0. *OASIS Standard* 1 (2008). OASIS. Retrieved from http://docs.oasis-open.org/wsrp/v2/wsrp-2.0-spec-os-01.pdf.

Sebastian Thöne, Ralph Depke, and Gregor Engels. 2003. Process-oriented, flexible composition of web services with UML. In *ER*. 390–401.

Ajay Tipnis and Ivan Lomelli. 2009. *Security - A Major Imperative for a Service-Oriented Architecture*. Technical Report. HP.

Huy Tran, Uwe Zdun, and Schahram Dustdar. 2008. View-based integration of process-driven soa models at various abstraction levels. In *MBSDI*. 55–66.

Wil M. P. Van der Aalst, Marlon Dumas, and Arthur H. M. ter Hofstede. 2003. Web service composition languages: old wine in new bottles? In *Euromicro Conference*. 298–305.

Wil M. P. Van Der Aalst and Arthur H. M. Ter Hofstede. 2005. YAWL: Yet another workflow language. *Information Systems* 30, 4 (2005), 245–275.

Wil M. P. van Der Aalst, Arthur HM Ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. 2003. Workflow patterns. *Distributed and Parallel Databases* 14, 1 (2003), 5–51.

Wil M. P. Van Der Aalst, Arthur H. M. Ter Hofstede, and Mathias Weske. 2003. *Business Process Management: A survey*. Springer.

Peter Van Roy. 2009. Programming paradigms for dummies: What every programmer should know. *New Computational Paradigms for Computer Music* (2009), 9–47.

Asir S. Vedamuthu, David Orchard, Frederick Hirsch, Maryann Hondo, Prasad Yendluri, Toufic Boubez, and Umit Yalçinalp. 2007. Web services policy 1.5-framework. *W3C Recommendation* 4 (2007), 1–41.

Kunal Verma, Karthik Gomadam, Amit P. Sheth, John Miller, and Zixin Wu. 2005. The METEOR-S approach for configuring and executing dynamic web processes. Technical Report. LSDIS Lab, University of Georgia. Retrieved from http://lsdis.cs.uga.edu/projects/meteor-stechrep6-24-05.pdf.

Dennis M. Volpano and Richard B. Kieburtz. 1985. Software templates. In *ICSE*. 55–60.

Larry Wall, Tom Christiansen, and Jon Orwant. 2000. *Programming Perl*. O'Reilly.

Guiling Wang, Shaohua Yang, and Yanbo Han. 2009. Mashroom: End-user mashup programming using nested tables. In *18th International Conference on World Wide Web*. ACM, 861–870.

Lizhe Wang, Gregor Von Laszewski, Andrew Younge, Xi He, Marcel Kunze, Jie Tao, and Cheng Fu. 2010. Cloud computing: A perspective study. *New Generation Computing* 28, 2 (2010), 137–146.

Ingo Weber, Hye-Young Paik, and Boualem Benatallah. 2013. Form-based web service composition for domain experts. *TWEB* 8, 1 (2013), 2.

Yi Wei and M. Brian Blake. 2010. Service-oriented computing and cloud computing: Challenges and opportunities. *IEEE Internet Computing* 14, 6 (2010), 72–75.

Michael Wilde, Mihael Hategan, Justin M. Wozniak, Ben Clifford, Daniel S. Katz, and Ian Foster. 2011. Swift: A language for distributed parallel scripting. *Parallel Comput.* 37, 9 (2011), 633–652.

Adam Wyner, Krasimir Angelov, Guntis Barzdins, Danica Damljanovic, Brian Davis, Norbert Fuchs, Stefan Hoefler, Ken Jones, Kaarel Kaljurand, Tobias Kuhn, and others. 2010. On controlled natural languages: Properties and prospects. In *Controlled Natural Language*. Springer, 281–289.

Jun Yan, Ryszard Kowalczyk, Jian Lin, Mohan B. Chhetri, Suk Keong Goh, and Jianying Zhang. 2007. Autonomous service level agreement negotiation for service composition provision. *Future Generation Computer Systems* 23, 6 (2007), 748–759.

Jin Yu, Boualem Benatallah, Regis Saint-Paul, Fabio Casati, Florian Daniel, and Maristella Matera. 2007. A framework for rapid integration of presentation components. In *WWW*. 923–932.

Liangzhao Zeng, Boualem Benatallah, Marlon Dumas, Jayant Kalagnanam, and Quan Z. Sheng. 2003. Quality driven web services composition. In *WWW*. 411–421.