On the Systematic Development of Domain-Specific Mashup Tools for End Users

Muhammad Imran, Stefano Soi, Felix Kling, Florian Daniel,
Fabio Casati and Maurizio Marchese

Department of Information Engineering and Computer Science
University of Trento, Via Sommarive 5, 38123, Trento, Italy
`lastname@disi.unitn.it`

Abstract. The recent emergence of mashup tools has refueled research on *end user development*, i.e., on enabling end users without programming skills to compose their own applications. Yet, similar to what happened with analogous promises in web service composition and business process management, research has mostly focused on technology and, as a consequence, has failed its objective. Plain technology (e.g., SOAP/WSDL web services) or simple modeling languages (e.g., Yahoo! Pipes) don't convey enough meaning to non-programmers.
In this paper, we propose a *domain-specific* approach to mashups that "speaks the language of the user", i.e., that is aware of the terminology, concepts, rules, and conventions (the domain) the user is comfortable with. We show what developing a domain-specific mashup tool means, which role the mashup meta-model and the domain model play and how these can be merged into a domain-specific mashup meta-model. We exemplify the approach by implementing a mashup tool for a specific scenario (research evaluation) and describe the respective user study. The results of a first user study confirms that domain-specific mashup tools indeed lower the entry barrier to mashup development.

1 Introduction

Mashups are typically simple web applications that, rather than being coded from scratch, are developed by integrating and reusing available data, functionalities, or pieces of user interfaces accessible over the Web. ***Mashup tools***, i.e., online development and runtime environments for mashups, ambitiously aim at enabling non-programmers (regular web users) to develop their own applications.Yet, similar to what happened in web service composition, the mashup platforms developed so far either expose too much functionality and too many technicalities so that they are powerful and flexible but suitable only for programmers, or only allow compositions that are so simple to be of little use for most practical applications. For example, mashup tools typically come with *SOAP services*, *RSS feeds*, *UI widgets*, and the like. Non-programmers do not understand what they can do with these kinds of compositional elements. We experienced this with our own mashup and composition platforms, mashArt [4] and MarcoFlow [5], which we believe to be simpler and more usable than many composition tools but that still failed in being suitable for non-programmers. Yet, being amenable to non-programmers is increasingly important as the opportunity given by the wider and wider range of available online applications and the increased flexibility that is required in both businesses and personal life management raise the need for situational (one-use or short-lifespan) applications that cannot be developed or maintained with the traditional requirement elicitation and software development processes.

We believe that ***the heart of the problem*** is that it is impractical to design tools that are *generic enough* to cover a wide range of application domains, *powerful enough* to enable the specification of non-trivial logic, and *simple enough* to be actually accessible to non-programmers. At some point, we need to give up something. In our view, this something is generality, since reducing expressive power would mean supporting only the development of toy applications, which is useless, while simplicity is our major aim. Giving up generality in practice means narrowing the focus of a design tool to a well-defined *domain* and tailoring the tool's development paradigm, models, language, and components to the specific needs of that domain only.

In this paper, we therefore champion the notion of ***domain-specific mashup tools*** and describe what they are composed of, how they can be developed, how they can be extended for the specificity of any particular application context, and how they can be used by non-programmers to develop complex mashup logics within the boundaries of one domain. We provide the following contributions:

- We describe a *methodology* for the development of domain-specific mashup tools, defining the necessary concepts and design artifacts. As we will see, one of the most challenging aspects is to determine what is a domain, how it can be described, and how it can both constrain a mashup tool (to the specific purpose of achieving simplicity of use) and ease development. The methodology targets expert developers, who implement mashup tools.
- We detail and exemplify all *design artifacts* that are necessary to implement a domain-specific mashup tool, in order to provide expert developers with tools they can reuse in their own developments.
- We apply the methodology in the context of an *example mashup platform* that aims to support a domain most scientists are acquainted with, i.e., research evaluation. This prototypal tool targets domain experts.
- We perform a *user study*, in order to assess the viability of the developed platform and of the respective development methodology.

While we focus on mashups, the techniques and lessons learned in the paper are general and can be applied to other composition or modeling environments.

Next, we first introduce our reference scenario. In Section 3, we introduce our basic definitions and provide our problem statement. In Section 4 we outline the methodology we follow to implement the scenario and provide the necessary details. In Section 5 we describe the actual implementation of our prototype tool, and in Section 6 we report on a preliminary user study. In Section 7, we review related works. We conclude the paper in Section 8.

2 Scenario: Research Evaluation

As an example of a domain specific task, we now describe the evaluation procedure used by the central administration of the University of Trento (UniTN) for checking the productivity of each researcher who belongs to a particular department with respect to the average quality of researchers belonging to similar departments (i.e., departments in the same disciplinary sector) in all Italian universities. The comparison uses the following procedure based on one simple bibliometric indicator, i.e., a weighted publication count metric.

1. A list of all researchers working in Italian universities is retrieved from a national registry, and a reference sample of faculty members with similar statistical features (e.g., belonging to the same *disciplinary sector*) of the evaluated department is compiled.
2. Publications for each researcher of the selected department and for all Italian researchers in the selected sample are extracted from an agreed-on data source (e.g., Microsoft Academic, Scopus, DBLP, etc.).
3. The publications obtained in the previous step are then weighted using a venue classification. For each researcher a single weighted publication count parameter is thus obtained with a simple weighted sum of her publications.
4. The statistical distribution – more specifically, a negative binomial distribution – of the weighted publication count metric is then computed for the researchers' reference sample.
5. Each researcher in the selected department is ranked, based on her individual weighted publication count, over the computed statistical distribution, thereby estimating her percentile, i.e. the percentage of the researchers in the same disciplinary sector having equal or lower values for the specific metric.

The requirement we extract from this domain-specific scenario is that we need to empower people involved in the evaluation process (that is, the average faculty member or the administrative persons in charge of research evaluation) so that they are able to define and compare relatively complex evaluation processes, taking and processing data in various ways from different sources, and visually analyze and understand the results. As we need to extract, combine, and process data and services from multiple sources and represent the information with visual components, this task has all the characteristics of a *mashup*.

3 Analysis and Problem

If we carefully look at the described mashup scenario, we see that it is purely *domain-specific*, i.e. it processes domain objects (researchers, publications, metrics, and so on), uses domain-specific processes (e.g., the computation of weighted publication count metric) and complies with a set of domain-specific composition rules. Throughout this paper, we will see how the development of this scenario can be aided by a domain-specific mashup tool. Before going into the details, we introduce the necessary concepts:

We define a ***web mashup*** (or *mashup*) as a web application that integrates data, application logic, and/or user interfaces (UIs) sourced from the Web [15]. Typically, a mashup integrates and orchestrates two or more elements. Our reference scenario requires all three ingredients of the definition: we need to fetch researchers and publication information from various Web-accessible sources (the data); we need to compute indicators and rankings (the application logic); and we need to render the output to the user for inspection (the UI). We generically refer to the services or applications implementing these features as *components*.

In order to support the described evaluation algorithm, components must be composed and put into communication. Simplifying this task by tailoring a mashup tool to the specific domain of research evaluation first of all requires understanding what a domain is. We define a ***domain*** as a delimited sphere of concepts and processes; *domain concepts* consist of data and relationships; *domain processes* operate on domain concepts and are either atomic (activities) or composite (processes integrating multiple activities).

A ***domain-specific mashup*** is therefore a mashup that describes a composite *domain process* that manipulates *domain concepts* via *domain activities and processes*. It is typically specified using a domain-specific, graphical model notation. A ***domain-specific mashup tool*** (DMT) is a development and execution environment that enables *domain experts*, i.e., the actors operating in the domain, to develop and execute *domain-specific mashups* via a *syntax* that exposes all features of the *domain*.

A DMT is initially "empty". It then gets populated with specific components that provide functionality needed to implement mashup behaviors. For example, software developers (not end-users) will define libraries of components for research evaluation, such as components to extract data from Google Scholar, or to compute the h-index. The domain model can be arbitrarily extended, though the caveat here is that a domain model that is too rich can become difficult for software developers to follow.

Given these definitions, the ***problem*** we solve in this paper is that of providing the necessary concepts and a methodology for the development of domain-specific mashup models and DMTs. The problem is neither simple nor of immediate solution. While domain modeling is a common task in software engineering, its application to the development of mashup platforms is not trivial. For instance, we must precisely understand which domain properties are needed to exhaustively cover all necessary domain aspects that are necessary to tailor a mashup platform to a specific domain, which property comes into play in which step of the development of the platform, how domain aspects are materialized (e.g., visualized) in the mashup platform, and so on.

4 Methodology

Throughout this paper we show how we have developed a mashup platform for our reference scenario, in order to exemplify how its development can approach the above challenges systematically. The development of the platform has allowed us to conceptualize the necessary tasks and to structure them into the following ***methodology*** steps:

1. Definition of a *domain concept model* (CM) to express domain data and relationships. The concepts are the core of each domain. The specification of domain concepts allows the mashup platform to understand what kind of *data objects* it must support. This is different from generic mashup platforms, which provide support for generic data formats, not specific objects.
2. Identification of a generic *mashup meta-model*[1] (MM) that suits the composition needs of the domain and the selected scenarios. A variety of different mashup approaches, i.e., meta-models, have emerged over the last years and before thinking about domain-specific features, it is important to identify a meta-model that accommodates the domain processes to be mashed up.
3. Definition of a *domain-specific mashup meta-model*. Given a generic MM, the next step is understanding how to inject the domain into it. We approach this by specifying and developing:
   (a) A *domain process model* (PM) that expresses classes of domain activities and, possibly, ready processes. Domain activities and processes represent the dynamic aspect of the domain. They operate on and manipulate the domain concepts.
   (b) A *domain syntax* that provides each concept in the domain-specific mashup meta-model (the union of MM and PM) with its own symbol. Domain concepts and activities must be represented by visual metaphores conveying their meaning to domain experts.
   (c) A set of *instances of domain-specific components*. This is the step in which the reusable domain-knowledge is encoded, in order to enable domain experts to mash it up into new applications.
4. *Implementation* of the DMT as a tool whose expressive power is that of the domain-specific mashup meta-model and that is able to host and integrate the domain-specific activities and processes.

In the next subsections, we expand each of these steps.

4.1 The Domain Concept Model

The domain concept model is obtained via interactions between an IT expert and a domain expert. The heart of each domain is represented by the information items each expert of that domain knows and understands. Modeling this kind of information requires understanding which these information items are and how they relate to each other, eventually producing a conceptual model that represents the knowledge base that is shared among the experts of the domain.

In domain-specific mashups, the concept model has three kinds of ***readers*** (and usages), and understanding this helps us to define how the domain should

[1] We use the term *meta-model* to describe the constructs (and the relationships among them) that rule the design of mashup *models*. With the term *instance* we refer to the actual mashup application that can be operated by the user.

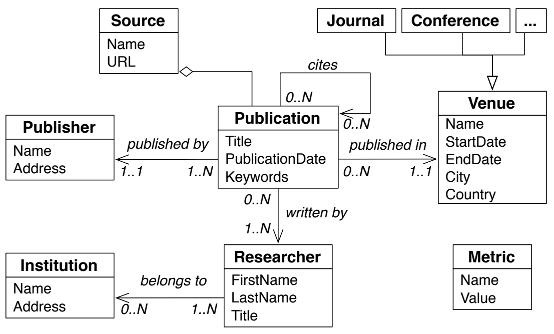


Fig. 1. Domain concept model for the research evaluation scenario

be represented. The first readers are the mashup modelers (domain experts). For them an entity-relationship diagram is a commonly adopted technique, able to let them understand quite easily conceptual models. The second readers are the developers of components (programmers), which need to be aware of the data format in which entities and relationships can be represented, e.g., in terms of XML schemas. The third reader is the DMT itself, which enforces compliance of data exchanges with the concept model.

Therefore, the ***domain concept model (CM)*** describes the *conceptual entities* and the *relationships* among them, which, together, constitute the domain knowledge. We express the domain concept model as a conventional entity-relationship diagram. It also includes a representation of the entities as XML schemas. For instance, in Figure 1 we put the main concepts we can identify in our reference scenario into a CM, detailing entities, attributes, and relationships. The core element in the evaluation of scientific production and quality is the *publication*, which is typically published in the context of a specific *venue*, e.g., 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 or Google Scholar) from which publications are accessed is gaining importance. The actual evaluation is represented in the model by the *metric* entity, which can be computed over any of the other entities. We notice that each concept model implicitly includes the concept of *grouping* the entities in arbitrary ways, so groups are also an implicitly defined entity.

4.2 The Generic Mashup Meta-Model

To define the ***type of mashups*** and, hence, the modeling formalism that is required, it is necessary to model which features (in terms of software capabilities) the mashups should be able to support. Mashups are particular types of web applications. They are component-based, may integrate a variety of services, data sources, and UIs. They may need an own layout for placing components, require control flows or data flows, ask for the synchronization of UIs and the orchestration of services, allow concurrent access or not, and so on. Which exact features a mashup type supports is described by its *mashup meta-model.*

We first define a generic mashup meta-model, which may fit a variety of different domains, then we show how to define the domain-specific mashup meta-model, which will allow us to draw domain-specific mashup models. Specifically, the generic ***mashup meta-model (MM)*** specifies a *class* of mashups and, thereby, the *expressive power*, i.e., the concepts and composition paradigms, a mashup platform must know in order to support the development of that class of mashups. The MM implicitly specifies the expressive power of the mashup platform. Identifying the right features of the mashups that fit a given domain is therefore crucial.

For instance, our research evaluation scenario asks for the capability to integrate data sources (to access publications and researchers via the Web), web services (to compute metrics and perform transformations), and UIs (to render the output of the assessment). We call this capability *universal integration.* Next, the scenario asks for data processing capabilities that are similar to what we know from Yahoo! Pipes, i.e., data flows. It requires dedicated software components that implement the basic activities in the scenario, e.g., compute the impact of a researcher (the sum of his/her publications weighted by the venue ranking), compute the percentile of the researcher inside the national sample (producing outputs like "top 10%"), or plot the department ranking as a chart.

For our research evaluation scenario, we start from a very simple MM, both in terms of notation and execution semantics, which enables end-users to model their own mashups. Indeed, it can be fully specified in one page:

- A ***mashup*** $m = \langle C, P, VP, L\rangle$, defined according to the meta-model MM, consists of a set of *components* C, a set of data *pipes* P, a set of view ports VP that can host and render components with own UI, and a *layout* L that specifies the graphical arrangement of components.
- A ***component*** $c = \langle IPT, OPT, CPT, type, desc\rangle$, where $c \in C$, is like a task that performs some data, application, or UI action.

  Components have *ports* through which pipes are connected. Ports can be divided in *input* (IPT) and *output* ports (OPT), where input ports carry data into the component, while output ports carry data generated (or handed over) by the component. Each component must have at least either an input or an output port. Components with no input ports are called *information sources.* Components with no output ports are called *information sinks.* Components with both input and output ports are called *information processors.* UI components are always information sinks.

  Configuration ports (CPT) are used to configure the components. They are typically used to configure filters (defining the filter conditions) or to define the nature of a query on a data source. The configuration data can be a constant (e.g., a parameter defined by the end-user) or can arrive in a pipe from another component. Conceptually, constant configurations are as if they come from a component feeding a constant value.

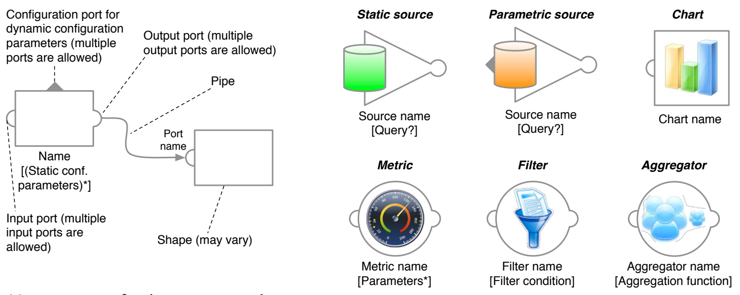
  The type (*type*) of the components denotes whether they are *UI components*, which display data and can be rendered in the mashup's layout, or *application components*, which either fetch or process information.
  Components can also have a description *desc* at an arbitrary level of formalization, whose purpose is to inform the user about the data the components handle and produce.
- A ***pipe*** $p \in P$ carries data (e.g., XML documents) between the ports of two components, implementing a data flow logic. So, $p \in IPT \times (OPT \cup CPT)$.
- A ***view port*** $vp \in VP$ identifies a place holder, e.g., a DIV element or an IFRAME, inside the HTML template that gives the mashup its graphical identity. Typically, a template has multiple place holders.
- Finally, the ***layout*** L defines which component with own UI is to be rendered in which view port of the template. Therefore $l \in C \times VP$.

Each mashup following this MM must have at least a source and a sink, and all ports of all components must be attached to a pipe or manually filled with data (the configuration port).

In the model above there are *no variables* and *no data mappings*. This is at the heart of enabling end-user development as this is where much of the complexity resides. It is unrealistic to ask end users to perform data mapping operations. Because there is a CM, each component is required to be able to process any document that conforms to the model. This does not mean that a component must process every single XML element. For example, a component that computes the h-index will likely do so for researchers, not for publications, and probably not for publishers (though it is conceivable to have an h-index computed for publishers as well). So the component will "attach" a metric only to the researcher information that flows in. Anything else that flows in is just passed through without alterations. The component description will help users to understand what the component operates on or generates, and this is why an informal description suffices. What this means is that each component in a domain-specific mashup must be able to implement this *pass-through* semantics and it must operate on or generate one or more (but not all) elements as specified in the CM. Therefore, our MM assumes that all components comply to understand the CM.

The ***operational semantics*** of the MM is as follows:

1. Execution of the mashup is *initiated* by the user.
2. Components that are *ready* for execution are identified. A component is ready when all the input and configuration ports are filled with data, that is, they have all necessary data to start processing.
3. All ready components are *executed*. They process the data in input ports, consuming the respective data items form the input feed, and generate output on their output ports. The generated outputs fill the inputs of other components, turning them executable.
4. The execution proceeds by identifying ready components and executing them (i.e., reiterating steps 2 and 3), until there are no components to be executed


Configuration port for dynamic configuration parameters (multiple ports are allowed)
Output port (multiple output ports are allowed)
Pipe
Port name
Name [(Static conf. parameters)*]
Input port (multiple input ports are allowed)
Shape (may vary)
Static source
Source name [Query?]
Parametric source
Source name [Query?]
Chart
Chart name
Metric
Metric name [Parameters*]
Filter
Filter name [Filter condition]
Aggregator
Aggregator name [Aggregation function]

(a) Basic syntax for the concepts in the mashup meta-model that are to be exposed to the user. Data mappings are configured in a dedicated pop-up window.

(b) Domain-specific syntax for the concepts in the domain-specific meta-model extension

Fig. 2. Generic and domain-specific syntax for research evaluation

left. At this point, all components have been executed, and all the sinks have received and rendered information.

Developing mashups based on this meta-model, i.e., graphically composing a mashup in a mashup tool, requires defining a ***syntax*** for the concepts in the MM. In Figure 2(a) we map the above MM to a basic set of generic graphical symbols and composition rules. In the next section, we show where to configure domain-specific symbols.

4.3 The Domain-Specific Mashup Meta-Model

The mashup meta-model (MM) described in the previous section allows the definition of a class of mashups that can fit in different domains. Thus, it is not yet tailored to a specific domain, e.g. research evaluation. Now we want to push the domain into the mashup meta-model constraining the class of the mashups that can be produced to that of our specific domain. The next step is therefore understanding the dynamics of the concepts in the model, that is, the typical classes of processes and activities that are performed by domain experts in the domain, in order to transform or evolve concrete instances of the concepts in the CM and to arrive at a structuring of components as well as to an intuitive graphical notation. What we obtain from this is a *domain-specific mashup meta-model*. Each domain-specific meta-model is a specialization of the mashup meta-model along three dimensions: (i) domain-specific activities and processes, (ii) domain-specific syntax, and (iii) domain instances.

The ***domain process model (PM)*** describes the classes of *processes* or *activities* that the domain expert may want to mash up to implement composite, domain-specific processes. Operatively, the PM is again derived by specializing the generic meta-model based on interactions with domain experts, just like for the domain concept model. This time the topic of the interaction is aimed at defining classes of components, their interactions and notations. In the case of research evaluation, this led to the identification of the following classes of activities, i.e., classes of components. For instance, *Source extraction*, *Metric computation*, and *Filtering* and *Aggregation* activities.

A possible ***domain-specific syntax*** for the classes in the PM is shown in Figure 2(b), which is used for our reference scenario. Its semantic is the one described by the MM in Section 4.2. In practice, defining a PM that fully represents a domain requires considering multiple scenarios for a given domain, aiming at covering all possible classes of processes in the domain.

A set of ***instances*** of domain activities must be implemented, providing concrete mashup components. For example, the *Microsoft Academic Publications* component is an instance of *source extraction* activity with a configuration port (*SetResearchers*) that allows the setup of the researchers for which publications are to be loaded from Microsoft Academic.

In summary, we limit the flexibility of a generic mashup tool to a specific class of mashups, gaining however in intuitiveness, due to the strong focus on the specific needs and issues of the target domain. Given the models introduced so far, we can therefore say that a ***domain-specific mashup tool (DMT)*** is a development and execution environment that (i) implements a domain-specific mashup meta-model, (ii) exposes a domain-specific modeling syntax, and (iii) includes an extensible set of domain-specific component instances.

5 The ResEval Mashup Tool

Given the above methodology and artifacts, we now summarize how we used these ingredients to develop a mashup platform for research evaluation.

Design Principles. The described mashup scenario is based on a data flow paradigm, which is typically not very intuitive to non-programmers, especially at runtime when processing the data flow logic lasts several seconds. In order to have a more accessible tool, we bet on *transparency*, i.e., we provide its users with insight into the state of a running mashup. We identify two key points where transparency is important in the mashup model: components and processing state. At each instant of time during the execution of a mashup, we allow a user to inspect the data processed and produced by each component, and we graphically communicate the state of the control flow by animating the mashup model with suitable colors.

Another peculiar aspect of the chosen domain is that it may require the processing of large amounts of data (e.g., we need to extract all the publications of the Italian researchers of a given sector). Loading these data from the server, processing them on the client side, and moving them from one component to another in the browser is unfeasible, due to bandwidth, resource, and time restrictions. Data processing must therefore occur *at the server side*, and only control data and excerpts of the actual data processed need to be exchanged between the client and the server, in order to achieve transparency. There are two exceptions to this approach. First, when the client-side component must process the data, such as in the case of a UI component (e.g., a chart showing

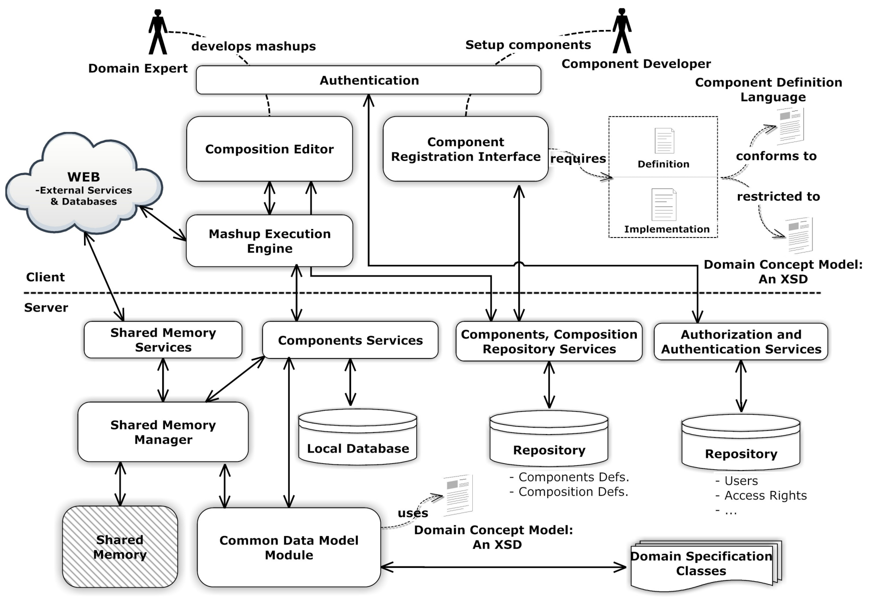
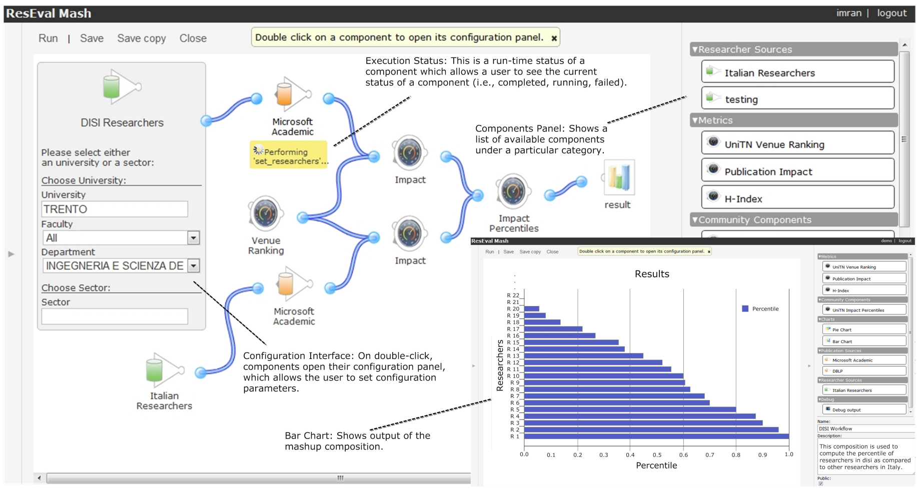


Fig. 3. Internal architecture of the ResEval Mash platform.

some data). Second, when the developer chooses to implement a data processing logic as a JavaScript function instead of implementing it as a web service.

Architecture and Implementation. The architecture of the implemented platform is shown in Figure 3. The most important part of the platform is the *mashup execution engine*, which is developed for client-side processing, that is we control data processing on the server from the client. The engine is primarily responsible for running a mashup composition, triggering the component's actions and managing the communication between client and server. The *composition editor* provides the mashup canvas to the user. It shows a components list from which users can drag and drop components onto the canvas and connect them. The composition editor implements the *domain-specific mashup meta-model* and exposes it through the *domain syntax.* From the editor it is also possible to launch the execution of a composition through a run button and hand the mashup over to the *mashup engine* for execution. The composition editor and its various parts are shown in Figure 4. The platform also comes with a *component registration interface* for developers, which aids them in the setup and addition of new components to the platform.

On the server side, we have a set of RESTful web services, i.e., the *components services*, *authentication services*, *components and composition repository services* and *shared memory services.* Components services manage and allow the invocation of those components whose business logic is implemented as a server-side web service. These web services, together with the client-side components, implement the *domain process model.* Authentication services are used for user authentication and authorization. Components and composition repository ser-



Double click on a component to open its configuration panel.

Please select either an university or a sector:

Choose University:

University: TRENTO

Faculty: All

Department: INGEGNERIA E SCIENZA DE

Choose Sector:

Sector:

Execution Status: This is a run-time status of a component which allows a user to see the current status of a component (i.e., completed, running, failed).

Components Panel: Shows a list of available components under a particular category.

Configuration Interface: On double-click, components open their configuration panel, which allows the user to set configuration parameters.

Bar Chart: Shows output of the mashup composition.

Fig. 4. Composition editor and example mashup output.

vices enable CRUD operations for components and compositions. Shared memory services provide an interface for external web services (i.e., services which are not deployed on our platform) to use the shared memory in order to increase the performance. The *shared memory manager* provides and manages a space for each mashup execution instance on server side. The *common data model (CDM) module* implements the *domain concept model* (CM) and supports the checking of data types in the system. CDM configures itself using an XSD (i.e., an XML schema representing domain concept model) and generates the *domain specification classes* (e.g., in our case these classes are POJO, annotated with JAXB annotations). In Figure 4 we illustrate the mashup model that implements the solution to our researchers evaluation scenario for a specific department. The model starts with two parallel flows: one computing the weighted publication count (the “impact” metric in the specific scenario) for all Italian researchers in a selected disciplinary sector (e.g., Computer Science); the other flow computes the same “impact” metric for the researchers belonging to the UniTN Computer Science department. The former branch defines the distribution of the Italian researchers for the Computer Science disciplinary sector, the latter is used to compute the impact percentiles of UniTN’s researchers and to determine their individual percentiles, which are finally visualized in a bar chart.

A demo of ResEval Mash is described in [8] and a continuously updated prototype of the tool is available online at `http://open.reseval.org/`.

6 User study and Evaluation

In order to evaluate our domain-specific mashup approach, we conducted a user study. The results reported in this paper concentrate on usability, with an emphasis on the role of prior experience on learning. Prior experience was differentiated in two categories which are fundamental in our approach to mashup design: domain knowledge and computing skills. As described later, for each user we measured the levels of experience for both categories.

Method. Ten participants covering a broad range of academic and technical expertise were invited to use ResEval Mash. At the beginning participants were asked to fill in a questionnaire reporting their computing skills and to watch a video tutorial, introducing the basic functionalities of the tool (e.g., it showed how to create a simple mashup). After this training, participants were asked to use the system to accomplish 3 tasks. The first one, starting from the composition presented in the video tutorial, to modify the configuration parameter of a component, the second to replace some components inside the composition, and the third to build a mashup composition from scratch. Whilst completing these tasks, participants were asked to "talk aloud" regarding their thoughts and actions. This interaction was filmed, as was the interview that followed task completion. The interview focused on interactional difficulties experienced and the evolution of participants' conceptual understanding over time.

Results. Overall, the tool was deemed as usable and something with which participants were comfortable. Independently of their level of computing knowledge, all participants were able to accomplish the tasks with minimal or no help at all. The only visible difference reflected a variable level of confidence in task execution. IT experts appeared to be more confident during the test.

The results of our study indicate real potential for the domain-specific mashup approach to allow people with no computing skills to create their own applications. The comparison between the two groups of users (IT and non IT experts) highlighted good performance independently of participants' computing skills. The request for higher training emerging from a few less expert users appeared to be rather linked to a weaker domain knowledge than to their computing capabilities (a result we will further explore in future research). A major finding of this study is related to the ease with which our sample understood that components had to be linked together so that information could flow between different services. This is a well-acknowledged problem evinced in several user studies of EUD tools (e.g., [12]), which did not occur at all in the current study.

Overall, this study suggests that ResEval Mash is a successful tool appealing to both expert programmers and end users with no computing skills.

7 Related Work

Domain-specific modeling. 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 [9] [3] [11] [6]. Mainly these areas are related to Domain Specific Modeling (DSM) and Domain Specific Language (DSL).

In DSM, domain concepts, rules, and semantics are represented by one or more models, which 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 his/her 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 [9]) have shown that developers' productivity can be increased up to an order of magnitude.

In the DSL context, although we can find solutions targeting end users (e.g., Excel macros) and medium skilled users (e.g., MatLab), most of the current DSLs target expert developers (e.g., Swashup [10]). Also here the introduction of the "domain" raises the abstraction level, but the typical textual nature of these languages makes them less intuitive and harder to manage and less suitable for end users compared to visual approaches. Benefits and limits of the DSM and DSL approaches are summarized in [6] and [11].

Web service composition. BPEL (Business Process Execution Language) [13] is currently one of the most used solutions for web service composition, and it is supported by many commercial and free tools. BPEL provides powerful features addressing service composition and orchestration but no support is provided for UI integration, as, for instance, required in our reference scenario. This shortcoming is partly addressed by the BPEL4People [2] and WS-HumanTask [1] specifications, which aim at introducing also human actors into service compositions. Yet, the specifications focus on the coordination logic only and do not support the design of the UIs for task execution. In the MarcoFlow project [5] we provide a solution that bridges the gap between service and UI integration, but the approach is however complex and only suited for expert programmers.

Mashups. Web mashups [15] emerged as an approach to provide easier ways to connect together services and data sources available on the Web [7], together with the claim to target non-programmers. 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 [14] 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 normal web users 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.

8 Status and Lessons Learned

The work described in this paper resulted from actual needs within the university and within the context of an EU project, which were not yet met by current technology. It also resulted from the observation that in general composition technologies failed to a large extent to strike the right balance between ease of use and expressive power. They define seemingly useful abstractions and tools, but in the end developers still prefer to use (textual) programming languages, and at the same time domain experts are not able to understand and use them. What we have pursued in our work is, in essence, to constrain the language to the domain (but not in general in terms of expressive power) and provide a domain-specific notation so that it becomes easier to use and in particular does not require users to deal with one of the most complex aspect of process modeling (at least for end users), that of data mappings, as the components and the DMT take care of this, thanks to the common data model. This is a very simple, but very powerful, concept, because now users just need to take components, place them next to each other and simply connect them, something very different from what traditional mashup or service composition tools require.

References

1. Active Endpoints, Adobe, BEA, IBM, Oracle, SAP. Web Services Human Task (WS-HumanTask) Version 1.0. Technical report, June 2007.
2. Active Endpoints, Adobe, BEA, IBM, Oracle, SAP. WS-BPEL Extension for People (BPEL4People) Version 1.0. Technical report, June 2007.
3. M. F. Costabile, D. Fogli, G. Fresta, P. Mussio, and A. Piccinno. Software environments for end-user development and tailoring. *PsychNology Journal*, pages 99–122, 2004.
4. F. Daniel, F. Casati, B. Benatallah, and M.-C. Shan. Hosted Universal Composition: Models, Languages and Infrastructure in mashArt. In *ER'09*, pages 428–443.
5. F. Daniel, S. Soi, S. Tranquillini, F. Casati, C. Heng, and L. Yan. From People to Services to UI: Distributed Orchestration of User Interfaces. In *BPM'10*, pages 310–326.
6. R. France and B. Rumpe. Domain specific modeling. *Software and Systems Modeling*, 4:1–3, 2005.
7. B. Hartmann, S. Doorley, and S. Klemmer. Hacking, Mashing, Gluing: A Study of Opportunistic Design and Development. *Pervasive Computing*, 7(3):46–54, 2006.
8. M. Imran, F. Kling, S. Soi, F. Daniel, F. Casati, and M. Marchese. ResEval Mash: A Mashup Tool for Advanced Research Evaluation. In *Proceedings of WWW 2012*.
9. Á. Lédeczi, A. Bakay, M. Maroti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing domain-specific design environments. *IEEE Computer*, 34(11):44–51, 2001.
10. E. M. Maximilien, H. Wilkinson, N. Desai, and S. Tai. A domain-specific language for web apis and services mashups. In *ICSOC*, pages 13–26, 2007.
11. M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.
12. A. Namoun, T. Nestler, and A. De Angeli. Service Composition for Non Programmers: Prospects, Problems, and Design Recommendations. In *Proceedings of ECOWS*, pages 123–130. IEEE, 2010.
13. OASIS. Web Services Business Process Execution Language Version 2.0. Technical report, April 2007.
14. S. Pietschmann, M. Voigt, A. Rümpel, and K. Meißner. Cruise: Composition of rich user interface services. In *ICWE'09*, pages 473–476. 2009.
15. J. Yu, B. Benatallah, F. Casati, and F. Daniel. Understanding Mashup Development. *IEEE Internet Computing*, 12:44–52, 2008.