Developing Mashup Tools for End-Users: On the Importance of the Application Domain

Florian Daniel, Muhammad Imran, Stefano Soi, Antonella De Angeli, Christopher R. Wilkinson, Fabio Casati and Maurizio Marchese
University of Trento, Via Sommarive 5, 38123, Trento, Italy
lastname@disi.unitn.it

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 article, 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 confirm that domain-specific mashup tools indeed lower the entry barrier to mashup development.

General Terms: Design, Experimentation

1. INTRODUCTION

Mashups are typically simple web applications (most of the times consisting of just one single page) 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. For instance, `housingmaps.com` integrates housing offers from Craigslist with a Google map, adding value to the two individual applications.

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, sometimes even *situational* applications developed ad hoc for a specific immediate need. Yet, similar to what happened in web service composition, the mashup platforms developed so far tend to expose too much functionality and too many technicalities so that they are powerful and flexible but suitable only for programmers. Alternatively, they 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 [Namoun et al. 2010a; 2010b]. We experienced this with our own mashup and composition platforms, mashArt [Daniel et al. 2009] and MarcoFlow [Daniel et al. 2010], which we believe to be simpler and more usable than many composition tools but that still failed in being suitable for non-programmers [Mehandjiev et al. 2011]. Yet, being amenable to non-programmers

Author's address: University of Trento - Department of Information Engineering and Computer Science, Via Sommarive 5, 38122 Povo (TN), Italy.

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:

(1) 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.

(2) 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.

(3) 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.

(4) We perform a *user study* in order to assess the usability of the developed platform and the viability of the respective development methodology.

While we focus on mashups, the techniques and lessons learned in the paper are general in nature and can easily be applied to other composition or modeling environments, such as web service composition or business process modeling.

Next, we first introduce a reference scenario. In Section 3, we present key definitions and provide the problem statement. Section 4 outlines the methodology followed to implement the scenario. In Section 5 we describe ResEval Mash, the actual implementation of our prototype tool, and in Section 6 we report on a user study conducted with the tool and the respective results. 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 application scenario, let us 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. The evaluation is used to allocate resources and funding for the university departments. In essence, the algorithm compares the quality of the scientific production of each researcher in a given department of UniTN 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 publication list obtained in the previous step is then weighted using a venue classification. That is, the publications are classified by an internal committee in three quality categories mainly based on ISI Journal Impact Factor: A/1.0(top), B/0.6(average), C/0.3(low). For each researcher a single weighted publication count parameter is thus obtained with a weighted sum of his/her publications.

(4) The statistical distribution – more specifically, a negative binomial distribution – of the weighted publication count metric is then computed out of the Italian researchers' reference sample.

(5) Each researcher in the selected department is ranked based on his/her weighted publication count by comparing this value with the statistical distribution. That is, for each researcher the respective percentile (e.g., top 10%) in the distribution of the researchers in the same disciplinary sector is computed.

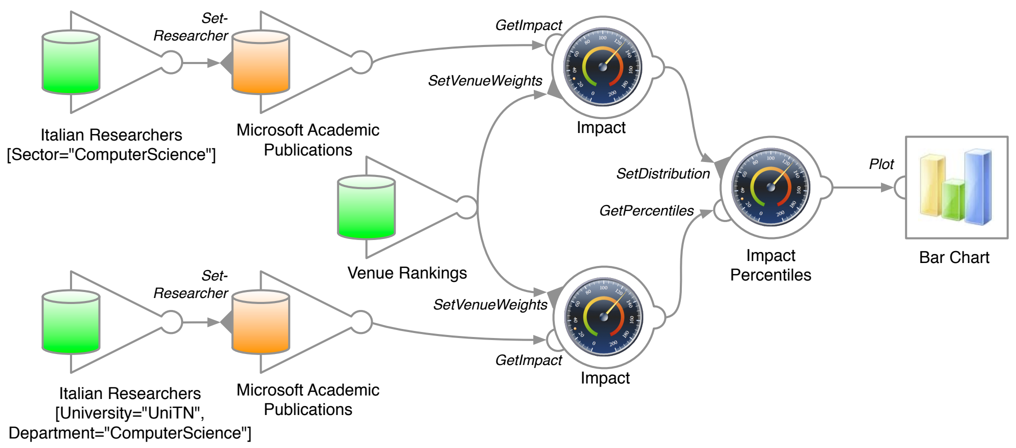

Set-Researcher
Italian Researchers [Sector="ComputerScience"]
Microsoft Academic Publications
GetImpact
SetVenueWeights
Impact
Venue Rankings
SetDistribution
GetPercentiles
Impact Percentiles
Plot
Bar Chart
Set-Researcher
Italian Researchers [University="UniTN", Department="ComputerScience"]
Microsoft Academic Publications
SetVenueWeights
GetImpact
Impact

Figure. 1. University of Trento's internal researcher evaluation procedure.

The percentile for each researcher in the selected department is considered as an estimation of the publishing profile of that researcher and is used for comparison with other researchers in the same department. As one can notice, plenty of effort is required to compute the performance of each researcher, which is currently mainly done manually. Fervid discussion on the suitability of the selected criteria often arise, as people would like to understand how the results would differ changing the publications ranking, the source of the bibliometric information, or the criteria of the reference sample. Indeed all these factors have a big impact on the final result and have been locally at the center of a heated debate. Many researchers would like to use different metrics, like citation-based metrics (e.g., h-index). Yet, computing different metrics and variations thereof is a complex task that costs considerable human resources and time.

The requirement we extract from this scenario is that we need to empower people involved in the evaluation process (i.e., the average faculty member or the administrative persons in charge of it) so that they can be able to define and compare relatively complex evaluation processes, taking and processing data in various ways from different sources, and visually analyze the results. This task, requiring to extract, combine, and process data and services from multiple sources, and finally represent the information with visual components, has all the characteristics of a mashup, especially if the mashup logic comes from the users.

In Figure 1 we illustrate the mashup model we are aiming at for our researchers evaluation scenario within a specific department. The model starts with two parallel flows: one computing the weighted publication number (the "impact" metric in the specific scenario) for all Italian researchers in a selected disciplinary sector (e.g., Computer Science). The other 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.

Although the composition model in Figure 1 is apparently similar to conventional web service composition or data flow models, in the following we will show why we are confident that also end-users will be able to model this or similarly complex evaluation scenarios, while, instead, they are not yet able to compose web services in general.

3. ANALYSIS AND PROBLEM

If we carefully look at the described mashup scenario, we see that the proposed model is *domain-specific*, i.e., it is entirely based on concepts that are typical of the research evaluation domain. For instance, the scenario processes domain objects (researchers, publications, metrics, and so on), uses domain-specific components (the *Italian researchers* data source, the *Impact* metric, etc.), and complies with a set of domain-specific composition rules (e.g., that publications can be ranked based on the importance of the venue they have been published in).

In order to enable the development of an application for the described evaluation procedure, there is no need for a composition or mashup environment that supports as many composition technologies or options as possible. The intuition we elaborate in this article is that, instead, a much more limited environment that supports exactly the basic tasks described in the scenario (e.g., fetch the set of Italian researchers) and allows its users to mash them up in an as easy as possible way (e.g., without having to care about how to transform data among components) is more effective. The challenge lies in finding the right trade-off between flexibility and simplicity. The former, for example, pushes toward a large number of basic components, the latter toward a small number of components. As we will see, it is the nature of the specific domain that tells us where to stop.

Throughout this paper, we will therefore show how the development of the example scenario can be aided by a domain-specific mashup tool. Turning the previous consideration into practice, the development of this tool will be driven by the following key principles:

(1) ***Intuitive user interface.*** Enabling domain experts to develop their own research evaluation metrics, i.e., mashups, requires an intuitive and easy-to-use user interface (UI) based on the concepts and terminology the target domain expert is acquainted with. Research evaluation, for instance, speaks about metrics, researchers, publications, etc.
(2) ***Intuitive modeling constructs.*** Next to the look and feel of the platform, it is important that the functionalities provided through the platform (i.e., the building blocks in the composition design environment) resemble the common practice of the domain. For instance, we need to be able to compute metrics, to group people and publications, and so on.
(3) ***No data mapping.*** Our experience with prior mashup platforms, i.e., mashArt [Daniel et al. 2009] and MarcoFlow [Daniel et al. 2010], has shown that data mappings are one of the least intuitive tasks in composition environments and that non-programmers are typically not able to correctly specify them. We therefore aim to develop a mashup platform that is able to work without data mappings.

Before going into the details, we introduce the necessary concepts, starting from our interpretation of web mashups [Yu et al. 2008]:

Definition A ***web mashup*** (or *mashup*) is a web application that integrates data, application logic, and/or user interfaces (UIs) sourced from the Web. Typically, a mashup integrates and orchestrates two or more elements.

Our reference scenario requires all three ingredients listed in 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*. Components must be put into communication, in order to support the described evaluation algorithm.

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 and, then, a domain-specific mashup as follows:

Definition A ***domain*** is 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).

Definition A ***domain-specific mashup*** is a mashup that describes a composite *domain process* that manipulates *domain concepts* via *domain activities and processes*. It is specified in a domain-specific, graphical modeling notation.

The domain defines the "universe" in the context of which we can define domain-specific mashups. It defines the information that is processed by the mashup, both conceptually and in terms of concrete data types (e.g., XML schemas). It defines the classes of components that can be part of the process and how they can be combined, as well as a notation that carries meaning in the domain (such as specific graphical symbols for components of different classes). For instance, in our reference scenario, concepts include *publications*, *researchers*, *metrics*, etc. The process models define classes of components such as data extraction from digital libraries, metric computation, or filtering and aggregation components. These domain restrictions and the exposed domain concepts at the mashup modeling level is what enables simplification of the language and its usage.

Definition 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, or to group researchers based on their institution, or to visualize results in different ways. Because all components fit in the classes and interact based on a common data model, it becomes easier to combine them and to define mashups, as the DMT knows what can be combined and can guide the user in matching components. 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 those 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.

The DMT idea is heavily grounded on a rich corpus of research in ***Human-Computer Interaction*** (HCI), demonstrating that consideration of user knowledge and prior experience are required to create truly usable and inclusive products, and are key considerations in the performance of usability evaluations [Nielsen 1993]. The prior experience of products is important to their usability, and the transfer of previous experience depends upon the nature of prior and subsequent experience of similar tasks [Thomas and van-Leeuwen 1999]. Familiarity of the interface design, its interaction style, or the metaphor it conforms to if it possesses one, are key features for successful and intuitive interaction [Okeye 1998]. More familiar interfaces, or interface features, allow for easier information processing in terms of user capability, and the subsequent human responses can be performed at an automatic and subconscious level. Karlsson and Wikstrom [2006] identified that the use of semantics could be an effective tool for enhancing product design and use, particularly for novel users, as they can indicate how the product or interface will behave and how interaction is likely to occur. Similarly, Monk [1998] stressed that to be usable and accessible, interfaces need to be easily understood and learned, and in the process, must cause minimal cognitive load. Effective interaction consists of users understanding potential actions, the execution of specific action, and the perception of the effects of that action.

As we cannot exploit the users' technical expertise, we propose here to exploit their knowledge of the task domain. In other words, we intend to transform mashups from technical tools built around a computing metaphor to true cognitive artifacts [Norman 1991], capable to operate upon familiar information in order to "serve a representational function that affect human cognitive performance."

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 ingredients and to structure them into a ***methodology*** for the development of domain-specific mashup platforms. The methodology encodes a top-down approach, which starts from the analysis of the target domain and ends with the implementation of the specifically tailored mashup platform. Specifically, developing a domain-specific mashup platform requires:

(1) Definition of a *domain concept model* (DCM) 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, e.g., ranging from data mashups, over user interface mashups to process mashups. Before thinking about domain-specific features, it is important to identify a meta-model that is able to accommodate 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 so that all features of the domain can be communicated to the developer. 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

[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.

domain. They operate on and manipulate the domain concepts. In the context of mashups, we can map activities and processes to reusable components of the platform.

(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. The claim here is that just catering for domain-specific activities or processes is not enough, if these are not accompanied with visual metaphors that the domain expert is acquainted with and that visually convey the respective functionalities.

(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.

The above steps mostly focus on the *design* of a domain-specific mashup platform. Since domains, however, typically *evolve* over time, in a concrete deployment it might be necessary to periodically update domain models, components, and the platform implementation (that is, iterating over the above design steps), in order to take into account changing requirements or practices. The better the analysis and design of the domain in the first place, the less modifications will be required in the subsequent evolution steps, e.g., limiting evolution to the implementation of new components only.

In the next subsections, we expand each of the above design steps; we do not further elaborate on evolution.

4.1 The Domain Concept Model

The domain concept model is constructed by the IT experts via verbal interaction with the domain experts or via behavioral observation of the experts performing their daily activities and performing a suitable task-analysis. The concept model represents the information experts know, understand, and use in their work. Modeling this kind of information requires understanding the fundamental information items and how they relate to each other, eventually producing a 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 ***stakeholders*** (and usages), and understanding this helps us to define how the domain should be represented. The first stakeholders are the mashup modelers (domain experts), i.e., the end-users that will develop different mashups from existing components. For them it is important that the concept model is easy to understand, and an entity-relationship diagram (possibly with a description) is a commonly adopted technique to communicate conceptual models. The second kind of stakeholders are the developers of components, which are programmers. They need to be aware of the data format in which entities and relationships can be represented, e.g., in terms of XML schemas, in order to implement components that can interoperate with other components of the domain. The third stakeholder is the DMT itself, which enforces compliance of data exchanges with the concept model. Therefore:

Definition The ***domain concept model (DCM)*** describes the *conceptual entities* and the *relationships* among them, which, together, constitute the domain knowledge.

We express the domain-model as a conventional entity-relationship diagram. It also includes a representation of the entities as XML schemas. For instance, in Figure 2 we put the main concepts we can identify in our reference scenario into a DCM, 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

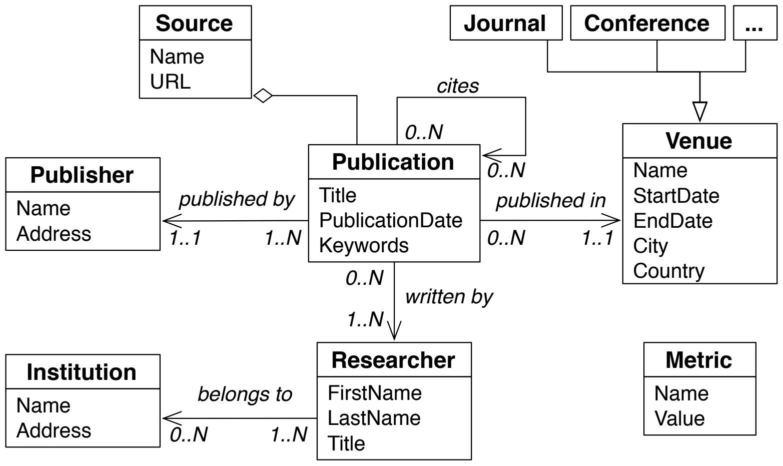


Figure. 2. Domain concept model for the research evaluation scenario

evaluation – also the *source* (e.g., Scopus, the ACM digital library 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 and, hence, the choice of the source will affect the evaluation result. The actual evaluation is represented in the model by the *metric* entity, which can be computed over any of the other entities.

In order to develop a DMT, the ER (Entity-Relationship) model has to be generated through several interactions between the domain expert and the IT expert, who has knowledge of conceptual modeling. The IT expert also generates the XML schemas corresponding to the ER model, which are the actual artifacts processed by the DMT. In fact, although the ER model is part of the concept model, it is never processed itself by the DMT. It rather serves as a reference for any user of the platform to inform them on the concepts supported by it. In principle, other formalisms can be adopted (such as UML Class diagrams). 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

When discussing the domain concept model we made the implicit choice to start from *generic* (i.e. domain-independent) models like Entity-Relationship diagrams and XML, as these are well established data modeling and type specification languages amenable to humans and machines. For end-user development of mashups, the choice is less obvious since it is not easy to identify a modeling formalism that is amenable to defining end-user mashups (which is why we endeavor to define a domain-specific mashup approach). If we take existing mashup models and simply inject specific data types in the system, we are not likely to be successful in reducing the complexity level. However, the availability of the DCM makes it possible to derive a different kind of mashup modeling formalism, as discussed next.

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*.

In the following, 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.

Definition The generic ***mashup meta-model (MM)*** specifies a *class* of mashups and, thereby, the *expressive power*, i.e., the concepts and composition paradigms, the mashup platform must know in order to support the development of that class of mashups.

The MM therefore 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 in a bar chart.

4.2.1 *The meta-model.* We start from a very simple MM, both in terms of notation and execution semantics, which enables end-users to model own mashups. Indeed, it can be fully specified in one page:

(1) 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.

(2) 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.

(3) A ***pipe*** $p \in P$ carries data (e.g., XML documents) between the ports of two components, implementing a data flow logic. So, $p \in IPT \times (OPT \cup CPT)$.

(4) 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.

(5) 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).

This is all we need to define a mashup and as we will see, this is an executable specification. There is nothing else besides this picture. This is not that far from the complexity of specifying a flowchart, for example. It is very distant from what can be an (executable) BPMN specification or a BPEL process in terms of complexity.

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 DCM, 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 DCM. Therefore, our MM assumes that all components comply to understand the DCM.

Furthermore, in the model there are also *no gateways* a-la BPMN, although it is possible to have dedicated components that, for example, implement an if-then semantics and have two output ports for this purpose. In this case, one of the output ports will be populated with an empty feed. Complex routing semantics are virtually impossible for non-experts to understand (and in many cases for experts as well) and for this reason if they are needed we delegate them to the components which are done by programmers and are understood by end-users in the context of a domain.

4.2.2 *Operational semantics.* The behavior of the components and the semantics of the MM are as follows:

(1) Executions of the mashups are *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 then *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 left. At this point, all components have been executed, and all the sinks have received and rendered information.

4.2.3 *Generic mashup syntax.* 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 3 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 DCM and to arrive at a structuring

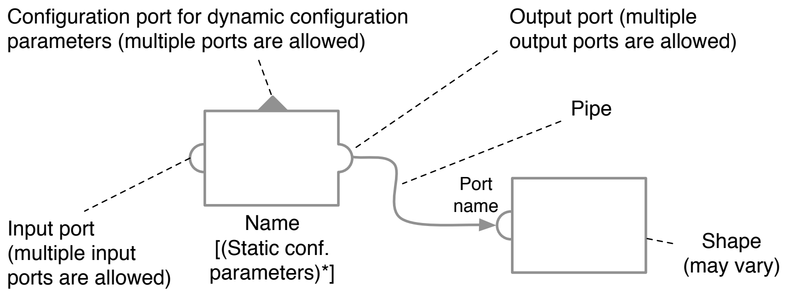

Configuration port for dynamic configuration parameters (multiple ports are allowed)
Output port (multiple output ports are allowed)
Pipe
Port name
Input port (multiple input ports are allowed)
Name [(Static conf. parameters)*]
Shape (may vary)

Figure. 3. Basic syntax for the concepts in the mashup meta-model.

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.

4.3.1 *Domain process model*

Definition 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 process model 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:

(1) *Source extraction* activities. They are like queries over digital libraries such as Scopus or Scholar. They have no input port, and have one output port (the extracted data). These components may have one or more configuration ports that specify in essence the “query”. For example a source component may take in input a set of researchers and extract publications and citations for every researcher from Scopus.

(2) *Metric computation* activities, which can take in input institutions, venues, researchers, or publications and attach a metric to them. The corresponding components have at least one input and one output ports. For example, a component determines the h-index for researchers, or determines the percentile of a metric based on a distribution.

(3) *Aggregation* activities, which define groups of items based on some parameter (e.g., affiliation).

(4) *Filtering* activities, which receive an input pipe and return in output a filtering of the input, based on a criterion that arrives in a configuration port. For example we can filter researchers based on the nationality or affiliation or based on the value of a metric.

(5) *UI widgets*, corresponding to information sink components that plot or map information on researchers, venues, publications, and related metrics.

For simplicity, we discuss only the processes that are necessary to implement the reference scenario.

4.3.2 *Domain syntax.* A possible ***domain-specific syntax*** for the classes in the PM (derived from the generic syntax presented in Figure 3) is shown in Figure 4, which is used for our reference scenario in Figure 1 shown earlier. 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.

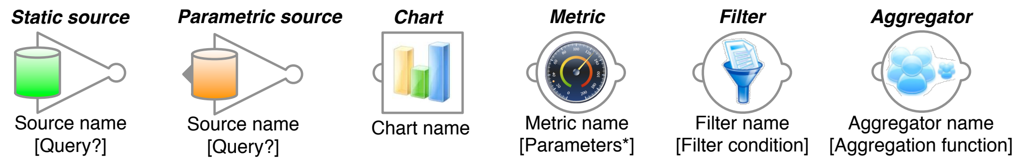


Figure. 4. Domain-specific syntax for the concepts in the domain-specific meta-model extension

4.3.3 *Domain instances.* Figure 1 actually exemplifies the use of *instances* of domain-specific 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. The component is implemented as web service, and its symbol is an instantiation of the parametric source component type in Figure 4 without static query. Similarly, we need to implement web services for the *Italian Researchers* (source extraction activity), the *Venue Ranking* (source extraction activity), the *Impact* (metric computation activity), the *Impact Percentiles* (metric computation activity), and the *Bar Chart* (UI widget) components.

In summary, what we do is limiting the flexibility of a generic mashup tool to a specific class of mashups, gaining however in intuitiveness, due the strong focus on the specific needs and issues of the target domain. Given the models introduced so far, we can therefore refine our definition of DMT given earlier as follows:

Definition 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 MASH TOOL FOR RESEARCH EVALUATION

The methodology described above is the result of our experience with the implementation of our own domain-specific mashup platform, ResEval Mash[2]. ***ResEval Mash*** is a mashup platform for research evaluation, i.e., 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 with a client-side editor and runtime engine, both running inside a common web browser. It supports the processing of also large amounts of data, a feature that is achieved via the sensible distribution of the respective computation steps over client and server.

In the following, we show how ResEval Mash has been implemented, starting from the domain models introduced throughout the previous sections.

5.1 Design Principles

Starting from the considerations in Section 3, the implementation of ResEval Mash is based on a set of design principles (described next), which we think are crucial for the success of a mashup platform like ResEval Mash. The first and last principles stem from our prior work and user studies [Namoun et al. 2010b] in the context of web service composition and end-user development, while the second and third principles respond to domain-specific requirements identified during the analysis of the domain and are based on our past experience with similar problems in the context of the LiquidPub European project[3].

(1) ***Hidden data mappings.*** In order to prevent the users from defining data mappings, the mashup component used in the platform are all able to understand and manipulate

[2]http://open.reseval.org/
[3]http://liquidpub.org/

the domain concepts expressed in the DCM, which defines the domain entities and their relations. That is, they accept as input and produce as output only domain entities (e.g., researchers, publications, metric values). Since all the components, hence, speak the same language, composition can do without explicit data mappings and it is enough to model which component feeds input to which other component.

(2) ***Data-intensive processes.*** Although apparently simple, the chosen domain is peculiar in that it may require the processing of large amounts of data. For instance, we may need to extract and process all the publications of the Italian researchers, i.e., on average several dozens of publications by about sixty-one thousand researchers (and this only for Italian scenarios). Loading these large amounts of data from remote services and processing them in the browser at the client side is unfeasible due to bandwidth, resource, and time restrictions. Data processing should therefore be kept at the server side (we achieve this via dedicated RESTful web services running on the mashup server).

(3) ***Platform-specific services.*** As opposed to common web services, which are typically designed to be independent of the external world, the previous two principles instead demand for services that are specifically designed and implemented to efficiently run in our domain-specific architecture. That is, they must be aware of the platform they run in. As we will see, this allows the services to access shared resources (e.g., the data passed among components) in a protected and speedy fashion.

(4) ***Runtime transparency.*** Finally, research evaluation processes like our reference scenario focus on the processing of data, which – from a mashup paradigm point of view – demands for a data flow mashup paradigm. Although data flows are relatively intuitive at design time, they typically are not very intuitive at runtime, especially when processing a data flow logic takes several seconds (as could happen in our case). In order to convey to the user what is going on during execution, we therefore want to provide transparency into the state of a running mashup.
We identify two key points where transparency is important in the mashup model: component state and processing state. At each instant of time during the execution of a mashup, the runtime environment should allow the user to inspect the data processed and produced by each component, and the environment should graphically communicate the processing progress by animating a graphical representation of the mashup model with suitable colors.

These principles require ResEval Mash to specifically take into account the characteristics of the research evaluation domain. Doing so produces a platform that is fundamentally different from generic mashup platforms, such as Yahoo! Pipes[4].

5.2 Architecture

Figure 5 illustrates the internal architecture that takes into account the above principles and the domain-specific requirements introduced throughout the previous sections: Hidden data mappings are achieved by implementing mashup components that all comply with the *domain conceptual model* described in Figure 2. The processing of large amounts of data is achieved at the server side by implementing platform-specific services that all operate on a *shared memory*, which allows the components to read and write back data and prevents them from having to pass data directly from one service to another. The components and services implement the *domain process model* discussed in Section 4.3, i.e., all the typical domain activities that characterize the research evaluation domain. Runtime transparency is achieved by controlling data processing from the client and animating accordingly the mashup model in the Composition Editor. Doing so requires that each design-time modeling construct has an equivalent runtime component that is able to render its runtime state to the user. The modeling constructs are the ones of the *domain-specific syntax* illustrated in Figure 4, which can be used to compose mashups like the

[4]`http://pipes.yahoo.com/pipes/`

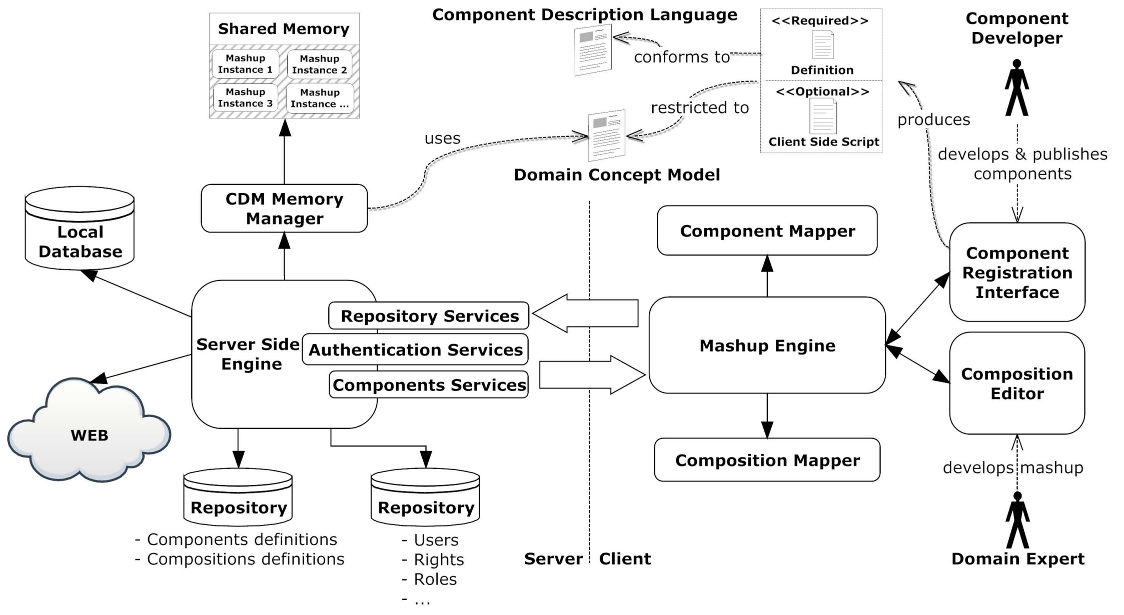


Figure. 5. Mashup Platform Architecture

one in our reference scenario (see Figure 1). Given such a model, the Mashup Engine is able to run the mashup according to the *meta-model* introduced in Section 4.2.

The role of the individual elements in Figure 5 is as follows:

Mashup Engine: The most important part of the platform is the *mashup engine*, which is developed for the 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. As a component either binds with one or more services or with a JavaScript implementation, the engine is responsible for checking the respective binding and for executing the corresponding action. The engine is also responsible for the management of complex interactions among components. A detailed view of these possible interaction scenario is given later in this section.

Component and Composition mappers: The *component* and *composition mappers* parse component and composition descriptors to represent them in the *composition editor* at design time and to bind them at run time.

Composition editor: 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.

Server-Side Services: On the server side, we have a set of RESTful web services, i.e., the *repository services*, *authentication services*, and *components services.* Repository services enable CRUD operations for components and compositions. Authentication services are used for user authentication and authorization. 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.* A detail explanation of how to develop a service for a component is given in section 5.4.

CDM Memory Manager: The *common data model (CDM) memory manager* implements the domain concept model (DCM) and supports the checking of data types in the system. All data processing services read and write to this shared memory through the CDM memory manager. In order to configure the CDM, the *CDM memory manger* generates corresponding Java classes (e.g., in our case these classes are POJO, annotated with JAXB annotations) from an XSD that encodes the domain concept model. The CDM interacts with a *shared memory* that provides a space for each mashup execution instance. In our first proof-of-concept prototype we use the server's working memory (RAM) as *shared memory*, which allows for high performance. Clearly, this solution fits the purpose of our prototype but it may not scale to in-production installations, which may need to deal with large numbers of users and large amounts of data that only hardly can be kept in RAM. In our future work, we will therefore develop a persistent database-based *shared memory.*

Server-engine: All services are managed by the *server-side engine*, which is responsible for managing all the modules that are at the server side, e.g., the CDM memory manager, the repository, and so on. The server-side engine is the place where requests coming from the client side are dispatched to the respective service implementing the required operations.

Local Database and the Web: Both the *local database* and the *Web* represent the data which is required and used by the components services. We as platform provider provides an initial database and a basic set of services on top of it. A third-party service can be deployed and thus it can use an external database anywhere on the Web.

Component registration interface: 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. The interface allows the developer to define components starting from ready templates. In order to develop a component, the developer has to provide two artifacts: (i) a component definition and (ii) a component implementation. The implementation consists either of JavaScript code for client-side components or it is linked by providing a binding to a web service for server-side components.

5.3 Components Models and Data Passing Logic

There are two component models in ResEval Mash, depending on whether the respective business logic resides in the client or in the server side: *server components* (SC) are implemented as RESTful web services that run at the server side; *client components* (CC) are implemented in JavaScript file and run at the client side. Independently of the component model, each component has a client-side component front-end, which allows (i) the Mashup Engine to enact component operations and (ii) the user to inspect the state of the mashup during runtime. All communications among components are mediated by the Mashup Engine, internally implementing a dedicated event bus for shipping data via events. Server components require interactions with their server-side business logic and the shared memory; this interaction needs to be mediated by the Mashup Engine. Client components directly interact with their client-side business logic; this interaction does not require the intervention of the Mashup Engine.

Components consume or produce different *types of data*: actual *data* (D), *configuration parameters* (CP), and control data like *request status* (RS), a flag telling whether actual *data is required* in output (DR), and a *key* (K) identifying data items in the shared memory. All components can consume and produce actual data, yet, as we will see, not always producing actual data in output is necessary. The configuration parameters enable the setup of the components. The request status enables rendering the processing status at runtime. The key is crucial to identify data items produced by one component and to be "passed" as input to another component. As explained earlier, instead of directly passing data from one service to another, for performance reasons we use a shared memory that all services can access and only pass a key, i.e., a reference to the actual data from component to component.

Based on the flow of components in the mashup model, we can have different data passing

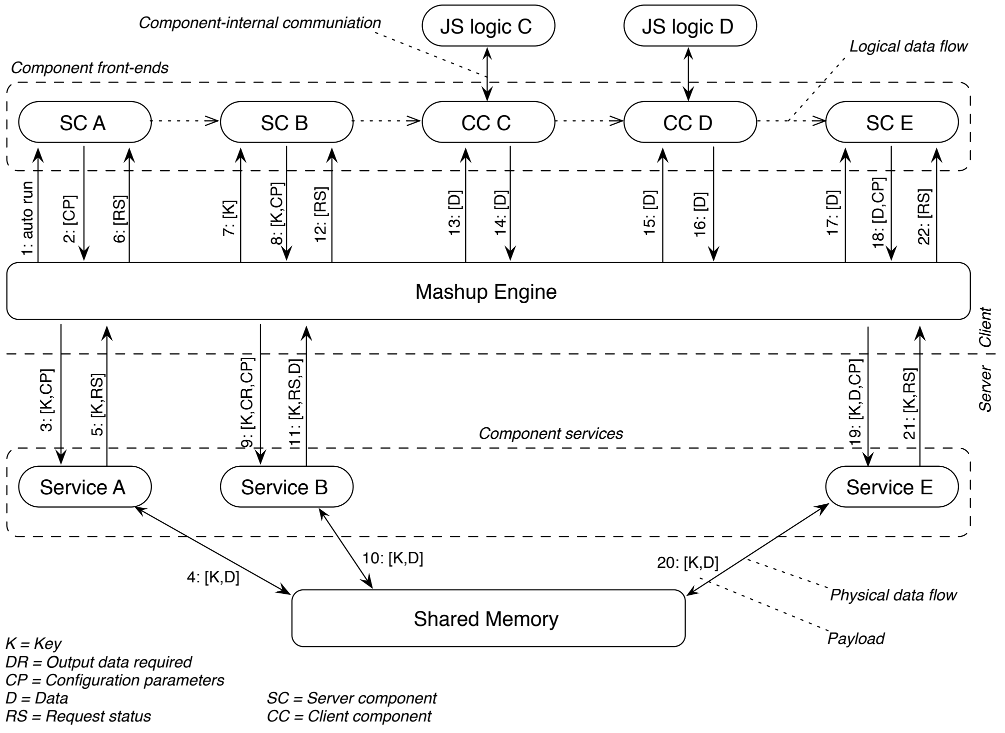


Figure. 6. ResEval Mash's internal data passing logic.

patterns. Given the two different types of components, we can recognize four possible interaction patterns. The four patterns are illustrated in Figure 6 and described in the following paragraphs:

(1) ***SC-SC interaction:*** Both the components are of type SC. In Figure 6, component A is connected with component B. Since component A is the first component in the composition and it does not require any input, it can start the execution immediately. It is the responsibility of the Mashup Engine to trigger the operation of the component A (step 1). At this point, component A calls its back-end web service through the Mashup Engine, passing only the configuration parameters (CP) to it (2). The Mashup Engine, analyzing the composition model, knows that the next component in the flow is also a server component (component B), so it extends component A's request adding a *key* control information to the original request, which can be used by component A's service to mark the data it produces in the shared memory. Hence, the Mashup Engine invokes service A (3). Service A receives the control data, executes its own logic, and stores its output into the Shared Memory (4). Once the execution ends, Service A sends back the control data (i.e., key and request status) to the Mashup Engine (5), which forwards the request status to component A (6); the engine keeps track of the key. With this, component A has completed and the engine can enable the next component (7). In the SC-SC interaction, we do not need to ship any data from the server to the client.

(2) ***SC-CC interaction:*** Once activated, component B enacts its server-side logic (8, 9, 10). The Mashup Engine detects that the next component in the flow is a client component, so it adds the DR control data parameter in addition to the key and the configuration parameters, in order to instruct the web service B to send actual output data back to the client side after it has been stored in the Shared Memory. In this way, when service B finishes its execution, it returns the control data and the actual output data of the service (i.e., key, request status and output data) to the Mashup Engine (11), which then passes the request status to component

B (12) and the actual data to the next component in the mashup, i.e., component C (13).

(3) ***CC-CC interaction:*** Client component to client component interactions do not require to interact with the server-side services. Once the component C's operation is triggered in response to the termination of component B, it is ready to start its execution and to pass component B's output data to the JavaScript function implementing its business logic. Once component C finishes its execution, it sends its output data back to the engine (14), which is then able to start component D (15) by passing C's output data.

(4) ***CC-SC interaction:*** After the completion of component D (16), the Mashup Engine passes the respective data to component E as input (17). At this point, component E calls its corresponding service E, passing to it the actual data and possible configuration parameters (18), along with the key appended by the Mashup Engine (19). Possibly, also the Output Data Request flag could be included in the control data but, as explained, this depends on the next component in the flow, which for presentation purpose is not further defined in Figure 6. Eventually, service E returns its response (i.e., key and request status – plus possible output data if the DR flag is present) to the Mashup Engine (21), which is then delivered to component E (22).

While ResEval Mash fully supports these four data passing patterns and is able to understand whether data are to be processed at the client or the server side, it has to be noted that the actual decision of where data are to be processed is up to the developer of the respective mashup component. Client components by definition require data at the client side; server components at the server side. Therefore, if large amounts of data are to be processed, a sensible design of the respective components is paramount. As a rule of thumb, we can say that data should be processed at the server side whenever possible, and component developers should use client components only when really necessary. For instance, visualization components of course require client-side data processing. Yet, if they are used as sinks in the mashup model (which is usually the case), they will have to process only the final output of the actual data processing logic, which is typically of smaller size compared to the actual data sourced from the initial data sources (e.g., a table of h-indexes vs. the lists of publications by the set of the respective researchers).

5.4 The Domain-Specific Service Ecosystem

An innovative aspect of our mashup platform is its approach based on the concept of *domain-specific components*. In Section 5.2 we described the role of the Components services in the architecture of the system. These are not simply generic web services, but web services that constitute a *domain-specific service ecosystem*, i.e., a set of services respecting shared models and conventions and that are designed to work collaboratively where each of them provides a brick to solve more complex problems proper of the specific domain. Having such an ecosystem of compatible and compliant services, introduces several advantages that make our tool actually usable and able to respond to the specific requirements of the domain we are dealing with.

Given the important role domain-specific components and services play in our platform, next we describe how they are designed and illustrate some details of their implementation and their interactions with the other parts of the system.

A ResEval Mash component requires the definition of two main artifacts: the component descriptor and the component implementation.

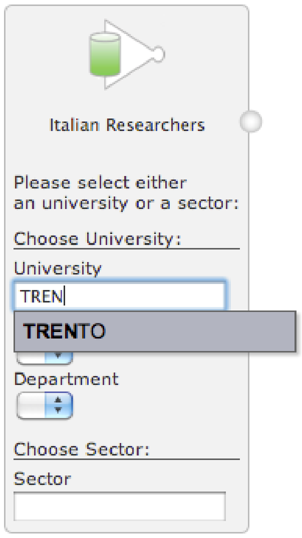
The ***component descriptor*** describes the main properties of a component, which are:

(1) *Operations.* Functions that are triggered as consequence of an external event that take some input data and perform a given business logic.

(2) *Events.* Messages produced by the component to inform the external world of its state changes, e.g., due to interactions with the user or an operation completion. Events may carry output data.

(3) *Implementation binding.* A binding defining how to reach the component implementation.

```
<component id="org.reseval.ItalianResearchers" name="Italian Researchers">
    <description>Produces the list of Italian researchers provided by ...

    <config ref="uniId">
        <option name="label" value="University"/>
        <option name="renderer">
            <option name="type" value="jsm.ui.input.Autocomplete"/>
            <option name="url"
                    value="http://.../italianSource/university/autocomplete"/>
            <option name="search_parameter" value="input"/>
            <option name="idMapper" value="id"/>
            <option name="valueMapper" value="name"/>
        </option>
    </config>

    [... OTHER CONFIGURATION PARAMETERS DEFINITION ...]

    <configTemplate>
        <![CDATA[
            {uniId}
            [... OTHER CONFIGURATION PARAMETERS ...]
        ]]>
    </configTemplate>

    <event name="Researchers loaded" ref="researchers_loaded">
        <output name="researchers"
                type="xsd:domain:Researcher" collection="true"/>
    </event>

    <request ref="researchers" runsOn="" triggers="researchers_loaded">
        <url>http://.../italianSource/researchers</url>
        <parameters>
            <parameter name="uniID" value="{uniId}"/>
            <parameter name="facID" value="{facultyId}"/>
            <parameter name="depID" value="{departmentId}"/>
            <parameter name="secID" value="{sectionId}"/>
        </parameters>
    </request>
</component>
```


Italian Researchers

Please select either
an university or a sector:

Choose University:

University

TREN

TRENTO

Department

Choose Sector:

Sector

Figure. 7. The descriptor of the Italian Researchers component along with its representation in the Composition Editor

(4) *Configuration parameters.* Parameters that, as opposed to input data, are set up at composition design time by the designer to configure the component's behavior.

(5) *Meta-data.* The component's information, such as name and natural language description of the component itself.

In our platform the component descriptors are implemented as XML file, which must comply with an XML Schema Definition (XSD). The XSD defines both the schema for the component descriptors and the admitted data types. Validating the descriptor against the data types definition we can actually enforce the adoption of the common domain concept model (DCM), which enable smooth composability and no need for data mapping in the Composition Editor, as discussed in Section 5.1.

For example, an excerpt of the Italian Researchers component descriptor along with its representation in the Composition Editor is shown in Figure 7. The component is implemented through a server-side web service. Its descriptor does not present any operation and it has an event called `Researchers Loaded`, which is used to spit out the list of researchers that are retrieved by the associated back-end service. The binding among the service and its client-side counterpart is set up in the descriptor through the `<request>` tag. As shown, this tag includes the information needed to invoke the service, i.e., its end-point URL and the configuration parameters that must be sent along with the request. In addition, the attribute `triggers` specifies the event to be raised upon service completion. The attribute `runsOn`, instead, specifies the component's operation that must be invoked to start the service call. In this particular case, since the component has no operations and no inputs to wait for, when the mashup is started the Mashup Engine automatically invokes the back-end service associated to the component, causing the process execution to start. If we were dealing with a component implemented via client-side JavaScript, we would not need the `<request>` tag, and the implementation binding would be represented by the `ref` attribute of the component operation or event, whose value would be the name of the JavaScript function implementing the related business logic.

The component in Figure 7 has different configuration parameters, which are used to define the search criteria to be applied to retrieve the researchers. We can see the `uniId` parameter. Beside the name of the related label, we must specify the `renderer` to be used, that is, the way in which the parameter will be represented in the Composition Editor. In this case, we are using a text input field with auto-completion features. The auto-completion feature is provided by a dedicated service operation that can be reached at the address specified in the `url` option. Finally, we can see the presence of the `configTemplate` tag, which is just used to set the order in which the parameters must be presented in the component representation in the Composition Editor.

The other main artifact that constitutes a ResEval Mash component is its ***implementation***. As already discussed above, a component can be implemented in two different ways: through client-side JavaScript code (client component) or through a server-side web service (server component). The choice of having a client-side or a server-side implementation depends mainly on the type of component to be created, which may be a UI component (i.e., a component the user can interact with at runtime through a graphical interface) or a service component (i.e., a component that runs a specific business logic but does not have any UI). UI components (e.g., the Bar Chart of our scenario) are always implemented through client-side JavaScript files since they must directly interact with the browser to create and manage the graphical user interface. Service components (e.g., the Microsoft Academic Publications of our scenario), instead, can be implemented in both ways, depending on their characteristics. In the research evaluation domain, since they typically deal with large amounts of data, service components are commonly implemented through server-side web services. In such a way, they do not have the computational power constraints present at the client-side and, moreover, they can exploit the platform features offered at the server-side, like the Shared Memory mechanism, which, e.g., permit to efficiently deal with data-intensive processes. In other cases, where we do not have particular computational requirements, a service component can be implemented via client-side JavaScript, which runs directly in the browser. The JavaScript implementation, both in case of UI and service components, must include the functions implementing the component's business logic.

For example, our Italian Researchers service component is implemented at server-side since it has to deal with large amounts of data (i.e., thousands of researchers), so it belongs to the server components category (introduced in Section 5.3). This type of components, to correctly work within our domain-specific platform, must be implemented as Java RESTful web service following specific implementation guidelines. In particular, the service must be able to properly communicate with the other parts of the system and, thus, it must be aware of the data passing patterns discussed before and the shared memory. Figure 8 shows the interaction protocol with the other components of the platform the service must comply with.

The service is invoked through an HTTP POST request by the client-side Mashup Engine, performed through an asynchronous Ajax invocation (the half arrowheads in the figure represent asynchronous calls). The need to expose all the operations through HTTP POST comes from the fact that in many cases it must be possible to send complex objects as parameters to the service, which would not be possible in general using a GET request. For instance, in our example, the operation is invoked through a POST request at the URL `http://.../resevalmash-api/resources/italianSource/researchers` and the component's configuration parameters (e.g., selected university or department) are posted in the request body. Besides the parameters, the body also includes control data, that is the `key` and the `OutputDataRequired` flag.

Once the request coming from the Mahup Engine is received by the service, the service code must process it following the sequence diagram shown in Figure 8. If the service is designed to accept input data, first it will get the data from the Shared Memory through the API provided by the Server-Side Engine, using the received key as parameter.

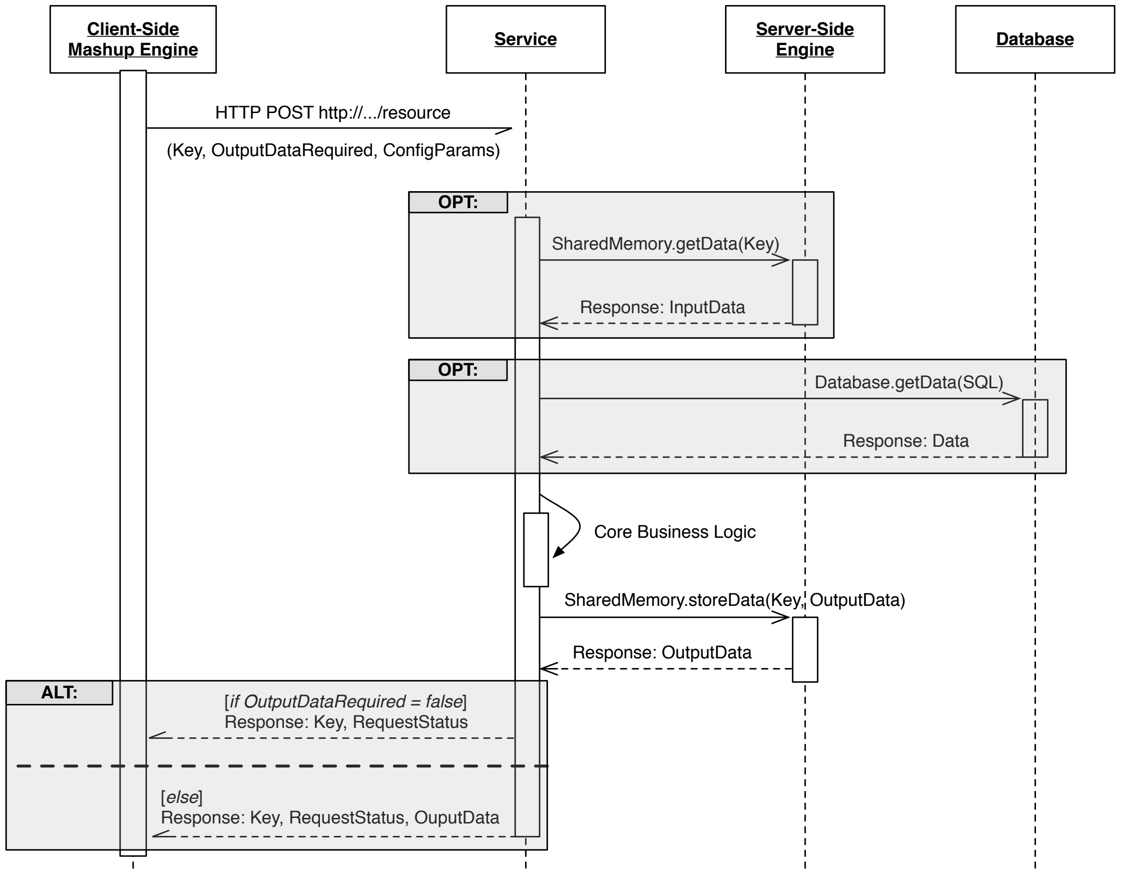

Client-Side Mashup Engine
Service
Server-Side Engine
Database
HTTP POST http://.../resource
(Key, OutputDataRequired, ConfigParams)
OPT:
SharedMemory.getData(Key)
Response: InputData
OPT:
Database.getData(SQL)
Response: Data
Core Business Logic
SharedMemory.storeData(Key, OutputData)
Response: OutputData
ALT:
[if OutputDataRequired = false]
Response: Key, RequestStatus
[else]
Response: Key, RequestStatus, OuputData

Figure. 8. Platform-specific interaction protocol each service must comly with

Then, the service may need to have access to other data for executing its core business logic. The services developed and deployed by us (as platform owners) can use the system database to persistently store their data, as show in the second optional box. This is, for instance, the case of our Italian Researchers component that retrieves the researchers from the system database, where the whole Italian researchers data source has been pre-loaded for efficiency reasons. Third-party services, instead, do not have access to the system database but they can use external data sources as external databases or online services available on the Web. Clearly, the usage of the system database guarantees higher performances and avoids possible network bottlenecks.

Once the service has retrieved all the necessary data, it starts executing its core business logic (for our example component, it consists in the filtering of the researchers of interest based on the configuration parameters). The business logic execution results are then stored in the Shared Memory using the Server-Side Engine API methods. Typically, all the services will produce some output data, although, possibly, there could be exceptions like, for instance, a service that is only designed to send emails.

Finally, the service must send a response back to the Mashup Engine. The response content depends on the `OutputDataRequest` flag value. If it is set to false, as shown in the upper part of the alternative box in the figure, the response will contain the `Key` and the `RequestStatus` of the service (success or error). If the flag is set to true, in addition to those control data, the response will also contain the actual `OutputData` produced by the service logic.

So far, all components and services for ResEval Mash have been implemented by ourselves, yet the idea is to open the platform also to external developers for the development of *custom* *components*. In order to ease component development, e.g., the setup of the connection with the Shared Memory and the processing of the individual control data items, we will provide a dedicated Java interface that can be extended with the custom logic. The description, registration, and deployment of custom components is then possible via the dedicated Component Registration Interface briefly described in Section 5.2.

6. USER STUDY AND EVALUATION

A summative evaluation was conducted to analyse the user experience with ResEval Mash. 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. Domain knowledge was controlled by selecting all users with expertise in research evaluation, computing skills varied in the sample from people with no programming knowledge at all, to expert programmers.

The study applied a concurrent talk-aloud protocol, a technique requiring users to verbalise all their thoughts and opinions while performing a set of tasks. Verbalisation capture techniques have been found to be particularly effective when conducting experimental investigations, which provide an opportunity to study communication between products, designers and users [Jarke et al. 1998; Rouse and Morris 1986]. Responses given during task completion are considered more representative of the behavior and problems users have during assessment [Hands and Davidoff 2001] and concurrent talk-aloud protocols have been shown to encourage participants to go into greater detail, to provide more in-depth evaluation, and help pin-point usability problems and places where their expectations fail to be met [Teague et al. 2001].

6.1 Method

Ten participants covering a broad range of academic and technical expertise were invited to use ResEval Mash. At the beginning of the study, they signed a consent form presenting ResEval Mash as a tool for allowing non-programmers to develop their own computing applications. Then, they were asked to fill in a questionnaire reporting their computing skills and knowledge about research evaluation alongside some basic demographic information (e.g., age and job position). Specifically, participants were asked to estimate their skills with the use of software similar to the Microsoft Office Suite tools, programming languages, flowcharts and mashup tools, on a 4-point scale, ranging from very skilled to no skilled at all. They were also presented with a list of 21 concepts related to research evaluation and asked to indicate for each of them whether they were aware of it and able to understand its meaning, on a 2 point scale (yes vs. no).

After the questionnaire, participants watched a video tutorial (lasting approximately 10 minutes) that instructed them how to operate ResEval Mash. The video introduced the basic functionalities of the tool, quickly explaining the concept of components, configuration parameters, and data compatibility. It then showed how to create a simple mashup of 4 components to display the H-index of the researchers of the Department of Computer Science and Engineering of the University of Trento on a bar chart according to the Microsoft Academics publication source. Finally, the video presented another mashup example used to summarize and reinforce the concepts shown up to this point, where 4 components were connected to visualize on a bar chart the G-index of a researcher (Figure 9.a).

After training, participants were asked to use the system. The first task asked people to start from the first composition presented in the video tutorial and to modify the year parameter of the Microsoft Academic component, to select a different department from the Italian Researchers component and finally to replace the publication source component currently used in the composition with the Google Scholar component. The second task required them to design a composition to compute the participant's own publication count and visualize it on a chart. The correct solution required linking together 4 components, as highlighted in Figure 9.b.

Whilst completing these two tasks, participants were asked to "talk aloud" regarding their thoughts and actions. This interaction was filmed, as was the interview that followed task com-

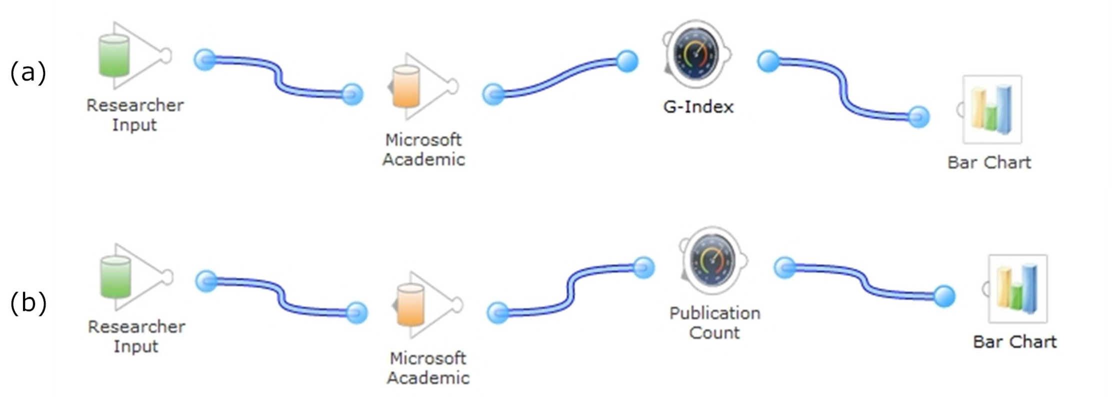


Figure. 9. Mashup compositions to compute G-Index (a) and publication count (b)

pletion. The interview focused on interactional difficulties experienced, the evolution of participants' conceptual understanding over time, and a detailed usability evaluation stressing a feature based assessment reporting which features were considered to be beneficial to interaction, which were understood, and what participants, as users, would like to see in the system.

6.2 Results

The video-capture and talk aloud protocols were used to establish strengths and weaknesses in design and conceptual understanding. A subsequent usability assessment was used to identify the difficulties participants reported experiencing and their understanding of key features of mashup tool interaction. Anonimized data related to the initial and final questionnairs are available on the ResEval Mash project website[5]. Videos recording user interactions with the tool can not be provided for privacy reasons.

6.2.1 *Sample description.* The sample covered a broad range of job positions and technical skills. Half of it was composed of people who reported not being skilled in programming languages, the other half reported being very skilled or good in relation to programming languages. All the participants possessed moderate to no experience with mashup tools. The breakdown of participants according to Position is reported in Table I.

IT Skills	**Position**
High Computing Skills	PhD Students (3), Post-doc (1), Senior Faculty Member (1)
No Computing Skills	Administrative People (3), PhD student (1), Senior Faculty Member (1)

Table I. User categories

On average as a group, participants had a good understanding of the domain. They possessed experience of 80% of the 21 domain specific conceptual components listed during the pre-interaction assessment. This value ranged from a minimum of 48% to a maximum of 100%.

6.2.2 *Usability evaluation.* 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. The IT expert users reflected less before performing their actions and appeared to be more confident during the test. Overall, among the users with lower computing skills there was agreement that more training in the use of the tool

[5]`http://open.reseval.org/project-updates/user-studies/`

would be beneficial, whereas this requirement did not emerge from the more skilled sample. It is worth noticing however that the people reporting this need also indicated a lower level of domain knowledge as compared to the other users.

All participants understood the concept of "component" and had no specific issues in terms of configuring or connecting components. However, the post-doc researcher suggested that it might be beneficial for the system to indicate clearly when a proposed connection was inappropriate or illegal by using colour to differentiate the states of legality or appropriateness. Another participant suggested the possibility of disabling the illegal components from the selection panel when a component was selected in the composition canvas. Selection of components was highlighted as a potential problem, as identification of the right component required some time to be performed. During the study, this did not appear to be a major problem, as only a selected number of components (N= 8) were tested. Yet, it is reasonable to assume that this problem will increase as the number of available components grows. One participant suggested a search feature, to complement the current menu selection interaction mode.

The task requiring tailoring an existing mashup was generally performed better than the task requiring creating a new mashup. In the latter case, a problem emerged with the selection of the first component (i.e., Researcher Input), as several participants selected the Italian Researchers component expecting to be capable to personalise their query there. Saving of configurations was also a source of uncertainty for several participants. The configuration parameters only needed to be filled in by the users and no other action from them was required. This was not clear to the users that in many cases expected an explicit saving action to be performed (e.g., through a "Save configuration" button) and that also expected a feedback to be returned on configuration completion. Several people used the "Close" button after updating the configuration, leading to deletion of the component.

Furthermore, most participants reported some difficulty interacting with the tool due to the physical interaction of double-clicking on the component image in order to open it and been capable to configure its parameters. This constraint was referenced as taking time to learn.

6.3 Discussion

Our study indicates 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 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. Further research will explore the relative role of these two factors by a full factorial experimental study on a larger sample. However, this preliminary study suggested that ResEval Mash is a successful tool appealing both to expert programmers and end-users with no computing skills.

All participants reported a good level of understanding of the basic concepts implemented in ResEval Mash, although some suggestions for improvement were collected, mainly related to verbal labels used to denote components. Most usability issues evinced from behavioral observations can be easily solved. For instance, the uncertainty experienced by several users with saving the configuration parameters can be counteracted by adding an explicit saving option in the interface of the components. A more serious issue was highlighted as regards selection of components, which was found to be an error prone and time demanding task. This problem is likely to increase exponentially with the availability of more components, but it can be partially counteracted by a smart advice system decreasing the number of items available for selection based on a comparison between the current application context and previous successful implementations, as presented in [De Angeli et al. 2011]. For instance, illegal components could be automatically disabled and the one used more often made salient.

Overall, the study provided some interesting results and highlighted the important role of user evaluation in the design of interactive systems. A major finding is related to the ease with which our sample (independently of their technical skills) 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., the ServFace Builder, Namoun et al 2011), which surprisingly did not occur at all in the current study. The mismatch can be due to a different level of complexity of the evaluation tasks, but also to an important design difference. Indeed, ResEval Mash only requires users to connect components as holistic concepts, whereas other tools, such as the ServFace builder required the user to perform complex connections between individual fields of user interfaces. More research is needed to understand the boundaries of ResEval Mash, testing it with more complex development scenarios.

7. 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 currently two main approaches to enable less skilled users to develop programs: 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 and Business Process Management (BPM) community was one of the first to propose that the abstraction of business processes into tasks and control flows would allow also less skilled users to define their own processes. Yet, according to 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 program 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 domain-specific mashup environments, 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, which can be mashed up.

As such, the work presented in this paper is related to three key areas, i.e., domain-specific modeling, web service composition, and mashups, which we briefly overview in the following.

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 [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 (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 [Lédeczi et al. 2001]) 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 [Maximilien et al. 2007]). 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 [France and Rumpe 2005] and [Mernik et al. 2005].

Web service composition. BPEL (Business Process Execution Language) [OASIS 2007] 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 [Active Endpoints, Adobe, BEA, IBM, Oracle, SAP 2007b] and WS-HumanTask [Active Endpoints, Adobe, BEA, IBM, Oracle, SAP 2007a] 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 [Daniel et al. 2010] 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 [Yu et al. 2008] emerged as an approach to provide easier ways to connect together services and data sources available on the Web [Hartmann et al. 2006], together with the claim to target non-programmers. Yahoo! Pipes [6] 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 [7], 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 that 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. In particular, the language does not require users to deal with one of the most complex aspects 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.

The results of our user study regarding the ResEval Mash tool, our domain-specific mashup platform for research evaluation, show that end-users feel comfortable in a mashup environment that resembles the domain they are acquainted with. The intuitiveness of the used components, which represent well-known domain concepts and actions, prevails over the lack of composition knowledge the users (the domain experts) may have and help them to acquire the necessary composition knowledge step by step by simply “playing” with ResEval Mash. Components in ResEval Mash have real meaning to users.

Yet, we also acknowledge that there is still work to be done, in order to turn ResEval Mash

[6] `http://pipes.yahoo.com`
[7] `http://www.servface.eu`

into a even more powerful instrument for research evaluation. Before going publicly online, we still would like to improve the intuitiveness of its user interface, especially as for what regards the configuration of component parameters, a task that was not perceived as intuitive by users. We also have to complete the implementation of some of the components. In the context of both this work and other research conducted in parallel, we have learned that users with only little IT skills may benefit from contextual help [De Angeli et al. 2011], e.g., provided in the form of recommendations that suggest the user which next composition action could make sense. We already designed the respective client-side knowledge base for storing composition knowledge and a respective recommendation engine to provide interactive, contextual help [Roy Chowdhury et al. 2011]; next, we will work on the extraction (mining) of reusable composition knowledge (in the form of composition patterns) from existing mashup models. Joining the power of both domain-specific design and suitable assistance technologies will allow us to widen even further the spectrum of people that are able to develop mashups.

At a more technical level, we will study how to relax our design principle that asks for platform-specific web service implementations, that is, we will try to understand *which* platform-independent services (i.e., services that are not designed to know the platform's concept model and to use platform utilities like the shared memory) we can leverage on for the definition of research evaluation mashups and *how* to integrate them technically into the platform (e.g., via a suitable mediator or dedicated wrappers). This will allow us to improve reuse of existing services and to further assist service developers in implementing research evaluation functionalities (in that we would take over part of the work that makes them platform-specific). The challenge will be to identify the right trade-off between platform dependence and platform independence.

Acknowledgment: This work was supported by EU project OMELETTE (contract no. 257635).

REFERENCES

Active Endpoints, Adobe, BEA, IBM, Oracle, SAP. 2007a. Web Services Human Task (WS-HumanTask) Version 1.0. Tech. rep. June.

Active Endpoints, Adobe, BEA, IBM, Oracle, SAP. 2007b. WS-BPEL Extension for People (BPEL4People) Version 1.0. Tech. rep. June.

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

Daniel, F., Casati, F., Benatallah, B., and Shan, M.-C. 2009. Hosted Universal Composition: Models, Languages and Infrastructure in mashArt. In *ER'09*. Berlin, Heidelberg, 428–443.

Daniel, F., Soi, S., Tranquillini, S., Casati, F., Heng, C., and Yan, L. 2010. From People to Services to UI: Distributed Orchestration of User Interfaces. In *BPM'10*. 310–326.

De Angeli, A., Battocchi, A., Roy Chowdhury, S., Rodriguez, C., Daniel, F., and Casati, F. 2011. End-user requirements for wisdom-aware eud. In *Proceedings of IS-EUD 2011*. 245–250.

France, R. and Rumpe, B. 2005. Domain specific modeling. *Software and Systems Modeling 4*, 1–3.

Hands, D., A. S. and Davidoff, J. 2001. Recency and duration neglect in television picture quality evaluation. applied cognitive psychology. *Applied Cognitive Psychology* 15, 639–657.

Hartmann, B., Doorley, S., and Klemmer, S. 2006. Hacking, Mashing, Gluing: A Study of Opportunistic Design and Development. *Pervasive Computing 7,* 3, 46–54.

Jarke, M., Bui, X., and Carroll, J. 1998. Scenario management: An interdisciplinary approach. *Requirements Engineering 3,* 3, 155–173.

Karlsson, M. and Wikstrom, L. 2006. *Contemporary Ergonomics.* Taylor and Francis, Great Britain, Chapter Safety semantics: A study on the effect of product expression on user safety behaviour, 169–173.

Lédeczi, Á., Bakay, A., Maroti, M., Völgyesi, P., Nordstrom, G., Sprinkle, J., and Karsai, G. 2001. Composing domain-specific design environments. *IEEE Computer 34,* 11, 44–51.

Maximilien, E. M., Wilkinson, H., Desai, N., and Tai, S. 2007. A domain-specific language for web apis and services mashups. In *ICSOC*. 13–26.

Mehandjiev, N., De Angeli, A., Wajid, U., Namoun, A., and Battocchi, A. 2011. Empowering end-users to develop service-based applications. *End-User Development*, 413–418.

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

Monk, A. 1998. Cyclic interaction: a unitary approach to intention, action and the environment. *Cognition 68*, 95–110.

Namoun, A., Nestler, T., and De Angeli, A. 2010a. Conceptual and Usability Issues in the Composable Web of Software Services. In *Current Trends in Web Engineering - 10th International Conference on Web Engineering ICWE 2010 Workshops*. Springer, 396–407.

Namoun, A., Nestler, T., and De Angeli, A. 2010b. Service Composition for Non Programmers: Pro-spects, Problems, and Design Recommendations. In *Proceedings of the 8th IEEE European Conference on Web Services (ECOWS)*. IEEE, 123 – 130.

Nielsen, J. 1993. *Usability Engineering.* Academic Press, California.

Norman, D. A. 1991. *Cognitive artifacts.* Cambridge University Press, New York, NY, USA, 17–38.

OASIS. 2007. Web Services Business Process Execution Language Version 2.0. Tech. rep., `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`. April.

Okeye, H. 1998. Metaphor mental model approach to intuitive graphical user interface design. Ph.D. thesis, Cleveland State University, USA.

Pietschmann, S., Voigt, M., Rümpel, A., and Meissner, K. 2009. Cruise: Composition of rich user interface services. In *ICWE'09*. 473–476.

Rouse, W. and Morris, N. 1986. On looking into the black box: Prospects and limits in the search for mental models. *Psychological bulletin 100,* 3, 349.

Roy Chowdhury, S., Daniel, F., and Casati, F. 2011. Efficient, Interactive Recommendation of Mashup Composition Knowledge. In *Proceedings of ICSOC 2011*. Springer, 374–388.

Teague, R., De Jesus, K., and Ueno, M. 2001. Concurrent vs. post-task usability test ratings. In *CHI'01 extended abstracts on Human factors in computing systems*. ACM, 289–290.

Thomas, B. and van-Leeuwen, M. 1999. *The user interface design of the fizz and spark GSM telephones.* Taylor & Francis, London.

Yu, J., Benatallah, B., Casati, F., and Daniel, F. 2008. Understanding Mashup Development. *IEEE Internet Computing 12*, 44–52.