# Conceptual Development of Custom, Domain-Specific Mashup Platforms



STEFANO SOI, FLORIAN DANIEL, and FABIO CASATI, University of Trento

Despite the common claim by *mashup platforms* that they enable end-users to develop their own software, in practice end-users still don't develop their own mashups, as the highly technical or inexistent user bases of today's mashup platforms testify. The key shortcoming of current platforms is their *general-purpose* nature, that privileges expressive power over intuitiveness. In our prior work, we have demonstrated that a *domain-specific* mashup approach, which privileges intuitiveness over expressive power, has much more potential to enable *end-user development* (EUD). The problem is that developing mashup platforms—domain-specific or not—is *complex* and *time consuming*. In addition, domain-specific mashup platforms by their very nature target only a small user basis, that is, the experts of the target domain, which makes their development not sustainable if it is not adequately supported and automated.

With this article, we aim to make the development of *custom, domain-specific mashup platforms* cost-effective. We describe a *mashup tool development kit* (MDK) that is able to *automatically generate* a mashup platform (comprising custom mashup and component description languages and design-time and runtime environments) from a conceptual design and to provision it *as a service*. We equip the kit with a dedicated *development methodology* and demonstrate the applicability and viability of the approach with the help of two case studies.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering (CASE)*; H.5.4 [**Information Systems**]: Information Interfaces and Presentation—*Hypertext/Hypermedia*; H.3.5 [**Information Systems**]: Information Storage and Retrieval—*Online information services*

General Terms: Design, Languages

Additional Key Words and Phrases: Mashups, domain-specific mashups, mashup tools/platforms, conceptual development, metadesign, mashup platforms as a service

**ACM Reference Format:**
Stefano Soi, Florian Daniel, and Fabio Casati. 2014. Conceptual development of custom, domain-specific mashup platforms. ACM Trans. Web 8, 3, Article 14 (June 2014), 35 pages.
DOI: http://dx.doi.org/10.1145/2628439

## 1. INTRODUCTION

During the last decade, both industry and academia proposed a variety of different mashup tools, such as Yahoo! Pipes (http://pipes.yahoo.com/pipes/), JackBe Presto (http://www.jackbe.com/products), ServFace Builder [Feldmann et al. 2009], Karma [Tuchinda et al. 2011], CRUISe [Pietschmann et al. 2009], MashLight [Baresi and Guinea 2010], and the like. *Mashup tools* (or platforms) are integrated development and runtime environments that are typically accessed online as a hosted service and allow their users (ranging from target users with low software development skills to experienced programmers) to develop and run their own mashups in a graphical, most of the times model-driven fashion. *Mashups* are composite Web applications integrating

Authors' addresses: S. Soi, F. Daniel (corresponding author), and F. Casati, University of Trento, Via Sommarive 9, 38123, Povo (TN), Italy; email: daniel@disi.unitn.it.

DOI: http://dx.doi.org/10.1145/2628439

data (e.g., RSS or Atom feeds), application logic (e.g., via SOAP or RESTful Web services), and/or user interfaces (e.g., graphical widgets) accessible over the Web [Daniel and Matera 2014]. That is, mashups are developed by reusing and integrating existing resources.

In our own research, we worked on mashup paradigms and tools targeting both end-users (e.g., in the MashArt project[1] [Daniel et al. 2009]) and skilled developers (e.g., in the MarcoFlow project[2] [Daniel et al. 2011]). One of the lessons we learned in this context is that mashup tools (including our own) are too technical for end-users and, as a consequence, end-users are not able to: (i) understand what exactly they can do with the tool and (ii) how to do it. This observation is backed by Namoun et al. [2010b], who performed a set of user studies to understand how end-users perceive mashups and concluded that they generally lack an understanding of what, for instance, Web services are, and of how these can be integrated to form new logics (e.g., via data or control flows). Yet, they also identified a positive attitude of end-users toward these new kinds of software development tools and the necessary curiosity to start "playing" with them, which we consider a necessary requirement for our work.

The level of technicality of a mashup tool, that is, the type and number of development-specific concepts and features exposed, is determined by its development flexibility and expressive power. Most of the existing mashup platforms are *general-purpose* mashup platforms that enable mashing up all possible kinds of resources available on the Web, but do not sufficiently abstract away from technologies. Mastering concepts like Web services, dataflows, variables, and the like is simply out of reach of the average end-user without development knowledge.

Based on these considerations, we started investigating the power of what we call *domain-specific mashup platforms* for End-User Development (EUD), where a domain-specific mashup platform is a mashup platform specifically tailored to a given domain, to its terminology, its tasks, and its needs. The ideal domain-specific mashup platform presents its users only with concepts, activities, and practices that are well known in its target domain and that *domain experts* (the end-users in the target domain) are aware of and are used to dealing with in their everyday work. And it does so by hiding all the technicalities actually implementing the features of the domain (e.g., a domain expert is simply not interested in knowing whether a payment activity is provided as SOAP or RESTful Web service).

As a proof of concept, we developed our own domain-specific mashup platform for a domain we are well acquainted with ourselves, namely research evaluation, but that also involves people without software development skills (e.g., administrative staff and research staff from non-IT departments). The platform is called ResEval Mash [Daniel et al. 2012], and the user study we performed with our target domain experts supports our hypothesis that domain-specific mashup platforms are another step forward toward EUD. Test groups with and without IT skills performed similarly well, and both were able to implement a set of test research evaluations.

With the development of ResEval Mash, though, we also experienced how complex and time consuming the design and implementation of a domain-specific mashup platform can be. In addition to the general technical aspects common to all mashup platforms tailoring a platform to a specific domain does not only require a thorough analysis and formalization of the domain (typically also involving interaction with domain experts), but also a clear understanding of how to "inject" the acquired domain knowledge into a mashup platform and how to hide unnecessary technicalities. We developed ResEval Mash for research purposes but, given the relatively small user

[1]https://sites.google.com/site/mashtn/industrial-projects/mashart.

[2]https://sites.google.com/site/mashtn/industrial-projects/marcoflow.

basis of domain-specific tools, this effort is not sustainable in general for the large-scale development of domain-specific mashup platforms.

With this article, we conceptualize all the major aspects that characterize the development of mashup platforms, whether generic or domain specific, and describe an approach to the development of domain-specific mashup platforms that: (i) features a conceptual platform development paradigm and (ii) a complete generative platform architecture able to automatically produce suitable design time and runtime mashup environments. Specifically, we provide the following contributions.

(1) We analyze the current mashup ecosystem, characterize the most used types of mashups (e.g., data versus user interface mashups), and derive a large set of composition features (e.g., the types of components supported or the way data is propagated among components) that mashup platforms may have to support.
(2) We define a unified mashup model that brings together all identified features in one model and expresses them as mashup model patterns and feature constraints.
(3) We describe a conceptual approach to the development of custom mashup languages, based on composition features and the automatic resolution of conflicts.
(4) We describe the implementation of a mashup runtime environment that is able to execute mashups expressed in any of the conceptually designed mashup languages. We also describe a prototype mashup editor that can be tailored to a given domain.
(5) We put everything into context in the form of a methodology for the development of domain-specific mashup platforms and exemplify the application of the approach with the development of two mashup platforms.

Next, we elaborate further on the background and motivations of this work. In Section 3, we state requirements and design principles and outline our approach. In Section 4, we specifically focus on mashup language features, the design of the unified metamodel, and the conceptual development of custom mashup languages. In Section 5, we clarify the respective operational semantics by explaining the runtime environment and describe the customizable mashup editor and implementation of the overall supporting infrastructure. In Section 6, we bring all development aspects and instruments under one hood and describe a methodology for the development of domain-specific mashup platforms. We apply the approach to two use cases in Section 7 and evaluate the method's viability and limitations in Section 8. Then, we discuss related works (Section 9) and the lessons we learned (Section 10).

## 2. RATIONALE AND BACKGROUND

A prerequisite of our work is a basic understanding of mashups, which is not trivial.

### 2.1. Reference Scenario: Research Evaluation Mashups

A concrete domain for mashups we have been working on is research evaluation. We identify with this term all those procedures aimed at providing a quantitative value representing the quality of a researcher, a research artifact, a group of researchers (e.g., a department), a group of research artifacts (e.g., conference proceedings), and similar entities. Producing and having access to these kinds of data is very important in different contexts, ranging from self-assessment to the hiring of new personnel and the distribution of money in departments.

For example, in Figure 1 we provide a graphical representation of the evaluation process adopted in the University of Trento (UniTn) for distributing resources and research funds among departments based on their research productivity. In essence, the process compares the quality of the scientific production of each researcher in a given department with the average quality of the research production of researchers

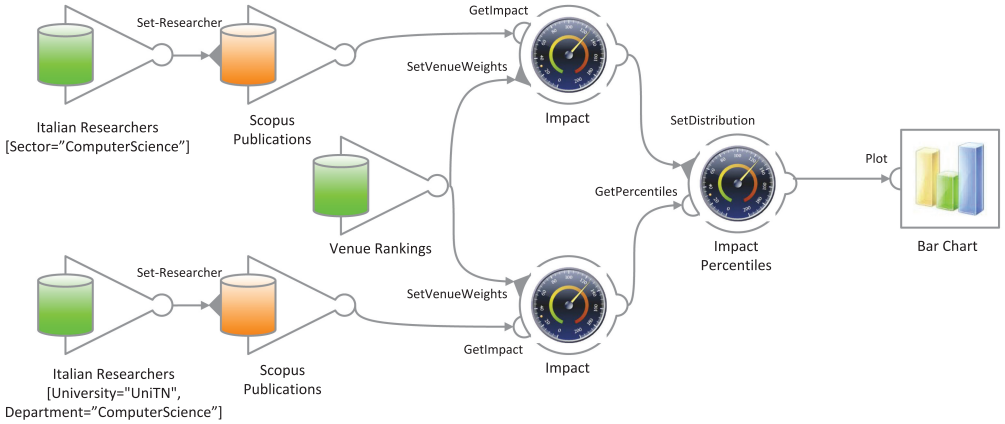


Fig. 1. Model of University of Trento's internal department evaluation procedure.

belonging to similar departments (i.e., departments in the same disciplinary sector) in all Italian universities. The comparison is based on the weighted publication count.

Today the computation of research evaluation processes is done manually by, for example, the university administrative employees, using fixed criteria for the selection of, for instance, bibliographic information sources or evaluation metrics. Evaluating research, though, is not so easy and straightforward. Many aspects can heavily impact the evaluation results, for instance, the publication venues' ranking, the source of the bibliometric information, the algorithms used for the metrics, etc. Fervid discussion on the suitability of the selected criteria often arises, as people would like to understand how the results would differ if one changed some of these criteria. This would require, however, the computation of many variations of the process (each one adopting different criteria). Doing so would cost huge human resources and time. It is clear, therefore, that developing all these process variations through manual implementation or outsourcing is not a viable solution.

The task has, however, all the characteristics of a mashup, especially if the mashup logic comes from the users themselves. Developing a mashup tool that enables administrators and generic faculty members to run evaluation logics, such as the one in Figure 1, is challenging and requires: (i) supporting all technical intricacies of the scenario (e.g., the integration of data, Web services, and UIs as well as data transformations) while (ii) hiding them from the user (e.g., Figure 1 does not present any technical aspects besides sources, metrics, visualization components, and dataflows). End-users not only do not know about technicalities, they also don't want to know.

In order to identify the level of abstraction end-users are comfortable with, we developed ResEval Mash (http://open.reseval.org/), a mashup platform for research evaluation. The development of ResEval Mash can be divided into two phases: the analysis of the aforesaid requirements and the implementation of the platform.

Understanding which requirements to elicit and how to formalize them so that they can be used to drive the design of a mashup platform was harder than expected and required significant abstraction and modeling skills. Identifying the key aspects of research evaluation, the nature of the data involved, and the tasks that characterize it took several months. This kind of analysis required significant formalization efforts and deep knowledge about research evaluation, which we built partly based on our own needs and partly by involving the experts in research evaluation of our department.

Turning requirements into suitable designs for a mashup platform and eventually implementing the platform took even longer than the first phase, that is, five to six months. The design and implementation of the various mashup components that populate the mashup platform took an additional month to complete. A big effort we spent on identifying where the specifics of research evaluation manifest themselves inside a mashup platform (e.g., in the development of components or in the mashup language underlying the platform) and on designing suitable architectural solutions bringing together the specifics of research evaluation with the characteristics of a mashup platform (e.g., component-based development). This activity required deep knowledge of composition language design, model-driven software development, client- and server-side Web development, client-server communication patterns, Web service orchestration, data integration, user interface design, and so on.

### 2.2. Domain-Specific Mashups

Abstracting from the concrete case of research evaluation, that is, the domain of our mashup tool, we define a *domain* as a delimited sphere of activity and knowledge described via concepts and processes. *Domain concepts* express the static aspects of a domain, that is, they tell which elements (e.g., researchers and publications) and relationships thereof characterize the domain and introduce the necessary domain-specific terminology. *Domain processes* express the dynamic aspects of the domain, that is, they tell how domain concepts are operated and manipulated (e.g., the computation of an h-index from a set of publications). Domain processes can be atomic (activities) or composite (processes integrating multiple activities).

A domain-specific mashup is an example of composite domain process. More precisely, a *domain-specific mashup* (DSM) is a mashup that implements a composite domain process manipulating domain concepts via domain processes. As such, a domain-specific mashup is an application that solves a problem in a given domain, such as the computation of a custom evaluation metric for research evaluation.

A *domain-specific mashup tool*[3] (DMT) is now a development and execution environment that allows *domain experts* (the end-users of the target domain) to develop all possible types of domain-specific mashups (and nothing more). In order to do so, a DMT may feature a *domain-specific composition language* (a language specifically tailored to the development of the composite processes that characterize the domain) and a *domain-specific syntax* (a syntax that makes use of domain terminology and conveys the semantics of domain activities) that enable domain experts to develop their own mashups in an as-familiar-as-possible environment.

This definition of DMT does not necessarily imply specific technical or technological choices. It rather emphasizes the difference of a DMT from generic tools that are instead domain agnostic and aim to support as many domains as possible by not making any assumption about the target domain of the tool, neither in terms of interesting reusable functionalities nor in terms of target mashups. Generic tools therefore focus more on technologies, such as SOAP services or W3C widgets, while DMTs rather focus on specific activities, such as the fetching of data from Google Scholar or the computation of an h-index (independently of how these are implemented). The research evaluation process shown in Figure 1 is an example of a domain-specific mashup expressed in a domain-specific, graphical modeling notation (note the terminology used and the three types of components: data sources, metrics, and charts).

As for the effectiveness of domain-specific mashup tools in enabling end-user development, in Daniel et al. [2012] we report on a dedicated user study. The data we

[3]Throughout this article we use the terms *tool* and *platform* interchangeably, and we specifically focus on mashup tools producing mashup specifications that can be interpreted and executed by an engine.

collected in collaboration with the HCI group of University of Trento provide concrete evidence that domain-specific mashup tools like ResEval Mash are more accessible to end-users than generic mashup tools. Given different mashup scenarios, surprisingly, participants quickly understood that components had to be linked together so that information could flow between components thus making the necessary connections. The lack of this understanding and ability is a well-acknowledged problem evidenced in several other user studies of EUD tools [Namoun et al. 2010b]. Our observations indicate that the key to solve this problem is: (i) to alleviate users from specifying data mappings by adopting a *by-reference* data passing paradigm and (ii) the purposeful design of the Web services to be mashed up. In fact, eliminating data mapping tasks from the mashup development process is a major simplification end-users benefit from and a feature generic tools cannot provide, as they lack the necessary knowledge of the domain data.

### 2.3. Types of Mashups

Research evaluation is only one domain that may benefit from mashups. Many other domains exist. However, it is impossible to list and characterize them all here. What we can do to collect further requirements is summarize the spectrum of mashup types that have emerged over the last years and that may be tailored to specific domains. For the purpose of this work, we identify the following types of mashups.

—*Simple data mashups*. One of the first and most popular mashup types aims at the integration of data, typically coming in the form of structured data like RSS or Atom feeds, JSON, or other XML data resources. Data mashup tools are typically based on a dataflow paradigm, where data is fetched from the resources and processed via different operators by passing data from one operator to the other. Common operators are filter, split, merge, truncate, and so on. The result of data mashups is generally a data resource itself, for example, an RSS feed that can be accessed via the Web. A well-known representative of mashup tool for this category is Yahoo! Pipes.
—*Data-intensive data mashups*. This type of mashup is a specialization of data mashups which is characterized by large volumes of data to be processed. Data-intensive mashups may work with data amounts that cannot be easily transferred over the Web, the preferred solution of generic, hosted data mashup solutions based on dataflows. A solution to this problem is, for example, the use of components and a mashup environment that pass data by reference, instead of by value. Scientific workflow tools are examples that adopt this paradigm.
—*User interface (UI) mashups*. UI mashups integrate UI components into a shared layout so that the user can directly interact with them, provide inputs, and get visual outputs. A key characteristic of UI mashups is the event-based synchronization of UI components (e.g., through a publish/subscribe infrastructure) which allows the user to control multiple components by interacting with only one of them. Web portals or widget containers (e.g., based on W3C widgets or OpenSocial gadgets) fall into this category of mashup tools.
—*Simple service mashups*. This kind of mashups focuses on Web service composition, where Web services are integrated and orchestrated in a process-like fashion. The integration logic is typically expressed as a control flow specifying when each service is to be invoked. Data is usually passed among services via intermediate variables that host the output of one service until the invocation of one or more other services (the so-called blackboard approach). Two key aspects of service mashups are long-running processes and asynchronous communications, the former requiring an execution environment independent of the client starting the service mashup, the

latter requiring message correlation. One example of service mashup tool is ServFace Builder, but also simple BPEL compositions fall into this category.

—*Hybrid mashups*. Mashups express their actual power only in a hybrid setting, in which a mashup integrates components at different layers of the architectural stack, that is, the data, logic, and UI layers. Not mandatorily all layers must be present for a mashup to be considered hybrid. For instance, a UI mashup with process-based Web service integration can be considered a hybrid mashup. The key is the seamless integration of all the other mashup types described, with the mashup logic possibly being distributed over client and server. An example of mashup tool for this category of mashups is mashArt, which features a so-called universal integration paradigm.

A DMT may be based on any of these basic types of mashups and include additional, domain-specific features, such as specific components, terminology, composition rules, and the like. The problem we address in this article is how to ease the development of DMTs, both technically and methodologically.

## 3. CONCEPTUAL DEVELOPMENT OF DOMAIN-SPECIFIC MASHUP TOOLS

The key idea to enable and ease the development of DMTs is: (i) to boil down the development of a whole DMT to the development of its internal mashup language, (ii) to devise a conceptual approach to the development of its mashup language based on the selection of so-called mashup language features, and (iii) to automatically generate the DMT starting from the designed mashup language. We jointly call the set of software artifacts supporting this process the *Mashup Tool Development kit* (MDK).

### 3.1. Goals and Requirements

The goals and requirements of the MDK can be summarized in four points (items are goals, subitems are concrete requirements). Specifically, we aim for the following.

(1) We support a large variety of mashup types to cover as many technical requirements and domains as possible. As hinted at in Section 2.3, mashups may require very different internal integration logics, and each domain may have its very own requirements. The more integration techniques we support, the more DMTs can be developed. Doing so concretely requires support for the following.
   —Data, application logic, and user interface (UI) component types: RSS and Atom feeds, SOAP and RESTful Web services, W3C widgets [W3C 2011], and generic JavaScript components [Daniel et al. 2009]
   —Flexible configuration of components (e.g., manual inputs at design time)
   —Synchronous and asynchronous interaction patterns (e.g., the request-response, solicit-response, one-way, and notification patterns for Web services, but also events for UI components)
   —Short-living (e.g., for the duration of a user session) and long-running mashups (lasting even beyond a user session)
   —Control-flow logics to structure the execution of the integrated components, such as via sequential or parallel executions of invocations (branching and joining the control flow), loops, conditions, support for correlation, etc.
   —Different data passing techniques, such as dataflows (also implying some control flow), shared variables, or reference passing
   —Presentation, that is, the layout and rendering of mashup outputs and/or UI components, for instance, spread over individual or multiple pages
   —Multiple users, such as in a process-like fashion (each user with his/her own task or view on the mashup) or concurrently (all users sharing the same view in parallel)

(2) We provide a conceptual approach to the design of custom mashup languages. Different integration techniques require different constructs and different constructs require different mashup language structures. Designing a sound, that is, consistent and nonambiguous, mashup language is nontrivial and requires a lot of data, service, UI integration expertise, and concretely, support for the following.
   —Conceptual language features that express typical mashup language capabilities and abstract away from their low-level implementation (e.g., dataflows or request-response operations are features a language may need to support)
   —Fast feature composition, that is, the easy selection and combination of language features into a consistent mashup language that can be used for the development of the supporting DMT
   —Automated soundness controls that check possible dependencies among language features (e.g., that UI components require a presentation) or incompatibilities therefore (e.g., that control flow and dataflow cannot be put together)
(3) We automate the implementation of DMTs. Given a mashup language, implementing a mashup tool that exactly supports all the capabilities of the language can be very time consuming. The goal is to alleviate the developer from this task by automatically generating a design and runtime environment, given a mashup language specification. Doing so requires support for the following.
   —The automatic generation of mashup language specifications in the form of XSD schemas (other schema languages could be adopted as well) that can be used to express mashups and to drive the implementation of DMTs
   —The automatic generation of component descriptor specifications, that express the component configurations performed at the conceptual level, guide the developer in the development of components, and enable the registration of components with the DMT
   —A configurable runtime environment able to support all mashup features identified previously, while also being able to constrain the features it offers to its users to only those specified in a given mashup language
   —A configurable design environment enabling mashup developers to design their own mashups in a model-driven fashion, where the exposed modeling constructs and available components depend on the particular mashup language chosen
(4) We deliver DMTs as a service. To ease the adoption of custom DMTs, the vision is to provision generated DMTs as a service so as to alleviate the developer from tedious deployment and maintenance tasks that could threaten the continuity of the service delivery. This requires support for the following.
   —Hosted deployment and execution of DMTs, where, instead of producing code packages or software modules in output of the automated generation process, the idea is to deploy generated DMTs in a hosted fashion
   —Multitenancy, in order to separate different DMTs running on the same DMT server from each other

First, we outline how we address these requirements, then we look into the details.

### 3.2. Approach

We approach the conceptual design of mashup languages by focusing on individual mashup language features, that is, the composition/integration features a mashup language should support. For instance, whether a mashup is built by specifying dataflow connectors among the components to be integrated or if, instead, it is built by specifying variables and control-flow connectors to express the mashup's integration logic depends on the design of the mashup language supported by the DMT. Support for dataflow, for control flow, for specific types of component technologies and the like are, for example, what we call mashup language features.

The goal of the conceptual design of the mashup language is to automatically generate a mashup language specification out of a selection of features, where such specification can come either as a textual language schema or a graphical mashup model. In our work, we specifically aim at the generation of an XSD schema that can be processed by a machine, while we use equivalent, graphical models for presentation purposes in this article. Language features can thus be represented as language patterns, namely patterns of the language specification that may comprise multiple language constructs and possible relationships among the constructs. A mashup language is then a sound composition of language patterns, where "sound" means noncontradicting at runtime.

Designing a mashup language is therefore a composition problem in its own, with the language patterns being the "components" with which to work. Compared to conventional component-based development approaches, however, language patterns have a distinctive feature that makes our problem very different (next to the fact that we do not handle software modules but model fragments): unlike, for example, Web services, language patterns are not independent and may present incompatibility and dependency relations. That is, the language patterns of different mashup features may overlap (e.g., interacting with a SOAP service is very similar to interacting with a RESTful service), include other features (e.g., the dataflow paradigm generally subsumes the presence of data source components), or exclude others (e.g., the dataflow paradigm does not make use of variables). This asks for a thorough design of the language patterns and their mutual interaction points.

We approach this issue by mapping each composition feature into a unified mashup model (UMM), that: (i) defines and integrates all basic language constructs syntactically, (ii) enables the definition of mashup language features as language patterns (model fragments) on top, and (iii) guarantees that patterns are compatible by design. This unified metamodel does not per se express an executable mashup language in that it contains all possible features we identify, which may also be conflicting if used together inside a single language. It rather serves as a means for integrating all constructs and features in a consistent fashion. Only the sensible selection of a consistent set of nonconflicting mashup features enables the generation of a sound specification of a mashup language and of a component descriptor language.

The availability of the unified mashup model further enables the generation of the actual DMT, that is, the mashup runtime environment and the mashup editor, in a relatively comfortable fashion: By design, all the mashup languages derived from the UMM will contain a subset of the constructs defined by the UMM, and nothing else. This observation allows us to implement a single runtime environment that: (i) supports all the constructs and features expressed in the UMM and (ii) can be easily constrained to only those features contained in a given mashup language specification. In other words, the runtime environment is able to execute all mashups designed in any of the languages based on the UMM. The entire runtime environment must be aware of a given mashup's language specification. For example, it must know whether to expect control-flow or dataflow constructs, or how data is passed among components. Similarly, it is possible to develop a single mashup editor that: (i) provides graphical modeling constructs for all the language constructs in the UMM and (ii) can be constrained to those constructs of the language features of the chosen mashup language only.

### 3.3. MDK Architecture

Figure 2 illustrates the functional architecture of the MDK implementing the preceding approach. The architecture is composed of four main components, namely, a DMT development tool, a configurable, domain-specific mashup editor, a configurable, domain-specific runtime environment, and a set of mashup components, as well as a set of artifacts, namely, the UMM, a DMT configuration package, a set of component

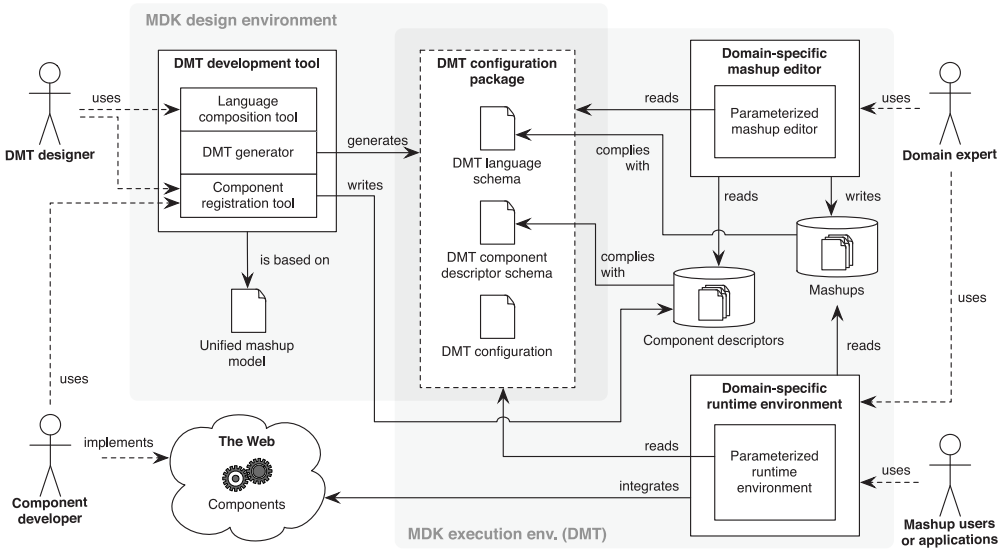


Fig. 2. Functional architecture of the MDK with main components, artifacts, and involved actors (we omit the details of user, access rights, and multitenancy management).

descriptors, and a set of mashup specifications. Note that the design environment of the MDK is represented by the DMT development tool and the output it produces, that is, the DMT configuration package, and that the execution environment of the MDK is represented by the configurable editor and runtime environment that read the DMT configuration package. Executing a DMT design thus effectively means instantiating a complete DMT.

The figure also illustrates the involvement of four actors (or roles), namely the DMT designer, the component developer, the domain expert, and the mashup user. The DMT designer develops new DMTs, starting from his/her domain analysis and a set of according components implementing domain activities. Mashup components are implemented by a component developer, who registers the components with the platform (the DMT developer and component developer may be the same physical person). Once a DMT is deployed, the domain expert can use it to develop domain-specific mashups that can then be used either by himself/herself or by a generic mashup user (without the domain knowledge necessary to develop the mashup) or an external application/service (e.g., as in the case of data or service mashups).

The DMT development tool is based on the unified mashup model and allows the DMT designer to develop a custom DMT by designing a custom mashup language (supported by the visual language configuration tool). The implementation of custom DMTs is done via the DMT generator, which generates a DMT configuration package containing three artifacts: (i) the XSD schema of the custom mashup language, (ii) the XSD schema of the component description language; and (iii) a DMT configuration file. This latter contains the list of mashup features selected by the DMT developer, an optional definition of a domain syntax for the mashup editor, an optional list of mashup components for the editor, a reference to the selected mashup repository, and a reference to the component repository (for the runtime environment).

The DMT configuration package is stored in a repository of the MDK and identified by a unique URL, that can be used for its retrieval. Figure 2 focuses on the architectural elements necessary for the development of one DMT; it neglects the repository

needed for user/organization (tenants) management as well as the multitenancy management interfaces. Suitably setting access rights allows tenants to restrict access to the mashups, component descriptors, and DMT configuration package repositories to specific tenants/users.

The domain-specific mashup editor is the result of the configuration of a parameterized mashup editor with the DMT configuration package. Based on the configuration of the given DMT and the component descriptors of the components defined in the configuration package, the editor provides graphical constructs to manipulate all necessary mashup components, possibly using the domain syntax artifacts also specified in the DMT configuration file (the domain syntax specifies domain-specific graphical constructs and references associated image files to be injected into the editor). The editor generates mashup model instances that are compliant with the DMT's mashup language and stores them in its own repository.

Also the domain-specific runtime environment is the result of the configuration of a parameterized runtime environment (that supports all constructs of the unified mashup model) with the configuration package. Given a mashup model instance, the runtime environment derives the specific mashup language it is specified with and configures its behavior accordingly. Executing a mashup causes the runtime environment to instantiate the mashup and to interact with the mashup components, whether remotely via message exchanges (e.g., for Web services) or locally via direct invocations (e.g., for UI components).

## 4. CONCEPTUAL DESIGN OF CUSTOM MASHUP LANGUAGES

The core of the MDK is the conceptual design approach for DMTs, which, as described earlier, materializes concretely in the design of the mashup language. Starting from the goals and requirements identified in Section 3.1, in the following we construct the Unified Mashup Model (UMM) that brings together all language concepts and constructs that emerge from these requirements. Then, we express the specific mashup language features we aim to support as patterns of the UMM and explain the resulting language design and generation process.

### 4.1. Unified Mashup Model (UMM)

The design of the UMM follows a threefold goal: first and foremost, it aims to elicit and interrelate all concepts necessary to implement mashups that satisfy the requirements described in Section 3.1 (we highlight them in the description that follows in italic); second, it aims to express a set of language constructs that can be used for the construction of a mashup language; third, it aims to bring all constructs under one consistent, integrated hood, thus enabling the correct selection of subparts of the UMM. The challenge is to support as many mashup capabilities as possible (so as to support a big number of possible DMTs) while keeping the model as simple as possible (so as to keep the language clean and usable). The result is illustrated in Figure 3.

The model supports a varied set of *component technologies* (distinguished via a `Type attribute`), that is, RSS feeds, Atom feeds/services, JavaScript data/logic components, RESTful Web services, SOAP Web services, W3C UI widgets, and our own JavaScript UI components [Daniel et al. 2009], that also support inter-widget communication.

SOAP Web services are expressed as components with a set of operations, where each operation may have one input and one output message. Input/output messages are expressed via a single input/output parameter per operation, while the data type of the parameters carries the messages' XSD schemas. SOAP Web services may come with operations supporting *synchronous* and *asynchronous interactions* (request-response, solicit-response, one-way, notification). Suitably setting the type of the operation and the cardinalities of the input/output parameters enables expressing

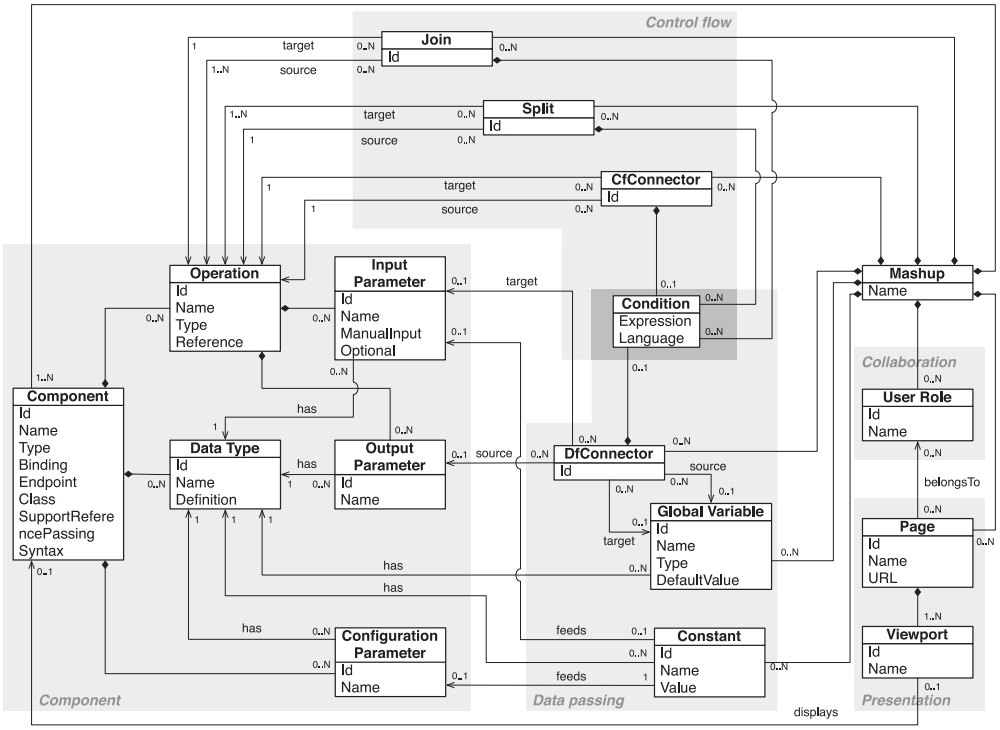


Fig. 3. The unified mashup model for custom mashup language generation. Gray boxes group entities into feature types. The component group is also used to derive component descriptor languages.

whether an operation has only an output (one-way), an input (notification), or an input and an output (request-response and solicit-response). The protocol binding (e.g., HTTP versus SMTP) and physical endpoint (the URL) are expressed via attributes.

RSS and Atom feeds are special instances of generic RESTful Web service (they have standardized data formats [RSS Advisory Board 2009; Nottingham and Sayre 2005] and operations [Gregorio and de Hora 2007]). We can therefore map them similarly: since RESTful Web services are stateless, that is, different invocations of the service (also via different operations) are independent of each other, each RESTful service can be expressed as a set of independent components—one for each URI of the service. Each such component may have a set of operations (get, post, put, delete) with input and output messages as described for SOAP Web services. RESTful services, though, do not support active behaviors, namely, solicit-response and notification operations.

For UI components with their own user interfaces, their *presentation* can be assigned to viewports inside pages (page templates with dedicated placeholders). W3C widgets can be seen as components with as set of configuration parameters for their *design-time configuration*, but without operations (the official widget specification [W3C 2011] does not support inter-widget communication). JavaScript components may come with or without UI and may also have a set of operations and configuration parameters.

Given these components, their integration logic may be achieved at the presentation layer via suitable layout templates, at the application-logic layer via the definition of the control flow that regulates the enactment of components, or at the data layer via a dataflow logic. *Control flow* is supported via control-flow connectors (`CfConnector`) that connect operation invocations, and split/join operators with possible control-flow conditions (e.g., for the definition of *or* or *and* splits or *loops*). *Dataflow* is supported via

dedicated dataflow connectors, that directly connect output to respective input parameters of different components, thereby also implying an invocation logic for the involved components. *Data passing* for control-flow-based mashups is enabled via dedicated global variables (holding dynamically changing data) and constants (for stable data), which can be connected to parameters via dataflow connectors (that in this modality don't imply any activation logic). The model supports three types of data transformations: parameter-parameter mappings, external transformations, and component-internal transformations. The former are supported via the `target` and `source` relationships of the `DfConnector` entity. External transformations can be plugged in as external services and used as independent components. Component-internal transformations leverage on the "by reference" data passing paradigm, requiring the availability of a shared memory whose structure is known and accessible to all components.

*Multiple users* are accommodated by the model via a simple, role-based access control logic: each mashup may have a set of user roles that in turn may have access to a (sub)set of pages. Each page may allow access through multiple roles. The lifetime of a mashup depends on the used components and the runtime environment of mashups.

The set of language constructs contained in the UMM does not come with the claim of absolute completeness. As we will see later, the UMM is meant to be extensible and to evolve over time, also with the help of the community. Yet, next we show how the UMM is already able to express a fairly large set of features for the design of mashup languages. We will also see that the UMM supports features that are mutually exclusive and that hence require careful selection when designing mashup languages.

### 4.2. Mashup Language Features

The gray background boxes in Figure 3 group the UMM constructs according to the categories of language features we identify, namely component, control-flow, data passing, presentation, and collaboration features. We recall that language features express capabilities of a mashup language and allow a developer to select those constructs of the UMM that implement his/her particular language requirements.

*4.2.1. Feature Model.* We formally represent a mashup language feature as a tuple $f = \langle name, label, description, specification, constraints \rangle$, where *name* is a unique identifier of the feature (e.g., "data_flow"); *label* is a human-readable name (e.g., "Dataflow"), *description* is a natural language description of the feature, *specification* is the specification of the UMM pattern implementing the feature, and *constraints* is a set of feature compatibility and dependency constraints.

*Feature constraints* are Boolean conditions that check: (i) whether all features required by a given selection of features are contained in the selection and (ii) whether the selection contains conflicting features. Feature constraints therefore guarantee the semantic soundness of a selection of features. Feature constraints are of the form $constr ::= fbool \mid \text{NOT } constr \mid constr \text{ op } constr$, where $fbool = \langle name, val \rangle$ is a Boolean variable representing the selection (or not) of a feature $f$, $fbool.name = f.name$ and $val = true \mid false$, and $op \in \{\wedge, \vee, \oplus\}$ is a logical AND, OR, or XOR operator.

The *pattern specification* for a feature is based on the structure of the UMM and expressed via three instructions: (i) the selection of constructs, (ii) the setting of possible attribute values, and (iii) the configuration of relationship cardinalities. The selection of constructs can be seen as a simple function $include(construct^{+})$, telling which constructs (entities of the UMM) are part of the pattern (relationships among two constructs are automatically added if both constructs are selected). Setting values for attributes corresponds to a function $addValue(attribute, value)$. The configuration of cardinalities can be seen as a function $setCardinality(relationship, minOccurs, maxOccurs)$ that, given

Table I. Excerpt of the Component Features

| Feature name | Mashup model pattern | Constraints |
|---|---|---|
| data_components | include (Component, Operation, DataType, OutputParameter) | RSS OR Atom OR REST OR SOAP |
| service_components | include (Component, Operation, DataType, InputParameter, OutputParameter) | REST OR SOAP OR JS |
| UI_components | include (Component, Data Type) | widget OR mashArt |
| RSS | include (Component, Operation, DataType, InputParameter, OutputParameter), addValue (Component.Type,"RSS") | – |
| request_response | include (Operation, InputParameter, OutputParameter, DataType), addValue (Operation.Type, "request_response") | – |
| solicit_response | include (Operation, InputParameter, OutputParameter, DataType), addValue (Operation.Type, "solicit_response") | SOAP OR JS OR mashArt |
| one_way | include (Operation, InputParameter, DataType), addValue (Operation.Type, "one_way"), setCardinality (Operation2OutputParameter, 0, 0) | – |
| notification | include (Operation, OutputParameter, DataType), addValue (Operation.Type, "notification"), setCardinality (Operation2InputParameter, 0, 0) | SOAP OR JS OR mashArt |
| configuration_parameters | include (ConfigurationParameter, Constant) | JS OR widget OR mashArt |
| 1_operation_per_component | setCardinality (Component2Operation, 1, 1) | – |
| n_operations_per_component | setCardinality (Component2Operation, 1, n) | – |
| 1_output_parameter | setCardinality (Operation2OutputParameter, 1, 1) | – |
| n_output_parameters | setCardinality (Operation2OutputParameter, 1, n) | – |
| ... | ... | ... |

Table II. Excerpt of the Control-Flow Features

| Feature name | Mashup model pattern | Constraints |
|---|---|---|
| control_flow | include (CfConnector) | NOT data_flow |
| split | include (Split) | control_flow |
| join | include (Join) | control_flow |
| conditions | include (Condition) | control_flow OR data_flow |
| ... | ... | ... |

a directed relationship, sets its minimum and maximum target cardinalities, possibly overwriting the default values contained in the UMM.

*4.2.2. Feature Specifications.* In the following, we describe the set of specific features we identified so far, grouped according to Figure 3. Tables I–V provide the respective pattern specifications and feature constraints. In the following, for space reasons, we discuss only the core language features of our approach. The full list of features identified so far can be inspected in http://goo.gl/uxhKQ.

*Component features* (Table I). The most important features of a mashup language are the types of target components to be mashed up (*data_component, service_component, UI_component*), along with the specific technology instances supported (*RSS, Atom, REST, SOAP, JS, widget, mashArt*). Next, for each of these technologies, it may be necessary to specify which exact communication patterns the language should support (*request_response, solicit_response, one_way, notification*). Supported communication patterns are added to the operation construct's Type attribute. Similarly, the language

Table III. Data Passing Features

| Feature name | Mashup model pattern | Constraints |
|---|---|---|
| blackboard | include (GlobalVariable, DfConnector) | NOT data_flow |
| by_reference | include (DfConnector) | NOT data_flow |
| data_flow | include (DfConnector) | NOT control_flow |

Table IV. Presentation Features

| Feature name | Meshup model pattern | Constraints |
|---|---|---|
| user_interface | include (Page, Viewport) | – |
| single_page | setCardinality (Mashup2Page, 1, 1) | user_interface |
| multiple_pages | setCardinality (Mashup2Page, 1, n) | ser_interface |

Table V. Collaboration Features

| Feature name | Mashup model pattern | Constraints |
|---|---|---|
| collaboration | include (UserRole) | user_interface |
| role_based_access | setCardinality (Page2UserRole, 1, 1) | collaboration |
| concurrent_access | setCardinality (Page2UserRole, 1, 1n) | collaboration |

may support configuration parameters (*configuration_parameters*) for JS or mashArt components or UI widgets, and it may constrain the number of operations per component (*1_operation_per_component, n_operations_per_component*) and the number of parameters per operation (e.g., *1_output_parameter, n_output_parameters*). For instance, in Yahoo! Pipes each component corresponds to one operation, while for SOAP services we have only single input/output parameters, namely, the messages.

*Control-flow features* (Table II). They specify in which order components are enacted. The use of simple control-flow connectors (*control_flow*) allows a mashup language to specify simple sequential executions. Adding also split and join operators (*split, join*) provides support for parallel executions. Adding conditions to control-flow connectors (*conditions*) further provides support for conditional branching and joining and for loops. Conditions are Boolean expressions over data flowing through the connector, variables, or constants and are written in some language that the runtime environment can evaluate (e.g., JavaScript).

*Data passing features* (Table III). They define how the language allows mashup developers to specify how data is passed among the mashed up components. Control-flow-based languages like BPEL typically use global variables (*blackboard*) to temporarily store outputs of one component and forward them as input to other components via suitable dataflow connectors. These data may represent the actual values of interest, or they may represent only references (*by_reference*) to values stored somewhere else, for example, in a database shared by all components. The UMM models the respective writing/reading operations with a dataflow connector between a variable and its target/source parameter; constants are used only for component configuration.

If the mashup language is based on the dataflow paradigm (*data_flow*), the dataflow connectors also imply an order of enactment of components, next to specifying how data are propagated among components. For instance, Yahoo! Pipes allows one to map output parameters to input parameters by specifying suitable dataflow connectors between parameters instead of between operations.

*Presentation features* (Table IV). They enable the specification of user interface integration logic and the layout of different components inside one or more Web pages, if the language aims at the development of UI mashups (*user_interface*). Typical mashups consist of only one page (*single_page*), but also mashups that distribute integrated components over multiple pages (*multiple_pages*) are possible.

*Collaboration features* (Table V). Mashups may be instantiated independently for each user without requiring any user management, or they may allow multiple users to share a same instance of the mashup (*collaboration*), requiring proper user and access rights management. If access to individual pages in a multipage mashup is restricted to individual roles, the language supports *role_based_access*. If multiple users are allowed to use the same page in parallel, the language supports *concurrent_access*.

This list of features, along with their mashup model patterns and constraints, shows how developing a good UMM is crucial, as is a trade-off between the simplicity and usability of the languages it supports (the fewer constructs the better) and the ease of mapping features onto it (the more constructs the better; in the extreme case, each feature could have its own construct). The simplicity of the language affects both the usability by human users and the complexity of the runtime environment. The ease of mapping features affects the complexity of the feature-driven design of the language: the more overlapping and conflicting language patterns, the harder the task. The challenge we faced is identifying the right balance between the two.

### 4.3. Feature-Driven Design and Generation of Mashup Languages

Developing a custom mashup language now means composing features, namely, their mashup model patterns, into a sound, custom mashup model. We consider a set of features *sound* if it respects all compatibility constraints (e.g., *data_flow* is incompatible with *blackboard*) and dependency constraints (e.g., *collaboration* requires a *user_interface*) associated to its features. Given our feature model, that already comes with a mashup model pattern and all necessary constrains, the actual "composition" of the language consists of the simple selection of features from the list of available features and the validation of the respective constraints.

We initiate the language composition process with a set of base constraints and a base language, that, respectively, enforce global constraints that guarantee the integrity of the overall language and contain the minimum set of language constructs. For instance, the constraint *(control_flow XOR data_flow) OR user_interface* requires the selection of at least one basic mashup paradigm (e.g., a simple state machine or a UI widget portal), while the constraint *data_components OR service_components OR UI_components* requires the selection of at least one type of component to be used. The base language contains at least the `Mashup` entity of the UMM.

Algorithm 1 illustrates our mashup language generation algorithm, given the names of a set of selected features $FnameSel$ and a knowledge base $F$ containing all feature definitions (including their patterns expressed in XSD). First, we initiate the set of Boolean variables $FB$ representing features, assigning "true" to selected features and "false" to nonselected features. Then, we construct the global set of constraints as a conjunction of all individual constraints and check their soundness. If the check fails, we interrupt and return *null*, otherwise we proceed and merge model fragments.

The outputs of the algorithm are two XSD language specifications, one for the mashup language, one for the description of mashup components. The generation of the latter is based only on component features (see Figure 3). The mashup language enables the design of mashups whereas the descriptor language enables the description and registration of components with the runtime platform (we will see this aspect later).

## 5. CUSTOM DMT IMPLEMENTATION

The feature-driven design approach together with the execution environment enable the conceptual design and automatic generation of custom DMTs. In the following, we describe the configurable runtime and design-time environments.[4]

[4]For code downloads and additional details (e.g., the complete list of feature definitions and the unified mashup language XSD), we refer the reader to our online resource (http://goo.gl/wxDOK).

**ALGORITHM 1:** generateLanguage

**Data:** Set of selected feature names *FnameSel,* feature knowledge base *F,* set of Boolean variables *FB*

**Results:** $\langle compositionLang, componentDescLang \rangle$ containing the generated XSD mashup language specification and XSD component description language specification, or *null* if the there are conflicts among the constraints of the selected features

```
1  compositionLang = languageBase; //languageBase is a global variable
2  CONSTR = baseConstraints; //baseConstraints is a global variable
3  Fsel = {f_j | f_j ∈ F, f_j.name ∈ FnameSel}; //load sel. features from knowledge base F
4  for each fb ∈ FB //set values of Boolean variables in FB
5     fb.val = (fb.name ∈ FnameSel) ? true : false;
6  for each f ∈ Fsel //construct set of constraints to be checked
7     CONSTR = CONSTR ∪ f.Constr;
8  if (checkSoundness(CONSTR, FB) == false) then //check soundness
9     return null; //interrupt processing if constraint conflicts occur
10 for each f ∈ Fsel
11    FRAGMENTS = FRAGMENTS ∪ returnIncludes(f); //construct set of fragments IDs
12    SETCARDINS = SETCARDINS ∪ returnSets(f); //construct set of setCardin. opers
13    ADDVALUES = ADDVALUES ∪ returnValues(f); //construct set of possible values for attributes
14 includeFragments(FRAGMENTS, compositionLang); //construct composition language
15 updateCardinalities(SETCARDINS, compositionLang); //update cardinalities
16 setAttributeDomains(ADDVALUES, compositionLang); //update domains of attributes
17 //construct result set by generating also the component description language
18 return ⟨compositionLang, extractDescLang(compositionLang)⟩;
```

### 5.1. The DMT Development Tool

In the previous section we explained that "composing" a custom mashup language with the described feature-driven approach corresponds to selecting the set of desired language features. Accordingly, the core of the DMT development tool is represented by a list of checkboxes, one for each conceptual feature in the feature knowledge base, that allow the designer to interactively select and deselect features and to check the soundness of a selected feature set (see Figure 4). If the soundness check is successful, the designer can trigger the mashup language generation algorithm (Algorithm 1). The generation algorithm generates the XSD specifications of the mashup language and then the component descriptor language along with the DMT configuration file (containing all feature selections) store them as a DMT configuration package on the MDK server. After completion, the generation process returns the URL of the configuration package, where the developer can retrieve his/her custom DMT design. Given this URL, the domain-specific mashup editor and runtime environment configure their parameterized cores.

In addition to the configuration of the mashup language, the DMT development tool also allows the developer to upload a domain-specific syntax descriptor file (in essence, a simple XML file that associates a given domain-specific graphical syntax, that is, an image) to any of the mashup language constructs described in the UMM in Figure 3, We will see later in this section how such custom syntax elements are used to decorate the built-in constructs of the mashup editor.

*Implementation.* The DMT development tool is implemented as a common Web application. The constraint verification algorithm is implemented in JavaScript and runs directly inside the client browser. The actual language generation algorithm is implemented as a Java servlet executed at the server side. Both algorithms accept as input the set of identifiers of the selected features and fetch the necessary feature details from an extensible, server-side feature knowledge base containing the XML

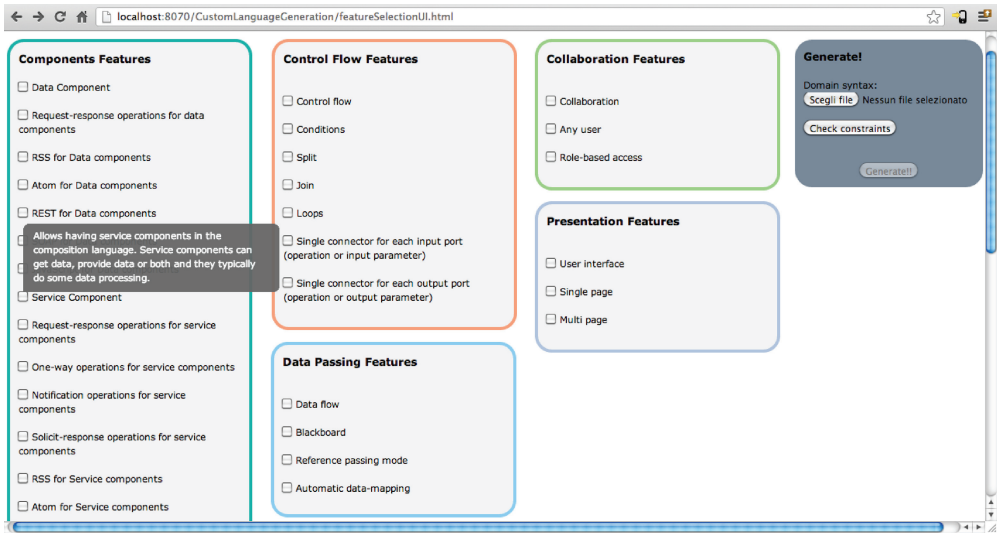


Fig. 4. The DMT development tool for language feature selection (features are grouped by design concerns). Cursor mouse-over events cause the tool to display feature-specific explanations. The upload feature at the right allows the configuration of the DMT editor with domain-specific syntax designs.

```
<feature name="data_flow" label="Data flow">

  <description> The composition paradigm is data flow, that is, it is possible
       to explicitly define the flow of the data among components opearations.
       In this case the data passing and the control flow overlap since
       operations triggering depends on the data flow.
  </description>

  <specification>

    <include fragments="dfConnectorDef, dfConnectorType,
         dfSourceOutputParameter, dfTargetInputParameter" />

  </specification>

  <constraints>NOT(control_flow)</constraints>
</feature>
```

Fig. 5. The dataflow composition feature definition as stored in the feature knowledge base.

definition of each feature (http://goo.gl/uxhKQ). Figure 5 shows an example XML feature definition. Language generation is based on an XSD serialization of the UMM described in Figure 3 [Soi et al. 2014].

### 5.2. The Parameterized Runtime Environment

The runtime environment is in charge of executing the domain-specific mashups. Mashup specifications can be expressed in any language the DMT development tool can produce. In order to be able to run different types of mashups, the runtime environment comes as a single implementation that implements all features introduced in Section 4.2 and that is able to adapt its behavior to a given DMT configuration. How exactly the individual features are executed, that is, how a mashup is executed starting from its specification, gives the specification its operational semantics.

The functional architecture of the runtime environment, that determines the operational semantics is illustrated in Figure 6. The environment is split into two parts, a client-side and a server-side part, and has at its core a mashup engine, in charge of parsing a mashup specification and instantiating the mashup accordingly. The engine

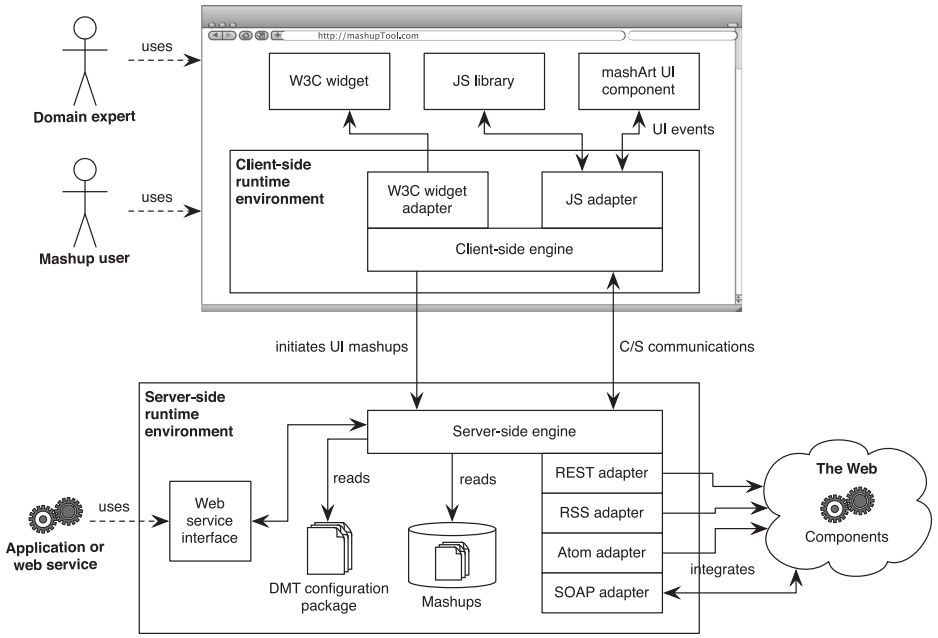


Fig. 6. Functional architecture of the runtime environment for domain-specific mashups (see Figure 2).

too is split into a server-side engine and client-side engine, where the server-side engine manages backend features (e.g., Web services orchestrations) and the client-side engine manages frontend features (e.g., UI component synchronization). It is the engine that understands the constructs in the mashup specification and knows how to execute the respective actions, implementing the various mashup features (invoking components, managing control flow, passing data, etc.). A set of adapters enables the engine to communicate with the different mashup component types supported, both at the client side (widgets, JS, and mashArt components) and the server side (RSS/Atom feeds, RESTful, and SOAP Web services). The adapters mediate between the engine-internal data format and the individual components' data formats and communication protocols.

Mashups stored in the internal mashup repository can be instantiated via two different channels, depending on the type of mashup: UI mashups are initiated by the domain expert or mashup user via a common Web browser using a simple management UI, while data or service mashups (without interactive UI) can be instantiated via a dedicated Web service interface for external applications or Web services, or via the management UI that allows the user to manually provide the inputs, otherwise expected to arrive through the Web service interface, and to view computed results.

The execution logic of a mashup upon an instantiation request is as follows.

(1) The server-side engine fetches the requested mashup specification from the repository and loads the respective DMT configuration package telling the engine which runtime features are required.
(2) Only for mashups with UI:
    (a) the server-side engine returns to the browser an HTML template including the JavaScript implementation of the client-side runtime environment along with the DMT configuration of the client side;
    (b) the client-side engine instantiates all UI components, causing their rendering inside the HTML template, initializes them according to their startup

configuration, and sets up all necessary event handlers. Client-side UI events are kept inside the client, unless an event specifically triggers a server-side action.

(3) From this point on, requests arriving to the server-side engine from the browser or arriving via the Web service interface are treated similarly (only responses are delivered via different channels):
   (a) the engine identifies the components to be invoked on the server-side; in the case of control-flow-based mashups, the first nonexecuted operation of each flow is chosen; while in the case of dataflow-based mashups, all the operations that have their necessary inputs filled are chosen;
   (b) the engine invokes the chosen components by suitably instructing the respective adapters, feeding them with the expected data as input, and buffering possible outputs for use in subsequent invocations;
   (c) step 3 is repeated until all operations of the invoked control/dataflows are processed and no asynchronous callback notifications are pending;

(4) the mashup terminates automatically for data and logic mashups. For UI mashups, the mashup stays alive listening for possible new requests from the client browser (step 3) and terminates only upon the closure of the client browser.

This instantiation logic allows for different instantiation models: short-living, stateless processing of server-side data or service mashups, stateful user sessions depending on the user interface instantiated in the client browser, and possibly long-living, stateful Web service orchestrations with asynchronous communications.

*Implementation.* Both the client- and server-side runtime environments are implemented in JavaScript. The server-side environment is based on Node.js[5], whose modularity and extensibility allow for component adapters (in charge of interacting with RSS, Atom, REST, and SOAP services) also being developed in different languages. In our current prototype, we implement them as Java-based RESTful Web services deployed in their own Apache Tomcat Web server running on the same machine as Node.js. The client-side adapters (for JavaScript components and W3C widgets) are instead entirely implemented in JavaScript.

The communication between the client-side and the server-side engine is enabled by a WebSocket [W3C 2013] exposed by the server-side engine, whose address is sent back to the client-side engine at startup. Communications among the two parts occur via direct messages over the WebSocket protocol. This enables efficient support for long-running processes, asynchronous interactions, and real-time applications.

### 5.3. The Parameterized Mashup Editor

The natural companion of a mashup runtime environment is a design environment, namely a mashup editor, that enables developers to design mashups that can be executed in the runtime environment. In the case of domain-specific mashups, this design environment assumes a special role in that it is the frontend of the mashup platform toward the domain experts. As such, its design requires particular attention so as to effectively convey the domain concepts, activities, and terminology that make the domain expert feel comfortable. Typically, mashup editors are graphical and model-based.

The MDK supports two different kinds of editors: (i) *custom editors* developed by third parties and able to produce mashups in any of the languages designed with the MDK and (ii) a *parameterized editor* that can be tailored to specific domains and that comes as part of the MDK. The former are able to provide the best tailoring of the design environment to the particular needs, practices, and terminology of a given

[5]Node.js is a framework for scalable, server-side JavaScript Web applications: http://nodejs.org/.

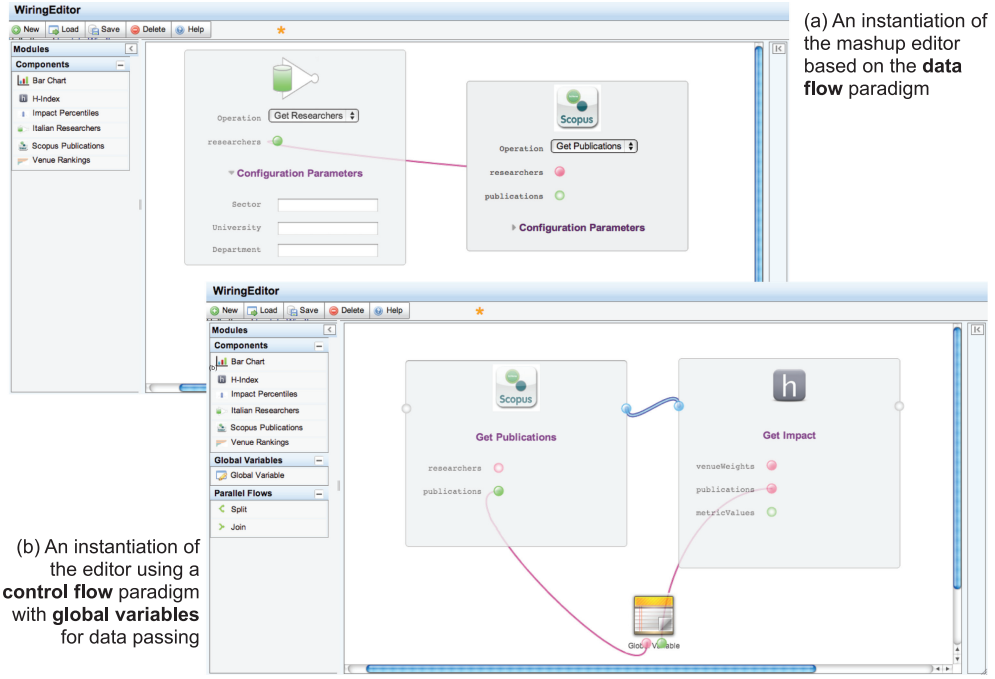


Fig. 7. The customizable mashup development environment. The two screenshots illustrate how the environment changes in function of the chosen mashup language.

domain. The latter is meant for fast prototyping and as an extensible basis for the development of custom editors. It provides a simple, Web-based graphical modeling environment supporting all the modeling constructs expressed in the UMM and that can be constrained to support only subsets of constructs, namely, those implementing a given mashup language. The outputs of the editor are mashups that comply with the chosen mashup language. Like for the runtime environment, the parameterized editor is configured with a DSM configuration package that uniquely identifies a designed mashup language and tells the editor which of its features to expose. The configuration package also includes a reference to a possible domain syntax specification, that is, an association of domain-specific graphical icons to the constructs in the UMM.

Figure 7 shows two screenshots of the editor when configured with two different mashup languages. Available modeling constructs are listed in the toolbar at the left of the figure and can be dragged and dropped onto the modeling canvas. Figure 7(a) supports dataflow-based mashups (wires are defined among parameters) with components possibly having multiple operations and configuration parameters. Figure 7(b) instead shows the support of control-flow-based mashups (wires are defined among operations) with global variables, joins, splits, and components with only single operations and without configuration parameters. In both cases, components display a domain-specific icon.

*Implementation.* The implementation of the parameterized editor builds on the WireIt library (http://neyric.github.com/wireit/docs/), a JavaScript library for the development of Web-based graphical editors. We extended the core of the library, that is, the reusable editor with support for drag-and-drop modeling, with: (i) support for all the language constructs of the UMM (e.g., with global variables and different types of wires to distinguish control-flow and data passing connectors) and (ii) a

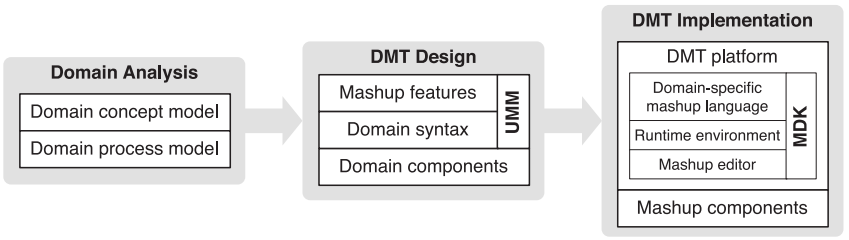


Fig. 8. Methodology for the development of domain-specific mashup tools with the proposed MDK.

self-configuration module, that loads the DMT configuration package at startup and initializes the editor's modeling language accordingly. The module also connects to the component registry to fetch the component descriptors of the components referenced in the configuration package and to be made available as modeling constructs in the toolbar.

## 6. METHODOLOGY

We have seen that developing a DMT is not an easy task and requires taking many design decisions related to the mashup language and its underlying mashup model, the development and runtime environments, the mashup components, etc. In addition, it is crucial to understand how and where to add domain specifics to the tool so as to effectively address the needs of the target domain and its domain experts. Next, we detail the lessons learned in the last six years and during the development of ResEval Mash [Daniel et al. 2012] into a methodology for the development of DMTs.

### 6.1. Developing Domain-Specific Mashup Tools

Figure 8 graphically summarizes the methodology, which contextualizes the instruments described throughout this article in the overall development process that we structure into three core activities.

(1) *Domain analysis*. This is the most important activity of the whole development. Its goal is to create an understanding of the domain and to identify and model those aspects that can be supported via a mashup platform. The outputs of this activity are two artifacts:
   (a) a domain concept model that expresses domain data and relationships (the concepts are the core of each domain and the specification of domain concepts allows the mashup platform to understand what kind of data objects it must support, both at platform and component level); generic mashup platforms typically do not have any knowledge of the data processed in their mashups, for example, domain-specific platforms may focus on a limited set of data objects only and take over tedious tasks like data mapping among components;
   (b) a domain process model, that expresses classes of domain activities and, possibly, ready processes; domain activities and processes represent the dynamic aspect of the domain and operate on and manipulate the domain concepts (in the context of mashups, we can map activities and processes to reusable components of the platform or classes of components).

(2) *DMT design*. Given a domain concept model and a domain process model, the next step is to understand how this representation of the domain maps to the capabilities provided by mashups and mashup platforms. This is the goal of the DMT design, which again produces three artifacts.
   (a) A set of *mashup features* is necessary to translate the development of composite domain processes into the development of mashups. The set of features is chosen

from the set of features supported by the MDK, possibly taking into account the structure and constructs of the UMM. It is here where decisions like dataflow versus control flow, RSS versus Atom, one operation per component versus multiple operations per component, etc., are made.

(b) A definition of domain-specific components represents recurrent domain processes to be made available as reusable resources of the mashup tool. The domain-specific components enable domain experts to mash them up into new, composite domain process, that is, domain-specific mashups. Components are typically described in terms of provided functionalities (operations) and data items operated on (inputs and outputs).

(c) A domain syntax definition equips each construct of the UMM for the chosen mashup features with its own, graphical symbol proper of the domain (where possible). Syntax elements can be assigned to classes of components (e.g., data sources) or directly to components (e.g., a DBLP source). The idea underlying the domain syntax is that visual metaphors the domain expert is acquainted with better covey the respective component functionalities.

(3) *DMT implementation*. This is the activity where the actual DMT is produced. The implementation is generally the most time-consuming activity, but also the activity that benefits most from the availability of the MDK. In fact, starting from the mashup features and domain syntax designed in the previous step, the MDK enables the automatic generation of the domain-specific mashup language and the respective runtime environment and mashup editor. The development of the mashup components is outside the scope of the MDK but assisted by the availability of the component description language, that tells developers which features of each given component technology can be used in the DMT. Ready components can then be made available to domain experts by registering them with the platform through the MDK's component registration tool.

Of course, the preceding activities do not mandatorily have to follow the waterfall model hinted at by Figure 8. The figure rather illustrates the conceptual flow of activities and artifacts. In practice, all activities may overlap or be iterated several times until a stable and satisfactory result is achieved.

### 6.2. Developing Mashup Components

What is out of the scope of the aforesaid methodology and this article is the development of the components to be mashed up. These encapsulate important pieces of domain knowledge (domain processes) in a reusable form and are the basic bricks for the development of domain-specific mashups. Their development can therefore not be automated as the development of the DMT using them, and instead requires profound domain and technology knowledge by expert component developers implementing, for example, Web services or UI widgets.

The availability of the MDK and its methodology, however, facilitates mashup component development in three ways.

—The domain process model and the according identification of domain components provide clear and well-defined requirements for the development of components. This development may require either the mere identification of readily available components, the wrapping of existing components into any of the supported component technologies, or the development from scratch of new components.

—The component description language generated contextually to the mashup language tells the developer which features of each component technology (e.g., synchronous versus asynchronous operations) are compatible with a given DMT design. This fosters consistent and easy-to-test component designs.

```
<component id="C1" name="Impact" type="service" binding="SOAP"
    endpoint="http://.../impact" class="metrics" supportReferencePassing="yes"
    syntax="http://.../impact.png">

  <operation id="OP1-1" name="Get Impact" type="request-response"
       reference="getImpact">
    <inputParameter id="I1-1" name="venueWeights" manualInput="no"
       optional="no">
      <has_dataType ref="venueWeights" />
    </inputParameter>
    <inputParameter id="I2-7" name="publications" manualInput="no"
       optional="no">
      <has_dataType ref="publications" />
    </inputParameter>

    <outputParameter id="O1-1" name="impact" >
      <has_dataType ref="metricValues" />
    </outputParameter>
  </operation>

</component>
```

Fig. 9. A simplified component descriptor for the registration of an impact SOAP service with a DMT.

—The support for standard technologies by the MDK eases the development task, in that it does not require component developers to learn proprietary component models or technologies.

In order to make components available to a given DMT, two additional artifacts are needed, next to the components themselves. First, a component descriptor, tells the DMT which features and technologies a component uses; second, a set of technology adapters is able to manage the interaction with components and to make them accessible from the DMT (see Figure 2). These latter already come with the MDK.

For example, to "componentize" an existing SOAP service, it is sufficient to create an XML descriptor similar to the one shown in Figure 9, which describes the Impact component needed in the research evaluation scenario presented earlier. The descriptor specifies the properties and interface for the computation of impact value, starting from a list of publications and a set of venue weights passed as input.

Component registration is done through a simple Web interface (the *component registration tool*), allowing the developer to upload a component descriptor and to associate it with a given DMT. Components are stored in their own repository of the MDK. During the registration phase, components can also be grouped into logical categories based on the structure of the domain process model (e.g., data sources, metrics, visualization widgets, etc.). The category of a component is stored in the class property of the component entity of the UMM. Similarly, the syntax property may associate a domain-specific graphical syntax element to a component.

## 7. CASE STUDIES

Next, we show the MDK at work with the help of two case studies.

### 7.1. ResEval Mash Reloaded

We recall the *ResEval Mash* platform described in Section 2.1. It is a mashup platform for research evaluation, that is, for the assessment of the productivity or quality of researchers, teams, institutions, journals, and the like. The platform is specifically tailored to the need of sourcing data about scientific publications and researchers from the Web, aggregating them, computing metrics (also complex and ad hoc ones), and visualizing them. ResEval Mash is a hosted mashup platform (http://open.reseval.org/) with a client-side editor and runtime engine, both running inside a common Web browser.

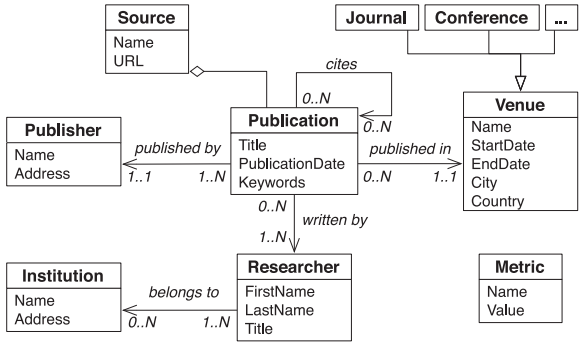


Fig. 10. The domain concept model of ResEval Mash as derived in Daniel et al. [2012].

*7.1.1. Analysis.* The first step in the MDK-based development of a domain-specific mashup tool similar to ResEval Mash is the analysis of the domain, producing the domain concept model and the domain process model as output.

We express the domain concept model as a conventional entity-relationship diagram. For instance, Figure 10 illustrates the main concepts we can identify in our reference scenario, detailing entities, attributes, and relationships. The core element in the evaluation of scientific production and quality is the *publication*, typically published in the context of a specific venue, such as, a conference or journal, by a publisher. It is written by one or more researchers belonging to an institution. Increasingly—with the growing importance of the Internet as information source for research evaluation—also the source (e.g., Scopus, the ACM Digital Library, hence or Microsoft Academic) from which publications are accessed is gaining importance, as each of them typically provides only a partial view on the scientific production of a researcher, hence the choice of the source will affect the evaluation result.

As for the activities that act upon these concepts, we can identify the following high-level domain process model in the form of a simple list of activity types typical for research evaluation.

(1) *Source extraction activities*. They are like queries over digital libraries such as Scopus or Scholar. For example, a typical query may take a set of researchers and extract publications and citations for every researcher from Scopus.
(2) *Metric computation activities*. Given an institution, venue, researcher, publication, or a set thereof, metric computation activities compute a metric measuring some form of scientific impact. For example, a typical metric is the h-index for researchers or the one that determines the percentile of a given value based on a given distribution.
(3) *Aggregation activities*. These define groups of items (e.g., researchers or scientific articles) based on some property (e.g., affiliation).
(4) *Filtering activities*. They process some input data and return filtered data as output based on some filtering criterion. For example, we can filter researchers based on their nationality or affiliation or based on the value of a metric.
(5) *Presentation activities*. Presentation activities correspond to activities that plot information on researchers, venues, publications, and related metrics for human inspection.

The domain concept model and the domain process model together are the input for the next phase in our methodology, namely the design of the DMT.

*7.1.2. Design: ResEval Mash, Simply.* If we instantiate the preceding domain process model, we obtain a set of requirements for the concrete implementation of domain components. For example, we might want to use a Microsoft Academic Publications component as an instance of source extraction activity with a dedicated configuration port that allows the setup of the researchers for which publications are to be loaded from Microsoft Academic. This component can be implemented as SOAP Web service. Similarly, we can implement Web services for an Italian Researchers component (source extraction activity), a Venue Ranking component (source extraction activity), an Impact component (metric computation activity), an Impact Percentiles component (metric computation activity), and a UI widget for a Bar-Chart component.

The choices of which concrete components to use and how to implement them (e.g., SOAP versus RESTful Web services versus UI widgets) along with the target mashups to be developed imply the need for a set of mashup features to be supported by the DMT under development. Specifically, for the implementation of ResEval Mash we identify the following mashup features.

—*Component features* consist of data_component (to fetch data from the Web), request_response_for_data, REST_for_data, SOAP_for_data, service_component (for the implementation of metrics, filters, and aggregations), request_response_for_service, one_way_for_service, notification_for_service, REST_for_service, SOAP_for_service, UI_component (for the visualization of outputs), one_way_for_UI, javascript_for_UI, manual_input, configuration_param, 1_operation_per_component, N_input_param_per_operation, N_output_param_per_operation.
—*Data passing features* include data_flow, by_reference (given the high amount of data involved in the research evaluation scenario described in Figure 1, we opt for components that run on a centralized server, operate on a shared memory implementing a data schema similar to the domain concept model illustrated in Figure 10, and exchange only references to the shared memory).
—*Presentation features* are comprised of user_interface, single_page (being that our target research evaluation mashups are an instance of data mashups, the visualization of computed outputs is relatively simple and does not require complex interactive user interfaces).

The last step of the design is the definition of a domain syntax for the mashup editor, that is, of a set of graphical icons for components. The screenshot in Figure 12 illustrates a set of possible icons for our research evaluation domain.

*7.1.3. Implementation and Use.* At this point, the platform designers can start with the actual implementation of the new DMT. As discussed in Section 5.1, DMT implementation with the MDK only requires the configuration of the DSM language via the selection of its mashup language features, a task supported via the language composition tool shown in Figure 4. The result of this conceptual development process is the domain-specific mashup language, along with an according runtime environment and mashup editor.

Figure 11 illustrates an excerpt of the XML serialization of a mashup implementing the research evaluation scenario introduced in Figure 1; the shown XML fragment is an instance of the XSD schema of the domain-specific mashup language automatically generated by the MDK from the aforesaid set of mashup features.

Figure 12 shows a screenshot of the generated domain-specific mashup editor, customized based on the features listed earlier. In particular, the figure illustrates the model of our reference research evaluation scenario. The editor provides all the functionalities necessary to implement the scenario and equips each component with

```
<mashup name="DepartmentProductivity">
  <component id="C1" name="Italian Researchers" type="data" binding="REST"
             endpoint="http://...">

    <configurationParameter id="CP1-1" name="Sector" manualInput="yes">
      <has_dataType ref="string" />
    </configurationParameter>
    [...]

    <operation id="OP1-1" name="Get Researchers" type="request-response"
               reference="getResearchers">
      <input id="I1-1" name="sector" dataType="string" optional="no"
           manualInput="yes" />
      <output id="O1-1" name="researchers" dataType="researchers"/>
    </operation>
  </component>

  [...]

  <component id="C9" name="bar Chart" type="ui" binding="javascript"
             endpoint="http>//...">
    <operation id="OP1-9" name="Plot" type="one-way"
               reference="plot">
      <input id="I1-9" name="data" dataType="dataSeries" optional="no"
           manualInput="no" />
    </operation>

  </component>

  [...]
  <constant id="CNST1" name="Sector" dataType="string" value="ComputerScience"
            feeds_configurationParameter="CP1-1"/>
  [...]

  <dfConnector id="DF1" source_output="O1-1" target_input="I1-2" />
  [...]
  <dfConnector id="DF9" source_output="O1-8" target_input="I1-9" />
</mashup>
```

Fig. 11. Excerpt of the XML serialization of the mashup implementing the scenario presented in Figure 1.

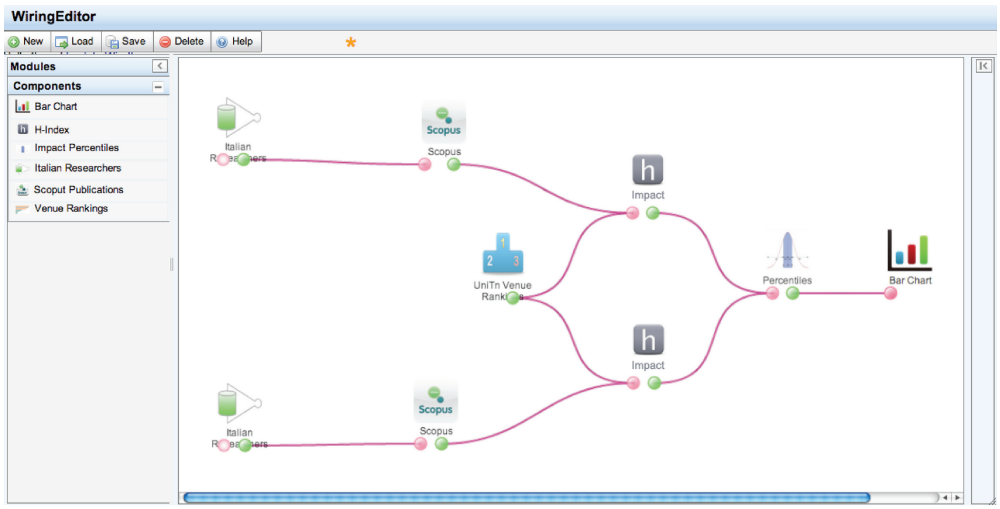


Fig. 12. The model of the mashup implementing our reference research evaluation scenario.

its own domain syntax representation. The runtime environment is obtained similarly to the mashup editor and is able to parse and run the XML definition of Figure 11.

For the implementation of the mashup components, is it enough to reuse the components already implemented for the original ResEval Mash project and to register them with the new DMT for research evaluation, without new development effort.

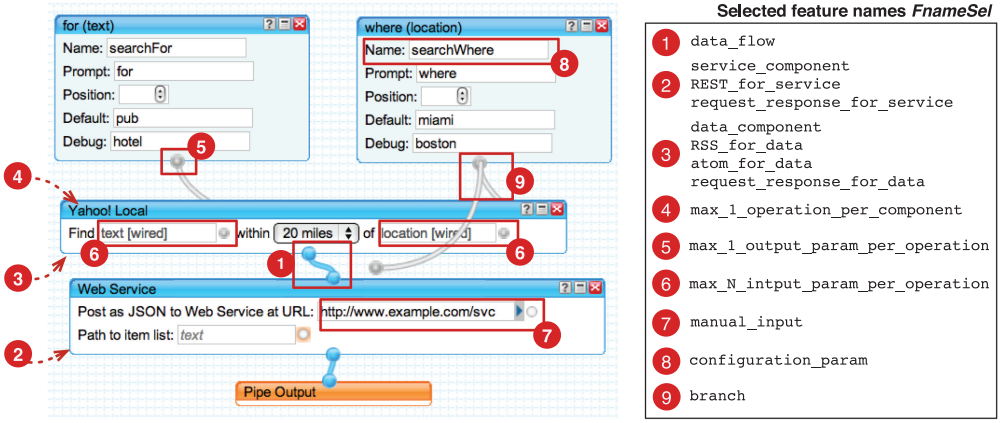


Fig. 13. Yahoo! Pipes example composition and set of respective language features.

### 7.2. Yahoo! Pipes

As a second example, we derive part of the mashup language underlying the popular mashup platform Yahoo! Pipes from an example modeled in its graphical editor. Pipes is a data mashup tool for the retrieval and processing of data feeds. Figure 13 shows an example Pipes model, that we use to analyze Pipes' language features (for simplicity, we focus on the features evident in the figure only).

Pipes is based on the data_flow paradigm. It supports data_component and service_component types to retrieve and process data. Data source component types are RSS_for_data and atom_for_data, while the only supported service component type is REST_for_service. Each component in Pipes provides exactly one function, that is, each component represents one single operation, namely max_1_operation_per_component. All operations are of type request-response (request_response_for_data and request_response_for_service). Each operation may have one or more inputs (max_N_input_param_per_operation) but one and only one output (max_1_output_param_per_operation). Manual inputs (manual_input) are used to fill the values of input fields, that is, of configuration_parameter(s). Some inputs can be fed with both an input pipe and a manually set constant value. Also in this example, the output of a component can be the source for an arbitrary number of dataflow connectors, allowing one to branch the dataflow into parallel flows. Input parameters, instead, have at most one input pipe, so there is no need for any merge feature.

Figure 14 illustrates the resulting mashup model as selection of constructs of the UMM, while Figure 7(a) shows a sreenshot of the suitably configured mashup editor.

## 8. STRENGTHS AND WEAKNESSES

With this article, we aim to ease the development of custom, domain-specific mashup tools and to make it cost effective, fast, and robust. The two case studies show how thinking in terms of the mashup language to develop, starting from a set of conceptual mashup language features, significantly lowers the complexity of developing mashup tools. In Soi et al. [2014], we provide another example of how to implement a platform like our former mashArt platform [Daniel et al. 2009], which also features UI and service components.

The benefiter of this work is the developer of mashup platforms, who can leverage on a sophisticated mashup platform development kit in an intuitive fashion: the proposed conceptual development approach does not only raise the level of abstraction of the

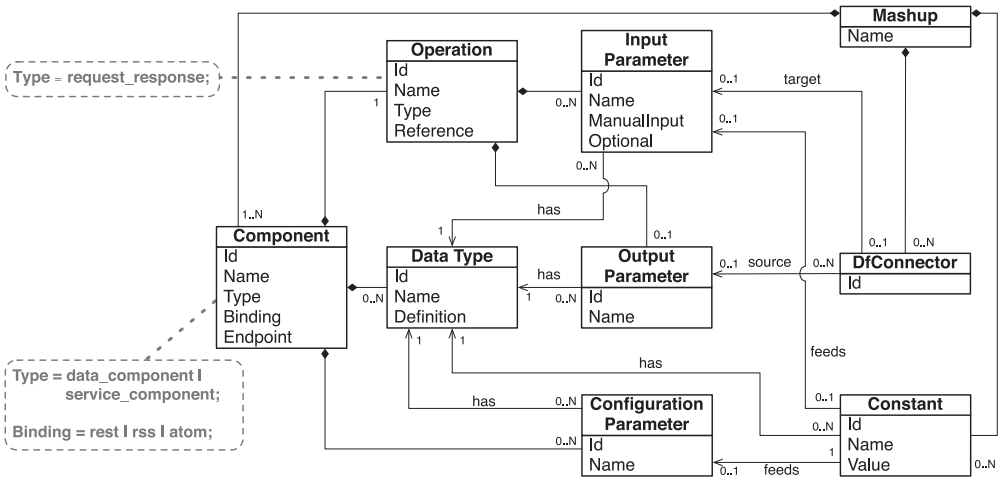


Fig. 14. Graphical model of the generated mashup language mimicking Yahoo! Pipes.

development task, but also does so without requiring the developer to learn any new development language or paradigm. The selection of language features is intuitive and self-explaining. The four tasks the developer needs to master are the following.

—*Domain and requirements analysis*. The thorough analysis of the domain of the mashup platform to develop and the identification of requirements are tasks that are common to all software projects. The MDK does not provide any specific tool support for this step; it does, however, clearly state which artifacts (domain concept model, domain process model, components, domain syntax) it requires as input.
—*Generation of mashup language and tool*. This step is the core of the proposed approach and is where the MDK provides most support. The simple translation of requirements into mashup language features allows the developer to obtain a working runtime environment with his/her own mashup language, a component registration dashboard with its own description language, and a preconfigured mashup editor.
—*Implementation of domain-specific components*. This step if fully up to the developer and needs to be done carefully. The input provided by the MDK is the component description language telling the developer how to implement components (e.g., which technologies and interaction patterns to use) and how to describe them.
—*Tailoring of the mashup editor*. Although the MDK generates a skeleton of a readily usable mashup editor, we consider its extension and tailoring almost mandatory to adequately address the needs and capabilities of domain experts.

As for the extensibility of the overall MDK, we acknowledge that in a context as young and evolving as mashups, it is easy for new requirements to emerge over time, thus requiring new features to be added to the platform. We cater for new requirements in two ways: First, the UMM and the feature knowledge base are designed in an extensible manner, allowing developers to add their own constructs and features in a declarative fashion (via suitable XSD and XML specifications). Runtime and design-time support for new features require the implementation of feature-specific handlers for the runtime environment (e.g., adapters) and for the mashup editor (e.g., new graphical constructs). This extensibility is further facilitated by our second measure, that is, the availability of the whole MDK as an open-source project hosted on GitHub and accessible via http://goo.gl/wxDOK.

As for the quality of generated mashup platforms and their suitability for end-user development, these are out of the control of the MDK, as they strongly depend on each individual implementation. The key concerns in this respect are the domain analysis and the design of the editor's UI. A profound understanding of a given domain and a well-conceived mapping of domain concepts and processes to mashup capabilities and suitable visualization paradigms are the prerequisites for the success of any domain-specific software instrument. In absence of a tailoring of the mashup editor to the target domain, the complexity of the editor is that of common model-driven mashup tools (see Figure 7).

Regarding the limitations of the presented work, we highlight four major aspects that we think could benefit from further development.

—*Mashup editor implementation*. The mashup editor is partly still under development, as it was not part of our original plan for the MDK. The current implementation still lacks support for minor features, such as, a custom syntax not only for components, and the overall user interface or user experience requires further improvement. However, as explained earlier, the editor is the most crucial interface toward the domain expert. As such, its UI paradigms, metaphors, constructs, terminology, etc., strongly affect the success of a whole DMT project. As explained before, we therefore consider the parameterized editor described in this article mostly as a starting point for the development of custom editors tailored to specific domains.

—*User management and multitenancy*. The current implementation of the MDK does not yet provide for proper user management and multitenancy. User management is, however, needed, for example, to associate users (domain experts) with their DMTs, or to set up access restrictions for the mashups developed by domain experts, especially if the MDK is to be provided as a service in a hosted fashion. This will also allow users to access their DMT without having to provide a reference to their DMT configuration package at each access, such as when opening the mashup editor.

—*Collaborative mashup development*. Our final goal is to foster end-user development via mashups. Independently of how simple mashup tools are, domain experts may still need some help, for instance, when they start using the tool or later on when facing tricky development scenarios. Collaborative development, that is, the possibility to have multiple domain experts designing a mashup together, is an effective instrument to facilitate knowledge exchange. Suitable support for collaboration may be added to both the UMM and the mashup editor.

—*Stand-alone deployment*. Alternatively to hosting the MDK and generated mashup tools on a remote server, the MDK could also generate a package containing all the software artifacts of a whole physical DMT (including the generated languages and the code of the editor and the runtime environment), that could be downloaded and deployed independently of the MDK on its own machine. This solution would also allow the developer to apply his/her own extensions to the mashup tool locally.

Our current focus is specifically on the improvement of the editor and on user management.

## 9. RELATED WORK

Although the requirement for more intuitive development environments and design support for end-users clearly emerges from research on End-User Development (EUD), for example, for Web services [Namoun et al. 2010a; 2010b], little is available to satisfy this need. There are essentially two main approaches to enable less skilled users to develop their own software: in general, development can be eased either by simplifying it (e.g., limiting the expressive power of a programming language) or by reusing knowledge (e.g., copying and pasting from existing algorithms). Among

the simplification approaches, the workflow/Business Process Management (BPM) community was one of the first to claim that the abstraction of business processes into tasks and control flows would allow also less skilled users to define their own processes. Yet, in our opinion, this approach achieved little success and modeling still requires training and knowledge. The advent of the Service-Oriented Architecture (SOA) substituted tasks with services, yet composition is still a challenging task even for expert developers [Namoun et al. 2010a; 2010b]. The reuse approach is implemented by programming libraries, services, or templates (such as generics in Java or process templates in workflows). It provides building blocks that can be composed to achieve a goal, or the entire composition (the algorithm, possibly made generic if templates are used), which may or may not suit a developer's needs.

Mashups aim to bring together the benefits of both simplification and reuse. In the case of DMTs, we aim to push simplification even further compared to generic mashup platforms by limiting the environment (and hence its expressive power) to the needs of a single, well-defined domain only. Reuse is supported in the form of reusable domain activities, that can be mashed up.

More specifically, Web mashups [Yu et al. 2008] emerged as an approach to provide easier ways to integrate services and data sources available on the Web [Hartmann et al. 2006], together with the claim to enable nonprogrammers to develop their own mashups. Three notable attempts in this context are Yahoo! Pipes, CRUISe, and the ServFace Builder. Yahoo! Pipes (http://pipes.yahoo.com) provides an intuitive visual editor that allows the design of data processing logics. Support for UI integration is missing, and support for service integration is still poor. Pipes operators provide only generic programming features (e.g., feed manipulation, looping) and typically require basic programming knowledge. The CRUISe project [Pietschmann et al. 2009] specifically focuses on composability and context-aware presentation of UIs, but does not support the seamless integration of UI components with Web services. The ServFace project (http://www.servface.eu) instead aims to support domain experts in composing semantically annotated Web services. The result is a simple, user-driven Web service orchestration tool, but UI integration and process logic definitions are rather limited and again basic programming knowledge is still required.

The idea of focusing on a particular domain and exploiting its specificities to create more effective and simpler development environments is supported by a large number of research works [Lédeczi et al. 2001; Costabile et al. 2004; Mernik et al. 2005; France and Rumpe 2005]. Mainly these areas are related to domain-specific modeling and domain-specific languages. In *Domain-Specific Modeling* (DSM), domain concepts, rules, and semantics are represented by one or more models, that are then translated into executable code. Managing these models can be a complex task that is typically suited only to programmers but that, however, increases productivity. This is possible thanks to the provision of domain-specific programming instruments that abstract from low-level programming details and powerful code generators that "implement" on behalf of the modeler. Studies using different DSM tools (e.g., the commercial MetaEdit+ tool and academic solution MIC [Lédeczi et al. 2001]) have shown that developers' productivity can be increased up to an order of magnitude. The goal of domain-specific mashup tools is to make domain-specific modeling accessible to end-users. In the *Domain-Specific Languages* (DSLs) context, we can find solutions targeting end-users, such as MS Excel macros for spreadsheets or MatLab for calculations. Although the introduction of the domain raises the level of abstraction of these languages, they still require dedicated training. Maximilien et al. propose Swashup [Maximilien et al. 2007] specifically for end-user service mashups, however, empirical evidence of which types of end-users the approach enables is not provided.

The specific problem of facilitating the development of custom DMTs has not been addressed before. Most contributions in the area of mashups and service-oriented computing focus on the design of individual languages taking into account, for example, quality of service [Mohabbati et al. 2011], adaptivity or context awareness [Hermosillo et al. 2012], energy efficiency [Hoesch-Klohe and Ghose 2010], and the like. We instead propose a language for the design of languages. The problem is very complex, but our analysis of a large set of mashup tools and practices has shown that the design space for nonmission-critical mashups (without fault handling, compensations, transactions, etc.) is limited and manageable, up to the point where we can provide mashup execution as a service for a large class of custom languages.

A proposal toward the standardization of a generic mashup language (the opposite of custom mashup languages), covering as many different uses as possible, is the Open Mashup Alliance's EMML (Enterprise Mashup Markup Language) specification [Open Mashup Alliance 2012]. The target of the initiative is, however, different: data mashups. In our view the key novelty mashups brought to software integration is integration at the UI layer. Therefore, the focus on data mashups only is too narrow and the language has already grown very complex (without concrete vendor adoption). Aghaee and Pautasso [2011], on the other hand, focus on the design of a generic mashup component description language and do not elaborate on their composition into mashups. Their model contains technology aspects (e.g., component wrappers), that we think are instead a runtime aspect that should not affect the component description. We only propose the use of component types and bindings.

Differently from these efforts and considering the growing importance of cloud computing and composition as a service (such as mashups or scientific workflows [Blake et al. 2010]), we expect the importance of the customization of composition languages—as a means of diversification—to grow. In this context, Trummer and Faltings [2011] work, for example, toward composition as a service; yet, they approach the problem from the provider side and study the optimal selection of service composition algorithms—a task that could be eased if customers were allowed to customize the composition language to be executed.

## 10. LESSONS LEARNED AND FUTURE WORK

The evaluation of the suitability of domain-specific mashups to nonprogrammers and especially the user studies we conducted in Daniel et al. [2012] and De Angeli et al. [2011] confirm the viability of the intuition that going from generic to domain-specific mashup tools lowers the barriers to mashup development, up to the point where domain experts are able to develop their own mashups. However, we also learned that nonprogrammers generally lack: (i) data modeling knowledge necessary to understand how to transform or map outputs to inputs, (ii) algorithmic knowledge necessary to express complex mashup logics, and (iii) generic software development knowledge needed to distinguish design-time from runtime concerns and to deploy ready mashups.

We approached the lack of data modeling knowledge in ResEval Mash by passing data by reference and implementing research evaluation services that all comply with the domain concept model depicted in Figure 10 and operate on a shared memory using that model as data model. Passing data by reference also enables the development of data-intensive mashup tools, but in the context of EUD, its major contribution is to eliminate the need for data mappings and transformations.

The lack of algorithmic knowledge we approach in another, related thread of research: In Roy Chowdhury et al. [2012], we describe a plugin for mashup tools that enables the interactive, contextual recommendation and reuse of mashup model patterns [Roy Chowdhury et al. 2011]. The goal of this line of research is to foster reuse of composition knowledge, to facilitate the transmission of good modeling practice (that

is, learning), and to speed up mashup development. The user studies we conducted in the context of the EU FP7 project OMELETTE (http://www.ict-omelette.eu) by running the plugin in two different mashup tools (Yahoo! Pipes and Apache Rave) provide statistically relevant evidence that recommending composition knowledge can significantly accelerate development times and lower development complexity (number of user interactions required to complete a mashup). We strongly believe that the joint application of domain-specific mashup approaches and assisted development techniques may further help lowering the barrier to EUD.

We did not yet approach the lack of software development knowledge, but from the feedback we got during our user studies we derived the requirement for live mashup environments, that is, for mashup environments that do not explicitly distinguish design time from runtime or deployment time, as is typical in conventional, model-driven approaches. The DashMash system by Cappiello et al. [2011] provides a first example of this paradigm and a future direction for our MDK.

Another important lesson we learned in this research is that EUD tools must be so-called *gentle slope systems* [Myers et al. 1992] that allow the users to learn the tool step-by-step without a steep learning curve. We have experienced this during our own user studies and understood that it is particularly true for end-users. End-users are generally not prepared to or interested in learning new conceptual and/or technological notions to use a tool. Users must be able to learn by attempts and immediately perceive the results of their learning effort, also without specific training.

As for the development of DMTs, whose facilitation is the final goal of this article, the development of the MDK was an extraordinary learning process in its own right. From the development of our own mashup tools (Mixup, mashArt, MarcoFlow, ResEval Mash, the OMELETTE tools) we easily derived a set of recurrent patterns for both language and architecture. Yet, the finding that if mashup features are properly identified and formalized as reusable mashup model fragments, the whole development process of a complete DMT can essentially be boiled down to the conceptual design of its internal mashup language was unexpected even to ourselves. We consider this one of the major lessons learned through this work.

Although the main motivation of the work is to facilitate the development of domain-specific development instruments for end-users, it is important to note that the described approach is general in nature and can also be applied to the development of custom mashup tools for expert programmers, for the fast prototyping of composition platforms, as starting point for the development of more advanced composition instruments or of customizable Business-Process-as-a-Service (BPaaS) cloud platforms, for the development of scientific workflow systems similar to myExperiments (http://www.myexperiment.org/), or simply for didactic purposes.

Next, we aim to expand the set of features supported by our MDK and to refine the prototype implementations of the developed tools, in particular of the editor, so as to be able to make the whole platform available as an open-source project via http://goo.gl/wxDOK and to evolve it with the help of the community. In a second step, we aim to complement the MDK with the assisted development approach that interactively recommends mashup model patterns and to perform suitable user studies to assess the benefits of the two approaches together.

## REFERENCES

S. Aghaee and C. Pautasso. 2011. The mashup component description language. In *Proceedings of the 13th International Conference on Information Integration and Web-Based Applications and Services (iiWAS'11).* ACM Press, New York, 311–316.

L. Baresi and S. Guinea. 2010. Consumer mashups with mashlight. In *Proceedings of the 3rd European Conference on ServiceWave (ServiceWave'10).* Springer, 112–123.

M. Blake, W. Tan, and F. Rosenberg. 2010. Composition as a service. *IEEE Internet Comput.* 14, 1, 78–82.

C. Cappiello, M. Matera, M. Picozzi, G. Sprega, D. Barbagallo, and C. Francalanci. 2011. DashMash: A mashup environment for end user development. In *Proceedings of the 11th International Conference on Web Engineering (ICWE'11)*. Springer, 152–166.

M. F. Costabile, D. Fogli, G. Fresta, P. Mussio, and A. Piccinno. 2004. Software environments for end-user development and tailoring. *PsychNol. J.* 2, 1, 99–122.

F. Daniel, F. Casati, B. Benatallah, and M. Shan. 2009. Hosted universal composition: Models, languages and infrastructure in mashArt. In *Proceedings of the 28th International Conference on Conceptual Modeling (ER'09)*. Springer, 428–443.

F. Daniel, M. Imran, S. Soi, A. D. Angeli, C. R. Wilkinson, F. Casati, and M. Marchese. 2012. Developing mashup tools for end-users: On the importance of the application domain. *Int. J. Next-Generat. Comput.* 3, 2.

F. Daniel and M. Matera. 2014. *Mashups: Concepts, Models and Architectures*. Springer.

F. Daniel, S. Soi, S. Tranquillini, F. Casati, C. Heng, and L. Yan. 2011. Distributed orchestration of user interfaces. *Inf. Syst.* 37, 6, 539–556.

A. De Ngeli, A. Battocchi, S. Roy Chowdhury, C. Rodriguez, F. Daniel, and F. Casati. 2011. End-user requirements for wisdom-aware EUD. In *Proceedings of the 3rd International Conference on End-User Development (IS-EUD'11)*. 245–250.

M. Feldmann, T. Nestler, U. Jugel, K. Muthmann, G. Hubsch, and A. Schill. 2009. Overview of an end user enabled model-driven development approach for interactive applications based on annotated services. In *Proceedings of the 4th Workshop on Emerging Web Services Technology (WEWST'09)*. ACM Press, New York, 19–28.

R. France and B. Rumpe. 2005. Domain specific modeling. *Softw. Syst. Model.* 4, 1–3.

J. Gregorio and B. De Hora. 2007. The atom publishing protocol. http://tools.ietf.org/html/rfc5023.

B. Hartmann, S. Doorley, and S. Klemmer. 2006. Hacking, mashing, gluing: A study of opportunistic design and development. *Pervas. Comput.* 7, 3, 46–54.

G. Hermosillo, L. Seinturier, and L. Duchien. 2012. Creating context-adaptive business processes. In *Proceedings of the 8th International Conference on Service-Oriented Computing (ICSOC'10)*. Springer, 228–242.

K. Hoesch-Klohe and A. Ghose. 2010. Carbon-aware business process design in Abnoba. In *Proceedings of the 8th International Conference on Service-Oriented Computing (ICSOC'10)*. Springer, 551–556.

A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. 2001. Composing domain-specific design environments. *IEEE Comput.* 34, 11, 44–51.

E. M. Maximilien, H. Wilkinson, N. Desai, and S. Tai. 2007. A domain-specific language for web APIS and services mashups. In *Proceedings of the 5th International Conference on Service-Oriented Computing (ICSOC'07)*. Springer, 13–26.

M. Mernik, J. Heering, and A. M. Sloane. 2005. When and how to develop domain-specific languages. *ACM Comput. Surv.* 37, 4, 316–344.

B. Mohabbati, D. Gasevic, M. Hatala, M. Asadi, E. Bagheri, and M. Boskovic. 2011. A quality aggregation model for service-oriented software product lines based on variability and composition patterns. In *Proceedings of the 9th International Conference on Service-Oriented Computing (ICSOC'11)*. Springer, 436–451.

B. Myers, D. C. Smith, and B. Horn. 1992. Chapter report of the 'end-user programming' working group. In *Languages for Developing User Interfaces*, Jones and Bartlett, Boston, 343–366.

A. Namoun, T. Nestler, and A. De Ngeli. 2010a. Conceptual and usability issues in the composable web of software services. In *Proceedings of the 10th International Conference on Current Trends in Web Engineering (ICWE'10)*. Springer, 396–407.

A. Namoun, T. Nestler, and A. De Ngeli. 2010b. Service composition for non programmers: Prospects, problems, and design recommendations. In *Proceedings of the 8th IEEE European Conference on Web Services (ECOWS'10)*. 123–130.

M. Nottingham and R. Sayre. 2005. The atom syndication format. http://www.ietf.org/rfc/rfc4287.txt.

Open Ashup Alliance. 2012. Enterprise mashup markup language (EMML). http://www.openmashup.org/omadocs/v1.0/index.html.

S. Pietschmann, M. Voigt, A. Rumpel, and K. Meissner. 2009. CRUISe: Composition of rich user interface services. In *Proceedings of the 9th International Conference on Current Trends in Web Engineering (ICWE'09)*. Springer, 473–476.

S. Roy Chowdhury, F. Daniel, and F. Casati. 2011. Efficient, interactive recommendation of mashup composition knowledge. In *Proceedings of the 9th International Conference on Service Oriented Computing (ICSOC'11)*. Springer, 374–388.

S. Roy Chowdhury, C. Rodriguez, F. Daniel, and F. Casati. 2012. Baya: Assisted mashup development as a service. In *Proceedings of the 21st International Conference Companion on World Wide Web (WWW'12Companion)*. ACM Press, New York, 409–412.

Rss Dvisory Board. 2009. RSS 2.0 specification. http://www.rssboard.org/rss-specification.

S. Soi, F. Daniel, and F. Casati. 2014. Conceptual design of sound, custom composition languages. In *Web Services Foundations*. Springer, 53–79.

I. Trummer and B. Faltings. 2011. Dynamically selecting composition algorithms for economical composition as a service. In *Proceedings of the 9th International Conference on Service-Oriented Computing (ICSOC'11)*. Springer, 513–522.

R. Tuchinda, C. A. Knoblock, and P. A. Szekely. 2011. Building mashups by demonstration. *ACM Trans. Web* 5, 3, 16.

W3C. 2011. Widget packaging and configuration. W3C working draft. http://www.w3.org/TR/widgets/.

W3C. 2013. The websocket API. http://dev.w3.org/html5/websockets/.

J. Yu, B. Benatallah, F. Casati, and F. Daniel. 2008. Understanding mashup development. *IEEE Internet Comput.* 12, 44–52.

Received June 2013; revised December 2013; accepted March 2014