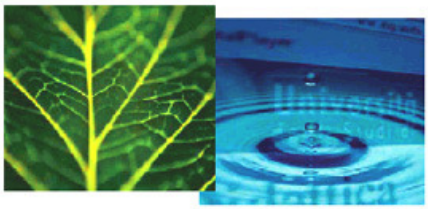
PhD Dissertation

International Doctorate School in Information and Communication Technologies

DISI - University of Trento

Domain Specific Mashup Platforms as a Service

Stefano Soi

Advisors:

Prof. Fabio Casati and Dr. Florian Daniel

Università degli Studi di Trento

March 2013

Abstract

Since the Web 2.0 advent, Web users have gained more and more power moving their role from simple information consumers to producers. In line with this trend, mashup technologies have immediately attracted a lot of attention and many research investments since their birth. The big mashup promise was to ***bring application development to the masses****, so that any Web-educated worker, also non-IT skilled, could implement his/her own situational applications (i.e., relatively small applications meant to address a temporary need of one or few persons) exploiting the simple paradigms and visual metaphors provided by mashup tools. After a decade from the first mashup tools, though,* ***mashups are not really part of people's everyday life*** *and are still rather unknown technologies that — beside some exceptions —* ***hardly find concrete application*** *in the real world.*

During our research in this field our high-level goal was to ***foster the adoption of mashup technologies by end users****. Aiming at this, we identified three main characteristics that must be reached by mashup technologies to get to the expected diffusion: (i) usefulness and (ii) usability for the end users and (iii) affordability for the developers of the respective mashup tools (in terms of required skills, time and cost). We identified lacks in these achievements as main hindering factors for the wide adoption of mashup technologies. Making mashup technologies* ***useful****,* ***usable*** *and* ***affordable****, therefore, are the three main challenges we addressed in our research work. This work contributes to the achievement of all these three major goals: first, by enabling a so-called* ***universal in-***

***tegration paradigm**, focussing on the creation of more powerful and complete mashups allowing data, application logic and user interface integration in one single language and tool; then, by introducing and developing **domain specific mashup** technologies, able to lower mashups' complexity and make them usable by domain experts (i.e., end users expert of a given target domain); finally, by realizing a system able to generate **domain specific mashup platforms as a service**, basically relieving developers of platforms implementation and, therefore, making platform development affordable. This thesis specifically focusses on the last two points, i.e., on the domain specific mashup approach and on the semi-automatic generation of domain specific mashup platforms.*

Keywords

Mashups, Mashup tools, Domain specific mashups, mashup platform design, mashup platform meta-design, mashup platform as a service

Contents

Structure of the Thesis

This thesis is structured as a collection of articles that we published during this work and that, therefore, have been reviewed and accepted by peers in the scientific community[1]. This dissertation presents an executive summary of our research work, providing an overview of the main problems and solutions, and the references to our articles describing in deeper details the various parts of the work. The executive summary is structured as follows. First, we provide the context of this thesis and some background work (Section 1 and 2, respectively). Then, we define the problems we identified in this context (in Section 3) and the proposed solutions to address them, which are discussed in Section 4 and 5. Section 6 provides the list of our main contributions and Section 7, finally, discusses the key lessons we learned during this research work and its main limitations. All the articles composing this work (cited in Figure 3) are included as appendixes at the end of this dissertation.

[1]The only not yet published article in this thesis is [23], which is a technical report and will be submitted for publication soon

Executive summary

1 Context

During the last decade a vast amount of functionalities have been made available as online services, in form of Web Services, REST or JavaScript APIs, RSS/Atom feeds and so on. While these services can also be used independently of each other, putting them together to create a value-adding combination could lead to much more fruitful results. This is exactly what mashups try to achieve. We define mashups as web applications that integrate data, application logic, and/or user interfaces (UIs) sourced from the Web. We define mashup tools as development environments allowing the creation and execution of mashup compositions integrating the three types of components just mentioned, typically through a graphical Web interface. In addition, most mashup tools aim at empowering non-IT skilled Web users to develop such composite applications, therefore, enabling *end user development* (EUD). A number of studies [13, 2] discuss the benefits of moving the development of this kind of composite applications from IT-experts to non-programmers. This would be a radical paradigm shift bringing two main advantages: (i) avoid requirements transfer from domain-experts to IT-experts and (ii) allow the effective development of situational applications, i.e., applications addressing transient or very specific needs regarding one or few persons for which a standard software

development lifecycle would not be time- and cost-effective. These reasons have motivated our and others' research in the mashup area and pushed the proliferation of mashup tools.

The goal of this thesis is to **bring mashup development to end users**. With the term end user we refer to non-programmers possessing a minimum level of IT skills, for example, that of the average Excel user or perhaps, using Nardi's terminology [20], that of the "local developers" (i.e., non-programmers passionate about new software technologies).

To achieve this goal, we identify two main characteristics that mashup technologies (intended in the broad sense, including mashup approaches, models and tools) must have to be suitable for end users. Mashup technologies must be:

- **powerful**: they must provide enough expressive power to allow the users to realize applications able to address and solve real-life practical problems;
- **simple**: they must be simple enough so that also non-IT skilled end users are able to use them to build their own — useful — compositions, achieving, thus, so-called *end user development* (EUD).

We identified lacks in these achievements as main hindering factors for the wide adoption of mashup technologies by end users. Many mashup solutions allow users to only define part of an application (e.g., the service composition) requiring the manual implementation and integration of other needed parts (e.g., the UIs). These solutions alone are therefore not powerful enough to let users create complete applications from start to end. Moreover, they are typically too complex to be used by end users to create real-life, useful applications; for example, Yahoo Pipes[1] presents compositional elements representing programming concepts, like feed fetching or regular expressions, that end users are not able to understand.

[1]Yahoo Pipes hompage: `http://pipes.yahoo.com`

Making mashup technologies **useful** (by making them more powerful) and **usable** (by making them simpler), therefore, is the main challenge we addressed in our research work. To further foster mashup technologies' adoption, we also aim at making mashup platform development easier and faster, allowing the achievement of efficient cost/benefit ratios. In other words, this means making them **affordable** for those who need to develop a mashup platform.

This dissertation focusses on the solutions we developed to address the usability and affordability challenges, discussed in Section 4 and 5, respectively. In the next section we present our background work, which contributes to improve the usefulness of mashups.

2 Background: Universal Integration

When we started our investigations in the mashup research area, we identified a gap in the integration possibilities offered by the mashup tools available at the time, which was limiting mashups' expressive power. Some of the available tools mainly focussed on the integration of data (e.g., Karma [25]), others more on the integration of services (e.g., ServFace Builder [21]) and yet others on the composition of user interfaces (e.g., Intel Mash Maker), but none of them was providing integration spanning over different layers allowing the composition of data, application logic and user interface (UI) components in one and the same language and tool. This was a big gap, since most applications (in particular in the Web context) need to integrate different types of components. This gap was significantly limiting the usefulness of the mashups that could be produced, which often required a manual — expert — intervention for the development and integration of the missing parts (e.g., to implement suitable UIs to be integrated with a service composition). The seamless integration at the data, application

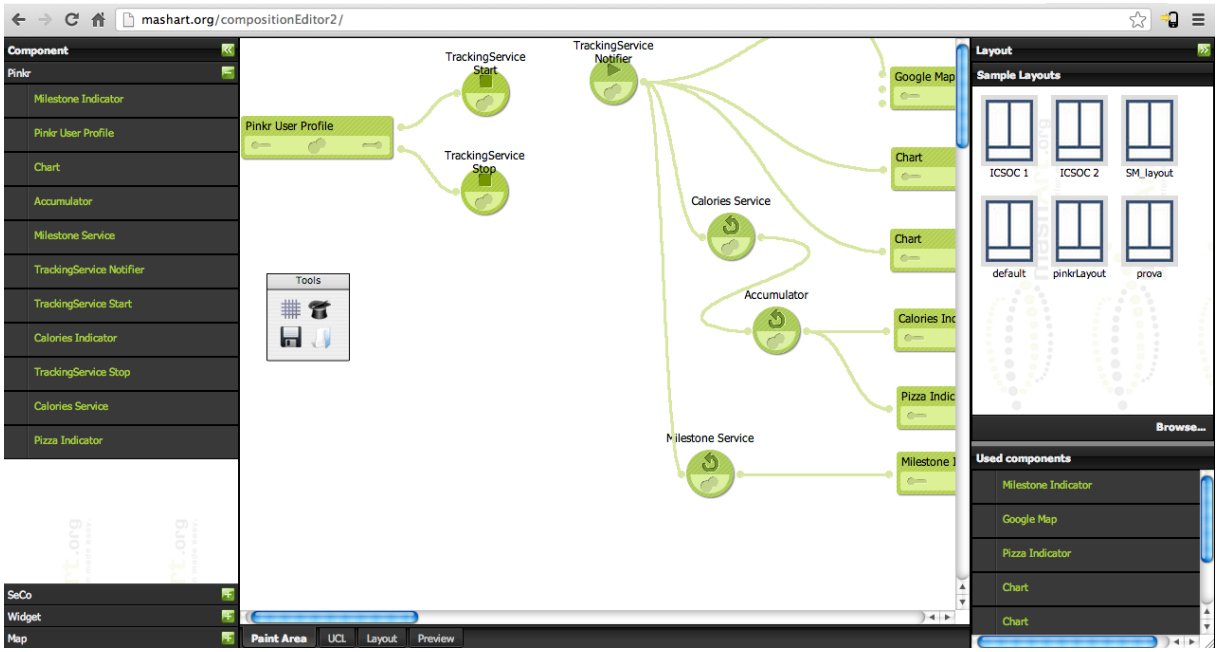


Figure 1: The mashArt mashup tool

logic and UI layer is what we called *universal integration*. To address the universal integration gap, we started an industrial project aiming at building a mashup platform providing universal integration and targeting end users. The resulting mashup tool is mashArt [4, 5] (shown in Figure 1), a hosted Web mashup tool allowing its users to develop and run compositions including components belonging to all the three mentioned layers[2]. The major challenge faced during the mashArt project was the design of both a model and a corresponding language able to accommodate the different types of components and the different composition paradigms they require (e.g., UI integration is event driven while service integration is typically managed through the control flow paradigm). Another challenging aspect was the design and implementation of a user friendly, graphical development environment and of a runtime engine able to execute the compositions generated by the development environment. We coherently included both

[2]The mashArt project has been funded and developed in collaboration with SAP, Palo Alto - California

the development and execution environments in the same tool, so as to enable a faster and simpler application development and testing lifecycle. Both environments run within the browser on the client side, although some functionalities are implemented on the server-side. The runtime engine, in particular, also comprises a set of server-side modules needed to support asynchronous services integration in the mashup compositions. This is another innovative feature of the mashArt tool, representing an important contribution of the project.

Following on the same research track, we started another industrial project, MarcoFlow [11, 10, 12], to further investigate the universal integration concept[3]. In this case, we decided to build upon standard service composition technologies, which are widely adopted and developers are already acquainted with. We extended the service composition layer to allow the easy integration of UI components, possibly distributed over multiple Web pages (what we called *distributed UI orchestration*), so that human actors could be integrated in the mashups, i.e, in UI-empowered business processes. This is different from other solutions like BPEL4People and WS-Human Tasks, since these two specifications only focus on the coordination of human tasks and do not support the design of the UIs for task execution. In this case, instead, we use the integration of the UI layer as means to integrate the "human layer". In the MarcoFlow project we can identify two main contributions: first, we extended the standard BPEL model and language to integrate UI-related constructs (introducing the concepts of UI component, page and user associated to a page). Then, we designed and implemented a suitable architecture including a standard BPEL engine (managing the process orchestration), a client-side runtime engine running in the browser (managing the UI components included in the process,

[3]The MarcoFlow project has been funded and developed in collaboration with Huawei, Shenzhen, China

which are implemented in JavaScript), and a middleware managing the communications among the two engines and carefully performing protocol translations and message buffering/proxying. We also suitably extended a BPEL editor and developed a deployment system, providing a platform covering the whole application lifecycle: development, deployment and runtime. In addition, MarcoFlow allows the design of *collaborative processes*, that is, different users can participate and interact with a same process instance, concretely managing it in a collaborative fashion. This is possible by assigning the different UI components included in a process (that, as we said, can be distributed over multiple pages) to different users, which can access their own interface to interact with the process and with the other collaborators. This feature is quite common in the service composition area, but is new in the mashup context. MarcoFlow is clearly targeting programmers (with service composition skills) and not end users.

As examples, in [9] and [8], we provide two use cases showing how mashArt and MarcoFlow tools, respectively, can be successfully used to build compositions in the search computing context.

3 The Problem

During our research work within the mashArt and MarcoFlow projects we contributed to address the first of the three high-level challenges we discussed above, i.e., making mashups useful. The other goal of the mashArt platform was to also tackle the second challenge in the list, that is, making mashup technologies usable by non-programmers, a still open issue that none of the available mashup solutions was able to effectively address. Despite our intentions, similarly to other tools, mashArt failed in enabling end user development.

Today, the situation is still the same. The main cause we identify behind

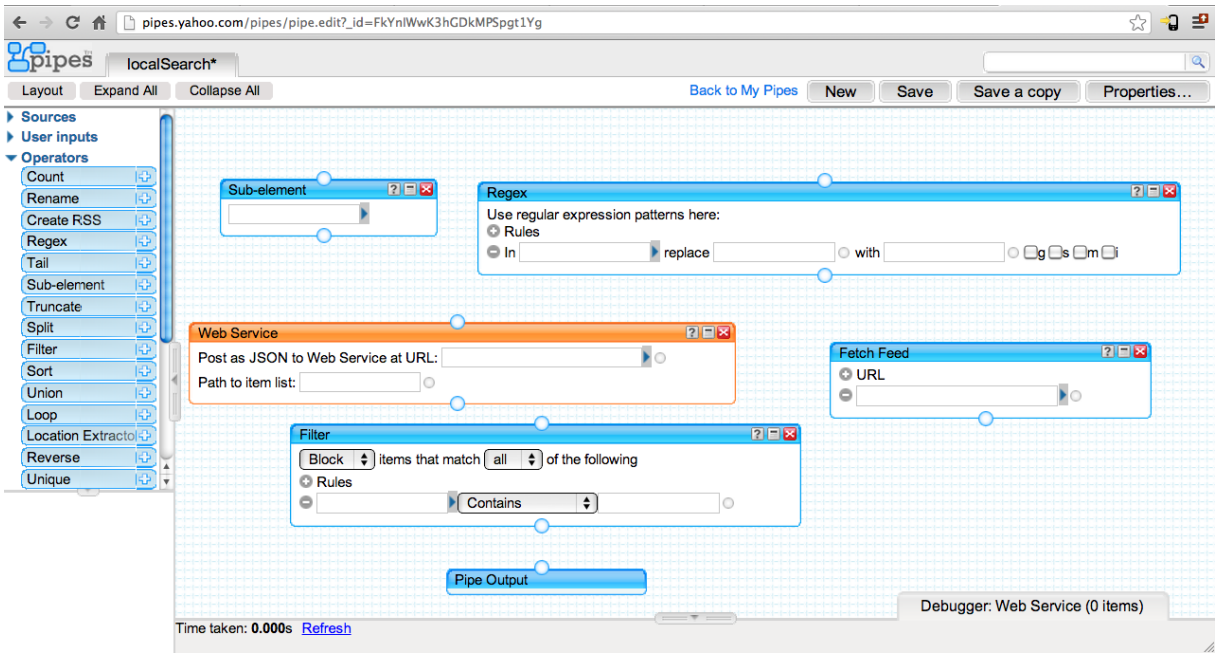


Figure 2: The Yahoo Pipes mashup tool

this problem is that the mashup platforms developed so far expose concepts, terminology and functionalities that are **too low-level and complex for end users**. Keeping a low level of abstraction allows having powerful and flexible tools that are, though, too complex for non-programmers. Non-IT skilled users are typically not able to use these tools or can achieve only the development of toy applications, i.e., applications too simple to be actually useful in real-life scenarios. Giving a concrete example, Figure 2 shows the popular Yahoo Pipes mashup environment and some of the components available in the tool. As exemplified in Figure 2, mashup tools typically require the user to be aware of concepts like Web Service, RSS feed, feed fetching, data filtering, data mapping and the like. As Namoun et al. demonstrated [19, 18], non-programmers do not understand them and, consequently, they do not manage their usage and, even less, their composition.

The core problem is to find the right balance among three dimensions: *expressiveness*, *flexibility* and *usability*. We think we cannot have tools

that are characterized by (i) a flexibility level allowing us to cover a wide range of application domains, (ii) an expressive power allowing users to go beyond simple, toy applications and (iii) a usability level allowing non-programmers to effectively use the tool. Therefore, we must understand which dimensions are more important for our goals and how to balance them accordingly. The only possible meaningful balance, in our opinion, is to limit flexibility, since we want our tools to allow the development of non-trivial mashup compositions (i.e., high expressiveness, which impacts on mashup usefulness) and to be actually usable by non-programmers (i.e., high usability). Limiting the flexibility dimension only means having mashup tools focussing on a well-defined application domain but still presenting high expressiveness and usability levels thanks to the implementation of solutions specifically addressing the composition needs of that domain. This is in line with the requirements discussed in Section 1, where we identified high levels of expressive power and simplicity of use as fundamental characteristics for mashup technologies to be effective. The intuition, therefore, is that we need *domain specific mashup* (DSM) solutions, which is the approach we propose to address the usability challenge, as we discuss in Section 4.

However, an objection that could be raised is that the development of this kind of solutions — which are as demanding as developing a general-purpose tool but designed to be used by a limited audience — may be in general **not affordable** (i.e., result in a non-sustainable cost/benefit ratio). We address this issue by providing specific solutions supporting and automating DSM platforms design and development, as discussed in Section 5.

The path that led us to the development of the solutions addressing the problems highlighted in this section is summarized in Figure 3. The figure shows our research path using the articles we published, providing a high-level overview of our work.

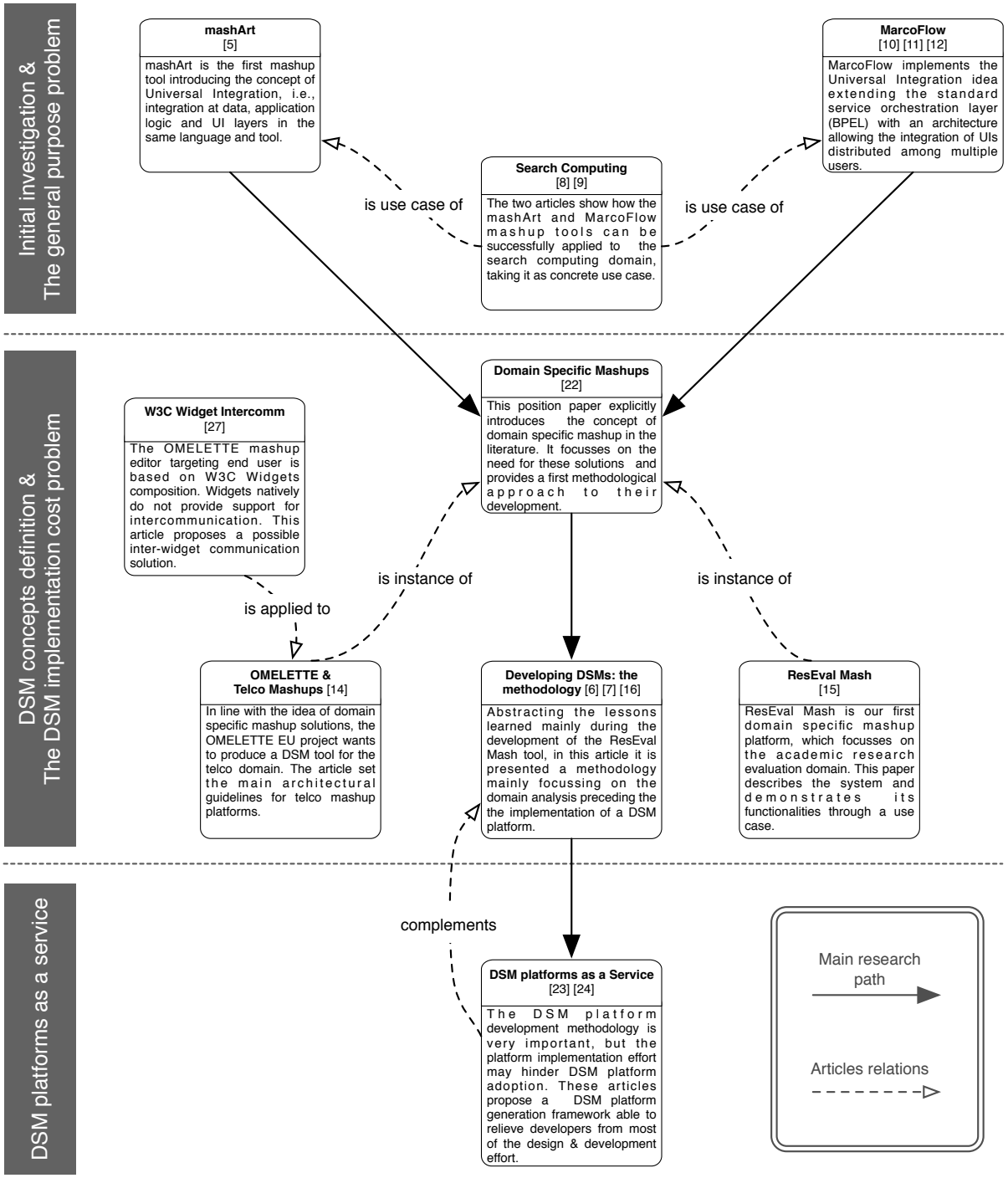


Figure 3: The research path evolution and the article relations

4 Domain Specific Mashups as EUD Enablers

Our solution to address the usability challenge without limiting mashup usefulness is to design mashup platforms focussing on one application domain only. This kind of tools is what we call *domain specific mashup tools*.

Since these tools are tailored to a specific domain, they expose concepts, terminology, functionalities and dynamics (i.e., semantics) that the domain experts using them (i.e., end users expert of a given target domain) are already acquainted with and that they can therefore understand and effectively manage. This is the key of the approach to enable end user development.

After introducing this new concept and approach in [22], we applied it to the design and development of a mashup tool for the research evaluation domain: ResEval Mash[4] [15, 16, 6, 7]. Designing this tool we faced three main challenges:

- understand what are the most important aspects to analyze of an application domain and how these can be formalized through suitable concrete artifacts, to be used to drive platform design;
- understand how to make mashup platforms (including, e.g., editors, runtime environments, models) more effective (i.e., powerful and usable) through the injection of the domain information encoded in the artifacts mentioned above (i.e., which aspects of the tool to modify based on this information and how);
- perform the analysis and formalization tasks for the research evaluation domain and, based on this task's outcomes, design and implement the complete domain specific platform for this domain.

[4] I worked on the design of the methodology discussed below in collaboration with Muhammad Imran. I did not directly work on the tool itself, which has been developed by Muahammad Imran only.

The analysis phase took us several months, also because, in parallel, we defined how to perform the analysis itself, i.e., which domain aspects to consider and how to formally represent them through concrete artifacts. This required many analysis, abstraction and modeling efforts. The actual design and implementation of the DSM tool took even longer, since we had to develop both a mashup development and runtime environment addressing the domain specific requirements elicited during the analysis phase. For example, from the analysis we found that our domain is characterized by the need of dealing with large amounts of bibliographic data. To address this specific requirement we designed an engine supporting data passing by reference, a peculiar feature proposed for Web Service composition [26], but that no other mashup platform provides. This way, there is no need to pass huge data loads among the services involved in the mashups, avoiding the possible associated network bottlenecks potentially disrupting the mashup execution performance.

Abstracting the lessons learned during the ResEval Mash development, we devised a structured **methodology** for the design of DSM platforms, which supports DSM platform developers through this phase. The methodology focusses on the domain analysis and formalization phase, defining the sequence of steps to be done and the artifacts describing the domain to be produced as output. This methodology [7, 16] represents a relevant contribution of this work.

Furthermore, we have also validated the DSM approach running a user study where two groups of domain experts, one with high computing skills and one with no computing skills, were asked to use ResEval Mash to compose some research evaluation processes. The user study results [7] show that also non IT-skilled users are able to build non-trivial processes using our DSM tool, validating therefore our approach as effective EUD enabler.

5 DSM Platforms as a Service

Building a mashup platform is definitely a non-trivial task, which requires to be aware of many design issues and related possible solutions, as described in [1]. Developing DSM platforms is, in general, even more complex.

Developing ResEval Mash required many months of work. Surely, the domain analysis and formalization phase can benefit from the availability of the methodology we designed. This methodology helps platform developers stating which domain aspects to analyze and how to formalize the analysis outcomes through suitable artifacts. Thanks to this, the domain analysis phase can be significantly shortened. The implementation phase, instead, does not have significant support allowing us to alleviate developers' work. As described in Section 3, in this situation, developing DSM platforms may result not affordable.

To overcome this affordability problem, we designed and implemented a **DSM platform generation framework** relieving developers of most of the effort required to implement a DSM platform. This system is detailed in [24] and [23]. Although this framework can be very beneficial also for the development of generic — non domain specific — platforms, it is particularly useful in the context of DSM platform development since it allows developers to create highly customized mashup platforms able to address domain specific requirements. Figure 4 shows the functional architecture of the generation framework.

The simple interface of the *language and platform design and generation tool* (shown in Figure 5), allows *platform designers* to select a set of conceptual features to be supported by the target DSM platform, letting them abstract from low level design and implementation details. Conceptual features allow developers to state high-level requirements like, e.g., the support for data flow or control flow paradigms, conditions, UIs, data

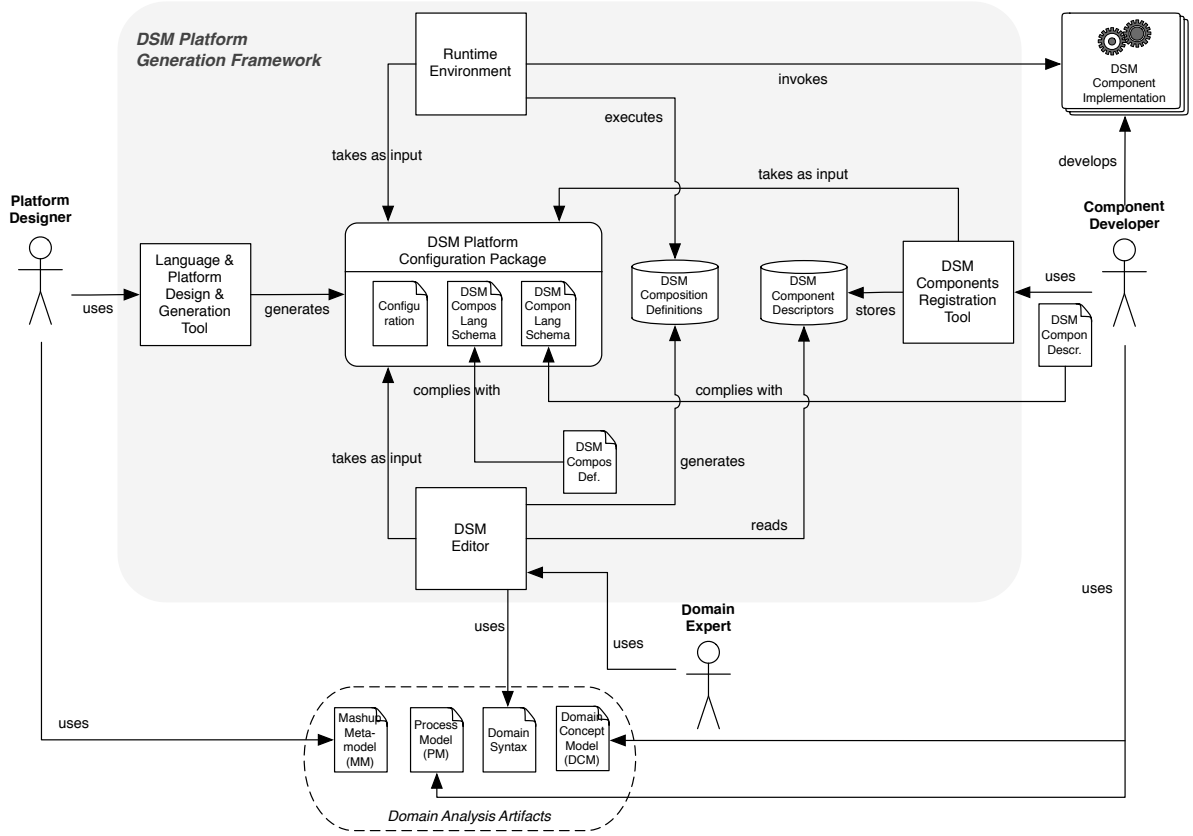


Figure 4: Functional system architecture.

passing by reference or by value, and similar features. Based on a set of selected features, the framework automatically generates a *DSM platform configuration package* containing:

- a *configuration* document. This document contains the list of the features selected by the platform designer, the references to the *domain analysis artifacts* needed by the *DSM editor* (provided by the designer), and the references to the component descriptors and compositions repositories;
- the schema definitions of the *DSM composition language* and *DSM component descriptor language*, which *DSM composition definitions* and *DSM component descriptors*, respectively, must comply to. The languages defined by these documents are generated using the *unified*

mashup meta-model as base and include all the constructs needed to support the set of selected features.

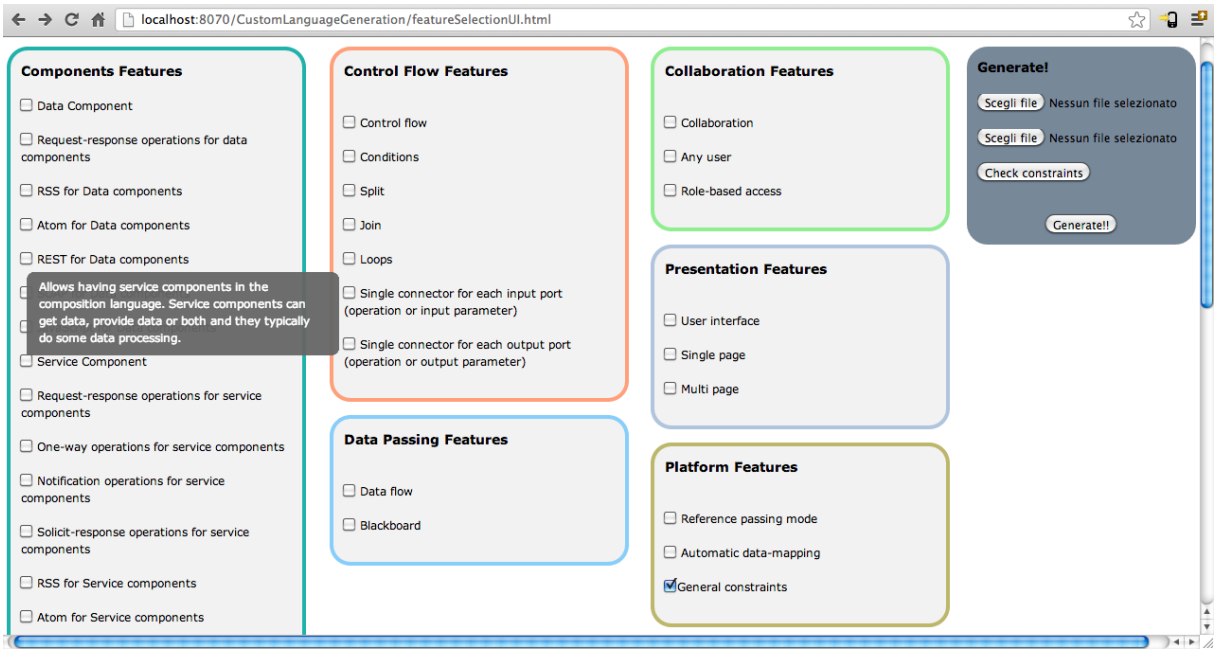


Figure 5: Features selection user interface used by the platform designer

In addition, the framework provides a *runtime environment* and a *DSM editor* that are able to adapt their functionalities and behavior based on the information included in a configuration package they take as input.

The *runtime environment* executes the *DSM composition definitions* stored in the composition repository. During composition execution it interacts with the *DSM components* invoking their operations.

The *DSM editor* is used by the *domain experts* to graphically design the mashups. To show the available DSM components to domain experts, the editor reads the *DSM component descriptors* from the component descriptors repository. Finally, the editor generates a representation of the designed mashup compliant with the DSM composition language defined within the configuration package taken as input, and stores it in the compositions repository.

Indeed, our DSM platform generation framework generates "logical platforms". In fact, it only generates DSM platform configuration packages and not the runtime engine and mashup editor program code. Passing a configuration package as input to the runtime engine and to DSM editor (which are physically deployed on our server) makes them work with the languages, features, components and compositions associated to the specific DSM logical platform described by the given configuration package. This approach allows us to provide hosted DSM platforms as a service, which are already deployed and ready to work.

The main challenges we faced building the generation framework are:

- understand the possible requirements that different target DSM platforms may have and how to support them;
- define a platform conceptual design approach allowing developers to design DSM platforms reasoning at the higher level of abstraction of conceptual features, letting them concentrate on the core requirements of the target platform without having to struggle with low-level design and implementation choices. This requires first to abstract the wide variety of identified requirements into a set of conceptual features, each one representing a subset of these requirements. Then, to associate each feature to a pattern of low-level design choices able to support the set of requirements represented by the feature itself. For example, supporting UI mashups requires designing a language including concepts like page and viewport, and design a platform supporting the integration, for instance, of widgets or JavaScript components;
- design a system enough flexible to integrate all the derived features and able to provide suitable DSM composition and component description languages, runtime environment, and mashup editor based on a given set of selected features. This includes the following challenges:

- design a mashup meta-model, that is able to integrate the different constructs needed to support all the conceptual features. Due to its comprehensiveness, this meta-model cannot be executable since it contains incompatible constructs (e.g., data flow paradigm and global variables cannot coexist since there could be conflicts in the data passing logic);

- design an approach and a set of algorithms allowing us to generate a DSM composition language and a DSM component description language, based on a set of selected features. The approach must guarantee that generated languages are consistent, i.e., do not include incompatible constructs;

- design a runtime environment able to execute the compositions defined through any language generated by our system, taking into account the selected features;

- design a mashup editor that, based on the selected features, exposes suitable functionalities and compositional elements to support all of (and only) them, finally allowing domain experts to produce mashup composition definitions complying with the generated DSM languages.

Addressing these challenges required substantial analysis, abstraction and modeling efforts (in particular, for designing the base models used for the language generation [24]) and strong software design and implementation skills (for all the generation algorithms and, in particular, for the "adaptive" runtime environment and editor [23]). Clearly, another fundamental enabler allowing us to successfully address these challenges has been our prior experience in the mashup world, thanks to the participation to several

mashup projects, i.e., mashArt[5], MarcoFlow[6], ResEval[7] and OMELETTE[8] [27, 14], where we have been directly involved in the design and implementation of different mashup tools.

The DSM platform generation framework is able to relieve developers of most of the implementation effort. Moreover, it also enables the conceptual design of DSM platforms, letting developers reason in terms of conceptual features and guaranteeing consistent languages and functional tools supporting the specific needs of the target domain. Clearly, non negligible efforts are still on developers' shoulders. In particular, they have to implement the mashup components, a fundamental and demanding task in terms of time and resources; however, for already implemented services, we provide simple "componentization" mechanisms only requiring the creation of a component descriptor specifying the main component's properties and interface. In addition, depending on the specific needs, also the editor may require some extension or personalization. We provide a basic editor able to support the identified features and able to work with any DSM language generated by our system. However, there could be particular cases where the editor may benefit from domain specific user interface customization or from additional mechanisms improving the user experience. In this cases, the editor can be modified and extended by developers building upon the basic editor provided.

[5]mashArt project's homepage:
`https://sites.google.com/site/mashtn/industrial-projects/mashart`
[6]MarcoFlow project's homepage:
`https://sites.google.com/site/mashtn/industrial-projects/marcolfow`
[7]ResEval Mash project's homepage: `http://open.reseval.org/`
[8]OMELETTE project's homepage: `http://www.ict-omelette.eu/`

6 Contributions

Throughout our research activities we designed and developed a range of models, techniques, approaches, algorithms and tools that, properly integrated together, compose the methodology and generation framework discussed in this dissertation. Next, we summarize the key contributions of our work.

- **DSM concept and approach**. We introduced the concept of domain specific mashup platform and the underlying approach [22]. Many other research fields exploit the "domain specificity strategy" to create more effective software solutions, but this approach in the literature was not associated to the mashup context yet.

- **User study**. We conducted a user study to validate the DSM approach using our DSM tool for research evaluation (ResEval Mash) [7]. The results of the study show that non-IT skilled users are able to manage composition tasks when provided with a DSM tool like ResEval Mash, supporting the viability of the DSM approach.

- **DSM methodology**. We defined a general methodology for DSM platform design [7, 16, 6]. This methodology describes the sequence of steps needed to analyze a domain and a set of artifacts to formalize the analysis outcomes to be used in the actual platform development phase.

- **Unified mashup meta-model**. We designed a conceptual, unified mashup meta-model able to seamlessly integrate the variety of constructs needed to support all the DSM platforms' requirements we identified analyzing existing mashups types and tools [24]. In addition, we defined a set of translation rules to transform the unified

mashup meta-model into a *unified mashup language*, i.e., a translation of the model into a more easily processable format (we use XSD for the language schema definition).

- **Conceptual mashup features**. We defined a set of conceptual mashup features [24] (abstracting the above identified requirements) allowing developers to reason at a conceptual level and forget about lower level language details. Each feature definition also includes: (i) the specification of possible compatibility or dependency constraints the feature may have with respect to any other feature (needed to guarantee languages consistency), and (ii) the specification of the unified language fragments required to support the feature itself (needed for the language generation algorithms) .

- **DSM platform conceptual design approach**. We designed a conceptual design approach allowing developers to design DSM platforms simply selecting a set of conceptual features representing their core requirements. Letting developers reason at the higher level of abstraction of conceptual features allows them to concentrate on the core requirements of the target platform without being diverted by low-level design and implementation choices.

- **DSM platform generation framework**. We designed and implemented a system (integrating with the above methodology) for DSM platforms generation that developers can use to build DSM platforms, addressing their target-domain specific needs, more easily and rapidly and, therefore, at a more affordable cost [23]. This framework includes and integrates:

  - a **platform and language design tool**, that is, a Web application including a simple interface allowing platform designers to

perform the conceptual design of the target DSM platform and languages. The design is realized simply selecting the set of features to be supported to address the platform requirements. In addition, platform designers must also upload a set of required artifacts previously produced following our methodology;

- a set of **DSM language generation algorithms** for the definition of consistent DSM languages [24], which represent a fundamental piece of any mashup platform since they set the platform's expressive power's bounds. The algorithms, based on a set of selected mashup features, extract DSM composition and component descriptor languages from the unified mashup language. All and only the constructs needed to support the selected features are included in the generated DSM languages. Languages' consistency is guaranteed by a validity check of the set of selected features, performed through feature constraints verification;
- a **runtime environment** for the execution of the mashups defined in any DSM language produced by our framework. It is able to support all the identified conceptual features and to adapt its behavior according to the set of selected features passed as input;
- a **mashup editor** able to adapt its functionalities and behavior based on a set of selected features passed as input. This editor exposes to the domain experts all and only the compositional elements and functionalities needed to support the selected features. The editor is able to work (i.e., read and generate) with mashup defined in any DSM language produced by our framework.

7 Conclusion

In this work we addressed the main challenge of bringing application development to end users. They key of our approach to address this big challenge is to develop domain specific mashup solutions to empower domain experts to develop their own applications. In this context, we provided important contributions going towards the EUD goal. Next we evaluate our approach and solutions and discuss the lessons learned during our research work. Finally, based on these lessons, we provide a set of promising directions for future works.

7.1 Validation and Limitations

We validated the general domain specific mashup approach through the user studies ran on ResEval Mash [7], which have shown that domain experts are able to manage composition tasks when provided with domain specific mashup tools recalling concepts and semantics of the domain they work in, therefore, validating the DSM approach as effective EUD enabler.

Moreover, in [24, 23] we have shown that the set of features we support allows the generation of DSM languages supporting the requirements of different types of mashup tools, thus, providing a validation for our conceptual language generation approach.

Validating the DSM platform generation framework itself is extremely complex. Doing it requires the availability of a number of developers (i) able to use our framework (i.e., fully aware of mashup technologies and able to design a mashup platform), (ii) experts in a given application domain and (iii) willing to work with us for a relatively long period (in the order of several weeks). In addition, a test group developing similar platforms through the standard, manual development approach would be also needed. Being able to perform a similar study would be clearly useful to

evaluate and collect feedbacks about the usability and the effectiveness of our DSM platform generation framework, but it is evident that this would be extremely costly in terms of resources (i.e., mashup platform developers) and, thus, almost impossible to realize.

The DSM platform generation framework provides a set of features allowing us to build languages and tools supporting many different domain requirements. However, the list of identified features comes without the claim of completeness and it is meant to grow over time to increase the spectrum of requirements the system can effectively support. The framework will be available as open-source project to let the community participate in its evolution, e.g., including the support for new features or improving the current runtime or development environments.

Regarding the mashup editor, we still provide a basic editor whose implementation has still to be completed to support all the identified features. As already discussed in Section 5, depending on the specific requirements developers may have, the editor may benefit from domain specific user interface customizations or from additional mechanisms improving the usability. In this cases, the editor can be modified and extended by developers building upon the basic editor provided. In addition, the editor uses the most adopted interaction paradigm in the mashup context, i.e., a visual language based on wiring. It does not support other interaction paradigms adopted by other tools, for instance, based on textual DSLs, editable examples, or spreadsheets.

The runtime environment is fully working and supports all the conceptual features we identified.

Finally, in general we do not argue we can build effective platforms for any domain, but we argue that our methodology and system can be applied to many domains allowing a faster and more affordable development of DSM platforms for them.

7.2 Lessons Learned

Our research activities were driven by the inspiring idea that mashup technologies can enable the radical paradigm shift making non-programmers the real designers and developers of their own applications, that is, enable end user development (EUD). Clearly, the development of, e.g., complex and/or large software systems will still require the work and expertise of software architects and professional developers. However, for the development of simpler applications (realizable through the lightweight composition of the huge variety of available data, services, APIs and UI widgets) this paradigm shift would have a huge impact, moving their development from IT departments directly to the final consumer (or *prosumer*, at this point) and enabling the development of situational applications that today cannot just be implemented, since they typically require too many resources to be developed following the standard software development lifecycle.

During our work we have learned that to make mashup tools more usable (i.e., to go towards EUD) should be the tools to adapt to users and to their mindset and habits, and not the opposite. This is the difference between DSM solutions and general-purpose ones. The former provide users with compositional elements resembling the concepts they face and manage in their everyday life, which, therefore, they are able to understand and manage. The latter, instead, require users to map the concepts they know to lower-level compositional elements, which, suitably composed, can be able to represent those concepts. This abstraction and mapping exercise, though, is far from the possibilities of the non-IT skilled end users, since it typically require programming knowledge (to understand the low-level compositional elements) and abstraction skills (to map them to the concepts familiar to the user) that end users do not have. These lessons

motivated us to design the DSM approach proposed in this work, which allows the creation of simpler mashup tools that, being domain specific, are closer to domain experts' mindset and, therefore, more usable for them.

We are convinced that the DSM approach is an important starting point to achieve our main goal, i.e., bring application development to end users, but it must be further developed and merged with other approaches and technologies. In our experience we have been involved in the design of other approaches going towards EUD and we realized that they can complement each other, leading to really effective solutions to enable a wider and wider range of users to develop their own applications. Our approach goes in the direction of simplifying to the possible extent the mashup tools. Other approaches follow different ways to enable EUD; for example, a promising research track focusses on the development of technologies to assist end users during the mashup development phase, e.g., recommending possible needed components or how to compose them. An example of this kind of tools is Baya [3], which has been developed by our research group and is being applied in the context of the OMELETTE project. The convergence of the DSM approach with other technologies like the one just described is, in our vision, the right way to follow to enable EUD in real-life scenarios.

Another important lessons we learned, and that we try to apply to our solutions, is that to achieve EUD, the tools we provide to end users must comply with the definition of *gentle slope systems* [17]. This means that the tools must allow users to learn how to effectively use the tool step-by-step without steep learning curves. We have experienced this also during the user studies we did during our research, understanding that this is particularly true for end users. This class of users is not prepared to and is not even interested in learning a large amount of conceptual and technology-related notions to use a tool. Users must be able to learn by attempts and constantly perceive the results of their learning efforts. This is very

important also in the mashup context. The inherent complexity of modeling a composition, establishing the components' execution sequence and dealing with the data passing require an algorithmic and technical mindset that end users do not have, but that, as we have seen, they can "gently learn" when provided with DSM mashup tools [7] (which are simpler to understand and work with) and, e.g., when assisted during the composition development (making the learning slope even more gentle).

As we have directly experienced during the development of several mashup tools (i.e., mashArt, MarcoFlow, OMELETTE Live mashup Environment, ResEval Mash) developing mashup platforms, in particular DSM ones, is definitely non-trivial and expensive. To foster and push the adoption of the DSM approach, which we consider a key ingredient towards the EUD goal, we designed supporting methodologies and tools for making their development simpler, faster and more affordable.

Finally, although the approaches and tools proposed in this paper focus on the development of DSM solutions targeting EUD, we observe that the generation framework we provide can also be applied to other contexts. In general, through its conceptual design approach and platform generation algorithms, it can simplify the development of many types of mashup tools. For example, it can be used for rapidly building mashup platforms aiming to ease the work of professional developers (thus, not targeting EUD), for the development of scientific workflow management systems (like myExperiments[9]) requiring to deal with data-intensive process or, in general, in any case a custom mashup platform may be useful.

7.3 Future works

As discussed in the previous section, the DSM approach and tools proposed in this work are a solid base for the achievement of our goals. However, they

[9]myExperiments homepage: `http://www.myexperiment.org/`

must be further developed and merged with other approaches and technologies to reach higher usability levels for end users. We plan to improve our DSM platform generation framework following the next key directions:

- expand the set of features supported by our generation framework. This will allow us to widen the range of requirements and domains the framework is able to cover;
- make the tools that the framework generates more stable and complete. In particular, the editor must be completed to support all the features we identified and must be improved in terms of graphical aspects and usability;
- make the whole generation framework available as open-source project. This will let the community use, extend and improve it. In addition, releasing the framework as open-source project we also facilitate and foster the adoption of the DSM approach;
- be directly involved in the development of new DSM platforms for different domains. This will allow us to test the framework, assess its coverage and then improve it based on the issues identified during its usage;
- integrate complementary technologies into the mashup editor to improve its usability level and make its learning curve more "gentle". In particular, we plan to integrate the above mentioned Baya system in our mashup editor, so that to assists end users during the composition tasks.

  Following this directions we will improve over time the robustness, coverage and usability of both the generation framework and the tools it provides. This will allow us to empower a wider and wider range of users to develop their own applications.

Bibliography

[1] S. Aghaee, M. Nowak, and C. Pautasso. Reusable decision space for mashup tool design. In *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems*, pages 211–220. ACM, 2012.

[2] S. Bitzer and M. Schumann. Mashups: An Approach to Overcoming the Business/IT Gap in Service-Oriented Architectures. In *Proceedings of AMCIS*, 2009.

[3] S. R. Chowdhury, C. Rodríguez, F. Daniel, and F. Casati. Baya: Assisted Mashup Development as a Service. In *Proceedings of WWW 2012 Companion*, pages 409–412, April 2012.

[4] F. Daniel, F. Casati, B. Benatallah, and M. C. Shan. Hosted Universal Composition: Models, Languages and Infrastructure in mashArt. In *Proceedings of ER'09*, 2009.

[5] F. Daniel, F. Casati, S. Soi, J. Fox, D. Zancarli, and M. C. Shan. Hosted Universal Integration on the Web: the mashArt Platform. In *Proceedings of ICSOC/ServiceWave*, Stockholm, Sweden, November 2009. Springer Verlag.

[6] F. Daniel, M. Imran, F. Kling, S. Soi, F. Casati, and M. Marchese. Developing Domain-Specific Mashup Tools for End Users. In *Proceedings of WWW Companion*, pages 491–492, 2012.

[7] F. Daniel, M. Imran, S. Soi, A. De Angeli, C. R. Wilkinson, F. Casati, and M. Marchese. Developing Mashup Tools for End-Users: On the Importance of the Application Domain. *International Journal of Next-Generation Computing (IJNGC)*, 3(2), July 2012.

[8] F. Daniel, S. Soi, and F. Casati. *New Trends in Search Computing*, chapter Distributed User Interface Orchestration: On the Composition of Multi-User (Search) Applications, pages 187–196. Springer, May 2010.

[9] F. Daniel, S. Soi, and F. Casati. *Search Computing - Challenges and Directions*, volume 5950 of *LNCS*, chapter From Mashup Technologies to Universal Integration: Search Computing the Imperative Way, pages 72–93. Springer, March 2010.

[10] F. Daniel, S. Soi, S. Tranquillini, F. Casati, C. Heng, and L. Yan. From People to Services to UI: Distributed Orchestration of User Interfaces. In Springer, editor, *Proceedings of BPM'10*, volume 6336 of *LNCS*, pages 310–326, September 2010.

[11] F. Daniel, S. Soi, S. Tranquillini, F. Casati, C. Heng, and L. Yan. MarcoFlow: Modeling, Deploying, and Running Distributed User Interface Orchestrations. In *Proceedings of BPM'10 Demo Track*, September 2010.

[12] F. Daniel, S. Soi, S. Tranquillini, F. Casati, C. Heng, and L. Yan. Distributed orchestration of user interfaces. *Information Systems, Elsevier*, 37(6):539–556, September 2011.

[13] I. Floyd, M. Jones, D. Rathi, and M. Twidale. Web Mashups and Patchwork Prototyping: User-driven technological innovation with Web 2.0 and Open Source Software. In *Proceedings of HICCS*, 2007.

[14] H. Gebhardt, M. Gaedke, F. Daniel, S. Soi, F. Casati, C. A. Iglesias, and S. Wilson. From Mashups to Telco Mashups: a Survey. *IEEE Internet Computing*, 16(3):70–76, May-June 2012.

[15] M. Imran, F. Kling, S. Soi, F. Daniel, F. Casati, and M. Marchese. ResEval Mash: A Mashup Tool for Advanced Research Evaluation. In *Proceedings of WWW Companion*, pages 361–364, April 2012.

[16] M. Imran, S. Soi, F. Kling, F. Daniel, F. Casati, and M. Marchese. On the Systematic Development of Domain-Specific Mashup Tools for End Users. In *Proceedings of ICWE*, pages 291–298. Springer, July 2012.

[17] B.A. Myers, D. C. Smith, and B. Horn. *Languages for Developing User Interfaces*, chapter Report of the 'End-User Programming' Working Group, pages 343–366. Jones and Bartlett, Boston, 1992.

[18] A. Namoun, T. Nestler, and A. De Angeli. 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*, pages 396–407. Springer, 2010.

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

[20] B. A. Nardi. *A small matter of programming: perspectives on end user computing*. MIT Press, Cambridge, MA, USA, 1993.

[21] T. Nestler, M. Feldmann, G. Hübsch, A. Preußner, and U. Jugel. The ServFace Builder - A WYSIWYG approach for building Service-based Applications. In *Proceedings of ICWE*, 2010.

[22] S. Soi and M. Báez. Domain-Specific Mashups: From All to All You Need. In *Proceedings of ICWE Workshops*, pages 384–395. Springer, 2010.

[23] S. Soi, F. Daniel, and F. Casati. Domain Specific Mashup Platforms: the Easy Way. to be submitted to ACM Transactions on the Web (TWEB).

[24] S. Soi, F. Daniel, and F. Casati. *Web Services Foundations*, chapter Conceptual Design of Sound, Custom Composition Languages. Springer, 2013 (in press).

[25] R. Tuchinda, P. Szekely, and C. A. Knoblock. Building Mashups by Example. In *Proceedings of IUI*, 2008.

[26] M. Wieland, K. Gorlach, D. Schumm, and F. Leymann. Towards Reference Passing in Web Service and Workflow-Based Applications. In *Proceedings of Enterprise Distributed Object Computing Conference (EDOC)*, pages 109–118. IEEE, 1-4 Sept. 2009.

[27] S. Wilson, F. Daniel, U. Jugel, and S. Soi. Orchestrated User Interface Mashups Using W3C Widgets. In *Proceedings of ICWE Workshops*, pages 49–61. Springer, June 2011.

APPENDIXES

Appendix A

Hosted Universal Integration on the Web: the mashArt Platform

Florian Daniel[1], Fabio Casati[1], Stefano Soi[1], Jonny Fox[1], David Zancarli[1], Ming-Chien Shan[2]

[1] University of Trento, Italy
{daniel,casati,soi,fox,zancarli}@disi.unitn.it

[2] SAP Labs- 3410 Hillview Avenue, Palo Alto, CA 94304, USA
ming-chien.shan@sap.com

Abstract. Traditional integration practices like Enterprise Application Integration and Enterprise Information Integration approaches typically focus on the application layer and the data layer in software systems, i.e., on limited and specific development aspects. Current web mashup practices, instead, show that there is also a concrete need for (i) integration at the presentation layer and (ii) integration approaches that conciliate all the three layers together. In this demonstration, we show how our *mashArt* approach addresses these challenges and provides skilled web users with *universal integration* in a hosted fashion.

Keywords: Hosted Universal Integration, Mashups, Services Composition.

1 Introduction and contributions

Mashups are online applications that are developed by composing contents and functions accessible over the Web [1]. The innovative aspect of mashups is that they also tackle integration at the user interface (UI) level, i.e., besides application logic and data, they also reuse existing UIs (e.g., many of today's applications include a Google Map). We call this practice of integrating data, application logic, and UIs for the development of a composite application *universal integration*.

Universal integration can be done (and is being done) today by joining the capabilities of multiple programming languages and techniques, but it requires significant efforts and professional programmers. There is, however, also a growing number of *mashup tools*, which aim at aiding mashup development and at simplicity more than robustness or completeness of features. For instance, Yahoo Pipes focuses on RSS/Atom feeds, Microsoft Popfly on feeds and JavaScript components, Intel Mash Maker on UIs and annotated data in web pages, while JackBe Presto also allows putting a UI on top of data pipes. None of these, however, covers the three application layers discussed above together in a convenient and homogeneous fashion.

Building on research in SOA and capturing the trends of Web 2.0 and mashups, in this demo we propose an integrated and comprehensive approach for universal integration, equipped with a proper hosted development and execution platform called *mashArt* (a significant evolution of the work described in [2]). Our aim is to do what service composition has done for integrating services, but to do so at all layers, not just at the application layer, and to do so by learning lessons and capturing the trends

of Web 2.0 and mashups, removing some of the limitations that constrained a wider adoption of workflow/service composition technologies.

The mashArt approach aims at empowering non-professional programmers with easy-to-use and flexible abstractions and techniques to create and manage composite web applications. Specifically, mashArt provides the following, unique contributions:

- A *unified component model* that is able to accommodate and abstract UI components (HTML), application logic components (SOAP or RESTful services), and data components (feeds or XML/relational data) using a unified model.
- A *universal composition model* that allows mashArt users to develop composite applications on top of the unified component model and conciliates the needs of both UI synchronization and service orchestration under one hood.
- A *development and execution platform* for composite applications that facilitates rapid development, testing, and maintenance. mashArt is entirely hosted and web-based, with zero client-side code.

2 Demonstration storyboard

The live demonstration introduces the three contributions of mashArt by means of a joint use of slides (for the conceptual aspects) and hands-on platform demos (for the practical aspects). In particular, the demonstration is organized as follows:

1. *Intro:* introduction of the conceptual and theoretical background of the project, its goals and ambitions, and its contributions.
2. *UI integration*: explanation of the idea of UI integration and how UI components and the composition logic look like.
3. *UI integration demo*: demonstration of how to do UI integration with mashArt starting from a set of existing mashArt UI components. Two minutes suffice to show how to develop and run a simple application that synchronizes a search component and a map component for geo-visualization of results.
4. *Universal integration*: description of mashArt's component model and its composition model, which characterize the universal integration approach.
5. *Universal integration demo*: demonstration of how to combine service, data, and UI integration in mashArt. Again, two minutes suffice to show how to add an RSS reader component to the previous scenario and to feed it with data sourced from a RESTful service and transformed via a Yahoo! pipe.
6. *Architecture*: functional architecture of mashArt to show that mashArt is (or will be) more than what is shown in the demo.
7. *Conclusion and future works*: summary and outline of future works.

A short version of the demo can be previewed here: *http://mashart.org/mashArt.wmv*.

References

[1] J. Yu, et al., "Understanding Mashup Development and its Differences with Traditional Integration," *Internet Computing*, vol. 12, no. 5, pp. 44-52.

[2] J. Yu, et al., "A Framework for Rapid Integration of Presentation Components," in *WWW'07*, 2007, pp. 923-932.

Appendix B

Chapter 5:
From Mashup Technologies to Universal Integration: Search Computing the Imperative Way

Florian Daniel, Stefano Soi, Fabio Casati

University of Trento - Via Sommarive 14, 38050 Trento - Italy
{daniel,soi,casati}@disi.unitn.it

Abstract. Mashups, i.e., web applications that are developed by integrating data, application logic, and user interfaces sourced from the Web, represent one of the innovations that characterize Web 2.0. Novel content wrapping technologies, the availability of so-called web APIs (e.g., web services), and the increasing sophistication of mashup tools allow also the less skilled programmer (or even the average web user) to compose personal applications on the Web. In many cases, such applications also feature search capabilities, achieved by explicitly integrating search services, such as Google or Yahoo!, into the overall logic of the composite application.
In this chapter, we first overview the state of the art in mashup development by looking at which technologies a mashup developer should master and which instruments exist that facilitate the overall development process. Then we specifically focus on our own mashup platform, *mashArt*, and discuss its approach to what we call universal integration, i.e., integration at the data, application, and user interface layer inside one and the same mashup environment. To better explain the novel ideas of the platform and its value in the context of search computing, we discuss an example inspired by the idea of search computing.

1 Introduction

The advent of Web 2.0 led to the participation of the user into the content creation and application development processes, also thanks to the wealth of social web applications (e.g., wikis, blogs, photo sharing applications, etc.) that allow users to become an active contributor of content rather than just a passive consumer, and thanks to *web mashups* [1]. Mashup tools enable fairly sophisticated development tasks inside the web browser. They allow users to develop their own applications starting from existing content and functionality. Some applications focus on integrating RSS or Atom feeds, others on integrating RESTful services, others on simple UI widgets, etc. Mashup approaches are innovative especially in that they tackle integration at the user interface level and do not "just" focus on data and in that they aim at simplicity more than robustness or completeness of features (up to the point to enable also non-professional programmers to develop own mashups). Integrating content and services from the Web also means integrating *search results* or *services*, which makes ma-

shups a natural candidate for search computing applications, but also poses novel requirements in terms of composition features – especially as for what regards UIs.

Inspired by and building upon research in SOA and capturing the trends of Web 2.0 and mashups, this chapter introduces the concept of *universal integration*, that is, the creation of composite web applications that integrate data, application, and user interface (UI) components, effectively enabling the imperative development of advanced search computing applications. Our aim is to do what service composition has done for integrating services, but to do so at all layers, not just at the application layer, and remove some of the limitations that constrained a wider adoption of workflow/service composition technologies. Universal integration can be done (and is being done) today by joining the capabilities of multiple programming languages and techniques, but it requires significant efforts and professional programmers. In this chapter we provide abstractions, models and tools so that the development and deployment of universal compositions is greatly simplified, up to the extent that even non-professional programmers can do it in their web browser.

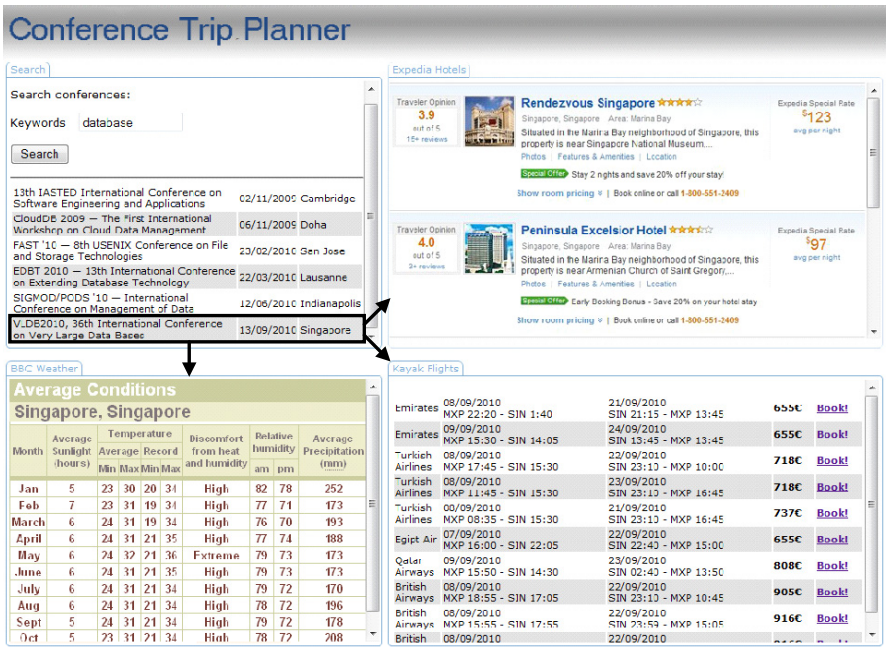


Figure 1 Reference scenario: the conference trip planner application. Selecting a conference from the list aligns the content shown by the components in the page.

Scenario. As a reference scenario throughout this chapter, we reuse the conference search scenario described in [18], based on the search query "*find all database conferences in the next six months in locations where the average temperature is 28°C degrees and for which a cheap travel solution including a luxury accommodation exists*". Answering this request requires (i) finding interesting conferences; (ii) understanding whether the conference location is served by low-cost flights; (iii) finding

luxury hotels close to the conference location with available rooms; and (iv) checking the expected average temperature of the location. Instead of automatically deriving a query plan to answer the request, in this chapter we focus on how the request can be answered through a composite application for the Web that interactively involves the user into the search process.

The screenshot in Figure 1 shows how such a *Conference Trip Planner* (CTP) application could look like. The application is composed of a variety of different components: In the upper left corner we have a *Conferences Search* component that allows the user of the application to specify a query string and to search for conferences that satisfy the query; retrieved results are displayed below the search form. This is a so-called UI component, as – besides supporting the conference search function – it also comes with its own UI, which is reused as-is by the composite application. Similarly, in the lower left corner, we have a *BBC Weather* UI component that shows the average weather conditions for a selected city, and in the upper right corner we have an *Expedia Hotel* UI component that provides a list of hotels given the name of a city. Finally, in the lower right corner, we have an *RSS Reader* UI component that displays a list of possible flight connections from Milano to the destination city.

The four UI components are synchronized via the *Conferences Search* component, which somehow represents the entry point for the evaluation of the overall "search query", i.e., the content displayed by the UI components. Specifically, by selecting an event of interest from the retrieved conferences, the user synchronizes the content of the other UI components in the page, resulting in a re-computation of the weather, hotel and flight components. By clicking on the proposed hotels or flights, the user is directly forwarded to the respective booking pages, where he/she can conclude the respective booking.

We assume that the *Conferences Search* component is implemented via a simple, generic search component in conjunction with an external conference search service; in our example, we use a Yahoo! Pipe to search for conferences and filter them according to the user's query. Similarly, we use a standard *RSS Reader* component to visualize flights that are retrieved via the *kayak.com* search engine. For the *BBC Weather* and the *Expedia Hotel* components, instead, we assume that they are both provided as readily usable UI components by the respective companies (just like common web services).

The application in Figure 1 represents only one possible application able to answer the initial query. In fact, other combinations of components and services could be adopted, e.g., using *lufthansa.com* instead of *kayak.com* or switching the position of the weather and the hotel components, but in this chapter we are not interested in identifying the best combination of components (i.e., the best "query plan" using the terminology of [18]). The challenge we address is how to *enable the average web user to compose* an application like the one in Figure 1, relying on his/her own judgment of how components are best glued together.

Approach and contributions. In the following we describe a universal composition model and tool, called *mashArt*, that support this kind of composition scenario. MashArt aims at *empowering users with easy-to-use and flexible abstractions and techniques to create and manage composite web applications.* In particular, in this chapter we make the following contributions:

- A *universal component model*, allowing the modeling of UI components, application components (e.g., services with an API) and data components (representing feeds or access to XML/relational data) using a unified model.
- A *universal composition model*, to combine the building blocks and expose the composition as a MashArt component, possibly accessible via rest/soap, and/or providing feeds, and/or having its own (composed) UI.
- The *mashArt platform* which is a service providing a number facilities for facilitating the rapid development and management of composite web applications. MashArt is hosted, web-based and with zero client-side code.

In this chapter we focus on the conceptual and architectural aspects of mashArt, which constitute the most innovative contributions of this work, namely the *component* and *composition model*s as well as the *development* and *runtime* part of the infrastructure. We next discuss the state of the art in mashups and composition technologies (Section 2) and then introduce the principles that guide our work (Section 3). In Section 4 and Section 5 we introduce the mashArt component and composition models. Section 6 describes the platform and hosted execution environment. Section 7 provides concluding remarks.

2 On composition and mashups for the Web

Several areas of research are related to (lightweight) composition and mashups on the Web. In this section, we briefly survey the areas of *service composition*, *UI composition*, *computer-aided web engineering tools*, *web portals and portlets*, and then put some more focus on *mashups*, all areas we feel particularly related to universal composition for the Web.

2.1 Service composition approaches

A representative of service orchestration approaches is BPEL [6], a standard composition language by OASIS. BPEL is based on WSDL-SOAP web services, and BPEL processes are themselves exposed as web services. Control flows are expressed by means of structured activities and may include rather complex exception and transaction support. Data is passed among services via variables (Java style). So far, BPEL is the most widely accepted service composition language. Although BPEL has produced promising results that are certainly useful, it is primarily targeted at professional programmers like business process developers. Its complexity (reference [6] counts 264 pages) makes it hardly applicable for web mashups.

Many variations of BPEL have been developed, e.g., aiming at invocation of REST services [7] and at exposing BPEL processes as REST services [8]. In [9] the authors describe Bite, a BPEL-like lightweight composition language specifically developed for RESTful environments. IBM's Sharable Code platform [10] follows a different strategy for the composition of REST or SOAP services: a domain-specific programming language from which Ruby on Rails application code is generated, also comprising user interfaces for the Web. In [11], the authors combine techniques from declara-

tive query languages and services composition to support multi-domain queries over multiple (search) services. All these approaches focus on the application and data layer; UIs can then be programmed on top of the service integration logic. mashArt features instead universal integration as a paradigm for the simple and seamless composition of UI, data, and application components. We argue that universal integration will provide benefits that are similar to those that SOA and process centric integration provided for simplifying the development of enterprise processes.

2.2 UI composition approaches

In [12] we discussed the problem of integration at the presentation layer and concluded that there are no real UI composition approaches readily available: Desktop UI component technologies such as .NET CAB [13] or Eclipse RCP [14] are highly technology-dependent and not ready for the Web. Browser plug-ins such as Java applets, Microsoft Silverlight, or Macromedia Flash can easily be embedded into HTML pages; communications among different technologies remain however cumbersome (e.g., via custom JavaScript). Java portlets [15] or WSRP [2] represent a mature and Web-friendly solution for the development of portal applications; portlets are however typically executed in an isolated fashion and communication or synchronization with other portlets or web services remains hard. Portals do not provide support for service orchestration logic.

2.3 Computer-aided web engineering tools

In order to aid the development of complex web applications, the web engineering community has so far typically focused on model-driven design approaches. Among the most notable and advanced model-driven web engineering tools we find, for instance, WebRatio [16] and VisualWade [17]. The former is based on a web-specific visual modeling language (WebML), the latter on an object-oriented modeling notation (OO-H). Similar, but less advanced, modeling tools are also available for web modeling languages/methods like Hera, OOHDM, and UWE. All these tools provide expert web programmers with modeling abstractions and automated code generation capabilities, which are however far beyond the capabilities of our target audience, i.e., advanced web users and not web programmers.

2.4 Portals and portlets

Still in the context of web applications, portals and portlets represent a different approach to the UI integration problem on the Web. Their approach explicitly distinguishes between UI components (the portlets) and composite applications (the portals) and it is probably the most advanced approach to UI composition as of today (We use the term “portlets” taken from the JSR-168 portlet specification [15], but our considerations also hold for ASP.NET Web Parts). Portlets are full-fledged, pluggable Web application components that generate document markup fragments (e.g.,

(X)HTML) and facilitate content aggregation in a portal server. Portlets are conceptually very similar to servlets. The main difference between them consists in the fact that while a servlet generates a complete web page, portlets generates just a piece of page (commonly called fragment) that is designed to be included into a portal page. Hence, while a servlet can be reached through a specific URL, a portlet can only be reached through the URL of the whole portal page. A portlet has no direct communication with the web browser, but this communications are managed by the portal and the portlet container that allow the request-response flows and the communication between portlets. A portal server typically allows users to customize composite pages (e.g., to rearrange or show/hide portlets) and provide single sign-on and role-based personalization.

Today, there are several standards for portlets, JSR-168 being the original specification. JSR-286 introduced inter-portlet communication via a portlet container that manages a publish-subscribe infrastructure that can be used by the portlets. Finally, WSRP [2] also added support for accessing remote portlets as web services over the Web. The portlet model is powerful as for what regards the presentation integration part, yet portals do not naturally support interactions with generic web services or the specification of orchestration logics.

2.5 Web mashups

Web mashups somehow address the above shortcomings. Web mashups are web applications that are developed by combining content, presentation, and application functionality from disparate Web sources [1]. The term mashup typically implies easy and fast integration based on open APIs and data sources, yielding applications that add value to the individual components of the application and thereby often use components in ways that differ from the actual reason that led to the original production of the raw sources.

Mashups are strongly related with the Web. The Web is the natural environment for publishing content and services today, and therefore it is the natural environment where to access and reuse them. Content and services are published in a variety of different forms and by using a multiplicity of different technologies; we can categorize the means to source content and services from the Web into three basic groups:

- *Data services* like RSS (Really Simple Syndication) or Atom feeds, JSON (JavaScript Object Notation) or XML resources, or simple text files. A typical example is newspapers and magazines that publish their news headers via RSS or Atom feeds that allow users to easily jump to the respective articles. These simple technologies are used to publish data on the Web that are meant for consumption by machines, not humans. In fact, they focus on the efficient distribution of content, rather than on the effective presentation of such contents to human users. Sourcing data via one of these technologies is typically very simple: it mostly requires accessing an online resource and processing the response. Data services to not have complex interaction patterns to be followed.
- *Web services or public APIs accessible over the Web*, such as SOAP (Simple Object Access Protocol) or RESTful (REpresentational State Transfer) web services or, to a lower degree, Java classes (accessed via the IIOP protocol) or simi-

lar. These technologies are used to publish application logic on the Web. Their goal is therefore not just to provide access to contents or data, but also to computing logic (e.g., the processing of an order for a book shop). Typically, the interaction with web services or APIs is ruled by so-called interaction protocols, which state which operations can be invoked, in which order, by which partners, etc. Not following the rules stated by the protocol may impede the correct functioning of the service or API.

- *User interface elements*, such as HTML clips or JavaScript APIs with own user interface (e.g., Google Maps), but also banners or advertisements. Content may also be represented by already formatted and graphically rendered data (typically in HTML). In many cases, accessing such kind of content means extracting them from a web page, as there is no equivalent data service available that can be used to source the same data. Typically, this occurs without the provider of the contents actually knowing that there is someone extracting data from its web pages. In other cases, e.g., Google Maps, the provider of the contents explicitly publishes its data at the user interface level only.

The very innovative aspect of web mashups is that they integrate sources also at the UI layer, not only at the data and application logic layers. Integration at the data and application logic layers has been extensible studied in the past, while integration at all three layers is still a goal that put architects and programmers in front of important conceptual and technical problems.

Mashup development is still an ad-hoc and time-consuming process, requiring advanced programming skills (e.g., wrapping web services, extracting contents from web sites, interpreting third-party JavaScript code, etc). There are a variety of mashup tools available online, but, as we will see, only few of them adequately address the problem of integration at all its layers. In this section, we will give an overview of the state of the art in the mashup world, spanning from manual development to semi-assisted and fully assisted development approaches.

Manual development

Developing applications that aggregate data, application logic and UIs coming from diverse sources requires deep knowledge about technologies like: (X)HTML, dynamic HTML, AJAX (Asynchronous JavaScript and XML), RSS, Atom; XML specifications like DTD, XSD, XSLT; protocols like SOAP or HTTP for SOAP and RESTful web services; programming languages like JavaScript, PHP, Ruby, Java, C#, and so on; relational or object-oriented databases, etc. In addition, it might be necessary to master the business protocols of employed services and to have knowledge about how to compose services into service orchestrations. This long and not exhaustive list of technologies highlights how mashing up even a simple application, such as the one in our reference scenario, is a hard and time-consuming task that can only be completed by skilled programmers.

The development of our Conference Trip Planner requires, for instance, the following skills: First of all, the developer needs to understand well the dynamics behind and interaction logic of the *Yahoo! Pipes* and *Kayak* services and the *BBC Weather*, *Expedia Hotels* and *RSS Reader* UI components of the application. In the specific case, *Expedia Hotels* and *BBC Weather* expose JavaScript APIs that allow the devel-

oper to use and interact with their services; *Pipes* and *Kayak*, instead, return their output as RSS feeds, which need to be appropriately parsed to extract all the necessary information. While the UI components already come with their own UIs, for the conference and flight search results an ad-hoc user interface has to be developed in HTML. Next, the developer needs to implement the necessary synchronization logic among the *Conferences Search* component and the others, such that on the selection of a conference the other components will coherently update their content. In addition to invoking some JavaScript functions of the UI components, this also implies interacting with the remote search services upon the selection of a conference from the list. Finally, the developer needs to create a suitable layout for the composite application, which is able to accommodate the developed components and to render the final mashup application.

The described situation is already an ideal one: all components provide some kind of componentization. If, instead, we imagine that the developer also needs to develop the components to be mashed up, things get even worse. For instance, it could be necessary to implement a wrapper for the *BBC Weather* component that is able to automatically request weather forecasts for the correct city, to extract the HTML code of the average weather conditions, and to expose a JavaScript interface that allows the interaction with other components in the application. Similar operation would be necessary also for the other components of the application.

Semi-assisted development

To speed up and simplify the development especially of components to be mashed up, some useful web tools and frameworks have been recently introduced. Typically, they address the problem of data extraction from web sites and the provisioning of such data in form of data services or re-usable user interface elements. In the following, we analyze two representative tools, i.e., Dapper[1] and Openkapow[2], which are very user-friendly.

Dapper is a free online instrument for the generation of data wrappers that extract data from well-structured web pages. Dapper is based on a point and click technique able to assist the user in the selection of the contents to be extracted and to infer suitable extraction rules (e.g., regular expressions). Specifically, data extraction leverages the structure of the HTML formatting to understand which elements to extract (e.g., the first cells of all the rows in a table). Once properly identified, extracted data fields can be named and structured and then published, for instance, as RSS or XML data services. Published services can easily be accessed via a unique URL and are processed each time the respective URL is accessed.

Openkapow is a similar open service platform based on the concept of extraction robot, that is, user-created wrappers. Users of Openkapow can build their own robots, expose their results via web services, and run them from openkapow.com for free. Robots are able to access web sites and support the extraction and reuse of data, functionality and even pieces of user interfaces. Robots are built through a visual devel-

[1] http://www.dapper.net/open/
[2] http://openkapow.com

opment environment called RoboMaker. RoboMaker allows the user navigate inside the target web site and to define a series of simple steps, each one representing an event in the page, until the target data is reached. The extraction results can be exposed in two main ways: as a RESTful service or as an RSS feed, depending on the extracted content and on the expected use of it. After their publication on the Openkapow servers, robots are accessible through a public URL, which identifies the specific robot to run. So exposed services may also need some input values (e.g., user-id and password) that can be used to parameterize the services. Inputs can easily be passed by appending them to the service URL as name-value pairs, following the standard URL model.

To better understand how these tools can be used in the mashup context, let's refer again to the Conference Trip Planner example. Let us suppose that the *Kayak* flight search site does not have an RSS output for its search results. In this case, a data extraction service can be used to automatically extract the flight combinations from the result page. With Dapper, for instance, a developer needs to load one or more example pages into the Dapper environment. The more example pages are loaded, the better the inferred rules. Then, the developer needs to identify the individual data items he/she wants to extract from the page by clicking on the respective HTML elements (e.g., airline, departure time, arrival time, price, intermediate stops, link to booking), to label them and to assemble the final output (e.g., an RSS feed). There is no need to write any own line of code, in order to publish the extraction results on the Web.

While this kind of tools undoubtedly speeds up the development of data extraction from existing web sites, the development effort regarding the composition of components into a new application remain in unchanged. Therefore, the developer still has to be familiar with the services and APIs to be integrated, to display sourced data in a suitable way, and to manage the communication and synchronization logic between the components. Even assuming that data extraction tools can be successfully used by non-programmers, the final mashup development therefore still remains the hard task that can be performed only by skilled programmers.

Fully-assisted development

The previous analyses and consideration show that mashup development is typically a knowledge-intensive work, involving a variety of technologies and components. In addition to simplifying the creation of data extraction instruments for web pages, which address the problem of developing *components* for mashups, it is important to also aid the actual *composition* of components into applications, which is as hard and time-consuming as developing components, if not properly supported. Mashup tools or mashup platforms address exactly this problem, each of them focusing on different composition aspects and following different mashup approaches. In the following, we analyze four of these tools, which we think are most representative for this kind of assisted mashup development: Yahoo! Pipes[3], JackBe Presto[4], Microsoft Popfly[5], and

[3] http://pipes.yahoo.com/pipes
[4] http://www.jackbe.com/
[5] http://popflyteam.spaces.live.com – MS Popfly has been discontinued since August 24, 2009.

Intel Mash Maker[6]. There are also other tools like Google App Engine[7] or IBM's Lotus Mashups[8] and so on, but their discussion exceeds the scope of this chapter.

Yahoo! Pipes provides a simple and intuitive visual editor that allows one to design data-centric compositions. It takes data as input and provides data as output; the most important supported formats are RSS/Atom, XML, and JSON. A pipe is a data processing pipeline in which input data (coming from diverse data sources) are processed, manipulated and used as input for other processing steps, until the target transformation is completed. This pipeline-style process is implemented through an arbitrary number of intermediate operators, which manipulate data items inside the data feeds or provide features like loops, regular expressions or more advanced features like automatic location extraction or connection to external services. The set of operators are predefined and fixed; new functionality can be included in form of web services. Also, stored pipes can be reused as sources of another pipe.

Yahoo! Pipes' development environment is characterized by a simple and intuitive development paradigm that is however targeted at advanced web users or programmers. In fact, the level of abstraction of its operations (e.g., the regular expression component) and the characteristic data flow logic is only hardly understandable to non-programmers. Pipe's output is not meant for human consumption (RSS, Atom, JSon, etc.) but rather for integration in other applications. This limits both the variety of input sources that can be used and the accessibility of its output. In fact, the absence of any support for UIs prevents the direct use of Pipe's output by common web users. However, Pipes is a very popular data-mashup development tool, very likely due to its efficient and intuitive component placing and connection mechanism.

The development tool does not need any installation or plug-ins; it runs in any AJAX-enabled web browser. The development environment comes with a very efficient, integrated debugging tool that helps the developer during the design phase. Pipes are stored online and accessible via an own URL. When invoking a pipe, an execution process is started on the server side, relieving the client from the execution overhead. This characteristic could represent a problem under a scalability perspective: if a large number of simultaneous accesses to a pipe are made, performance and stability might suffer.

Considering our example application, with Yahoo Pipes it would be unfeasible to realize the application as described in the reference scenario, as there is no support for the user interface of the application. However, what we can do, for instance, is using Pipes to simplify the collection, aggregation and filtering of conferences sourced from different web sources, such as *conference-service.com* and *allconferences.com*. On top of this pipe, it is then necessary to provide a suitable user interface.

JackBe Presto is a robust and complete mashup platform which provides enterprise-level solutions. Presto gives the possibility to easily produce (design, test and deploy) mashups merging data coming from disparate sources. In particular it can be also connected to data sources very common in the business world (like Excel spread-

[6] http://mashmaker.intel.com/web
[7] http://code.google.com/intl/it-IT/appengine/
[8] http://www-01.ibm.com/software/lotus/products/mashups/

sheets, Oracle data software, etc.), that most of mashup competitor's solutions cannot access. Simple mashup composition can be done, also by non-IT users, through the Presto Wires tool. More advanced composition can be obtained only by professional developers implementing them in EMML language with the support of the Presto Mashup Studio plug-in for Eclipse. This language is the main actor of the OMA (Open Mashup Alliance) project, which aims to define an open language allowing enterprise mashup interoperability and portability.

The development environment is constituted by several independent tools. Wires is a visual editor based on a simple and intuitive data pipeline composition approach. It allows one to merge data coming from disparate internal and external sources producing a final output that can be graphically displayed as a mashlet. Mashlets can be plugged into a dash-board like user interface or a portal, or they can be embedded into a regular web page. Mashlet development is assisted by the Presto Mashlet tool, while the Mashup Studio is an Eclipse plug-in providing Java programmers with complete control on the mashup development process. Connectors allow one to hook up Presto to diverse software, such as Microsoft Excel, web portals, any Oracle technology, and similar. Presto services can be accessed through APIs, available for main programming languages (Java, JavaScript, C#, Python, etc.).

The runtime server provides secure mechanisms to virtualize (abstract the user from actual implementation details) and normalize (put the service output into standard formats: JSON or XML) any kind of service or data (SOAP, REST, RSS, DB, Excel) and expose them in a secure and governed way. Presto is not a hosted service, like Yahoo! Pipes; it needs to be installed and configured in each company individually.

Let us briefly analyze the possibility to create our Conference Trip Planner application with Presto. Just like Yahoo! Pipes, Wires gives the opportunity to easily access, merge and filter the RSS channels of the conferences search services and the Kayak flights search service. Retrieved items can be displayed by means of two mashlets. The development of the other UI components in form of mashlets has to be done manually in Mashup Studio using a standard programming language like Java. At this point the produced mashlets can be put together inside one web page. However, this solution does not provide for the synchronization of the basic components in the application (the mashlets), so that the selection of a conference updates the data shown in the other components. There is not inter-mashlet communication.

Microsoft Popfly gained a great consensus in the mashup community and achieved good levels of popularity and usage. Although the Popfly project has been discontinued, we analyze this mashup tool because we consider it an interesting example for UI composition with peculiarities that cannot be found in other tools.

Popfly provides a visual development environment for the realization of mashups based on the concept of components, or *block* as they are called in Popfly. A composition is created by dragging and dropping blocks of interest onto a design canvas and by graphically connecting them to create the desired application logic. A block can take the role of connector to external services or it can represent some internal functionality (implemented through a JavaScript function). Each block provides input and output ports that enable its connection to other blocks. Blocks can also be used to provide a user interface that can display the result of some processing. Placing multiple visualization blocks into a same page allows one to define the overall layout of the page. The internal layout of blocks can be customized by inserting ad-hoc HTML,

CSS or JavaScript code. Popfly has a wide collection of available blocks, offering functionalities like RSS readers, service connectors, map components based on Virtual Earth, etc. New blocks and compositions can be defined (in JavaScript), saved, shared and managed in a dedicated section of the platform.

At runtime, the communication flow is event-driven, that is, the activation of a certain component depends on the raising of some event by another component. There is no support for exception and transaction handling, but Popfly provides a section dedicated to the test and preview of the composition. Ready compositions are stored on the Popfly server, but the execution is done on the client – as many of the built-in blocks are based on the Silverlight platform. The client-side execution of mashups alleviates the server from heavy loads and limits scalability and performance.

Considering the Conference Trip Planner application, Popfly is the first tool that can be used to fully implement the application. We assume that skilled programmers already developed and published all blocks needed for the composition, especially the UI components *Conferences Search*, *Expedia Hotels* and *BBC Weather*, while the RSS reader necessary to display the output of the conference and flight search services already exists. At this point, the developer of the composition can drag and drop these components onto the modeling canvas and connect the blocks, also providing for the necessary mapping of the data parameters from outputs to inputs. In particular, the *Conferences Search* block must be connected to all the other blocks, in order to provide for the synchronization of the whole composition. Finally, the graphical appearance of the application's layout can be set up by including a custom CSS style sheet into the page. What is missing in Popfly is the possibility to define more complex, process-like service compositions, as could for example be needed to process the conference search results directly in Popfly.

Intel Mash Maker provides a completely different mashup approach: an environment for the integration of data from annotated source web pages based on a powerful, dedicated browser plug-in for the Firefox web browser. Rather than taking input from structured data sources such as RSS/Atom feeds or web services, Mash Maker allows users to reuse entire web pages and, if suitably annotated, to extract data from the pages. That is, the "components" that can be used in Mash Maker are standard web pages. If a page has been annotated in the past, it is possible to extract the annotated data from the page and share it with other components in the browser. If the page has not been annotated, it is possible reuse the page as is without however supporting any inter-page communication.

In order to annotate a page, Mash Maker allows developers and users to annotate the structure of web pages while browsing and to use such annotations to scrap contents from annotated pages. Advanced users may leverage the integrated Structure Editor to input XPath expressions with the help from FireBug's DOM Inspector (another plug-in for the Firefox web browser). Annotations are linked to target pages and stored on the Mash Maker server in order to share them with other users.

Composing mashups with Mash Maker occurs via a copy/paste paradigm, based on two modes of merging contents: *whole page merging*, where the content of one page is inserted as a header into another page; and *item-wise merging*, where contents from two pages are combined at row level, based on additional user annotations. The two techniques can be used to merge also more than two pages. Data exchange among components is achieved by means of a blackboard-like approach, where data of com-

ponents integrated into an application are immediately available to all other components. Not only the development, but also the execution of mashups is entirely performed with the help from the browser plug-in at the client side; on the server side there are only the annotations for data extraction and the stored mashup definitions.

To build the Conference Trip Planner with MashMaker, first we need to devise the necessary components in form of annotated web pages. For instance, instead of using the RSS interface toward the conference search services or toward the flight search service, we need to navigate the respective web sites and annotate the data items that are necessary to answer our reference query. Similarly, we need to annotate the UI components of our application. Next, all these individual pieces of HTML markup and annotations must be joined following an item-wise merging strategy. It is possible to implement the needed synchronization mechanisms to coordinate the components of the application with each other by means of sophisticated merge operations. The whole development procedure is a non-trivial and time-consuming, it requires some non-intuitive skills to annotate, decompose, merge and reconstruct pages and web applications of arbitrary complexity. Without advanced programming skills it is hard to implement the synchronization of components upon selection of a conference.

3 Universal composition: guiding principles

As highlighted above, although existing mashup approaches have produced promising results techniques that cater for simple and universal integration of web components at all the three layers of the application stack are still missing. We think, such techniques are necessary to transition Web 2.0 programming from elite types of computing environments to environments where users leverage simple abstractions to create composite web applications over potentially rich web components developed and maintained by professional programmers.

We aim at universal integration, and this has fundamental differences with respect to traditional composition. In particular, the fact that we aim at also integrating UI implies (i) that *synchronization*, and not (only) orchestration a la BPEL, should be adopted as interaction paradigm, (ii) that components must be able to react to both human user input and programmatic interaction, and (iii) that we must be able to design the UI of the *composite* application, not just the behavior and interaction among the components. This shows the need for a model based on state, events and synchronization more than on method calls and orchestration. We recognize in particular that *events*, *operations*, a notion of *state* and *configuration properties* are all we need to model a universal component. With respect to the design of the composite UI, we assume developers will use their favorite Web development tool (we do not aim at competing with these tools, although we do offer a simple templating mechanism for rapid development of prototype applications that run in the browser). Rather, we make it easy to embed mashArt components inside a Web application.

On the data side, we realize that *data* integration on the Web may also require different models: for example RSS feeds are naturally managed via a pipe-oriented data flow/streaming model (a-la Yahoo Pipes) rather than a variable-based approach as done in conventional service composition.

Another dimension of universality lies in the interaction protocols. MashArt aims at hiding the complexity of the specific protocol or data model supported by each component (REST, SOAP, RSS, Atom, JSON, etc) so a design goal is that from the perspective of the composer all these specificities are hidden – with the exceptions of the aspects that have a bearing on the composition (e.g., if a component is a feed, then we are aware that it operates, conceptually, by pushing content periodically or on the occurrence of certain events).

Generality and universality are often at odds with the other key design goal we have: *simplicity*. We want to enable advanced web users to create applications (an old dream of service composition languages which is still somewhat a far reaching objective). This means that mashArt must be fundamentally simpler than programming languages and current composition languages. We target the complexity of creating web pages with a web page editor, or the complexity of building a pipe with Yahoo Pipes (something that can be learned in a matter of hours rather than weeks).

To achieve simplicity we make two design decisions: first, we keep the composition model lightweight: for example, there are no complex exception or transaction mechanisms, no BPEL-style structured activities or complex dead-path elimination semantics. This still allows a model that makes it simple to define fairly sophisticated applications. Complex requirements can still be implemented but this needs to be done in an "ad hoc" manner (e.g., through proper combinations of event listeners and component logic) but there are no specialized constructs for this. Such constructs may be added over time if we realize that the majority of applications need them.

The second decision is to focus on simplicity only *from the perspective of the user* of the components, that is, the designer of the composite applications. In complex applications, complexity must reside somewhere, and we believe that as much as possible it needs to be inside the components. Components usually provide core functionalities and are reused over and over (that's one of the main goals of components).Thus, it makes sense to have professional programmers develop and maintain components. We believe this is necessary for the mashup paradigm to really take off. For example, issues such as interaction protocols (e.g., SOAP vs. REST or others) or initialization of interactions with components (e.g., message exchanges for client authentication) must be embedded in the components.

4 The mashArt Component Model

The first step toward the universal composition model is the definition of a component model. *MashArt* components wrap UI, application, and data services and expose their features/functionalities according to the mashArt component model. The model described here extends our initial UI-only component model presented in [3] to cater for universal components. The model is based on four abstractions: state, events, operations, and properties:

- The *state* is represented as a set of name-value pairs. What the state exactly contains and its level of abstraction is decided by the component developer, but in general it should be such that its change represents something relevant and significant for the other components to know. For example, in our *Conference*

Search component we can change the search string of the query and re-compute the list of pertaining conferences; this component-internal activity is irrelevant for the other components who are not interested in such low level of detail. Instead, clicking on (selecting) a specific conference expresses an information that may lead other components to show related information or application services to perform actions (e.g., query for flights). This is a state change we want to capture. In our case study, the state for the *Conference Search* component is the set of conferences being displayed plus the selected conference.

Modeling state for application components is something debatable as services are normally used in a stateless fashion. This is also why WSDL does not have a notion of state. However, while implementations can be stateless, from a modeling perspective it can be useful to model the state, and we believe that its omission from WSDL and WS-* standards was a mistake (with many partial attempts to correct it by introducing state machines that can be attached to service models). Although not discussed here, the state is a natural bridge between application services and data-oriented services (services that essentially manipulate a data object).

- Events communicate state changes and other information to the composition environment, also as name-value pairs. External notifications by SOAP services, callbacks from RESTful services, and events from UI components can be mapped to events. When events represent state changes, initiated either by the user by clicking on the component's UI or by programmatic requests (through operations, discussed below), the event data includes the new state. Other components subscribe to these events so that they can change their state appropriately (i.e., they synchronize). For instance, when selecting a conference in the Conference Search component, an event is generated that carries details (e.g., name, city, start/end date) about the performed selection.
- Operations are the dual of events. They are the methods invoked as a result of events, and often represent state change requests. For example, the Conference-Search component has a state change operation ShowConferences that can be used to display retrieved conferences. In this case, the operation parameters include the necessary information about the state to which the component must evolve (the list of conferences). In general, operations consume arbitrary parameters, which, as for events, are expressed as name-value pairs to keep the model simple. Request-response operations also return a set of name-value pairs – the same format as the call – and allow the mapping of request-response operations of SOAP services, Get and Post requests of RESTful services, and Get requests of feeds. One-way operations allow the mapping of one-way operations of SOAP services, Put and Delete requests of RESTful services, and operations of UI components. The linkage between events and operations, as we will see, is done in the composition model. We found the combination of (application-specific) states, events, and operations to be a very convenient and easy to understand programming paradigm for modeling all situations that require synchronization among UI, application, or data components.
- Finally, configuration properties include arbitrary component setup information. For example, UI components may include layout parameters, while service components may need configuration parameters, such as the username and

password for login. The semantics of these properties is entirely component-specific: no "standard" is prescribed by the component model. Again, they are name-value pairs.

In addition to the characteristics described above, components have aspects that are *internal*, meaning that they are not of concern to the composition designer, but only to the programmer who creates the component. In particular, a component might need to handle the invocation of a service, both in terms of mapping between the (possibly complex) data structure that the service supports and the flat data structure of mashArt (name-value pairs), and also in terms of invocation protocol (e.g., SOAP over http). There are two options for this: The first is to develop ad hoc logic in form of a wrapper. The wrapper takes the mashArt component invocation parameters, and with arbitrary logic and using arbitrary libraries, builds the message and invokes the service as appropriate. The second is to use the built-in mashArt bindings. In this case, the component description includes component bindings such as *component/http*, *component/SOAP*, *component/RSS*, or *component/Atom*. Given a component binding, the runtime environment is able to mediate protocols and formats by means of default mapping semantics; mappings can also be customized (more details are provided in the implementation section).In summary, the mashArt model intuitively accommodates multiple component models, such as UI components, SOAP and RESTful services, RSS and Atom feeds.

In Figure 2(a) we introduce our graphical modeling notation for mashArt components that captures the previously discussed characteristics of components, i.e., state, events, operations, and UI. *Stateless* components are represented by circles, *stateful* components by rectangular boxes. Components with *UI* are explicitly labeled as such. We use arrows to model *data flows*, which in turn allow us to express events and operations: arrows going out from a component are *events*; arrows coming in to a component are *operations*. There might be multiple events and operations associated with one component. Depending on the particular type of operation or event of a stateless service, there might be only one incoming data flow (for one-way operations), an incoming and an outgoing data flow (for request-response operations), or only an outgoing data flow (for events). Operations and events are bound to their component by means of a simple dot-notation: *component.(operation|event)*.

The actual model of a specific component is specified by means of an abstract component *descriptor*, formulated in the *mashArt Description Language* (MDL) a simple, XML-based interface description language. MDL is for mashArt components what WSDL is for web services.

5 Universal Composition Model

Since we target universal composition with both stateful and stateless components, as well as UI composition, which requires synchronization, and service composition, which is more orchestrational in nature, the resulting model combines features from *event-based* composition with *flow-based* composition. As we will see, these can naturally coexist without making the model overly complex.

In essence, composition is defined by linking events (or operation replies) that one component emits with operation invocations of another component. In terms of flow control, the model offers conditions on operations and split/join constructs, defined by tagging operations as optional or mandatory. Data is transferred between components following a pipe/data flow approach, rather than the variables-based approach typical of BPEL or of programming languages. The choice of the data flow model is motivated by the fact that while variables work very well for programs and are well understood by programmers, data flows appear to be easier to understand for non-programmers as they can focus on the communication between a pair of components. This is also why frameworks such as Yahoo Pipes can be used by non-programmers.

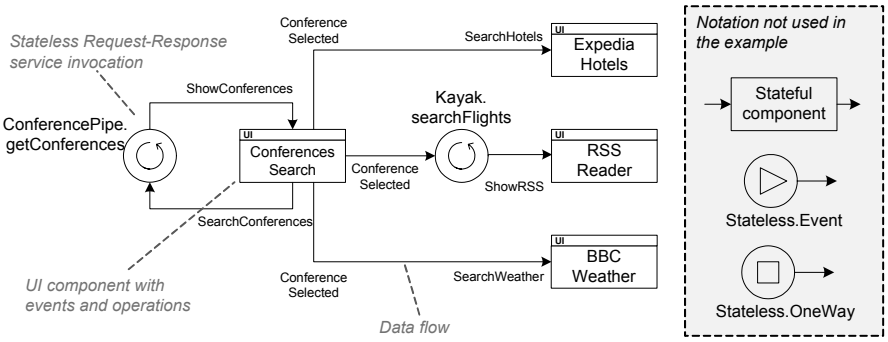
To keep the solution simple as per our requirements (yet, as complete and flexible as necessary) we had to make some compromises. For example, the model comes without any structured or complex system activities (e.g., scopes, nested scopes, sub-processes, timers) and does not include transaction management or exception handling. If more complex modeling constructs are necessary (e.g., a join construct with a special data merging function, a complex data transformation service, or a death-path elimination BPEL-style), they can be (i) implemented using the language constructs (although they could require many components and events and render the graph complex), (ii) integrated in the form of dedicated services (implemented as components), or (iii) by creating a BPEL subflow invoked by mashArt (this is supported by the tool but not described here, as it is implementation and not an original contribution). The model and the language described here provide for the necessary basic composition logic, while more complex logics are integrated without requiring any extension at the language level. As we go along and we realize that certain features are crucial, they will be added to the model.

The universal composition model is defined in the Universal Composition Language (UCL), which operates on MDL descriptors only. UCL is for universal compositions what BPEL is for web service compositions (but again, simpler and for universal compositions). A universal composition is characterized by:

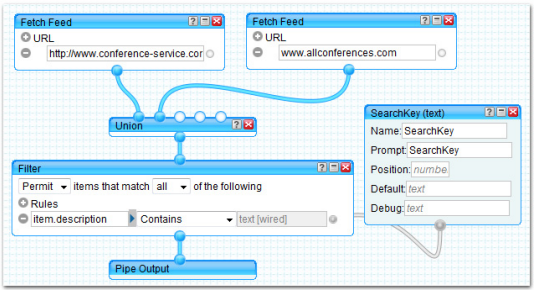
- *Component declarations*: Here we declare the components used in the composition and provide references to the MDL descriptor of each component. This allows access to all component details (e.g., the binding). Optionally, declarations may also contain the setting of constructor parameters.
- *Listeners*: Listeners are the core concept of the universal composition approach. They associate events with operations, effectively implementing simple publish-subscribe logics. Events produce parameters; operations consume them (static parameter values may be specified in the composition). Inside a listener, inputs and outputs can be arbitrarily connected (by referring to the respective IDs and parameter names) resulting into the definition of *data flows* among components. An optional condition may restrict the execution of operations; conditional statements are XPath statements expressed over the operation's input parameters. Only if the condition holds, the operation is executed.
- *Type definitions*: As for mashArt components, the structures of complex parameter values can be specified via dedicated data types.

We are now ready to compose our Conference Trip Planner. Composing an application means connecting events and operations via data flows, and, if necessary, spe-

cifying conditions constraining the execution of operations. The graphical model in Figure 2(a) represents, for instance, the "implementation" of the reference scenario described in the introduction. We can see the four UI components *Conferences Search*, *Expedia Hotels*, *RSS Reader* and *BBC Weather* and the two stateless service components *ConferencePipe* and *Kayak*.



(a) The mashArt composition model plus the modeling notation not used in the model



(b) The internals of the conference search aggregation and filtering pipe

Figure 2 Composition model for the Conference Trip Planner application

The composition has four listeners:

1. If a user enters a conference search string and starts the search (*SearchConference* event), the *ConferencePipe* service is invoked by processing a Yahoo! pipe that queries two other services: *conference-service.com* and *allconferences.com*. The internals of the pipe are shown in Figure 3(b). The pipe joins the results coming from the two services and applies the filter condition provided by the user; the result is passed back to the mashArt composition by invoking the *ShowConferences* operation of the *Conferences Search* UI component.
   Note that similar operators and feed processing logics as shown in Figure 3(b) could easily be implemented also directly in mashArt, but we prefer reusing Yahoo! Pipes to show an example of how mashup platforms can interoperate.
2. If a user selects a conference from the list of retrieved conferences (*ConferenceSelected* event), three listeners reacting to the same event are activated. The first lis-

tener propagates the selected conference location and dates to the *Expedia Hotel* service that retrieves a list of available hotels from the Expedia repository.

3. The second listener activated after the selection of a conference searches for matching flights and visualizes them in the *RSS Reader*. The flights are retrieved by invoking a flight search service providing access to the *kayak.com* database and delivering its results as RSS feed. Such feed is provided as input to the *RSS Reader* via the *ShowRSS* operation.
4. Finally, the last listener activated upon selection of a conference aligns the data shown in the *BBC Weather* component by forwarding the name of the city the conference is located in through the *SearchWeather* operation. This causes the component to visualize the average weather conditions for the selected city.

The graphical model represents the information that is necessary to understand the composition from the composer's point of view. Of particular interest for the structure of the composition is the distinction between stateful and stateless components: Stateful components handle multiple invocations during their lifetime; stateless components always represent only one invocation. For instance, the *ConferencePipe* service is invoked each time a user inputs a new search query, while the *Conferences Search* component is instantiated only once and handles multiple events and operatinos.

Regarding the semantics of the three data flows leaving the *Conferences Search* component upon a *ConferenceSelected* event, it is worth noting that we allow the association of *conditions* operations. A *condition* is a Boolean expression over the operation's input (e.g., simple expressions over name-value pairs like in *SQL where* clauses) that constrains the execution of the operation. The three data flows in Figure 2(a) represent a *parallel branch* (conjunctive semantics); if conditions where associated with either *SearchHotel*, *ShowRSS* or *SearchWeather* the flows would represent a *conditional branch* (disjunctive semantics). A similar logic applies to operations with multiple incoming flows that can be used to model *join* constructs. Inputs may be *optional*, meaning that they are not mandatory for the execution of the operation. If only mandatory inputs are used, the semantics is conjunctive; otherwise, the semantics is disjunctive.

A branch/join inside a listener (composed of multiple service invocations) corresponds to a *synchronous* branch/join. We speak instead of an *asynchronous* branch/join, when branching and joining a flow requires defining two listeners, one with the branch and one with the join. The listener with the branch terminates with multiple operations; the listener with the join reacts to multiple events or operation results. Again, events may be optional or mandatory. If only mandatory events are used, the semantics is conjunctive; if optional events are used, the semantics is disjunctive. There is no BPEL-style dead path elimination, and in case of conjunctive joins FIFO semantics is used for pairing events. The graph-based definition of optional events/operations conciliates a pub/sub approach with an orchestration approach.

Finally, data passing does not require any variables to store intermediate results. Parameter names and data types only refer to the data and the data structures exchanged via data flows. Data transformations are defined by connecting the event or feed parameters with the parameters of the operations invoked as a result of the event triggering. More complex mappings require knowledge about the exact data type of each of the involved parameters. In general, our approach supports a variety of data transformations: (i) simple parameter mappings as described above; (ii) inline script-

ing, e.g., for the computation of aggregated or combined values; (iii) runtime XSLT transformations; and (iv) dedicated data transformation services that take a data flow in input, transform it, and produce a new data flow in output. The use of the dedicated data transformation services is enabled by UCL's extensibility mechanism.

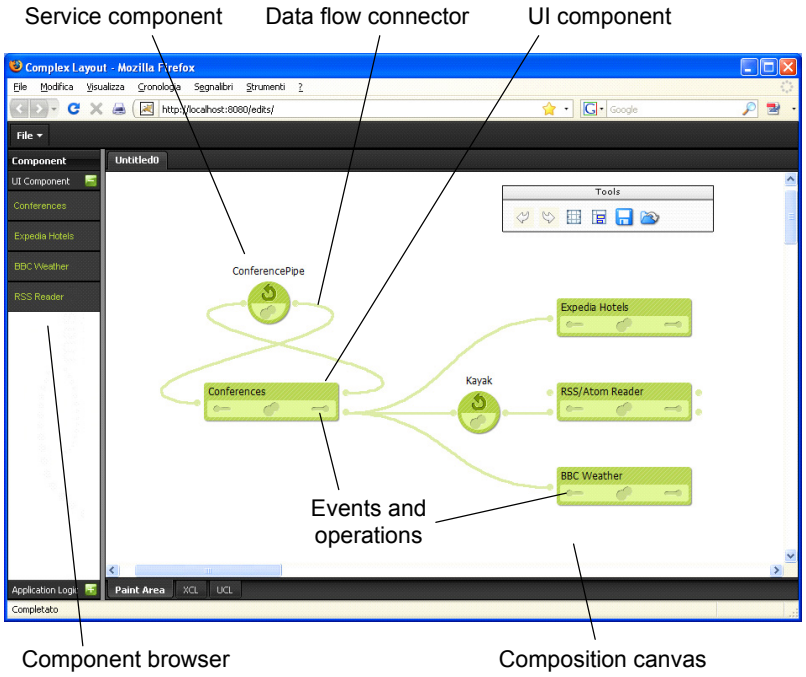


Figure 3 The mashArt editor

6 Implementing and Provisioning Universal Compositions

Development environment. In line with the idea of the Web as integration platform, the mashArt editor runs inside the client browser; no installation of software is required. The screenshot in Figure 3 shows how the universal composition of Figure 2(a) can be modeled in the editor. The modeling formalism of the editor slightly differs from the one introduced earlier, as in the editor we can also leverage interactive program features to enhance user experience (e.g., users can interactively choose events and operations from respective drop-down panels). But the expressive power of the editor is the same as discussed above.

The *list of available components* on the left hand side of the screenshot shows the components and services the user has access to in the online registry (e.g., the *Conferences Search* or the *BBC Weather* component). The *modeling canvas* at the right hand side hosts the composition logic represented by *UI components* (the boxes), *service components* (the circles), and *listeners* (the connectors). A click on a listener allows the user to map outputs to inputs and to specify optional input parameters.

In the lower part of the screenshot, tabs allow users to switch between different views on the same composition: visual model vs. textual UCL, interactive layout vs. textual HTML, and application preview. The layout of an application is based on standard HTML templates; we provide some default layouts, own templates can easily by uploaded. Laying out an application simply means placing all UI components of the composition into placeholders of the template (again, by dragging and dropping components). The preview panel allows the user to run the composition and test its correctness. Compositions can be stored on the mashArt server.

The implementation of the editor is based on JavaScript and the Open-jACOB Draw2D library (http://draw2d.org/draw2d/) for the graphical composition logic and AJAX for the communication between client and server. The registry on the server side, used to load components and services and to store compositions, is implemented as a RESTful web service in Java. The platform runs on Apache Tomcat.

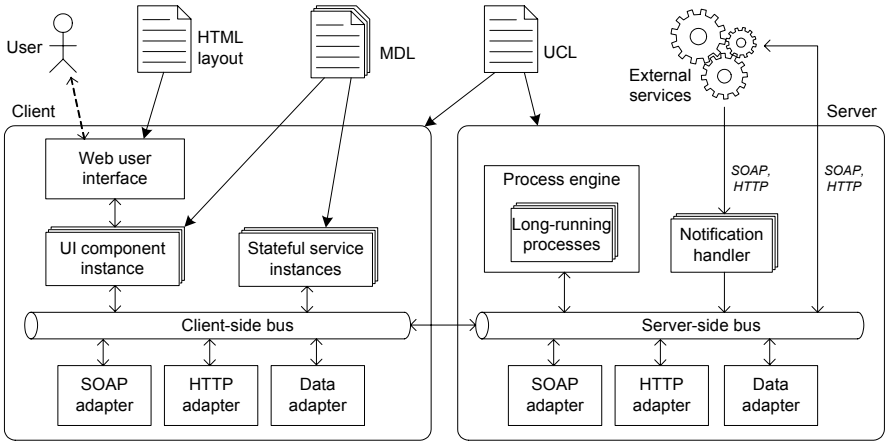


Figure 4 Universal execution framework

Execution environment. In developing a mashArt execution environment, the issues that need to be solved include (i) the seamless integration of stateful and stateless components and of UI and service components, (ii) the conciliation of short-lived and long-lasting business process logics in one homogeneous environment, (iii) the consistent distribution of actual execution tasks over client and server, and (iv) the transparent handling of multiple communication protocols. Details can be found in [19].

Figure 4 contextualizes the previous considerations in the functional architecture of our execution environment. The environment is divided into a client- and a server-side part, which exchange events via a synchronization channel. On the client side, the user interacts with the application via its UI, i.e., its UI components, and thereby generates events that are intercepted by the client-side event bus. The bus implements the listeners that are executed on the client side and manage the data and SOAP-HTTP adapters. The data adapter performs data transformations, the SOAP-HTTP adapters allow the environment to communicate with external services. Stateful service instances might also use the SOAP-HTTP adapters for communication purposes.

The server-side part is structured similarly, with the difference that the handling of external notifications is done via dedicated notification handlers, and long-lasting process logics that can be isolated from the client-side listeners and executed independently can be delegated to a conventional process engine (e.g., a BPEL engine).

The whole framework, i.e., UI components, listeners, data adapters, SOAP-HTTP adapters, and notification handlers are instantiated when parsing the UCL composition at application startup. The internal configuration of how to handle the individual components is achieved by parsing each component's MDL descriptor (e.g., to understand whether a component is a UI or a service component). The composite layout of the application is instantiated from the HTML template filled with the rendering of the application's UI components.

The client-side environment is an evolution of the already successfully implemented and tested UI integration framework of the Mixup project [3], that was however limited to UI components only. The environment comes with an AJAX implementation of the UCL and MDL parsers and is integrated with the mentioned online registry storing components and compositions. The server-side environment has successfully passed a prototype implementation (the effort of several Master theses) based on Java and the Tomcat web server. The integration with the external process engine (e.g., Active-BPEL) and of the client- and server-side parts is ongoing.

A first conclusion that can be drawn from our experiences is that performance does not play a major role on the client side. This is because in a given composition, only a limited number of components run on the client, and the client needs to handle only one instance of the application. On the server-side, performance becomes an issue if multiple composite applications with a high number of long-lasting processes are running in the same web server. Although we did not run scalability experiments yet, the re-use of existing and affirmed technologies, simple servlets for notification handlers, and BPEL engines for process logics will provide for the necessary scalability.

7 Conclusion

In this chapter, we have considered a novel approach to UI and service composition on the Web, i.e., *universal composition*. This composition approach is the foundation of the mashArt project, which aims at enabling even non-professional programmers (or Web users) to perform complex UI, application, and data integration tasks online and in a hosted fashion (integration as a service). Accessibility and ease of use of the composition instruments is facilitated by the simple composition logic and implemented by the intuitive graphical editor and the hosted execution environment. The platform comes with an online registry for components and compositions and will provide tools for monitoring and analysis of hosted compositions.

Throughout the chapter, we have constantly kept an eye on the connection between universal composition and *search computing*. The Conference Trip Planner tool implemented using the mashArt instruments and languages shows that it is indeed possible to develop a component-based application that provides answers to the conference search problem, provided that the necessary basic components are readily available. The application's integration logic is achieved by means of an imperative drag-and-drop composition paradigm that allows the users of the mashArt platform to compose applications according to their own knowledge about which components are needed

and about how to glue them together. There exist many alternative solutions to the implementation of the same application; yet, unlike in [18], where an optimal query plan is identified automatically, in mashArt it is up to the developer to decide which solution fits best his/her individual needs.

In terms of output of the composition, it is interesting to note that while in the traditional search scenario the output is a set of result tuples, the output in mashArt is rather represented by the whole application, i.e., the individual components and their interconnection. Given the search query introduced in the introduction of this chapter, its answer is therefore represented by the screenshot in Figure 1, which naturally combines simple search outputs with sophisticated UI components.

8 References

[1] J. Yu, et al., Understanding Mashup Development and its Differences with Traditional Integration, *Internet Computing*, vol. 12, no. 5, 2008, pp. 44-52.

[2] OASIS. Web Services for Remote Portlets, August 2003. [Online]. www.oasis-open.org/committees/wsrp

[3] J. Yu, et al., A Framework for Rapid Integration of Presentation Components, *WWW'07*, 2007, pp. 923-932.

[4] G. Alonso, F. Casati, H. Kuno, V. Machiraju, *Web Services: Concepts, Architectures and Applications*. Springer, 2003.

[5] S. Dustdar, W. Schreiner, A survey on web services composition, *Int. J. Web Grid Services*, vol. 1, no. 1, pp. 1-30, 2005.

[6] OASIS. Web Services Business Process Execution Language Version 2.0, April 2007. [Online]. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

[7] C. Pautasso, BPEL for REST, *BPM'08*, Milano, 2008.

[8] T. van Lessen, et al. A Management Framework for WS-BPEL, *ECoWS'08*, Dublin, 2008.

[9] F. Curbera, et al. Bite: Workflow Composition for the Web, *ICSOC'07*, Vienna, 2007, pp. 94-106.

[10] E. M. Maximilien, et al. An Online Platform for Web APIs and Service Mashups, *Internet Computing*, vol. 12, no. 5, pp. 32-43, Sep. 2008.

[11] D. Braga, et al. Optimization of Multi-Domain Queries on the Web, in *VLDB'08*, Auckland, 2008, pp. 562-573.

[12] F. Daniel, et al. Understanding UI Integration - A Survey of Problems, Technologies, and Opportunities, *IEEE Internet Computing*, pp. 59-66, May 2007.

[13] Microsoft Corporation. Smart Client - Composite UI Application Block, December 2005. [Online]. http://msdn.microsoft.com/en-us/library/aa480450.aspx

[14] The Eclipse Foundation. Rich Client Platform, October 2008. [Online]. http://wiki.eclipse.org/index.php/RCP

[15] Sun Microsystems. JSR-000168 Portlet Specification, October 2003. [Online]. http://jcp.org/aboutJava/communityprocess/final/jsr168/

[16] R. Acerbis, et al. Web Applications Design and Development with WebML and WebRatio 5.0. *TOOLS* (46) 2008, pp. 392-411.

[17] J. Gómez, et al. Tool Support for Model-Driven Development of Web Applications, *WISE'05*, pp. 721-730.

[18] D. Braga, S. Ceri, F. Daniel, D. Martinenghi. Optimization of Multi-Domain Queries on the Web. *VLDB'08*, August 2008, Auckland, New Zealand, Pages 562-573.

[19] F. Daniel, F. Casati, B. Benatallah, M.-C. Shan. Hosted Universal Composition: Models, Languages and Infrastructure in mashArt. *ER'09*, November 2009.

Appendix C

Distributed Orchestration of User Interfaces

Florian Daniel[a,1], Stefano Soi[a], Stefano Tranquillini[a], Fabio Casati[a], Chang Heng[b], Li Yan[b]

[a]*Department of Information Engineering and Computer Science, University of Trento*
Via Sommarive 5, 38123 Povo (TN), Italy
[b]*Huawei Technologies*
Shenzhen, P.R. China

Abstract

Workflow management systems focus on the coordination of people and work items, service composition approaches on the coordination of service invocations, and, recently, web mashups have started focusing on the integration and coordination also of pieces of user interfaces (UIs), e.g., a Google map, inside simple web pages. While these three approaches have evolved in a rather isolated fashion – although they can be seen as evolution of the componentization and coordination idea from people to services to UIs – in this paper we describe a component-based development paradigm that conciliates the core strengths of these three approaches inside a single model and language. We call this new paradigm *distributed UI orchestration*, so as to reflect the mashup-like and process-based nature of our target applications. In order to aid developers in implementing UI orchestrations, we equip the described model and language with suitable design, deployment, and runtime instruments, covering the whole life cycle of distributed UI orchestrations.

Keywords: UI orchestration, Distributed UIs, UI orchestration patterns, BPEL4UI, Mashups, UI components, MarcoFlow

1. Introduction

Workflow management systems support office automation processes, including the automatic generation of form-based user interfaces (UIs) for executing the human tasks in a process. *Service orchestrations* and related languages focus instead on integration at the application level. As such, this technology excels in the reuse of components and services but does not facilitate the development of UI front-ends for supporting human tasks and complex user interaction needs, which is one of the most time consuming tasks in software development [1].

Only recently, *web mashups* [2] have turned lessons learned from data and application integration into lightweight, simple composition approaches featuring a significant innovation: integration at the UI level. Besides web services or data feeds, mashups reuse pieces of UI (e.g., content extracted from web pages or JavaScript UI widgets) and integrate them into a new web page. Mashups, therefore, manifest the need for reuse in UI development and suitable UI component technologies. Interestingly, however, unlike what happened for services, this need has not yet resulted in accepted component-based development models and practices.

Email addresses: daniel@disi.unitn.it (Florian Daniel), soi@disi.unitn.it (Stefano Soi), tranquillini@disi.unitn.it (Stefano Tranquillini), casati@disi.unitn.it (Fabio Casati), changheng@huawei.com (Chang Heng), liyanmr@huawei.com (Li Yan)

[1]Corresponding author. Tel. +39 0461 283780, Fax +39 0461 282093

This paper tackles the development of applications that require service composition/process automation logic but that also include human tasks, where humans interact with the system via possibly complex and sophisticated UIs that are tailored to help them in performing the specific job they want to carry out. In other words, this work targets the development of ***mashup-like applications that require process support***, including applications that require distributed mashups coordinated in real time, and provides design and tool support for professional developers, yielding an original composition paradigm based on web-based UI components and web services.

This class of applications manifests a common need that today is typically fulfilled by developing UIs in ad hoc ways and using and manually configuring a process engine in the back-end for process automation. As an example, consider the ***scenario*** in Figure 1: The figure shows a home assistance application for the Province of Trento whose development we want to aid in one of our projects. A patient can ask for the visit of a home assistant (e.g., a paramedic) by calling (via phone) an *operator* of the assistance service. Upon request, the operator inputs the respective details and inspects the patient's data and personal health history in order to provide the assistant with the necessary instructions (steps 1-5). There is always one assistant on duty. The *home assistant* views the description, visits the patient, and files a report about the provided service (steps 6-7). The report is processed by the back-end system and archived (steps 8-9). If no further exams are needed, the process ends (steps 10-11). If exams are instead needed,

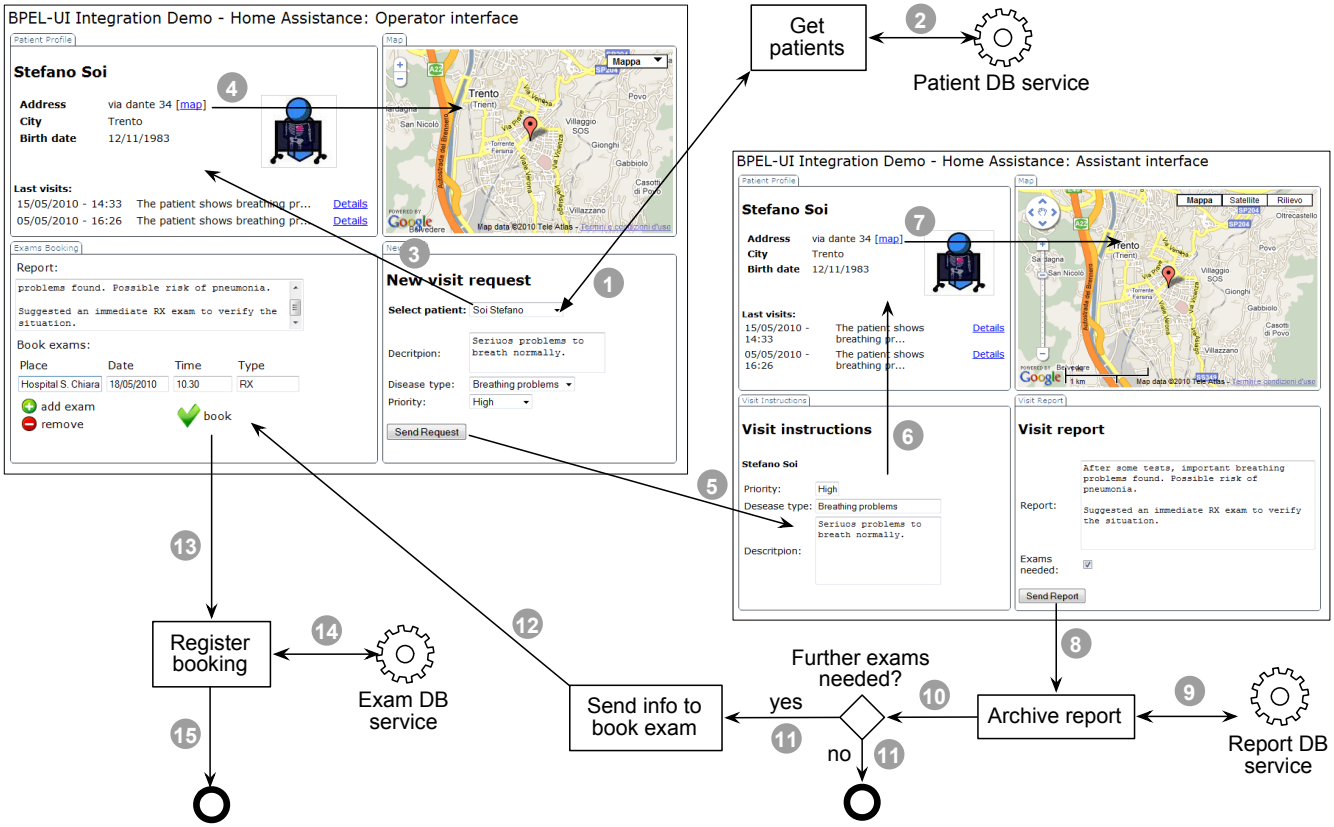


Figure 1: A home assistance application integrating both web services and UI components into a process-like orchestration logic.

the operator books the exam in the local hospital asking confirmation to the patient via phone (steps 12-13). Upon confirmation of the exam booking, the system also archives the booking, which terminates the responsibility of the home assistance service (steps 14-15).

The application in the scenario includes, besides the process logic, two mashup-like, web-based control consoles for the operator and the assistant that are themselves part of the orchestration, need to interact with the process, and are affected by its progress. In addition, the UIs are themselves component-based and created by reusing and combining existing UI components that are instantiated in the users' web browsers (both web pages in Figure 1 are composed of four components). The two applications, once instantiated, allow the operator and assistant to manage an individual request for assistance; each new request requires starting a new instance of the application.

In summary, the scenario requires the coordination of the individual actors in the process and the development of the necessary distributed user interface and service orchestration logic. Doing so requires addressing a set of ***challenges and contributions***:

1. Understanding how to *componentize UIs* and *compose* them into web applications;
2. Defining a logic that is able to *orchestrate both UIs and web services*;
3. Providing a *language and tool* for implementing distributed UI compositions; and
4. Developing a *runtime environment* that is able to execute distributed UI and service compositions.

Implementing the process of the scenario is a non-trivial composition problem that involves multiple development aspects that are only partly addressed by the current state of the art (Section 2). In Section 3, we identify the necessary requirements and outline the approach we follow in this paper, including the architecture of our *MarcoFlow* platform that will serve as guide throughout the rest of the paper. In Section 4, we then introduce the concept of HTML/JavaScript UI component and show how defining a new type of binding allows us to leverage the standard WSDL [3] language to abstractly describe them. We then build on existing composition languages (in particular WS-BPEL [4]) to introduce the notions of UI components, pages, and actors into service compositions (Section 6) and explain how such extension can be used to model UI orchestrations (Section 6). In Section 7 we discuss the different types of UI orchestrations that can be implemented. In Section 8, we show how we extended the Eclipse BPEL editor to support design, and we describe how to run UI orchestrations. Finally, in Section 9 we report on the lessons we learned in the development and use of MarcoFlow and then conclude the paper in Section 10.

2. State of the Art in Orchestrating Services, People and UIs

In most ***service orchestration*** approaches, such as BPEL [4], there is no support for UI design. Many variations of BPEL have been developed, e.g., aiming at the invocation of REST services [5] or at exposing BPEL processes as REST services [6]. IBM's Sharable Code platform [7] follows a slightly different strategy in the composition of REST and SOAP services and also allows the integration of user interfaces for the Web; UIs are however not provided as components but as ad-hoc Ruby on Rails HTML templates filled at runtime with dynamically generated content.

BPEL4People [8] is an extension of BPEL that introduces the concept of people task as first-class citizen into the orchestration of web services. The extension is tightly coupled with the WS-HumanTask [9] specification, which focuses on the definition of human tasks, including their properties, behavior and operations used to manipulate them. BPEL4People supports people activities in form of inline tasks (defined in BPEL4People) or standalone human tasks accessible as web services. In order to control the life cycle of service-enabled human tasks in an interoperable manner, WS-HumanTask also comes with a suitable coordination protocol for human tasks, which is supported by BPEL4People. The two specifications focus on the coordination logic only and do not support the design of the UIs for task execution.

The systematic development of ***web interfaces and applications*** has typically been addressed by the web engineering community by means of model-driven web design approaches. Among the most notable and advanced model-driven web engineering tools we find, for instance, WebRatio [10] and VisualWade [11]. The former is based on a web-specific visual modeling language (WebML), the latter on an object-oriented modeling notation (OO-H). Similar, but less advanced, modeling tools are also available for web modeling languages/methods like Hera [12], OOHDM [13], and UWE [14]. These tools provide expert web programmers with modeling abstractions and automated code generation capabilities for complex web applications based on a hyperlink-based navigation paradigm. WebML has also been extended toward web services [15] and process-based web applications [16]; reuse is however limited to web services and UIs are generated out of dynamically filled HTML templates.

A first approach to ***component-based UI development*** is represented by portals and portlets [17], which explicitly distinguish between UI components (the portlets) and composite applications (the portals). Portlets are full-fledged, pluggable Web application components that generate document markup fragments (e.g., in (X)HTML) that can however only be reached through the URL of the portal page. A portal server typically allows users to customize composite pages (e.g., to rearrange or show/hide portlets) and provides single sign-on and role-based personalization, but there is no possibility to specify process flows or web service interactions; also the new WSRP [18] specification only provides support for accessing remote portlets as web services.

Finally, the ***web mashup*** [2] community has produced a set of so-called mashup tools, which aim at assisting mashup development by means of easy-to-use graphical user interfaces targeted also at non-professional programmers. For instance, Yahoo! Pipes (`http://pipes.yahoo.com`) focuses on data integration via RSS or Atom feeds via a data-flow composition language; UI integration is not supported. Microsoft Popfly (`http://www.popfly.ms`; discontinued since August 2009) provided a graphical user interface for the composition of both data access applications and UI components; service orchestration was not supported. JackBe Presto (`http://www.jackbe.com`) adopts a Pipes-like approach for data mashups and allows a portal-like aggregation of UI widgets (so-called mashlets) visualizing the output of such mashups; there is no synchronization of UI widgets or process logic. IBM QEDWiki (`http://services.alphaworks.ibm.com/qedwiki`) provides a wiki-based (collaborative) mechanism to glue together JavaScript or PHP-based widgets; service composition is not supported. Intel Mash Maker (`http://mashmaker.intel.com`) features a browser plug-in that interprets annotations inside web pages supporting the personalization of web pages with UI widgets; service composition is outside the scope of Mash Maker.

In the mashArt [19] project, we worked on a so-called universal integration approach for UI components and data and application logic services. MashArt comes with a simple editor and a lightweight runtime environment running in the client browser and targets skilled web users. MashArt aims at simplicity: orchestration of distributed (i.e., multi-browser) applications and complex features like transactions or exception handling are outside its scope. The CRUISe project [20] has similarities with mashArt, especially regarding the componentization of UIs. Yet, is does not support the seamless integration of UI components with service orchestration, i.e., there is no support for complex process logic. CRUISe rather focuses on adaptivity and context-awareness. Finally, the ServFace project [21] aims to support even unskilled web users in composing web services that come with an annotated WSDL description. Annotations are used to automatically generate form-like interfaces for the services, which can be placed onto one or more web pages and used to graphically specify data flows among the form fields. The result is a simple, user-driven web service orchestration. None of these projects, however, supports the coordination of multiple different actors inside a same process.

As this analysis shows, existing development approaches for web-based applications lack an integrated support for service orchestration, component-based UI development, *and* coordination of users, three ingredients that instead are necessary to fully implement applications like the one described in our example scenario.

3. Distributed User Interface Orchestration: Definitions, Requirements, and Architecture

If we analyze the home assistance scenario, we see that the envisioned application (as a whole) is highly distributed over the Web: The UIs for the actors participating in the application are composed of UI components, which can be components developed in-house (like the `Patient Profile` component) or sourced from the Web (like the `Map` component); service orchestrations are based on web services. The UI exposes the state of the application and allows users to interact with it and to enact service calls. The two applications for the operator and the assistant are instantiated in different web browsers, contributing to the distribution of the overall UI and raising the need for synchronization.

The key idea to approach the coordination of (i) UI components inside web pages, (ii) web services providing data or application logic, and (iii) individual pages, as well as the people interacting with them, is to split the coordination problem into two layers: *intra-page UI synchronization* and *distributed UI synchronization and web service orchestration*. We call an application that is able to manage these two layers in an integrated fashion a ***distributed UI orchestration*** [22].

3.1. Requirements and approach

Supporting the development of distributed UI orchestrations is a complex and challenging task. Especially the aim of providing a development approach that is able to cover all development aspects in an *integrated* fashion poses requirements to the whole life cycle of UI orchestrations, in particular, in terms of design, deployment, and execution support.

Indeed, supporting the ***design*** of distributed UI orchestrations requires:

- Defining a new type of component, the *UI component*, which is able to modularize pieces of UI and to abstract their external interfaces. For the description of UI components, we slightly extend WSDL [3], obtaining what we call *WSDL4UI*, a language that is able to deal with the novel technological aspects that characterize UI components by reusing the standard syntax of WSDL.

- Bringing together the needs of *UI synchronization and service orchestration* in one single language. UIs are typically event-based (e.g., user clicks or key strokes), while service invocations are coordinated via control flows. In this paper, we show how to extend the standard BPEL [4] language in order to support UIs. We call this extended language *BPEL4UI*.

- Implementing a suitable, *graphical design environment* that allows developers to visually compose services and UI components and to define the grouping of UI components into pages. BPEL comes with graphical editors and ready, off-the-shelf runtime engines that we can reuse. For instance, we extend the Eclipse BPEL editor with UI-specific modeling constructs in order to design UI orchestrations and generate BPEL4UI in output.

Supporting the ***deployment*** of UI orchestrations requires:

- Splitting the BPEL4UI specification into the *two orchestration layers* for intra-page UI synchronization and distributed UI synchronization and web service orchestration. For the former we use a lightweight UI composition logic, which allows specifying how UI components are coordinated in the client browser. For the latter we rely on standard BPEL.

- Providing a set of *auxiliary web services* that are able to mediate communications between the client-side UI composition logic and the BPEL logic. We achieve this layer by automatically generating and deploying a set of web services that manage the UI-to-BPEL and BPEL-to-UI interactions.

Supporting the ***execution*** of UI orchestrations requires:

- Providing a *client-side runtime framework* for UI synchronization that is able to instantiate UI components inside web pages and to propagate events from one component to other components. Events of a UI component may be propagated to components running in the same web page or in other pages of the application as well as to web services.

- Providing a *communication middleware* layer that is able to run the generated auxiliary web services for UI-to-BPEL and BPEL-to-UI communications. We implement this layer by reusing standard web server technology able to instantiate SOAP and RESTful web services.

- Setting up a *BPEL engine* that is able to run standard BPEL process specifications. The engine is in charge of orchestrating web services and distributed UI-to-UI communications. Again, we rely on standard technology and reuse an existing BPEL engine.

- Implementing a *management console* for both developers and participants in UI orchestrations, enabling them to deploy UI orchestrations, to instantiate them, and to participate in them as required.

These requirements and the respective hints to our solution show that the main methodological goals in achieving our UI orchestration approach are (i) relying as much as possible on existing standards (to start from a commonly accepted and known basis), (ii) providing the developer with only few and simple new concepts (to facilitate fast learning), and (iii) implementing a runtime architecture

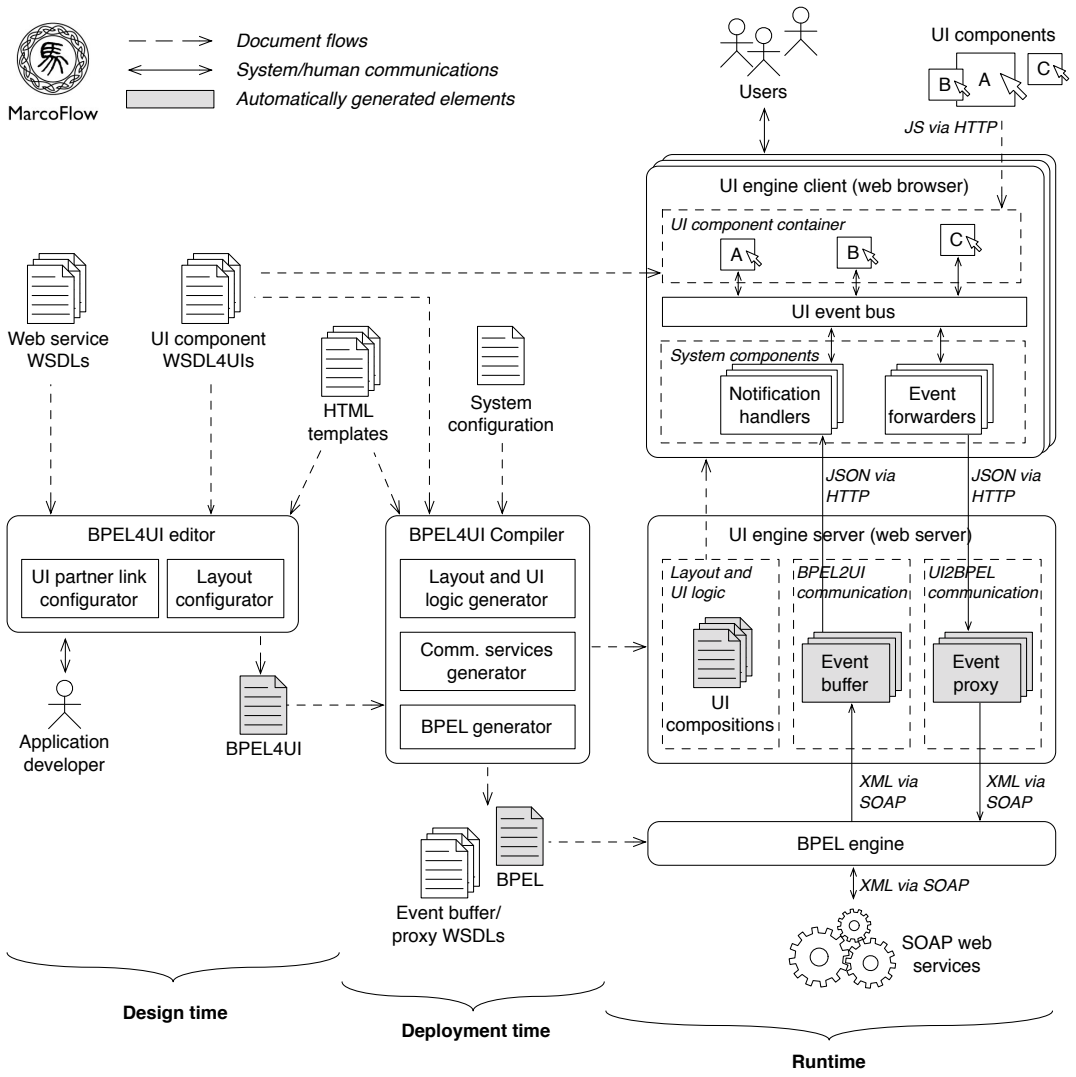


Figure 2: From design time to runtime: overall system architecture of MarcoFlow.

that associates each concern to the right level of abstraction and software tool (to maximize reuse), e.g., UI synchronization is handled in the browser, while service orchestration is delegated to the BPEL engine.

3.2. Architecture

A possible system architecture that meets the above requirements is shown in Figure 2. It's the architecture of our MarcoFlow platform, which has been developed jointly by Huawei Technologies and the University of Trento. For presentation purposes, we discuss a slightly simplified version and partition its software components into design time, deployment time, and runtime components.

The ***design*** part comprises a BPEL4UI editor, which comes with a UI partner link configurator, enabling the setup of UI components inside a UI orchestration, and a layout configurator, assisting the developer in placing UI components into pages. Starting from a set of web service WSDLs, UI component WSDL4UIs, and HTML templates the application developer graphically models the UI orchestration, and the editor generates a corresponding BPEL4UI specification in output, which contains in a single file the whole logic of the UI orchestration.

The ***deployment*** of a UI orchestration requires translating the BPEL4UI specification into executable formats. In fact, as we will see, BPEL4UI is not immediately executable neither by a standard BPEL engine nor by the UI rendering engine (the so-called UI engine in the right hand side of the figure). This task is achieved by the BPEL4UI compiler, which, starting from the BPEL4UI specification, the set of used HTML templates and UI component WSDL4UIs, and the system configuration of the runtime part of the architecture, generates three kinds of outputs:

1. A set of *communication channels* (to be deployed in the so-called UI engine server), which mediate communications between the UI engine client (the client browser) and the BPEL engine. These channels are crucial in that they resolve the technology conflict inherently present in BPEL4UI specifications: a BPEL engine is not able to talk to JavaScript UI components running inside a client browser, and UI components are not able to interact with the SOAP interface of a BPEL engine. For each UI component in a page, the compiler therefore generates (i) an event proxy that is able to forward events from the client browser to the BPEL engine and (ii) an event buffer that is able to accept events from the BPEL engine and store them on behalf of the UI engine client. The compiler also generates suitable WSDL files for proxies and buffers.
2. A *standard BPEL specification* containing the distributed UI synchronization and web service orchestration logic (see Section 6.1). Unlike the BPEL4UI specification, the generated BPEL specification does no longer contain any UI-specific constructs and can therefore be executed by any standards-compliant BPEL engine. This means that all references to UI components in input to the compilation process are rewritten into references to the respective communication channels of the UI components in the UI engine server, also setting the correct, new SOAP endpoints.
3. A set of *UI compositions*[2] (one for each page of the application) consisting of the layout of the page, the list of UI components of the page, the assignment of UI components to place holders, the specification of the intra-page UI synchronization logic (see Section 6.1), and a reference to the client-side runtime framework. Interactions with web services or UI components running in other pages are translated into interactions with local system components (the notification handlers and event forwarders), which manage the necessary interaction with the communication channels via suitable RESTful web service calls.

Finally, the BPEL4UI compiler also manages the deployment of the generated artifacts in the respective runtime environments. Specifically, the generated communication channels and the UI compositions are deployed in the UI engine server and the standard BPEL specification is deployed in the BPEL engine.

The ***execution*** of a UI orchestration requires the setting up and coordination of three independent runtime environments: First, the interaction with the users is managed in the client browser by an event-based JavaScript runtime framework that is able to parse the UI composition stored in the UI engine server, to instantiate UI components in their respective place holders, to configure the notification handlers and event forwarders, and to set up the necessary logic ruling the interaction of the components running inside the client browser. While event forwarders are called each time an event is to be sent from the client to the BPEL engine, the notification handlers are active components that periodically poll the event buffers of their UI components on the UI engine server in order to fetch possible events coming from the BPEL engine.

Second, the UI engine server must run the web services implementing the communication channels. In practice we generate standard Java servlets and SOAP web services, which can easily be deployed in a common web server, such as Apache Tomcat. The use of web server technology is mandatory in that we need to be able to accept notifications from the BPEL engine and the UI engine client, which requires the ability of constantly listening. The event buffer is implemented via a simple relational database (in PostgreSQL, `http://www.postgresql.org`) that manages multiple UI components and distinguishes between instances of UI orchestrations by means of a session key that is shared among all UI components participating in a same UI orchestration instance.

Third, running the BPEL process requires a BPEL engine. Our choice to rely on standard BPEL allows us to reuse a common engine without the need for any UI-specific extensions. In our case, we use Apache ODE (`http://ode.apache.org`), which is characterized by a simple deployment procedure for BPEL processes.

We discuss each of the ingredients in the following.

4. The Building Blocks: Web Services and UI Components

Orchestrating remote application logic and pieces of UI requires, first of all, understanding the exact nature of the components to be integrated, i.e., web services and UI components.

For the *integration of application logic*, we rely on standard web service technologies, such as ***WSDL-SOAP web services***, i.e., remote web services whose external interface is described in WSDL, which supports interoperability via four message-based types of operations: request-response, notification, one-way, and solicit-response. Most of today's web services of this kind are stateless, meaning that the order of invocation of their operations does not influence the success of the interaction, while there are also stateful services whose interaction requires following a so-called business protocol that describes the interaction patterns supported by the service.

For the *integration of UI*, we rely instead on ***JavaScript/HTML UI components***, which are simple, stand-alone web applications that can be instantiated and run inside any common web browser [19]. Figure 3 illustrates an example of UI component (the `Patient Profile` UI component of our reference scenario), along with an excerpt of its JavaScript code. The figure shows that, unlike web services, UI components are characterized by:

[2]Details about the format and logic of these UI compositions can be found in [19].

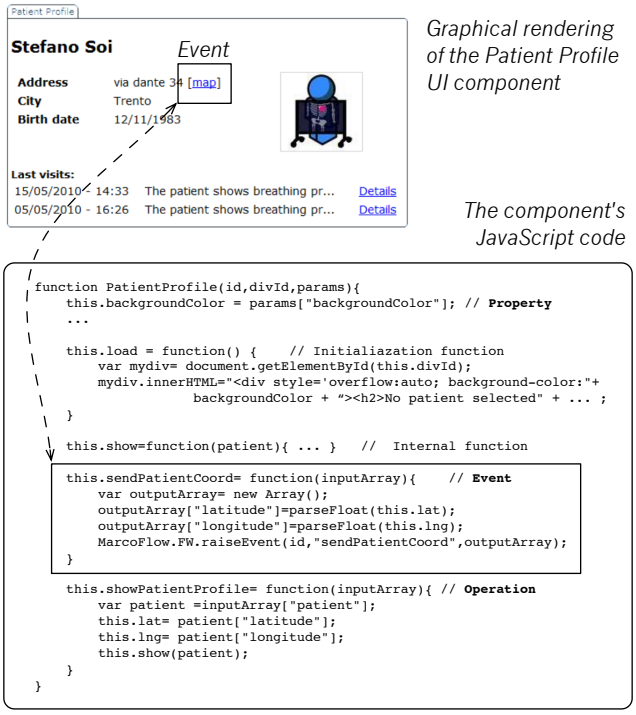


Figure 3: Graphical rendering and internal logic of a UI component

- A ***user interface***. UI components can be instantiated inside a web browser and can be accessed and navigated by a user via standard HTML. The UI allows the user to interactively inspect and alter the content of the component, just like in regular web applications. UI components are therefore stateful, and the component's navigation features replace the business protocol needed for services.
- ***Events***. Interacting with the UI generates system events (e.g., mouse clicks) in the browser used to manage the update of contents. Some events may be exposed as component events, in order to communicate state changes. For instance, a click on the "map" link in Figure 3 launches a `sendPatientCoord` event.
- ***Operations***. Operations enact state changes from the outside. Typically, we can map the event of one component to the operation of another component in order to synchronize the components' state (so that they show related information).
- ***Properties***. The graphical setup of a component may require the setting of constructor parameters, e.g., to align background colors or set other style properties.

In order to make UI components accessible to BPEL, each component must be equipped with a descriptor that describes its events, operations, and properties in terms of WSDL operations. As already anticipated in the previous section, doing so requires extending the standard

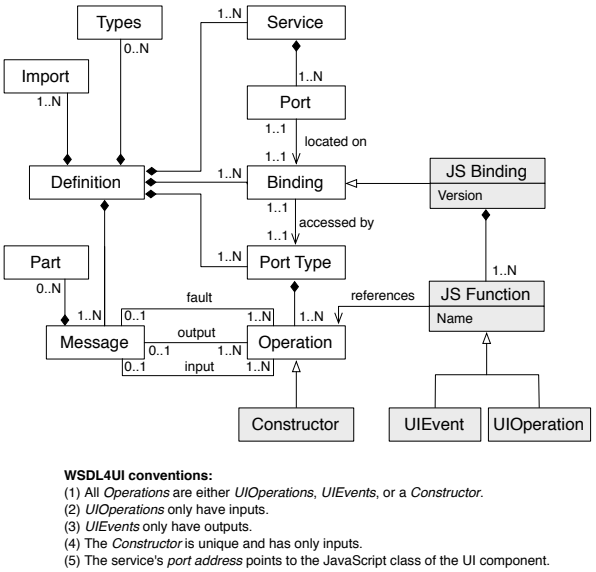


Figure 4: Simplified WSDL4UI meta-model (inspired by [23] and extended – via the gray boxes – toward UI components).

WSDL description logic, i.e., its meta-model, from web services to UI components. The result of this extension is called ***WSDL4UI***. Figure 4 illustrates its meta-model, from which we can see that the extension toward UI components occurs via two different techniques:

1. First, we introduce a set of *conventions* of how the abstract WSDL constructs can be used to describe UI components. The properties of the UI component are encapsulated by means of a dedicated *constructor* operation that can be used to set properties at instantiation time of the component. Next, all operations specified in the description are either UIOperations, UIEvents, or a constructor. *UIOperations* have only inputs; *UIEvents* have only outputs; the constructor is an operation. Finally, the *port address* of the described service corresponds to the URL at which the actual UI component can be downloaded for instantiation (in form of a JavaScript file).
2. Second, we introduce a new *JavaScript binding* that allows us to associate to each abstractly defined operation a JavaScript function of the UI component. Doing so enables the client-side runtime environment (the UI engine client) to parse the WSDL4UI description of a component, to invoke its constructor, and to correctly access events and operations in JavaScript.

Only WSDL files that conform to these rules are considered correct WSDL4UI descriptors of UI components. Figure 5, for instance, shows the descriptor of the `Patient Profile` UI component. Its interface is characterized by three WSDL operations: `ShowPatientProfile`, `SendPatientCoord`, and `constructor` (lines 9-17), corresponding, respectively, to a UIOperation, to a UIEvent and to the component's custructor, as stated in the

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <wsdl:definitions name="PatientProfile" targetNamespace="http://www.unitn.it/
3 JS/Patient" ... >
4   <!-- types definition -->
5   ...
6   <!-- massages definition -->
7   ...
8   <wsdl:portType name="PatientPortType">
9       <wsdl:operation name="constructor">
10          <wsdl:input message="tns:constructorMessage"/>
11      </wsdl:operation>
12      <wsdl:operation name="ShowPatientProfile">
13          <wsdl:input message="tns:ShowPatientProfileMessage"></wsdl:input>
14          </wsdl:operation>
15      <wsdl:operation name="SendPatientCoord">
16          <wsdl:output message="tns:SendPatientCoordMessage"></wsdl:output>
17      </wsdl:operation>
18   </wsdl:portType>
19
20   <wsdl:binding name="PatientJS" type="tns:PatientPortType">
21       <js:binding version="1.0" />
22       <wsdl:operation name="constructor">
23           <js:operation jsFunction="load" />
24       </wsdl:operation>
25       <wsdl:operation name="ShowPatientProfile">
26           <js:operation jsFunction="showPatientProfile" />
27       </wsdl:operation>
28       <wsdl:operation name="SendPatientCoord">
29           <js:event jsFunction="sendPatientCoord" />
30       </wsdl:operation>
31   </wsdl:binding>
32
33   <wsdl:service name="PatientProfile">
34       <wsdl:port name="PatientJS" binding="tns:PatientJS">
35           <soap:address location="http://www.unitn.it/JS/Patient.js" />
36       </wsdl:port>
37   </wsdl:service>
38 </wsdl:definitions>
```

Figure 5: Example of WSDL/UI description of a UI component.

JavaScript binding (lines 20-31). In the binding, there are also specified, through the related `jsFunction` attributes (e.g., line 23), the actual JavaScript functions implementing the operations, which are contained in the file located at the URL defined in the service's port address (line 35).

For the BPEL engine, in order to interact with a component, the BPEL4UI compiler introduced in Section 3.2 generates a respective event buffer and event proxy for the UI engine server and equips them with two standard WSDL descriptors. These descriptors contain the abstract service description as defined in the WSDL4UI file (the event buffer contains all operations of the UI components, the event proxy all events), yet their port addresses point to the newly generated services and their JavaScript binding is turned into a SOAP binding.

5. The UI Orchestration Meta-Model

Starting from web services and UI components, developing a UI orchestration requires modeling two fundamental aspects: (i) the *interaction logic* that rules the passing of data among UI components and web services and (ii) the *graphical layout* of the final application. Supporting these tasks in service orchestration languages (like BPEL) requires extending the expressive power of the languages with UI-specific constructs.

Figure 6 shows the simplified meta-model of BPEL4UI, addressing these two concerns. Specifically, the figure details all the new modeling constructs necessary to specify UI orchestrations (gray-shaded) and omits details of the standard BPEL language, which are reused as is by BPEL4UI (a detailed meta-model for BPEL can be found,

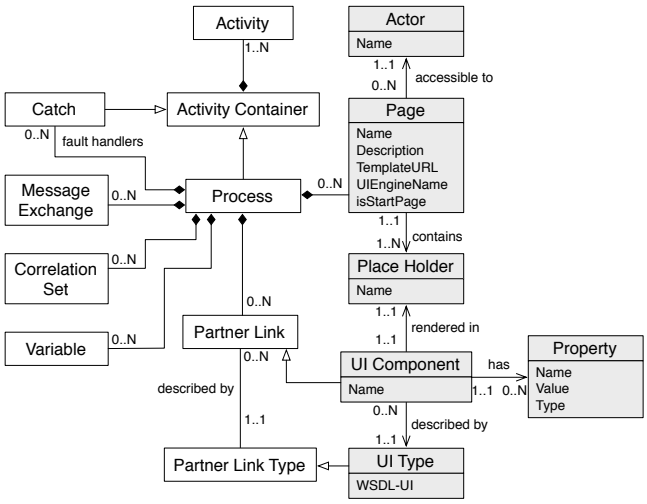


Figure 6: Simplified BPEL4UI meta-model in UML. White classes correspond to standard BPEL constructs [24]; gray classes correspond to constructs for UI and user management.

for instance, in [24]). The code snippet in Figure 7 exemplifies the syntax that we use, in order to express the novel concepts in BPEL4UI.

In terms of standard BPEL [4], a UI orchestration is a *process* that is composed of a set of associated *activities* (e.g., sequence, flow, if, assign, validate, or similar), *variables* (to store intermediate processing results), *message exchanges*, *correlation sets* (to correlate messages in conversations), and *fault handlers*. The services or UI components integrated by a process are declared by means of so-called *partner links*, while *partner link types* define the roles played by each of the services or UI components in the conversation and the port types specifying the operations and messages supported by each service or component. There can be multiple partner links for each partner link type.

Modeling UI-specific aspects requires instead introducing a set of new constructs that are not yet supported by BPEL. The constructs, illustrated in Figure 6, are:

- ***UI type***: The introduction of UI components into service compositions asks for a new kind of partner link type. Although syntactically there is no difference between web services and UI components (the JavaScript binding introduced into WSDL4UI comes into play only at runtime), it is important to distinguish between services and UI components as (i) their semantics and, hence, their usage in the model will be different from that of standard web services, and (ii) the UI orchestration editor must be aware of whether an object manipulated by the developers is a web service or a UI component, in order to support the setting of UI-specific properties.

  As exemplified in Figure 7, we specify the new partner link type like a standard web service type (lines 7-10).

In order to reflect the events and operations of the UI component, we distinguish the two roles. Lines 1-5 define the necessary name spaces and import the WSDL4UI descriptor of the UI component.

- ***Page***: The distributed UI of the overall application consists of one or more web pages, which can host instances of UI components. Pages have a *name*, a *description*, a reference to the pages' *layout template*, the name of the *UI engine* they will run on, and an indication of whether they are a *start page* of the application or not (as we will see in Section 7, inside a process model, not all pages allow the correct instantiation of the process).

  The code lines 13-20 in Figure 7 show the definition of a page called "operator", along with its layout template and the name of the UI engine on which the page will be deployed; the page is a start page for the process.

- ***Place holder***: Each page comes with a set of place holders, which are empty areas inside the layout template that can be used for the graphical rendering of UI components. Place holders are identified by a unique *name*, which can be used to associate UI components.

  Place holders are associated with page definitions and specified as sub-elements, as shown in lines 16-19 in Figure 7.

- ***UI component***: UI types can be instantiated as UI components. For instance, there may be one UI type but two different instances of the type running in two different web pages. Declaring a UI component in a BPEL4UI model leads to the creation of an instance of the UI component in one of the pages of the application. Each component has a unique *name*.

  We specify UI component partner links by extending the standard partner link definition of BPEL with three new attributes, i.e., *isUiComponent*, *pageName*, and *placeHolderName*. Lines 25-32 in Figure 7 show how to declare the `Patient Profile` component of our example scenario.

- ***Property***: As we have seen in the previous section, UI components may have a constructor that allows one to set configuration properties. Therefore, each UI component may have a set of associated properties than can be parsed at instantiation time of the component. We use simple *name-value* pairs to store constructor parameters.

  Properties extend the definition of UI component link types by adding property sub-elements to the partner link definition, one for each constructor parameter, as shown in lines 30-31 in Figure 7.

```
 1 <bpel:process name="HomeAssistance" targetNamespace="http://www.unitn.it/
 2 example/HomeAssistance" xmlns:wsdl6="http://www.unitnt.it/JS/Patient" ...>
 3  <bpel:import namespace="http://www.unitnt.it/JS/Patient"
 4               location="Patient.wsdl" importType="http://
 5               schemas.xmlsoap.org/wsdl/" />
 6  ...
 7  <bpel:partnerLinkType name="PatientPL">
 8      <bpel:role name="receive" portType="wsdl6:PatientPortTypeReceive"/>
 9      <bpel:role name="invoke" portType="wsdl6:PatientPortTypeInvoke"/>
10  </bpel:partnerLinkType>
11  ...
12  <bpel4ui:pages>
13      <bpel4ui:page name="operator" templateURL="operator.html"
14                uiEngineName="HAEngine" actorName="SteS"
15                description="the operator page" isStartPage="true" >
16          <bpel4ui:placeHolder name="marcoflow-top-left" />
17          <bpel4ui:placeHolder name="marcoflow-top-right" />
18          <bpel4ui:placeHolder name="marcoflow-bottom-left" />
19          <bpel4ui:placeHolder name="marcoflow-bottom-right" />
20      </bpel4ui:page>
21      ...
22  </bpel4ui:pages>
23
24  <bpel:partnerLinks>
25      <bpel:partnerLink name="PatientProfileUI_operator"
26                        partnerLinkType="tns:PatientPL"
27                        myRole="receive" partnerRole="invoke"
28                        isUiComponent="yes" pageName="operator"
29                        placeholderName="marcoflow-top-left">
30          <bpel4ui:property name="backgroundColor" type="xsd:string"
31                            value="white" />
32      </bpel:partnerLink>
33      ...
34  </bpel:partnerLinks>
35
36  <!-- orchestration logic definition -->
37  ...
38 </bpel:process>
```

Figure 7: Excerpt of the BPEL4UI home assistance process (new constructs in bold)

- ***Actor***: In order to coordinate the people in a process, pages of the application can be associated with individual actors, i.e., humans, which are then allowed to access the page and to interact with the UI orchestration via the UI components rendered in the page. As for now, we simply associate static actors to pages (using their *names*); yet, actors can easily be assigned also dynamically at deployment time or at runtime by associating roles instead of actors and using a suitable user management system.

  Actors are simply added to page definitions by means of the *actorName* attribute, as highlighted in line 14 in Figure 7.

The addition of these new concepts to BPEL turns the service orchestration language into a language that, in addition to service invocation logic, is also able to specify the organization of an application's UI and its distribution over multiple servers and actors. Our goal in doing so was to keep the number of new concepts as small as possible, while providing a fully operational specification language for UI orchestrations.

6. Modeling Distributed UI Orchestrations

The code example in Figure 7 shows that the UI-specific modeling constructs have a very limited impact on the syntax of BPEL and are mostly concerned with the abstract specification of the layout and the declaration of UI partner links. The actual composition logic, instead, relies exclusively on standard BPEL constructs. Yet, since UI components are different from web services (e.g., it is

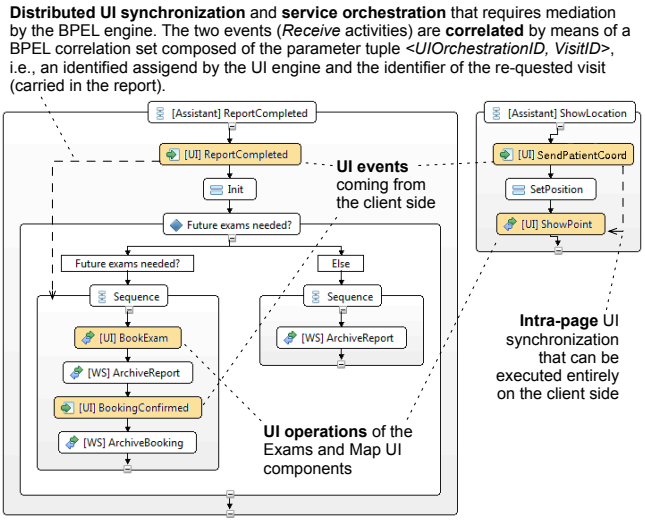


Figure 8: Part of the BPEL4UI model of the home assistance process as modeled in the extended Eclipse BPEL editor (the dashed and dotted lines/arrows have been overlaid as a means to explain the model).

important to know in which page they are running), modeling UI orchestrations requires a profound understanding of the necessary modeling constructs and their semantics. In particular, it is important to understand the effect that individual modeling patterns have on the execution of the final application, i.e., the semantics of the patterns, and which other modeling tasks (data transformations, message correlations, and layout design) are necessary to fully specify a working UI orchestration.

6.1. Core UI orchestration design patterns

The first step toward this understanding is mastering the core design patterns that characterize UI orchestrations. As hinted at in Section 3 and illustrated in Figure 8, we distinguish three main design patterns:

- ***Intra-page UI synchronization***: The small model block (a BPEL *sequence* construct) in the right part of Figure 8 shows the internals of step 7 in Figure 1. When the assistant clicks on the "map" link, the patient's address is shown on the Google map. In BPEL terms, we receive a message from the `Patient Profile` UI component (the event) and forward it to the operation of the `Map` component, both running inside the web page of the assistant. The pattern, hence, implements a so-called *intra-page UI synchronization*, i.e., a synchronization of UI components that run inside a same page. From a runtime point of view, this kind of UI synchronization can be performed entirely on the client side without requiring support from the BPEL engine.

- ***Distributed UI synchronization***: The bigger model block (again a BPEL *sequence* construct) in the left part of the figure, instead, contains a *distributed UI synchronization* that cannot be executed on the client side only, as the two UI components involved in the communication (`Visit Report` and `Exams Booking`) run in different web pages. The event generated upon submission of a new report is processed by the BPEL engine, which then decides whether an additional exam needs to be booked by the operator or not. As such, the BPEL engine manages two independent concerns, i.e., the forwarding of the event from one UI component to another and the evaluation of the condition, of which only the former is necessary to implement a distributed UI synchronization pattern. The execution of a distributed UI synchronization pattern always requires the cooperation of both the BPEL engine and the client-side runtime environment.

- ***Service orchestration***: The distributed UI synchronization also involves the orchestration of the `Report DB` and `Exam DB` web services, as well as some BPEL flow control constructs. In fact, the modeled logic checks whether the report expresses the need for further exams or not. In either case, the further processing of the report involves the invocation of either one or both the web services, in order to correctly terminate the handling of a visit request. The pure invocation of web services represents a service invocation pattern, whose execution can be entirely managed by the BPEL engine without requiring support from the client-side runtime environment.

The BPEL4UI excerpt in Figure 8 shows that, when modeling a UI orchestration, it is important to keep in mind who communicates with whom and which UI component will be rendered where. Depending on these two considerations, the modeled composition logic will either be executed on the client side, in the BPEL engine, or in both layers. For instance, it suffices to associate the `Map` component with a different page, in order to turn the intra-page UI synchronization in the right hand side of Figure 8 into a distributed UI synchronization and, hence, to require support from the BPEL engine.

6.2. Data transformations

When composing services or UI components, it is not enough to model the communication flow only. An important and time-consuming aspect is that of transforming the data passed from one component to another. With BPEL4UI we support all data transformation options provided by BPEL by means of its `Assign` construct. This allows us to leverage on technologies, such as XPath, XQuery, XSLT, or Java, for the implementation of also very complex data transformations.

Yet, it is important to keep in mind that the *type* of data transformation may affect the logic of the UI orchestration: For instance, if the `SetPosition` activity in the

top-right corner of Figure 8 does not transform data at all or only performs simple parameter mappings (with the BPEL `Copy` construct), we fully support the execution of the intra-page UI synchronization in the client browser. If instead a more complex transformation is needed, we rely on the BPEL engine to perform it.

The reason for this choice is that UI synchronization typically requires the exchange of only simple data (e.g., parameter-value pairs), which do not require complex transformation capabilities like the ones we need when interacting with web services. Supporting only simple parameter-parameter mappings on the client side allows us to keep the client-side runtime framework as lightweight as possible, without however giving up any of BPEL's data transformation capabilities.

6.3. Message correlation

Independently of the format of data, UI orchestrations may require a careful design of the messages used in the orchestration and of how these must be *correlated*, in order to enable the runtime environment to dispatch each message to its correct UI orchestration instance. In fact, just like in conventional workflow or service orchestration engines, there may be multiple instances of UI orchestrations running concurrently in a same BPEL/UI engine. Message correlation is required in all those cases where the orchestration involves *multiple entry points* into the orchestration logic (e.g., callbacks from external web services or a condition that requires input from two different events).

If we look at our modeling example in Figure 8, we see that the intra-page UI synchronization in the top-right corner does not involve multiple entry points. It is therefore not necessary to implement any correlation logic in BPEL4UI, in order to propagate the `SendPatientCoord` event from the `Patient Profile` UI component to the `ShowPoint` operation of the `Map` UI component. Since both UI components involved in this synchronization run inside the same web page and, therefore, there is no ambiguity regarding which instance of the `Map` UI component is the target of the `SendPatientCoord` event. In Section 7, we will see that this is not always the case.

The distributed UI synchronization, instead, involves two UI events from two different actors and, hence, different pages: `ReportCompleted` and `BookingConfirmed`. In this case, it is necessary to configure a so-called *correlation set* (in BPEL terminology) that allows the BPEL engine to understand when two instances of those events belong to a same process instance. In the example in Figure 8, we use `UIOrchestrationID` (provided by the UI engine) and `VisitID` (part of the report) as correlation set.

6.4. Graphical layout

Finally, the complete definition of a UI orchestration also requires the design of suitable *HTML templates* and the assignment of UI components to their place holders inside the pages. As our goal is the development of an enabling middleware layer for UI orchestrations, for the layout templates we rely on standard web design instruments and technologies (e.g., Adobe Dreamweaver). The only requirement the templates must satisfy is that they provide place holders in form of HTML DIV elements that can be indexed via standard HTML identifiers following a predefined naming convention: `<div id="marcoflow-..."></div>`.

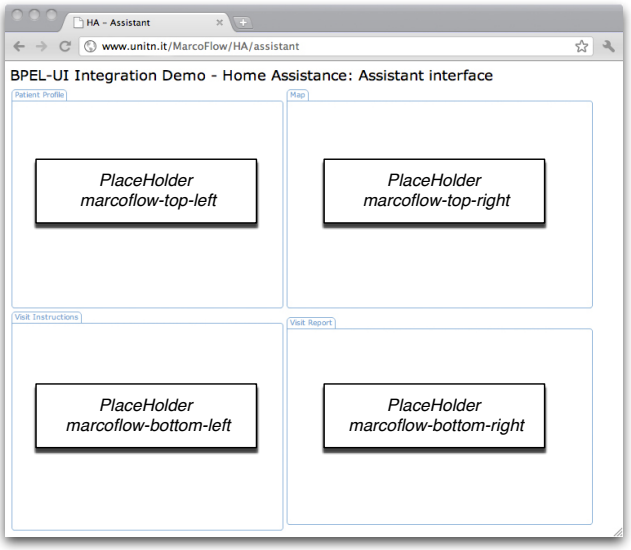


Figure 9: The HTML template of the assistant's web page highlighting the empty place holders for UI components.

Figure 9, for instance, depicts the empty HTML template of the assistant's web page, whose filled version we have already seen in Figure 1. The template is a simple HTML page with a page title and the four uniquely identified placeholders to be filled with UI components at runtime. Differently from dynamic HTML and most of the approaches discussed in Section 2, in which the template typically also contains the formatting logic for the data to be rendered inside the place holders, in our case the template only identifies the location of the UI components; the rendering of content is then managed autonomously by the UI components.

Once all HTML templates for all pages in the UI orchestration are defined, the definition of the pages and the association of UI partner links with place holders therein proceeds as exemplified in Section 6.

7. Types of UI orchestrations

So far we have seen how BPEL4UI supports the development of distributed UI orchestrations. Yet, developing correct UI orchestrations is still a non-trivial task, in that the distribution of UI synchronizations and service orchestrations over two different runtime engines (the UI engine and the BPEL engine) complicates the instantiation logic of distributed UI orchestrations, an aspect that developers

should understand thoroughly. As illustrated in Figure 10, we identify four main types of UI orchestrations that can be implemented by means of the core patterns described in Section 6.1, i.e., *pure UI synchronizations*, *pure service orchestrations*, *UI-driven UI orchestrations*, and *process-driven UI orchestrations*. The developer needs to master these configurations if he doesn't want to encounter unexpected behaviors or errors at runtime. We discuss each of these configurations next.

7.1. Pure UI synchronizations

From a UI point of view, the basic type of UI orchestration is represented by applications that involve *UI components only* and, hence, exclusively focus on the synchronization of UIs via events. Typical examples of this type of UI orchestration are UI-based mashups, portlets/portals, applications that integrate widgets/gadgets, or similar component-based UI applications.

Figure 10(a) illustrates a simple example: There are two concurrent pages, possibly associated with two different users and with a total of three UI components, one in `Page 1` and two in `Page 2`. By interacting with the UI component `A`, the user can generate an event that synchronizes component `B` in the other page; likewise, another user can interact with `B` and synchronize both `A` and `C`, while `C` allows the user to synchronize again `B`. The three UI components are instantiated in their web pages and run until the users close their web browsers or navigate to another web page. As such, UI components are stateful: their UI constantly reflects the interaction state of the users with the component (e.g., in terms of selections or navigation actions performed). During their lifetime, each UI component may generate multiple events as output and accept multiple events as input. That is, while in one instance of the UI orchestration in Figure 10(a) each UI component is instantiated only once, there may be multiple instances of synchronization events (the dashed arrows).

Supporting the *execution* of this type of UI orchestration requires the presence of both a client-side runtime environment and a server-side environment. Specifically, the intra-page UI synchronization of `B` and `C` can be handled in the client, since both UI components run inside the same web page, i.e., web browser. The synchronization of `A` and `B`, instead, requires help from the server side, in that they implement a distributed UI synchronization. Therefore, the event proxy on the server side (cf. Figure 2) is needed, in order to get the two web pages into communication.

Sending an event through the event proxy raises the need for *correlation*, in that there may be multiple instances of a same UI orchestration running concurrently and, therefore, it is necessary to identify which event belongs to which instance. The solution we adopt is to add to each generated UI event a so-called *UIOrchestrationID*, which uniquely identifies the UI orchestration instance. The identifier is generated by the UI engine at application startup and shared with all the users participating in the orchestration. This feature is automated in our runtime framework and does not require any specific modeling at design time.

7.2. Pure service orchestrations

From a web service point of view, the basic type of UI orchestration is the one that completely comes *without UI*, i.e., a common web service orchestration. Although this configuration represents a "degenerated" UI orchestration (given that there is no UI), it is fully supported by BPEL4UI and deserves an explanation in that it represents the building block for the next UI orchestration types. Typical examples are order processing logics or payment processes.

Figure 10(b) provides an example: There are six web service invocations (specifically, synchronous request-response invocations) and one incoming event arranged in a typical service orchestration. For presentation purpose, we adopt a *data flow* logic to model the orchestration, as for the discussion in this section it is not important to explicitly distinguish between control and data flow. The important aspect of the model is that, upon instantiation of the service orchestration, each element in the model is instantiated exactly once – including the data flow connectors (differently from what happened with the UI synchronization events in Figure 10(a)). The data flow connectors rule both which service invocation can be performed and how data are passed from one invocation to another.

Executing such a service orchestration requires support from an orchestration engine/server, such as a BPEL engine, which is able to instantiate on orchestration model, to invoke the services as prescribed by the model, to transform data formats between service invocations, to accept incoming notifications or events, and to keep the state of the progress in the orchestration instance. The actual services run remotely, and are outside the scope of the orchestration environment.

The important aspect of the model in Figure 10(b) is the incoming event (graphically represented by the letter in the circle), as the event raises the need for *correlation* in the service orchestration. In fact, without the incoming event, the model would consist only of synchronous service invocations, which could be processed easily step by step by the orchestration engine. The engine would simply invoke a service, wait for its response, pass the response to the next service, and so on till the whole orchestration logics ends. In the presence of the incoming event, instead, the engine must be able to correlate each incoming event it receives with the correct target orchestration instance of the event. Doing so requires sharing at least a simple key or identifier (the *correlation set*) among the running orchestration instance and the incoming event. For instance, the name of the person who starts the orchestration instance could be used as correlation identifier, as such could be known to both the engine and the external service sending the event – provided that there is always only one instance per person running in the engine.

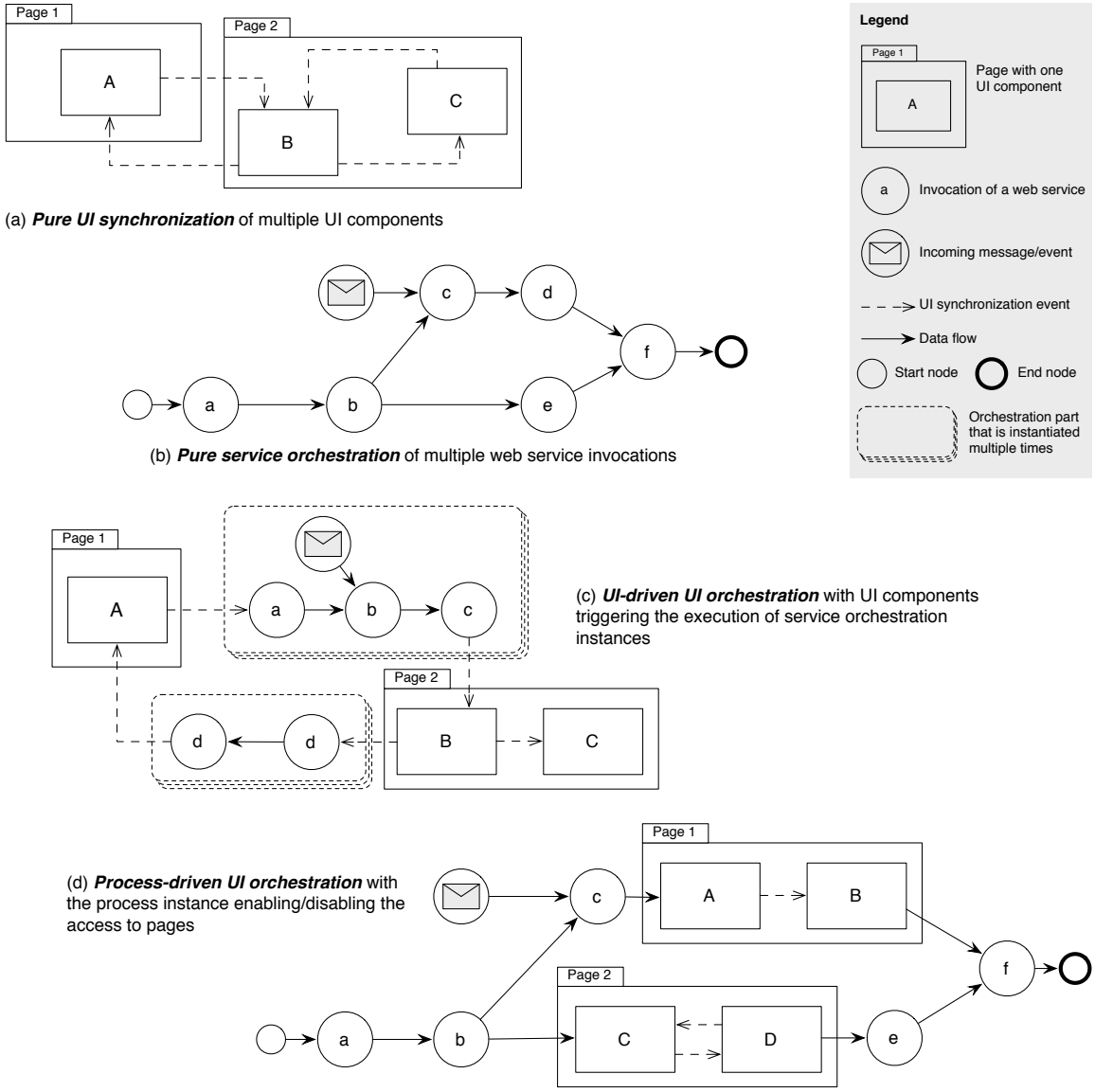


Figure 10: The four types of (UI) orchestration supported by BPEL4UI and the MarcoFlow system.

7.3. UI-driven UI orchestrations

A "full" UI orchestration, however, is characterized by the joint use of both UI synchronizations and service orchestrations inside a same application. Depending on which of these two ingredients dominates the behavior of the application, we can have either UI-driven orchestrations (where service orchestrations are enacted by the UI) or process-driven orchestrations (where the UIs are enacted by the service orchestration). Here we focus on the former type, in the next section we discuss the latter. For instance, a web mashup that integrates RSS data from a Yahoo! Pipe may invoke the pipe processing logic multiple times while running.

Figure 10(c) abstracts this type of UI orchestration: There are two pages with respective UI components and two service orchestration flows. While the intra-page UI synchronization of `B` and `C` does not involve any web service, the distributed UI synchronizations of `A` and `B` are based on intermediate service invocations in both directions. Just like we can have multiple UI synchronization events (the dashed arrows) for each instance of UI component, we now also have for each synchronization of `A` and `B` a new instance of the intermediate service orchestration logic (graphically represented by the dashed box around the service orchestrations).

In order to *execute* such a UI-driven UI orchestration, we need to join also the power of the runtime environments of the two previous configurations. Specifically, UI synchronizations involving service invocations can no longer be performed with a simple event proxy on the server side

only (like in pure UI orchestrations); instead, the synchronization requires a tight integration of the client-side runtime environment for UIs with the server-side service orchestration engine. Specifically, a UI synchronization event from one page must be able to instantiate and provide input to a service orchestration logic on the server side, which, in turn, must be able to deliver its output in form of a UI synchronization event sent to another page. That is, we need to have a full two-way communication channel between the two runtime environments, a feature that is implemented by the UI components' event proxies and event buffers in the UI engine server.

In terms of *correlation*, all UI synchronization events carry the `UIOrchestrationID`, as already introduced for pure UI orchestrations, while the service orchestration parts may require additional correlation information inside BPEL4UI, depending on their individual topology. For instance, the service orchestration enacted by propagating an event from `B` to `A` only involves synchronous service invocations and does therefore not require any additional correlation information. The other service orchestration in Figure 10(c), instead, also involves the reception of an external event, which requires the setup of an additional correlation identifier, as already described for Figure 10(b).

7.4. Process-driven UI orchestrations

Finally, we have a process-driven UI orchestration each time we have an application that brings together UI synchronizations and service orchestrations in which the service orchestration dominates over the UI synchronization. For instance, workflow management or, more in general, business process management applications that integrate both web services and UI components and that orchestrate tasks (work items) to be performed by either users or automated resources, such as our reference scenario, can be considered of this type of UI orchestration.

Figure 10(d) schematically illustrates the situation: The application starts with a pure service orchestration that enacts a set of services and, only after the successful processing of services `a`, `b`, `c`, and `d`, allows the users to access their respective web pages. Inside the pages, there are UI components that allow the users to interact with the pages and to perform and conclude their tasks, which causes the UI orchestration to leave again and disable the pages and to proceed with the processing of the remaining part of the service orchestration. That is, in process-driven UI orchestrations pages are invoked like services, but they are targeted at users and, therefore, expose a UI the users can interact with. The overall UI orchestration keeps waiting until the user successfully completes his/her task, which is communicated via an outgoing UI synchronization event.

In terms of required *execution* support, process-driven UI orchestrations are similar to UI-driven UI orchestrations, with the difference that the main service orchestration is instantiated only ones, not multiple times.

Correlation requirements are similar, too. As shown in Figure 10(d), if there is an incoming event that needs to be injected into a running instance of the UI orchestration, correlation is needed; otherwise, the whole UI orchestration can also be processed without correlation. UI synchronization events are again managed via the orchestration's unique identifier associated by the UI engine.

7.5. Complex UI orchestrations

The four types of UI orchestrations above represent those classes of UI orchestrations that characterize the most important application scenarios we encountered throughout the development of the MarcoFlow system. Yet, UI orchestrations may easily also get more complex. For instance, it is possible to use a process-driven UI orchestration (including again UIs and actors) in place of any of the simple service orchestrations in Figure 10(c), or it is possible to expand the simple pages in Figure 10(d) into complete UI-driven UI orchestrations (including new service orchestrations), or we could establish UI synchronizations among the two pages in Figure 10(d), and similar. While these kinds of UI orchestrations are theoretically possible and supported by BPEL4UI and MarcoFlow, luckily it is hard to find practical examples that indeed require such a level of complexity.

8. Implementing and Running UI Orchestrations

In order to ease the development, deployment, and execution of UI orchestrations, MarcoFlow comes with two tools that aid the different actors involved: a *graphical BPEL4UI editor* for developers and a *web-based management console* for both developers and users.

The ***graphical BPEL4UI editor*** for developers has been implemented as an extension of the Eclipse BPEL editor (`http://www.eclipse.org/bpel/`) and comes with (i) a panel for the specification of the pages in which UI components can be rendered and (ii) a property panel that allows the developer to configure the web pages, to set the properties of UI partner links, and to associate them to place holders in the layout.

The screenshot in Figure 11 shows the editor at work. The layout structure of the editor is the same of the standard Eclipse editor, except for some differences in the right and bottom side. On the right side, now it is also possible to define the pages of the UI orchestration (as elements of the *Pages* group). Selecting a page in the list shows the respective details in the *Properties* panel in the lower part of the figure and allows the developer to assign the actor, i.e., the user that will be allowed to access the page, and the HTML template for the page. Still on the right side, where usually there are only partner links for web services, now it is also possible to define UI partner links for UI components. Selecting a partner link from the list again shows its details in the *Properties* panel. Ticking the *UI component* checkbox turns the partner link into a

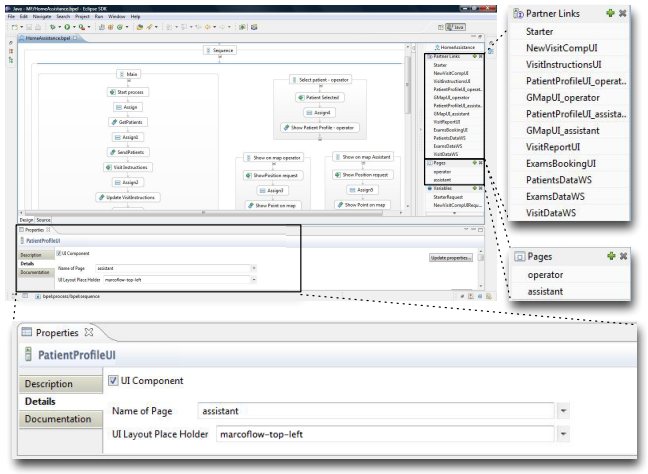


Figure 11: The extended Eclipse BPEL editor for developing UI orchestrations at work.

Figure 12: The management console for developers and users allowing them to deploy, instantiate, and participate in UI orchestrations.

UI partner link and allows the developer to define in which page and place holder inside the page the UI component will be rendered. The actual composition logic is specified in the modeling canvas in the central part of the editor.

The ***web-based management console*** helps (i) developers in deploying ready UI orchestrations and (ii) users in instantiating and participating in running UI orchestrations. Deploying a new UI orchestration requires the developer to pack all the project files (web service WSDLs, UI component WSDL4UIs, BPEL4UI specification, HTML templates, and the system configuration) into a single archive file and to upload it to the management console. Doing so allows the developer to deploy the application by means of a simple mouse click, which invokes the BPEL4UI compiler and generates the standard BPEL file, the event buffers and event proxies, their respective WSDL files, and the UI compositions and then deploys all generated artifacts in the respective runtime environments.

Figure 12, instead, shows the interface of the management console for regular users, where they can see which UI orchestrations have been deployed they have also access to. Specifically, a user can either start a new instance of UI orchestration (via the upper list in the figure) or participate in an already running instance of UI orchestration (via the lower list in the figure), which – in the case of the operator and assistant in our example scenario – leads him/her, for example, to one of the pages in Figure 1. The operator is allowed to instantiate the orchestration, and the assistant is enabled to participate.

The MarcoFlow system shown in Figure 2 is fully implemented and running (a demo of the tool is available at `http://mashart.org/marcoflow/demo.htm`). In our test setting, we run the UI engine server and the BPEL engine on the same machine, yet these components could also easily be distributed over different physical machines, a feature that is already supported by our code generator.

9. Lessons Learned

We conclude the paper with a few considerations on lessons learned while developing and applying MarcoFlow. In hindsight, these correspond to design decisions we would have done a little bit differently or to next steps that we are indeed undertaking.

One observation is that developers seemed to prefer a *web-based* environment rather than an Eclipse-based one. We had chosen Eclipse because it already comes with an open-source editor for BPEL, and we felt it was rather powerful and reasonably easy to extend as opposed to developing a new editor. In the end, working with the editor took a lot of time, so that we did not get the benefits of a web-based editor nor the time savings we hoped for.

A second issue relates to the large number of *conversions* of messages from SOAP to REST and vice versa. In the current approach, even when two rest services are communicating we always need to soap-ify them. While we aim to minimize this kind of conversions as much as possible (by keeping intra-page UI synchronizations on the client), this limits the scalability if a single UI engine is used.

Another interesting finding we did not realize in the beginning was that, since UI orchestrations intermix *stateless* elements (web service invocations) with *stateful* elements (UI components) the need for correlation in UI orchestrations is higher than in pure web service orchestrations. Design-time and runtime constructs here may be needed to simplify specifications and make the engine more scalable.

However the main considerations that will drive our research are in terms of usability and applicability. While working with BPEL was a strong requirement initially, many companies are increasingly considering mashup languages for non mission-critical applications, targeting relatively simple ways to integrate and present web-accessible data. This would fit well with the MarcoFlow approach, which can be extended to deal with mashup languages.

Finally, working with MarcoFlow and experimenting its usage helped us strengthen our belief that BPEL, its variations, and actually even mashup languages are not suitable for end users, no matter how good development tools are. Our conclusion here is that if we want to bring development power to the end users or at least to knowledge workers we need to define domain-specific models and tools rather than general purpose ones. This is the road we begun to undertake in our efforts within the Omelette EU FP7 project. Yet, we also recognize that the intrinsic complexity of UI orchestrations hardly suits the capabilities of less-skilled developers or end users.

10. Conclusion

The spectrum of applications whose design intrinsically depends on a structured flow of activities, tasks or capabilities is large, but current workflow or business process management software is not able to cater for all of them. Especially lightweight, component-based applications or Web 2.0 based, mashup-like applications typically do not justify the investment in complex process support systems, either because their user basis is too small or because there is need only for few, simple applications. Yet, these applications too demand for abstractions and tools that are able to speed up their development, especially in the context of the Web with its fast development cycles.

We introduced an approach to what we call *distributed UI orchestration*, a component-based development technique that introduces a new first-class concept into the workflow management and service composition world, i.e., UIs, and that fits the needs of many of today's web applications. We proposed a model for UI components and showed how dealing with them requires extending the expressive power of a standard service composition language, such as BPEL. We equipped the language with a modeling environment and a code generator able to produce artifacts that can be executed straightaway by our runtime environment, which separates intra-page UI synchronization from distributed UI synchronization and service orchestration. The result is an approach to distributed UI orchestration that is comprehensive and free.

A strong point of the described approach is that it recognizes the need for abstraction and more expressive models and languages at design time, while – thanks to its strong separation of concerns and powerful code generator – it does not require any new language or system at runtime.

References

[1] B. A. Myers, M. B. Rosson, User interface programming survey, SIGCHI Bull. 23 (1991) 27–30.

[2] J. Yu, B. Benatallah, F. Casati, F. Daniel, Understanding Mashup Development, IEEE Internet Computing 12 (2008) 44–52.

[3] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, Web Services Description Language (WSDL) 1.1, W3C Note, W3C, `http://www.w3.org/TR/wsdl`, 2001.

[4] OASIS, Web Services Business Process Execution Language Version 2.0, Technical Report, `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`, 2007.

[5] C. Pautasso, BPEL for REST, in: BPM'08, pp. 278–293.

[6] T. v. Lessen, F. Leymann, R. Mietzner, J. Nitzsche, D. Schleicher, A Management Framework for WS-BPEL, in: Proceedings of the 2008 Sixth European Conference on Web Services, IEEE Computer Society, Washington, DC, USA, 2008, pp. 187–196.

[7] E. M. Maximilien, A. Ranabahu, K. Gomadam, An Online Platform for Web APIs and Service Mashups, IEEE Internet Computing 12 (2008) 32–43.

[8] Active Endpoints, Adobe, BEA, IBM, Oracle, SAP, WS-BPEL Extension for People (BPEL4People) Version 1.0, Technical Report, 2007.

[9] Active Endpoints, Adobe, BEA, IBM, Oracle, SAP, Web Services Human Task (WS-HumanTask) Version 1.0, Technical Report, 2007.

[10] R. Acerbis, A. Bongio, M. Brambilla, S. Butti, S. Ceri, P. Fraternali, Web Applications Design and Development with WebML and WebRatio 5.0, in: Objects, Components, Models and Patterns, volume 11 of *LNBIP*, Springer, 2008, pp. 392–411.

[11] J. Gómez, A. Bia, A. Parraga, Tool Support for Model-Driven Development of Web Applications, in: Web Information Systems Engineering – WISE 2005, volume 3806 of *LNCS*, Springer Berlin / Heidelberg, 2005, pp. 721–730.

[12] R. Vdovjak, F. Frasincar, G.-J. Houben, P. Barna, Engineering Semantic Web Information Systems in Hera, Journal of Web Engineering 2 (2003) 3–26.

[13] D. Schwabe, G. Rossi, S. D. J. Barbosa, Systematic Hypermedia Application Design with OOHDM, in: HYPERTEXT '96: Proceedings of the the seventh ACM conference on Hypertext, ACM Press, New York, NY, USA, 1996, pp. 116–128.

[14] N. Koch, A. Kraus, R. Hennicker, The Authoring Process of the UML-based Web Engineering Approach, in: D. Schwabe (Ed.), First International Workshop on Web-oriented Software Technology (IWWOST01).

[15] I. Manolescu, M. Brambilla, S. Ceri, S. Comai, P. Fraternali, Model-driven design and deployment of service-enabled web applications, ACM Trans. Internet Technol. 5 (2005) 439–479.

[16] M. Brambilla, S. Ceri, P. Fraternali, I. Manolescu, Process modeling in Web applications, ACM Trans. Softw. Eng. Methodol. 15 (2006) 360–409.

[17] Sun Microsystems, JSR-000168 Portlet Specification, Technical Report, `http://jcp.org/aboutJava/communityprocess/final/jsr168/`, 2003.

[18] OASIS, Web Services for Remote Portlets, Technical Report, `www.oasis-open.org/committees/wsrp`, 2003.

[19] F. Daniel, F. Casati, B. Benatallah, M.-C. Shan, Hosted Universal Composition: Models, Languages and Infrastructure in mashArt, in: Proceedings of the 28th International Conference on Conceptual Modeling, ER '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 428–443.

[20] S. Pietschmann, M. Voigt, A. Rümpel, K. Meißner, CRUISe: Composition of Rich User Interface Services, in: Proceedings of the 9th International Conference on Web Engineering, ICWE '9, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 473–476.

[21] M. Feldmann, T. Nestler, K. Muthmann, U. Jugel, G. Hübsch, A. Schill, Overview of an end-user enabled model-driven development approach for interactive applications based on annotated services, in: Proceedings of the 4th Workshop on Emerging Web Services Technology, WEWST '09, ACM, New York, NY, USA, 2009, pp. 19–28.

[22] F. Daniel, S. Soi, S. Tranquillini, F. Casati, C. Heng, L. Yan, From People to Services to UI: Distributed Orchestration of User Interfaces, in: BPM'10, pp. 310–326.

[23] A. D'Ambrogio, A Model-driven WSDL Extension for Describing the QoS of Web Services, in: IEEE International Conference on Web Services (ICWS'06), pp. 789–796.

[24] WSPER.org, WS-BPEL 2.0 Metamodel, Technical Report, `http://www.ebpml.org/wsper/wsper/ws-bpel20.html`, 2007.

Appendix D

From People to Services to UI: Distributed Orchestration of User Interfaces

Florian Daniel, Stefano Soi, Stefano Tranquillini, Fabio Casati
University of Trento, Povo (TN), Italy
{daniel,soi,tranquillini,casati}@disi.unitn.it

Chang Heng, Li Yan
Huawei Technologies, Shenzhen, P.R. China
{changheng,liyanmr}@huawei.com

Abstract. Traditionally, workflow management systems aim at alleviating people's burden of coordinating repetitive business procedures, i.e., they coordinate *people*. Web service orchestration approaches, instead, coordinate pieces of software (the *web services*), hiding the human aspects that are intrinsically present in any business process behind the services. The recent emergence of technologies like BPEL4People and WS-HumanTask, which introduce human actors into service compositions, manifest that taking into account the people involved in business processes is however important. Yet, none of these approaches allow one to also develop the *user interfaces* (UIs) the users need to concretely participate in a business process.
With this paper, we want to go one step beyond state-of-the-art workflow management and service composition and propose an original model, language and running system for the composition of distributed UIs, an approach that allows us to bring together UIs, web services and people in a single orchestration logic and tool. To demonstrate the effectiveness of the idea, we apply the approach to a real-world home assistance scenario.

1 Introduction

Workflow management systems support office automation processes, including the automatic generation of form-based user interfaces (UIs) for executing the human tasks in a process. *Service orchestrations* and related languages focus instead on integration at the application level. As such, this technology excels in the reuse of components and services but does not facilitate the development of UI front-ends for supporting human tasks and complex user interaction needs, which is one of the most time consuming tasks in software development [1].

Only recently, *web mashups* [2] have turned lessons learned from data and application integration into lightweight, simple composition approaches featuring a significant innovation: integration at the UI level. Besides web services or data feeds, mashups reuse pieces of UI (e.g., content extracted from web pages or JavaScript UI widgets) and integrate them into a new web page. Mashups, therefore, manifest the need for reuse in UI development and suitable UI component technologies. Interes-

tingly, however, unlike what happened for services, this need has not yet resulted in accepted component-based development models and practices.

This paper tackles the development of applications that require service composition/process automation logic but that also include human tasks, where humans interact with the system via a possibly complex and sophisticated UI that is tailored to help them in performing the specific job they need to carry out. In other words, this work targets the development of mashup-like applications that require process support, including applications that require distributed mashups coordinated in real time, and provides design and tool support for professional developers, yielding an original composition paradigm based on web-based UI components and web services.

This is a common need that today is typically fulfilled by developing UIs in ad hoc ways and using a process engine in the back-end for process automation. As an example, consider the following scenario.

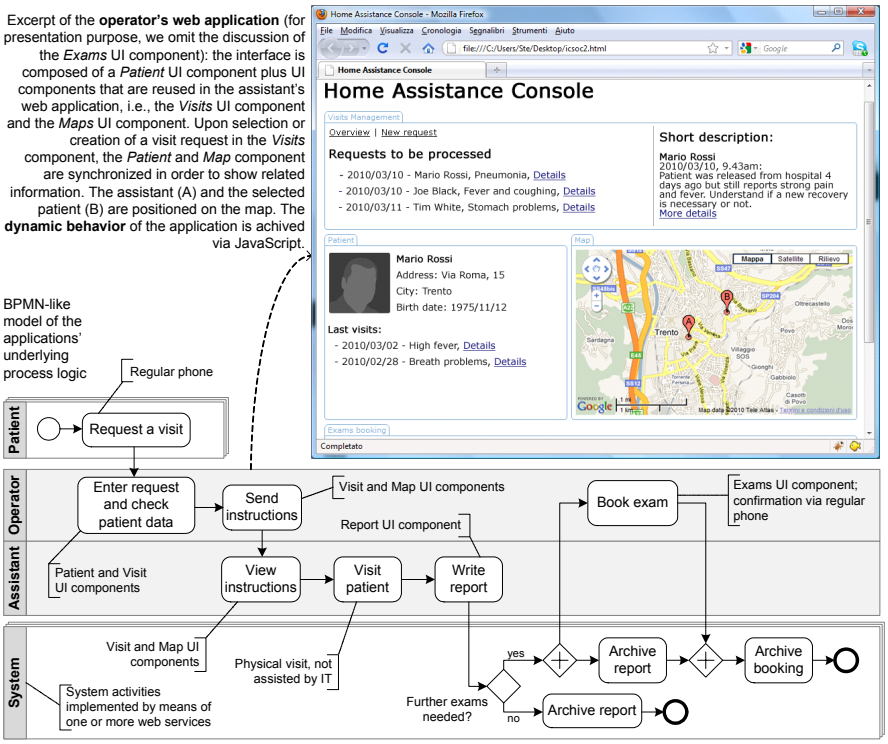


Figure 1 Simplified home assistance process: gray shaded swim lanes are instantiated only once (in form of suitable UIs) and handle multiple instances of white shaded swim lanes.

Scenario. Figure 1 shows the high-level model of a home assistance process in the Province of Trento we want to aid in one of our projects. A *patient* can ask for the visit of a home assistant (e.g., a paramedic) by calling (via phone) an operator of the assistance service. Upon request, the *operator* inputs the respective details and inspects the patient's data and personal health history in order to provide the assistant

with the necessary instructions. There is always one assistant on duty. The *home assistant* views the description, visits the patient, and files a report about the provided service. The report is processed by the *back-end system* and archived if no further exams are needed. If exams are instead needed, the operator books the exam in the local hospital asking confirmation to the patient (again via phone); in parallel, the system archives the report. Upon confirmation of the exam booking, the system also archives the booking, which terminates the responsibility of the home assistance service.

Our goal is to develop an application that supports this process. This application includes, besides the process logic, two mashup-like, web-based control consoles for the operator and the assistant that are themselves part of the orchestration and need to interact with (and are affected by) the evolution of the process. Furthermore, the UI can be itself component-based and created by reusing and combining existing UI components. The two applications, once instantiated, should be able to manage multiple requests for assistance, while the system activities will be instantiated independently for each report to be processed.

Challenges and contributions. The scenario requires the coordination of the individual actors in the process and the development of the necessary *distributed* user interface *and* service orchestration logic. Doing so requires (i) understanding how to *componentize UIs and compose them into web applications*, (ii) defining a logic that is able to *orchestrate both UIs and web services*, (iii) providing a language and tool for *specifying distributed UI compositions*, and (iv) developing a runtime environment that is able to *execute distributed UI and service compositions*.

Structure of the paper. Implementing the process of the scenario is a non-trivial composition problem. After describing the UI orchestration approach (Section 3), in this paper we show how defining a new type of binding allows us to leverage the standard WSDL [4] language to describe HTML/JavaScript UI components (Section 4). We then build on existing composition languages (in particular WS-BPEL [5]) to introduce the notions of UI components, pages, and actors to support the specification of distributed UI compositions (Section 5). The extended BPEL is compiled to generate the UI composition logic (that runs entirely on the browser, for performance reasons) and the server-side logic that performs service orchestration and distributed UI synchronization. Finally, we extend the Eclipse BPEL editor to support this extension, and we describe a system that is able to execute distributed UI compositions, starting from the extended BPEL specification. These models and tools are integrated in a hosted development and execution platform, called MarcoFlow (Section 6), jointly developed by Huawei Technologies and the University of Trento.

2 State of the Art in Orchestrating Services, People and UIs

In most **service orchestration** approaches, such as BPEL [5], there is no support for UI design. Many variations of BPEL have been developed, e.g., aiming at the invocation of REST services [6] or at exposing BPEL processes as REST services [7]. IBM's Sharable Code platform [8] follows a slightly different strategy in the composition of REST and SOAP services and also allows the integration of user interfaces

for the Web; UIs are however not provided as components but as ad-hoc Ruby on Rails HTML templates.

BPEL4People [9] is an extension of BPEL that introduces the concept of people task as first-class citizen into the orchestration of web services. The extension is tightly coupled with the **WS-HumanTask** [10] specification, which focuses on the definition of human tasks, including their properties, behavior and operations used to manipulate them. BPEL4People supports people activities in form of inline tasks (defined in BPEL4People) or standalone human tasks accessible as web services. In order to control the life cycle of service-enabled human tasks in an interoperable manner, WS-HumanTask also comes with a suitable coordination protocol for human tasks, which is supported by BPEL4People. The two specifications focus on the coordination logic only and do not support the design of the UIs for task execution.

The systematic development of web interfaces and applications has typically been addressed by the web engineering community by means of **model-driven web design approaches**. Among the most notable and advanced model-driven web engineering tools we find, for instance, WebRatio [11] and VisualWade [12]. The former is based on a web-specific visual modeling language (WebML), the latter on an object-oriented modeling notation (OO-H). Similar, but less advanced, modeling tools are also available for web modeling languages/methods like Hera, OOHDM, and UWE. These tools provide expert web programmers with modeling abstractions and automated code generation capabilities for complex web applications based on a hyperlink-based navigation paradigm. WebML has also been extended toward web services [13] and process-based web applications [14]; reuse is however limited to web services and UIs are generated out of HTML templates for individual components.

A first approach to component-based UI development is represented by **portals and portlets** [15], which explicitly distinguish between UI components (the portlets) and composite applications (the portals). Portlets are full-fledged, pluggable Web application components that generate document markup fragments (e.g., (X)HTML) that can however only be reached through the URL of the portal page. A portal server typically allows users to customize composite pages (e.g., to rearrange or show/hide portlets) and provides single sign-on and role-based personalization, but there is no possibility to specify process flows or web service interactions (the new WSRP [16] specification only provides support for accessing remote portlets as web services). Also **JavaServer Faces** [17] feature a component model for reusable UI components and support the definition of navigation flows; the technology is however hardly reusable in non-Java based web applications, navigation flows do not support flow controls, and there is no support for service orchestration and UI distribution.

Finally, the web mashup [2] phenomenon produced a set of so-called **mashup tools**, which aim at assisting mashup development by means of easy-to-use graphical user interfaces targeted also at non-professional programmers. For instance, Yahoo! Pipes (http://pipes.yahoo.com) focuses on data integration via RSS or Atom feeds via a data-flow composition language; UI integration is not supported. Microsoft Popfly (http://www.popfly.ms; discontinued since August 2009) provided a graphical user interface for the composition of both data access applications and UI components; service orchestration was not supported. JackBe Presto (http://www.jackbe.com) adopts a Pipes-like approach for data mashups and allows a portal-like aggregation of UI widgets (so-called mashlets) visualizing the output of such mashups; there is no

synchronization of UI widgets or process logic. IBM QEDWiki (http://services.alphaworks.ibm.com/qedwiki) provides a wiki-based (collaborative) mechanism to glue together JavaScript or PHP-based widgets; service composition is not supported. Intel Mash Maker (http://mashmaker.intel.com) features a browser plug-in which interprets annotations inside web pages allowing the personalization of web pages with UI widgets; service composition is outside the scope of Mash Maker.

In the mashArt [3] project, we worked on a so-called universal integration approach for UI components and data and application logic services. MashArt comes with a simple editor and a lightweight runtime environment running in the client browser and targets skilled web users. MashArt aims at simplicity: orchestration of distributed (i.e., multi-browser) applications, multiple actors, and complex features like transactions or exception handling are outside its scope. The CRUISe project [17] has similarities with mashArt, especially regarding the componentization of UIs. Yet, is does not support the seamless integration of UI components with service orchestration, i.e., there is no support for complex process logic. CRUISe rather focuses on adaptivity and context-awareness. Finally, the ServFace project [19] aims at supporting even unskilled web users in composing web services that come with an annotated WSDL description. Annotations are used to automatically generate form-like interfaces for the services, which can be placed onto one or more web pages and used to graphically specify data flows among the form fields. The result is a simple, user-driven web service orchestration. None of these projects, however, supports the coordination of multiple different actors inside a same process, and none of the approaches discussed in this section supports the distribution of UIs over multiple browsers.

3 Distributed User Interface Orchestration: Approach

If we analyze the home assistance scenario, we see that the envisioned application (as a whole) is *highly distributed* over the Web: The UIs for the actors participating in the application are composed of UI components, which can be components developed in-house (like the *Visit* component) or sourced from the Web (like the *Map* component); service orchestrations are based on web services. The UI exposes the state of the application and allows users to interact with it and to enact service calls. The two applications for the operator and the assistant are instantiated in different web browsers, contributing to the distribution of the overall UI and raising the need for synchronization.

The *key idea* to approach the coordination of (i) UI components inside web pages, (ii) web services providing data or application logic, and (iii) individual pages (as well as the people interacting with them) is to split the coordination problem into two layers: *intra-page UI synchronization* and *distributed UI synchronization and web service orchestration*.

We have seen that many of the research challenges raised by the home assistance application are not yet covered adequately by existing works. Especially the aim of providing a single development approach that is able to cover all development aspects in an integrated fashion poses requirements to the *whole life cycle* of UI orchestrations, especially in terms of design, deployment and execution support.

Indeed, supporting the *design* of distributed UI orchestrations such as the ones needed in the example scenario requires:

- Defining a new type of component, the **UI component**, which is able to modularize pieces of UI and to abstract their external interfaces in a way that conforms to the standard WSDL [4] format for service descriptions (to keep compatibility with the BPEL editors and language). We deal with the novel technological aspects introduced by UI components by defining a new type of WSDL binding, which allows us to specify how to translate the abstract WSDL operation descriptions into JavaScript function calls.
- Bringing together the needs of **UI synchronization and service orchestration** in one single language. UIs are typically event-based (e.g., user clicks or key strokes), while service invocations are coordinated via control flows. In this paper, we show how to extend the standard BPEL language in order to support UIs (BPEL comes with graphical editors and ready, off-the-shelf runtime engines that we want to reuse, not re-implement). We call this extended language *BPEL4UI*.
- Implementing a suitable, graphical **design environment** that allows developers to visually compose services and UI components and to define the grouping of UI components into pages. We achieve this by extending the Eclipse BPEL editor with UI-specific modeling constructs that are able to generate BPEL4UI in output.

Supporting the *deployment* of UI orchestrations requires:

- **Splitting the BPEL4UI specification** into the two orchestration layers for intra-page UI synchronization and distributed UI synchronization and web service orchestration. For the former we use a lightweight UI composition language (UICL), which allows specifying how UI components are coordinated in the client browser. For the latter we rely on standard BPEL.
- Providing a set of **auxiliary web services** that are able to mediate communications between the client-side UI composition logic and the BPEL logic. We achieve this layer by automatically generating and deploying a set of web services that manage the UI-to-BPEL and BPEL-to-UI interactions.

Supporting the *execution* of UI orchestrations requires:

- Providing a **client-side runtime framework** for UI synchronization that is able to instantiate UI components inside web pages and to propagate events from one component to other components, starting from a UICL specification. Events of a UI component may be propagated to components running in the same web page or in other pages of the application and to web services.
- Providing a **communication middleware layer** that is able to run the generated auxiliary web services for UI-to-BPEL and BPEL-to-UI communications. We implement this layer by reusing standard web server technology able to instantiate SOAP and RESTful web services.
- Setting up a **BPEL engine** that is able to run standard BPEL process specifications. The engine is in charge of orchestrating web services and distributed UI-UI communications. We rely on standard technology and reuse an existing BPEL engine.

These requirements and the respective hints to our solution show that the main methodological goals in achieving our UI orchestration approach are (i) relying as much as possible on existing *standards*, (ii) providing the developer with only *few and simple new concepts*, and (iii) implementing a runtime architecture that associates each concern to the *right level of abstraction and software tool* (e.g., UI synchronization is handled in the browser, while service orchestration is delegated to the BPEL engine).

4 The Building Blocks: Web Services and UI Components

Orchestrating remote application logic and pieces of UI requires, first of all, understanding the exact nature of the components to be integrated. For the integration of application logic, we rely on standard web service technologies, such as WSDL-SOAP services, i.e., remote web services whose external interface is described in WSDL, which supports interoperability via four message-based types of operations: *request-response*, *notification*, *one-way*, and *solicit-response*. Most of today's web services of this kind are *stateless*, meaning that the order of invocation of their operations does not influence the success of the interaction, while there are also *stateful* services whose interaction requires following a so-called business protocol that describes the interaction patterns supported by the service.

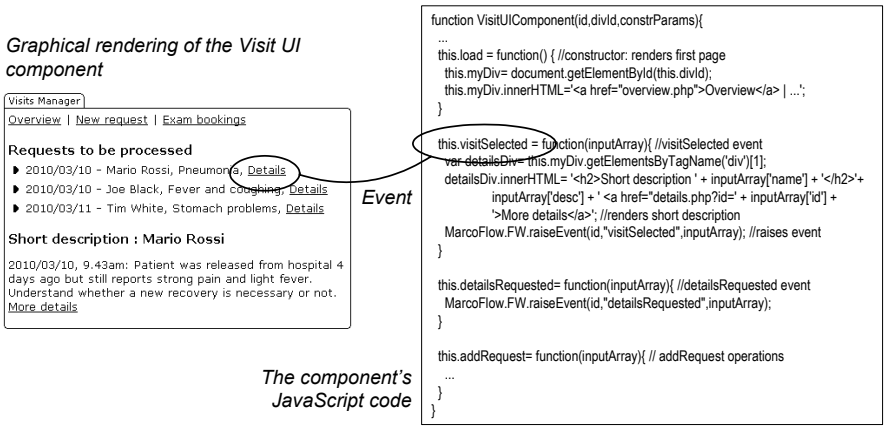


Figure 2 Graphical rendering and internal logic of a JavaScript/HTML UI component

For the integration of UI, we rely instead on JavaScript/HTML **UI components**, which are simple, stand-alone web applications that can be instantiated and run inside any common web browser. Figure 2 shows an example of UI component (the *Visit* UI component of our reference scenario), along with an excerpt of its JavaScript code. Unlike web services, UI components are characterized by:

– A **user interface**. UI components can be instantiated inside a web browser and can be accessed and navigated by a user via standard HTML. The UI allows the user to interactively inspect and alter the content of the component, e.g., the

short description in Figure 2. UI components are therefore *stateful*, and the component's navigation features replace the business protocol needed for services.

- **Events**. Interacting with the UI generates system events (e.g., mouse clicks) in the browser used to manage the update of contents. Some events may be exposed as component events in order to *communicate state changes*. For instance, a click on the *Details* link in Figure 2 launches a *visitSelected* event.
- **Operations**. Operations *enact state changes* from the outside. Typically, we can map the event of one component to the operation of another component in order to synchronize the components' state (so that they show related information).
- **Properties**. The graphical setup of a component may require the setting of *constructor parameters*, e.g., to align background colors or to specify the start page of a component.

In order to make UI components available in BPEL, each component is equipped with a standard WSDL descriptor that describes the events and operations (the constructor is expressed as operation) in terms of one-way and notification WSDL operations, respectively. To support the instantiation and execution of components, we have defined a new *JavaScript binding* for WSDL, which binds the abstract operations to the JavaScript functions of the component. The WSDL-UI descriptor can be used as is by the client-side runtime framework and adapted for its use by the BPEL engine.

5 Modeling UI Orchestrations

Specifying a UI orchestration requires modeling two fundamental aspects: (i) the *interaction logic* that rules the passing of data among UI components and web services and (ii) the *graphical layout* of the final application. Supporting these tasks in BPEL requires extending the expressive power of the language with UI-specific constructs.

5.1 BPEL4UI: concepts and syntax

Figure 3 shows the simplified meta-model of BPEL4UI. Specifically, the figure details all the new modeling constructs necessary to specify UI orchestrations (gray-shaded) and omits details of the standard BPEL language, which are reused as is by BPEL4UI (a detailed meta-model for BPEL can be found in [20]).

In terms of standard BPEL [5], a UI orchestration is a *process* that is composed of a set of associated *activities* (e.g., sequence, flow, if, assign, validate, or similar), *variables* (to store intermediate processing results), *message exchanges*, *correlation sets* (to correlate messages in conversations), and *fault handlers*. The services or UI components integrated by a process are declared by means of so-called *partner links*, while *partner link types* define the roles played by each of the services or UI components in the conversation and the *port types* specifying the operations and messages supported by each service or component. There can be multiple partner links for each partner link type.

Modeling UI-specific aspects requires instead introducing a set of new constructs that are not yet supported by BPEL. The constructs, illustrated in Figure 3, are:

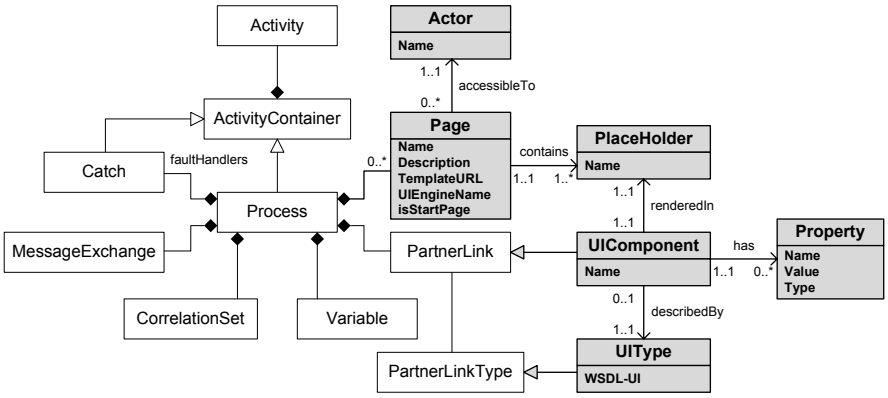


Figure 3 Simplified BPEL4UI meta-model in UML. White classes correspond to standard BPEL constructs [20]; gray classes correspond to constructs for UI and user management.

- **UI type**: The use of UI components in service compositions asks for a new kind of partner link type. Although syntactically there is no difference between web services and UI components (the JavaScript binding introduced into WSDL-UI comes into play only at runtime), it is important to distinguish between services and UI components as their semantics and, hence, their usage in the model will be different. Also, it is necessary to mark UI component types as such, in order to support the generation of standard BPEL, as described in Section 6.
  As exemplified in Figure 4, we specify the new partner link type like a standard web service type (lines 10-13). In order to reflect the events and operations of the UI component, we distinguish the two roles. Lines 1-8 define the necessary name spaces and import the WSDL-UI descriptor of the UI component.
- **Page**: The distributed UI of the overall application consists of one or more web pages, which can host instances of UI components. Pages have a *name*, a *description*, a reference to the pages' *layout template*, the name of the *UI engine* (see Section 6) they will run on, and an indication of whether they are a *start page* of the application or not (similar to the start activity in process models).
  The code lines 16-21 in Figure 4 show the definition of a page called "Operator", along with its layout template and the name of the UI engine on which the page will be deployed; the page is a start page for the process.
- **Place holder**: Each page comes with a set of place holders, which are empty areas inside the layout template that can be used for the graphical rendering of UI components. Place holders are identified by a unique *name*, which can be used to associate UI components.
  Place holders are associated with page definitions and specified as sub-elements, as shown in lines 19-20 in Figure 4.
- **UI component**: UI types can be instantiated as UI components. For instance, there might be one UI type but two different instances of the type running in two different web pages. Declaring a UI component in a BPEL4UI model leads to the creation of an instance of the UI component in one of the pages of the application. Each component is part of one process and has a unique *name*.

We specify UI component partner links by extending the standard partner link definition of BPEL with three new attributes, i.e., *isUiComponent*, *pageName* and *placeHolderName*. Lines 25-31 in Figure 4 show how to declare the *Visit UI component* of our example scenario.

- **Property**: As we have seen in the previous section, UI components may have a constructor that allows one to set configuration properties. Therefore, each UI component may have a set of associated properties than can be parsed at instantiation time of the component. We use simple *name-value* pairs to store constructor parameters.
  Properties extend the definition of UI component link types by adding *property* sub-elements to the partner link definition, one for each constructor parameter, as shown in lines 29-30 in Figure 4.
- **Actor**: In order to coordinate the people in a process, pages of the application can be associated with individual actors, i.e., humans, which are then allowed to access the page and to interact with the UI orchestration via the UI components rendered in the page. As for now, we simply associate static actors to pages (using their *names*); yet, actors can easily be assigned also dynamically at deployment time or runtime by associating roles instead of actors and using a suitable user management system.
  Actors are added to page definitions by means of the *actorName* attribute, as highlighted in line 18 in Figure 4.

```
1  <bpel:process name="HomeAssistance"
2    targetNamespace=www.unitn.it/bpel4ui/HomeAssistance
3    xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/
4    executable" xmlns:visit="http://www.unitnt.it/UI/VisitUIComponent" ...>
5
6  <bpel:import namespace="http://www.unitn.it/UI/VisitUIComponent"
7    location="VisitUI.wsdl" importType="http://schemas.xmlsoap.org/wsdl/">
8  </bpel:import>
9  ...
10 <bpel:partnerLinkType name="VisitUIComponent">
11   <bpel:role name="Receive" portType="visit:VisitUI_RECEIVE"/>
12   <bpel:role name="Invoke" portType="visit:VisitUI_INVOKE"/>
13 </bpel:partnerLinkType>
14 ...
15 <pages>
16   <page name="Operator" description="Operator's home page"
17     templateURL="http://www.unitn.it/BPEL4UI/operatorLayout.html"
18     uiEngineName="UNITN" isStartPage="yes" actorName="Paul">
19       <placeholder name="marcoflow-left"/>
20       <placeholder name="marcoflow-right"/>
21   </page>
22   ...
23 </pages>
24 <bpel:partnerLinks>
25   <bpel:partnerLink name="VisitUI_Operator"
26     partnerLinkType="VisitUIComponent" myRole="Receive"
27     partnerRole="Invoke" isUiComponent="yes" pageName="Patient"
28     placeHolderName="marcoflow-left">
29       <property name="StartPage" type="xsd:string">New Visit</property>
30       <property name="BackgroundColor" type="xsd:string">white</property>
31   </bpel:partnerLink>
32 </bpel:partnerLinks>
33 ...
34 </bpel:process>
```

Figure 4 Excerpt of the BPEL4UI home assistance process (new constructs in bold)

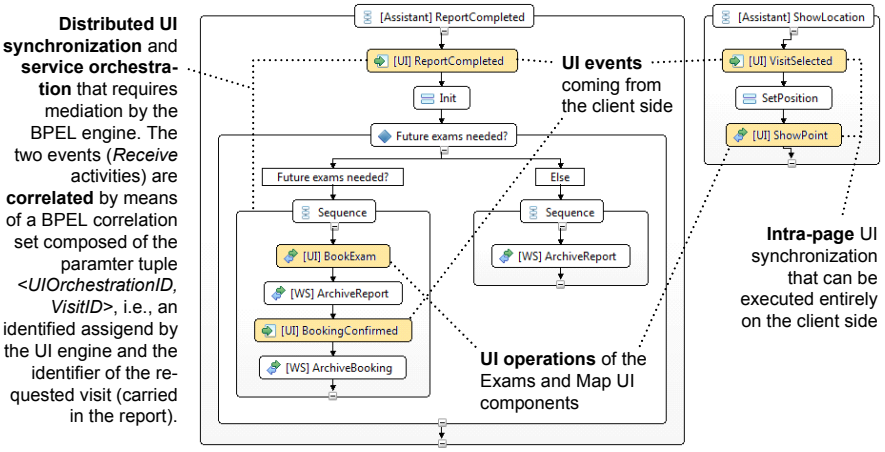


Figure 5 Part of the BPEL4UI model of the home assistance process modeled in the extended Eclipse BPEL editor (these and other *Sequence* constructs run inside a *Flow*).

5.2 Modeling the orchestration logic

The code example in Figure 4 shows that the UI-specific modeling constructs have a very limited impact on the syntax of BPEL and are concerned with the abstract specification of the layout and the declaration of UI partner links. The actual composition logic relies exclusively on standard BPEL constructs, yet – since UI components are different from web services (e.g., it is important to know in which page they are running) – it is important to understand the effect individual modeling patterns have on the execution of the final application, i.e., the *semantics* of the patterns. As hinted at in Section 3 and illustrated in Figure 5, we distinguish three main design patterns:

- **Intra-page UI synchronization**: The sequence construct in the right part of Figure 5 shows the internals of the *View instructions* task in Figure 1. When the assistant clicks on a visit request, the patient's address is shown on the Google map. In BPEL terms, we receive a message from the *Visit* UI component (the event) and forward it to the operation of the *Map* component, implementing an intra-page UI synchronization. Both UI components involved in the sequence are associated with the page of the assistant. Hence, this kind of UI synchronization can be performed on the *client side* without involving the BPEL engine.
- **Distributed UI synchronization**: The sequence construct in the left part of the figure, instead, contains a distributed synchronization that cannot be executed on the client only, as the two UI components involved in the communication (*Report* and *Exam*) run in different web pages. The event generated upon submission of a new report is processed by the *BPEL engine*, which then decides whether an additional exam needs to be booked by the operator or not.
- **Service orchestration**: The distributed UI synchronization also involves the orchestration of the *Report archiving* and *Booking archiving* web services, as well as some BPEL flow control constructs. For instance, the modeled logic checks whether the report expresses the need for further exams or not. In either case, the

further processing of the report involves the invocation of either one or both the web services, in order to correctly terminate the handling of a visit request.

The BPEL4UI excerpt in Figure 5 shows that, when modeling a UI orchestration, it is important to keep in mind who communicates with whom and where UI components will be rendered. Depending on these two considerations, the modeled composition logic will either be executed on the client side, in the BPEL engine, or in both layers. For instance, it suffices to associate the *Map* component with a different page so as to turn the intra-page UI synchronization in the right hand side of Figure 5 into a distributed communication and, hence, to require support from the BPEL engine.

Data transformations. When composing services or UI components, it is not enough to model the communication flow only. An important and time-consuming aspect is that of transforming the data passed from one component to another. With BPEL4UI we support all data transformation features provided by BPEL by means of its *Assign* activity. This allows us to leverage on technologies, such as XPath, XQuery, XSLT or Java, for the implementation of also very complex data transformations. Yet, the type of data transformation may affect the logic of the UI orchestration. For instance, if the *SetPosition* activity in Figure 5 does not transform data at all or only performs simple parameter mappings (with the BPEL *Copy* construct) the intra-page UI synchronization can be executed in the client browser. If instead a more complex transformation is needed, we rely on the BPEL engine to perform it.

The reason for this choice is that UI synchronization typically involves exchanging only simple data (e.g., parameter-value pairs) and does not require complex transformations like when interacting with web services. This choice allows us to keep the client-side framework as lightweight as possible, while not giving up any data transformation capabilities. The decision of where to transform data is taken based on the nature of the involved partner links and the type of transformation.

Correlation. The intra-page UI synchronization in Figure 5 does not involve any *asynchronous* communication pattern or multiple entry points into the process logic. It is therefore not necessary to implement any correlation logic in BPEL4UI in order to propagate the *VisitSelected* event to the *ShowPoint* operation. The correlation of the event and the operation in the two web pages is achieved outside the BPEL engine (in the *UI engine server* in Figure 6) by sharing a common key (the *UIOrchestrationID*) that is carried by each event and used to dispatch events. This kind of correlation is automated in our runtime environment and does not require specific modeling.

The distributed UI synchronization, instead, involves two UI events from two different actors: *ReportCompleted* and *BookingConfirmed*. In this case, it is necessary to configure a so-called *correlation set* (in BPEL terminology) that allows the BPEL engine to understand whether they belong to the same process instance or not. In Figure 5, we use *UIOrchestrationID* and *VisitID* (part of the report) as correlation set.

Graphical layout. Defining web pages and associating UI partner links with place holders therein requires implementing suitable HTML templates that are able to host UI components. As we focus on the middleware layer for UI orchestrations, for the layout templates we rely on standard web design instruments and technologies. The only requirement the templates must satisfy is that they provide place holders in form of HTML DIV elements that can be indexed via standard HTML identifiers following a predefined naming convention: * … *.

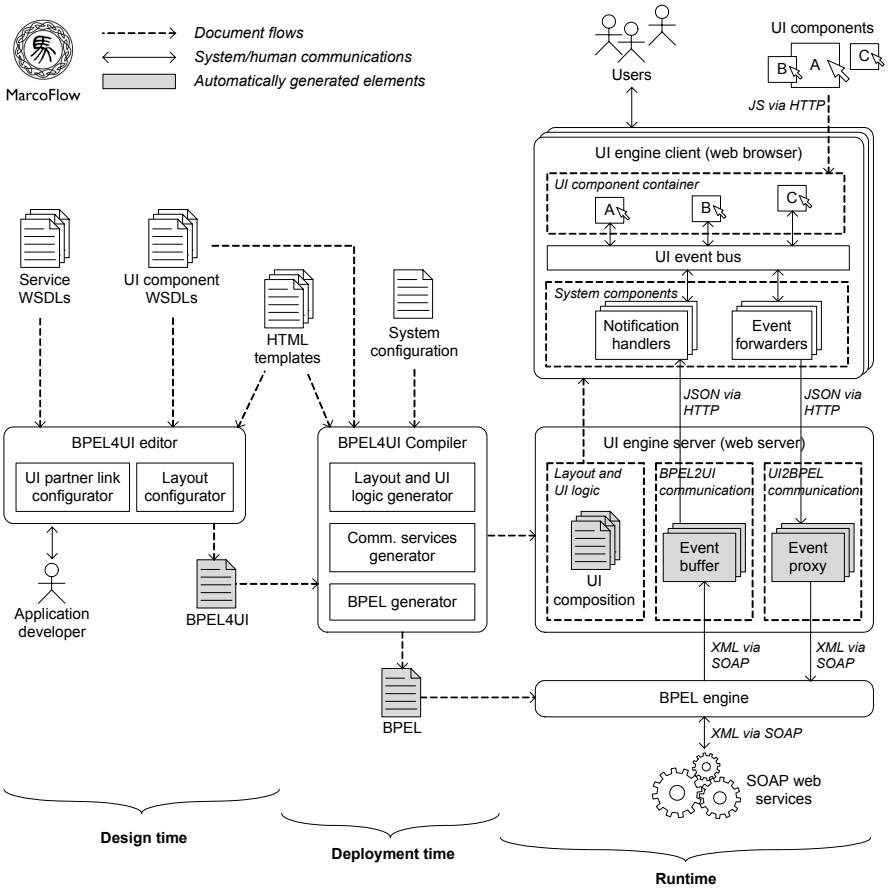


Figure 6 From design time to runtime: overall system architecture of MarcoFlow.

6 Deploying and Running UI Orchestrations

The BPEL4UI language is only a piece of the integrated system for UI orchestration, called *MarcoFlow*. The overall architecture of the system is shown in Figure 6 (for presentation purposes, we discuss a slightly simplified version), which partitions its software components into design time, deployment time, and runtime components.

The **design** part comprises the *BPEL4UI editor* with its *UI partner link configurator* and *layout configurator*. Starting from a set of *web service WSDLs*, *UI component WSDLs*, and *HTML templates* the application developer graphically models the UI orchestration, and the editor generates a corresponding *BPEL4UI* specification in output. The composition logic in Figure 5 has been modeled in our BPEL4UI editor, an extended Eclipse BPEL editor with (i) a panel for the specification of the pages in which UI components can be rendered and (ii) a property panel that allows the devel-

oper to configure the web pages, to set the properties of UI partner links, and to associate them to place holders in the layout.

The **deployment** of a UI orchestration requires translating the BPEL4UI specification, which is not immediately executable neither by a standard BPEL engine nor by the UI rendering engine (the so-called *UI engine*, which we discuss in the following), into executable formats. This task is achieved by the *BPEL4UI compiler*, which, starting from the *BPEL4UI* specification, the set of used *HTML templates* and *UI component WSDLs*, and the *system configuration* of the runtime part of the architecture, generates three kinds of outputs:

1. A set of *communication channels* (to be deployed in the so-called *UI engine server*), which mediate between the *UI engine client* (the client browser) and the *BPEL engine*. These channels are crucial in that they resolve the technology conflict inherently present in BPEL4UI specifications: a BPEL engine is not able to talk to JavaScript UI components running inside a client browser, and UI components are not able to interact with the SOAP interface of a BPEL engine. For each UI component in a page, the compiler therefore generates (i) an *event proxy* that is able to forward events from the client browser to the BPEL engine and (ii) an *event buffer* that is able to accept events from the BPEL engine and stores them on behalf of the *UI engine client*.
2. A *standard BPEL* specification containing the distributed UI synchronization and web service orchestration logic. Unlike the BPEL4UI specification, the generated BPEL specification does no longer contain any of the UI-specific constructs introduced in Section 5.1 and can therefore be executed by any standards-compliant BPEL engine. This means that all references to UI component partner links in input to the compilation are rewritten into references to the respective communication channels of the UI components in the *UI engine server*, also setting the correct, new SOAP endpoints.
3. A set of *UI compositions* (one for each page of the application) consisting of the layout of the page, the list of UI components of the page, the assignment of UI components to place holders, the specification of the intra-page UI synchronization logic, and a reference to the client-side runtime framework. Interactions with web services or UI components running in other pages are translated into interactions with local system components (the *notification handlers* and *event forwarders*), which manage the necessary interaction with the *communication channels* via suitable RESTful web service calls.

Finally, the *BPEL4UI compiler* also manages the deployment of the generated artifacts in the respective runtime environments. Specifically, the generated *communication channels* and the UI compositions are deployed in the *UI engine server* and the standard *BPEL specification* is deployed in the *BPEL engine*.

The **execution** of a UI orchestration requires the setting up and coordination of three independent runtime environments: First, the interaction with the users is managed in the client browser by an event-based JavaScript runtime framework that is able to parse the UI composition stored in the UI engine server, to instantiate UI components in their respective place holders, to configure the *notification handlers* and *event forwarders*, and to set up the necessary publish-subscribe logic ruling the event-to-operation mapping of the components running inside the client browser. While

event forwarders are called each time an event is to be sent from the client to the BPEL engine, the *notification handlers* are active components that periodically poll the event buffers of their UI components on the *UI engine server* in order to fetch possible events coming from the *BPEL engine* (we are currently studying suitable push mechanisms for events).

Second, the *UI engine server* must run the web services implementing the communication channels. In practice we generate standard Java servlets and SOAP web services, which can easily be deployed in a common web server, such as Apache Tomcat. The use of a web server is mandatory in that we need to be able to accept notifications from the BPEL engine and the UI engine client, which requires the ability of constant listening. The event buffer is implemented via a simple relational database (in PostgreSQL) that manages multiple UI components and distinguishes between instances of UI orchestrations by means of a session key that is shared among all UI components participating in a same UI orchestration instance.

Third, running the BPEL process requires a *BPEL engine*. Our choice to rely on standard BPEL allows us to reuse a common engine without the need for any UI-specific extensions. In our case, we use Apache ODE, which is characterized by a simple deployment procedure for BPEL processes.

The MarcoFlow system shown in Figure 6 is fully implemented and running. A demo of the tool is available at *http://mashart.org/marcoflow/demo.htm*.

7 Conclusion

The spectrum of applications whose design intrinsically depends on a structured flow of activities, tasks or capabilities is large, but current workflow or business process management software is not able to cater for all of them. Especially lightweight, component-based applications or Web 2.0 based, mashup-like applications typically do not justify the investment in complex process support systems, either because their user basis is too small or because there is need only for few, simple applications. Yet, these applications too demand for abstractions and tools that are able to speed up their development, especially in the context of the Web with its fast development cycles.

We introduced the idea of *distributed UI orchestration*, a component-based development technique that introduces a new first-class concept into the workflow management and service composition world, i.e., UIs, and that fits the needs of many of today's web applications. We proposed a model for UI components and showed how their use requires extending the expressive power of standard service composition languages. The language comes with a suitable modeling environment and a code generator able to produce code and instructions that can be executed straightaway by our runtime environment, which separates the problem of intra-page UI synchronization from that of distributed UI synchronization and service orchestration. The result is an approach to distributed UI orchestration that is comprehensive and free.

Unlike in our research on universal composition [3] and unlike mashup tools, in this paper we do not aim at enabling less skilled web users to develop simple applications. MarcoFlow targets skilled web developers that are familiar with BPEL and applications that are complex and possibly involve multiple actors that are distributed

over the Web, but that need orchestration. While the idea of event-based UI components has been around for some time now, distributed UI orchestration and multi-browser/multi-actor applications as proposed in this paper are new.

Next, we plan to support the *dynamic selection of actors* (during deployment or at runtime), advanced *access policies*, and *data flow* mechanisms that go beyond the current event-based communication (e.g., through a suitable persistence layer).

References

1. B.A. Myers, M.B. Rosson. Survey on user interface programming. *SIGCHI'92*, pp. 195-202.
2. J. Yu, B. Benatallah, F. Casati, F. Daniel. Understanding Mashup Development and its Differences with Traditional Integration. *IEEE Internet Computing*, Vol. 12, No. 5, September-October 2008, pp. 44-52.
3. F. Daniel, F. Casati, B. Benatallah, M.-C. Shan. Hosted Universal Composition: Models, Languages and Infrastructure in mashArt. *ER'09*, pp. 428-443.
4. E. Christensen, F. Curbera, G. Meredith, S. Weerawarana. Web Services Description Language (WSDL) 1.1. W3C Note, March 2001. [Online] *http://www.w3.org/TR/wsdl*
5. OASIS. Web Services Business Process Execution Language Version 2.0, April 2007. [Online]. *http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html*
6. C. Pautasso. BPEL for REST. *BPM'08*, Milano, pp. 278-293.
7. T. van Lessen, F. Leymann, R. Mietzner, J. Nitzsche, D. Schleicher. A Management Framework for WS-BPEL, *ECoWS'08*, Dublin, pp. 187-196.
8. E. M. Maximilien, A. Ranabahu, K. Gomadam. An Online Platform for Web APIs and Service Mashups, *Internet Computing*, vol. 12, no. 5, pp. 32-43, Sep. 2008.
9. Active Endpoints, Adobe, BEA, IBM, Oracle, SAP. WS-BPEL Extension for People (BPEL4People), Version 1.0. June 2007.
10. Active Endpoints, Adobe, BEA, IBM, Oracle, SAP. Web Services Human Task (WS-HumanTask), Version 1.0. June 2007.
11. R. Acerbis, A. Bongio, M. Brambilla, S. Butti, S. Ceri, P. Fraternali. Web Applications Design and Development with WebML and WebRatio 5.0. *TOOLS'08*, pp. 392-411.
12. J. Gómez, A. Bia, A. Parraga. Tool Support for Model-Driven Development of Web Applications, *WISE'05*, pp. 721-730.
13. I. Manolescu, M. Brambilla, S. Ceri, S. Comai, P. Fraternali. Model-Driven Design and Deployment of Service-Enabled Web Applications. *ACM Trans. Internet Technol.*, Vol. 5, No. 3, August 2005, pp. 439-479.
14. M. Brambilla, S. Ceri, P. Fraternali, I. Manolescu. Process Modeling in Web Applications. *ACM Trans. Softw. Eng. Methodol.*, Vol. 15, No. 4, October 2006, pp. 360-409.
15. Sun Microsystems. JSR-000168 Portlet Specification, October 2003. [Online]. *http://jcp.org/aboutJava/communityprocess/final/jsr168/*
16. OASIS. Web Services for Remote Portlets, August 2003. [Online]. *www.oasis-open.org/committees/wsrp*
17. Oracle. JavaServer Faces Technology. [Online] *http://java.sun.com/javaee/javaserverfaces/*
18. S. Pietschmann, M. Voigt, A. Rümpel, K. Meissner. CRUISe: Composition of Rich User Interface Services. *ICWE'09*, pp. 473-476.
19. M. Feldmann, T. Nestler, U. Jugel, K. Muthmann, G. Hübsch, A. Schill. Overview of an end user enabled model-driven development approach for interactive applications based on annotated services. *WEWST'09*, pp. 19-28.
20. WSPER.org. WS-BPEL 2.0 Metamodel. [Online] *http://www.ebpml.org/wsper/wsper/ws-bpel20.html*

Appendix E

MarcoFlow: Modeling, Deploying, and Running Distributed User Interface Orchestrations

(Demonstration Proposal)

Florian Daniel, Stefano Soi, Stefano Tranquillini, Fabio Casati
University of Trento, Povo (TN), Italy
{daniel,soi,tranquillini,casati}@disi.unitn.it

Chang Heng, Li Yan
Huawei Technologies, Shenzhen, P.R. China
{changheng,liyanmr}@huawei.com

Abstract. This demo introduces the idea of *distributed orchestration of user interfaces* (UIs), an application development approach that allows us to easily bring together UIs, web services and people in a single orchestration logic, language, and tool. The tool is called *MarcoFlow*, and it covers three main phases of the software development lifecycle: design (by means of a dedicated, visual editor), deployment (by means of a set of code generators), and execution (by means of a distributed runtime environment for UI orchestrations). We showcase the benefits of MarcoFlow in each of the phases by developing and running a practical and expressive application for the management of home assistance and by explaining, in each phase, which are the challenges, which the intuitions, and which the solutions. The demo targets the development of *mashup-like applications that require (distributed) process support* and, hence, targets researchers and practitioners interested in mashups, lightweight process design, web services, and innovative (and free) ways of providing process support.

1 Introduction

After workflow management (which supports the automation of business processes and human tasks) and service orchestration (which focuses on web services at the application layer), web mashups [1] feature a significant innovation: *integration at the UI level*. Besides web services or data feeds, mashups indeed reuse pieces of UIs (e.g., content extracted from web pages or JavaScript UI widgets) and integrate them into new web pages or applications. While mashups therefore manifest the need for reuse in UI development and for suitable UI component technologies, so far they only produced rather simple applications, most of the times consisting of only one web page and of little utility.

This demo complements the concepts introduced in [2], where we argue that there is a huge spectrum of applications that demand for development approaches that are similar to those of mashups but that go far beyond single page applications and in fact support multiple pages, multiple actors, complex navigation structures, and – more importantly – process-based application logic or navigation flows.

As of today, there is no single development instrument that allows one to develop this kind of applications using one language and one environment only. Filling this gap is the goal of MarcoFlow.

2 Demo scenario

In this demo, we want to develop an application that supports the scenario graphically described in Figure 1: A *patient* can ask for the visit of a home assistant (e.g., a paramedic) by calling (via phone) an operator of the home assistance service. Upon request, the *operator* inputs the respective details into his management console and inspects the patient's data and personal health history in order to provide the assistant with the necessary instructions to assist the patient. The *home assistant* views the description in his own application, visits the patient, and files a report about the provided service. The report is processed by the *back-end system* and archived. If no further exams are needed, the UI orchestration ends. If exams are instead needed, the respective details need to be sent to the operator so that he can book the exam in the local hospital, asking the patient for confirmation (again via phone). Upon confirmation of the exam booking, the system also archives the booking, which terminates the responsibility of the home assistance service.

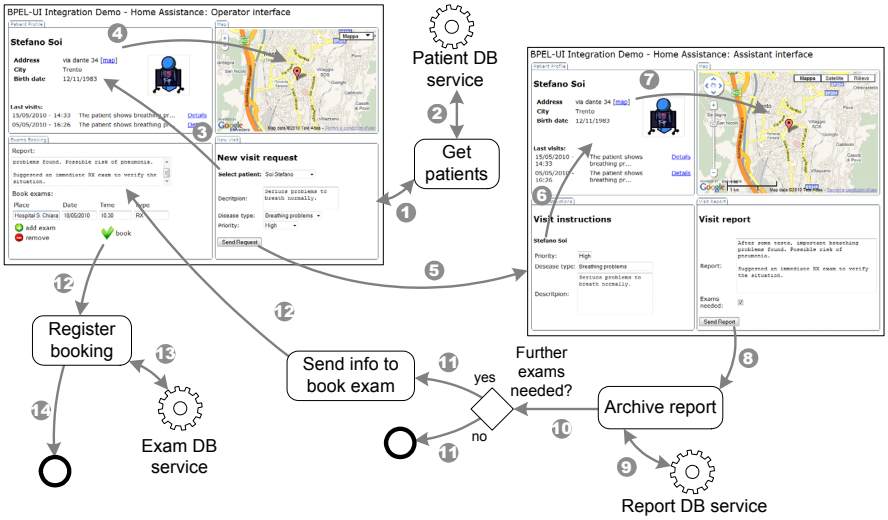


Figure 1 The distributed UI orchestration to be developed during the demo. The gray arrows indicate synchronization or orchestration points; they number labels indicate their order in time.

Our goal is to develop an application that supports this process. This application includes, besides the process logic, two mashup-like, web-based control consoles for the operator and the assistant that are themselves part of the orchestration and need to interact with (and are affected by) the evolution of the process. Furthermore, the UI is

itself component-based and created by reusing and combining existing UI components (each of the two pages in Figure 1 is, for instance, composed of four UI components). For each new request, the operator starts a new instance of the application, raising the need for correlation of services and UI components.

If we analyze the scenario, we see that the envisioned application (as a whole) is *highly distributed* over the Web: The UIs for the actors participating in the application are composed of UI components, which can be components developed in-house (like the *New Request* component) or sourced from the Web (like the *Map* component); service orchestrations are based on web services. The UI exposes the state of the application and allows users to interact with it and to enact service calls. The two applications for the operator and the assistant are instantiated in different web browsers, contributing to the distribution of the overall UI and raising the need for synchronization.

3 MarcoFlow: An Environment of Distributed UI Orchestration

The *key idea* to approach the coordination of (i) UI components inside web pages, (ii) web services providing data or application logic, and (iii) individual pages (as well as the people interacting with them) is to split the coordination problem into two layers: *intra-page UI synchronization* and *distributed UI synchronization and web service orchestration*.

Figure 2 shows the (simplified) architecture of the MarcoFlow environment, which aids the development and execution of the demo scenario. The architecture is partitioned into design time, deployment time, and runtime components, according to the three phases of the software development lifecycle supported by MarcoFlow.

The **design** part comprises the *BPEL4UI editor* that supports BPEL4UI [2], the composition language we use to specify distributed UI orchestrations. The editor is an extended Eclipse BPEL editor with (i) a panel for the specification of the pages in which UI components can be rendered and (ii) a property panel that allows the developer to configure the web pages, to set the properties of UI partner links, and to associate them to place holders in the layout.

The **deployment** of a UI orchestration requires translating the BPEL4UI specification into executable components: (i) a set of *communication channels* that mediate between the UI components in the client browser and the BPEL engine; (ii) a *standard BPEL specification* containing the distributed UI synchronization and web service orchestration logic; and (iii) a set of *UI compositions* (one for each page of the application) containing the intra-page UI synchronizations. This task is achieved by the *BPEL4UI compiler*, which also manages the deployment of the generated artifacts in the respective runtime environments.

The **execution** of a UI orchestration requires the setup and coordination of three independent runtime environments: (i) the interaction with users and intra-page UI synchronization is managed in the client browser by an *event-based JavaScript runtime framework*; (ii) a so-called *UI engine server* runs the web services implementing the communication channels; and (iii) a *standard BPEL engine* manages the distributed UI synchronization and web service orchestration.

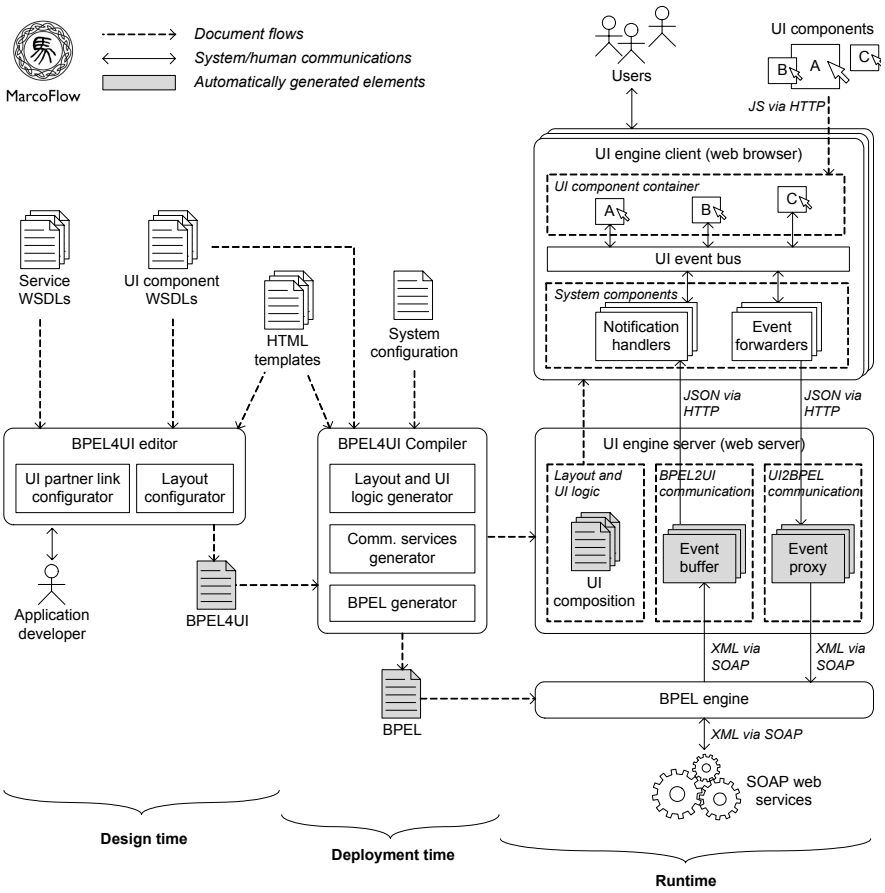


Figure 2 From design time to runtime: overall system architecture of MarcoFlow

The MarcoFlow system shown in Figure 2 is fully implemented and running. A patent application for parts of the system has been filed.

4 Demonstration Script

During the demo we intend to develop the UI orchestration described in Section 2 from scratch and run it live, while interacting with the audience and answering questions and clarifying details. The planned demo flow is as followed:

1. *Goals of MarcoFlow*: explanation of the motivation that led us to the development of the distributed UI orchestration approach and of the goals that we pursued in doing so.

2. *The home assistance scenario*: explanation of the scenario described in Section 2, the development challenges it implies, and the basic assumptions we have regarding people, user interfaces, and web services.
3. *The MarcoFlow approach*: explain the key idea behind the MarcoFlow approach to UI orchestration, i.e., splitting the coordination problem into intra-page UI synchronization and distributed UI synchronization and web service orchestration.
4. *System architecture*: description of the architecture and the internals of MarcoFlow. The description highlights which components support which phase in the development cycle: design, deployment, and runtime.
5. *The basic components*: here we explain how the UI components and web services we compose look like and how they can be abstractly described in a homogenous fashion, using standard WSDL and a new JavaScript binding for UI components.
6. *Designing distributed UI orchestrations*: explanation and live demo of the visual, BPEL4UI editor. The goal here is to develop step by step the UI orchestration of the reference scenario and to highlight modeling conventions and patterns.
7. *Code generation and deployment*: as highlighted when explaining the system architecture, MarcoFlow is a complex system in which many of the components are generated automatically starting from a BPEL4UI specification. Here we explain which components are created, how, and where they are deployed, in order to provide for the necessary execution support.
8. *Running the application*: after the deployment, the application is ready for execution. Running an application is supported by a simple web console for users that allows them to easily start UI orchestrations (the operator) and to participate in already running UI orchestrations (the assistant). Here we show how users can interact with their individual web pages and how these and the UI components inside the pages are orchestrated as part of the overall process logic. Special emphasis is given to the distinction of the two actors, i.e., the operator and the home assistant.
9. *Conclusion*: finally, we conclude the presentation with a recap of the demonstration and the MarcoFlow system and we outline our ideas for future works.

A demo of the tool is available at *http://mashart.org/marcoflow/demo.htm*

References

1. J. Yu, B. Benatallah, F. Casati, F. Daniel. Understanding Mashup Development and its Differences with Traditional Integration. *IEEE Internet Computing*, Vol. 12, No. 5, September-October 2008, pp. 44-52.
2. F. Daniel, S. Soi, S. Tranquillini, F. Casati, C. Heng, L. Yan. From People to Services to UI: Distributed Orchestration of User Interfaces. *BPM'10*, Hoboken, NJ, USA.

MarcoFlow: Modeling, Deploying, and Running Distributed User Interface Orchestrations

(Demonstration Paper)

Florian Daniel, Stefano Soi, Stefano Tranquillini, Fabio Casati
University of Trento, Povo (TN), Italy
{daniel,soi,tranquillini,casati}@disi.unitn.it

Chang Heng, Li Yan
Huawei Technologies, Shenzhen, P.R. China
{changheng,liyanmr}@huawei.com

Abstract. This demo introduces the idea of *distributed orchestration of user interfaces* (UIs), an application development approach that allows us to easily bring together UIs, web services, and people in a single orchestration logic, language, and tool. The tool is called *MarcoFlow*, and it covers three main phases of the software development lifecycle: design (by means of a dedicated, visual editor), deployment (by means of a set of code generators), and execution (by means of a distributed runtime environment for UI orchestrations). MarcoFlow targets the development of *mashup-like applications that require (distributed) process support* and, hence, targets researchers and practitioners interested in mashups, lightweight process design, web services, and innovative (and free) ways of providing process support.

1 Introduction

After workflow management (which supports the automation of business processes and human tasks) and service orchestration (which focuses on web services at the application layer), web mashups [1] feature a significant innovation: **integration at the UI level**. Besides web services or data feeds, mashups indeed reuse pieces of UIs (e.g., content extracted from web pages or JavaScript UI widgets) and integrate them into new web pages or applications. While mashups therefore manifest the need for reuse in UI development and for suitable UI component technologies, so far they produced rather simple applications consisting of one web page and of little utility.

This demo complements the concepts and solutions introduced in [2], where we argue that there is a huge spectrum of applications that demand for development approaches that are similar to those of mashups but that go far beyond single page applications and in fact support multiple pages, multiple actors, complex navigation structures, and – more importantly – process-based application logic or navigation flows. We call this type of applications **distributed UI orchestrations**, as (i) both components and the application itself may be distributed over the Web, (ii) in addition to traditional web services we also integrate novel JavaScript UI components, and (iii) services and UIs are orchestrated in an integrated fashion.

Challenges and contributions. Developing distributed UI orchestrations implies the coordination of individual actors and the development of a *distributed* user interface *and* service orchestration logic. Doing so requires (i) understanding how to *componentize UIs and compose them into web applications*, (ii) defining a logic that is able to *orchestrate both UIs and web services*, (iii) providing a language and tool for *specifying distributed UI compositions*, and (iv) developing a runtime environment that is able to *execute distributed UI and service compositions*.

Innovativeness of the tool. As of today, there is no single development instrument that answers these challenges and allows one to develop UI orchestrations using one language and one environment only. *BPEL* [3] focuses on web services only. *BPEL4People* [4] adds human tasks and actors as first-class concepts, but without supporting the development of suitable UIs. *Model-driven web design instruments*, such as WebRatio [5] or VisualWade [6], allow the development of advanced web applications, without however facilitating reuse of UI components sourced from the Web. *Portals and portlets* [7], instead, focus specifically on reuse, but they fail in supporting service integration and process flows. *Mashup tools* [1] support the integration of UIs and of services, but they typically do not support complex orchestration patterns (if not hand-coded). In mashArt [8], we did some first steps into that direction, but without considering multi-user and distributed UI support.

Significance to the BPM field. With MarcoFlow, we go one step beyond state-of-the-art BPM and service composition and propose an original model, language and running system for the composition of distributed UIs. The approach brings together UIs, web services and people in a single orchestration logic and tool and supports the development of mashup-like applications that require (distributed) process support, a kind of application that so far was not supported by BPM practices and software.

2 Distributed UI Orchestration

The **key idea** to approach the coordination of (i) UI components inside web pages, (ii) web services providing data or application logic, and (iii) individual pages (as well as the people interacting with them) is to split the coordination problem into two layers: *intra-page UI synchronization* and *distributed UI synchronization and web service orchestration*. UIs are typically event-based (e.g., user clicks or key strokes), while service invocations are coordinated via control flows. In this demo and in [2], we show how to describe UI components in terms of standard WSDL descriptors, how to bind them to JavaScript, and how to extend the standard BPEL language in order to support the two above composition layers. We call this extended language *BPEL4UI*.

Figure 1 shows the simplified meta-model of **BPEL4UI**. Specifically, the figure details all the new modeling constructs necessary to specify UI orchestrations (gray-shaded) and omits details of the standard BPEL language, which are reused as is by BPEL4UI. In terms of standard BPEL [3], a UI orchestration is a *process* that is composed of a set of associated *activities* (e.g., sequence, flow, if, assign, validate, or similar), *variables* (to store intermediate processing results), *message exchanges*, *correlation sets* (to correlate messages in conversations), and *fault handlers*. The services or UI components integrated by a process are declared by means of so-called

partner links, while *partner link types* define the roles played by each of the services or UI components in the conversation and the *port types* specifying the operations and messages supported by each service or component.

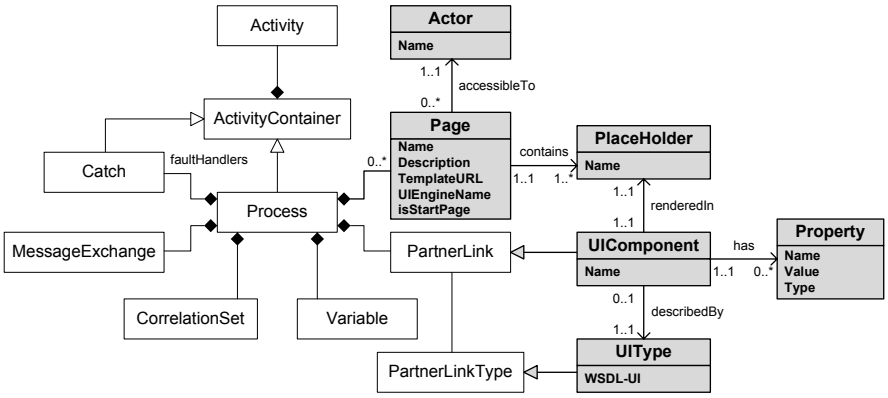


Figure 1 Simplified BPEL4UI meta-model in UML. White classes correspond to standard BPEL constructs; gray classes correspond to constructs for UI and user management.

Modeling UI-specific aspects requires instead introducing a set of **new constructs** that are not yet supported by BPEL. The constructs, illustrated in Figure 1, are: *UI type* (the partner link type for UI components), *page* (the web pages over which we distribute the UI of the application), *place holder* (the name of the place holders in which we can render UI components), *UI component* (the partner link for UI components), *property* (the constructor parameters of UI components), and *actor* (the human actors we associate with web pages).

It is important to note that although syntactically there is no difference between web services and UI components (the new JavaScript binding introduced into WSDL to map abstract operations to concrete JavaScript functions comes into play only at runtime), it is important to distinguish between services and UI components as their *semantics* and, hence, their usage in the model will be different. A detailed description of the new constructs and their usage can be found in [2].

As for the **layout** of distributed UI orchestrations, defining web pages and associating UI partner links with place holders requires implementing suitable HTML templates that are able to host the UI components of the orchestration at runtime. For the design of layout templates we rely on standard web design instruments. The only requirement the templates must satisfy is that they provide place holders in form of HTML DIV elements that can be indexed via standard HTML identifiers following a predefined naming convention, i.e., * … *.

The main **methodological goals** in implementing our UI orchestration approach were (i) relying as much as possible on existing *standards*, (ii) providing the developer with only *few and simple new concepts*, and (iii) implementing a runtime architecture that associates each concern to the *right level of abstraction and software tool* (e.g., UI synchronization is handled in the browser, while service orchestration is delegated to the BPEL engine).

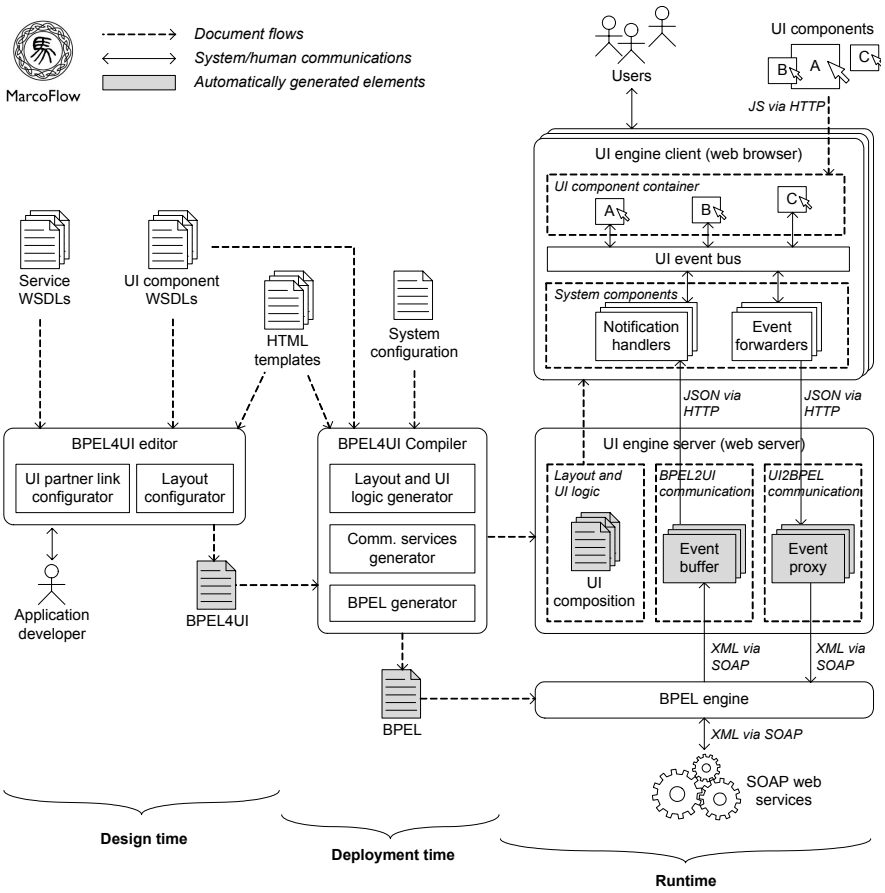


Figure 2 From design time to runtime: overall system architecture of MarcoFlow

3 The MarcoFlow Environment

Figure 2 shows the (simplified) architecture of the MarcoFlow environment, which aids the development and execution of distributed UI orchestrations. The architecture is partitioned into design time, deployment time, and runtime components, according to the three phases of the software development lifecycle supported by MarcoFlow.

The **design** part comprises the *BPEL4UI editor* that supports BPEL4UI [2], the composition language we use to specify distributed UI orchestrations. The editor is an extended Eclipse BPEL editor with (i) a panel for the specification of the pages in which UI components can be rendered and (ii) a property panel that allows the developer to configure the web pages, to set the properties of UI partner links, and to associate them to place holders in the layout.

The **deployment** of a UI orchestration requires translating the BPEL4UI specification into executable components: (i) a set of *communication channels* that mediate

between the UI components in the client browser and the BPEL engine; (ii) a *standard BPEL specification* containing the distributed UI synchronization and web service orchestration logic; and (iii) a set of *UI compositions* (one for each page of the application) containing the intra-page UI synchronizations. This task is achieved by the *BPEL4UI compiler*, which also manages the deployment of the generated artifacts in the respective runtime environments.

The **execution** of a UI orchestration requires the setup and coordination of three independent runtime environments: (i) the interaction with users and intra-page UI synchronization is managed in the client browser by an *event-based JavaScript runtime framework*; (ii) a so-called *UI engine server* runs the web services implementing the communication channels; and (iii) a *standard BPEL engine* manages the distributed UI synchronization and web service orchestration.

The MarcoFlow system shown in Figure 2 is fully implemented and running. A patent application for parts of the system has been filed.

4 Demo scenario

A example of how MarcoFlow can be used for the development of a distributed UI orchestration is available at http://mashart.org/marcoflow/demo.htm. The demo in form of a video illustrates in few minutes the main features of MarcoFlow in the context of a simple home assistance management application. Particular emphasis is given to the three development phases supported by the tool (design, deployment, and runtime) and to the use of the final application by the different actors involved in the distributed process logic.

References

1. J. Yu, B. Benatallah, F. Casati, F. Daniel. Understanding Mashup Development and its Differences with Traditional Integration. *IEEE Internet Computing*, Vol. 12, No. 5, September-October 2008, pp. 44-52.
2. F. Daniel, S. Soi, S. Tranquillini, F. Casati, C. Heng, L. Yan. From People to Services to UI: Distributed Orchestration of User Interfaces. *BPM'10*, Hoboken, NJ, USA.
3. OASIS. Web Services Business Process Execution Language Version 2.0, April 2007. [Online]. *http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html*
4. Active Endpoints Inc., Adobe Systems Inc., BEA Systems Inc., International Business Machines Corporation, Oracle Inc., SAP AG. WS-BPEL Extension for People (BPEL4People), Version 1.0. June 2007.
5. R. Acerbis, A. Bongio, M. Brambilla, S. Butti, S. Ceri, P. Fraternali. Web Applications Design and Development with WebML and WebRatio 5.0. *TOOLS'08*, pp. 392-411.
6. J. Gómez, A. Bia, A. Parraga. Tool Support for Model-Driven Development of Web Applications, *WISE'05*, pp. 721-730.
7. Sun Microsystems. JSR-000168 Portlet Specification, October 2003. [Online]. *http://jcp.org/aboutJava/communityprocess/final/jsr168/*
8. F. Daniel, F. Casati, B. Benatallah, M.-C. Shan. Hosted Universal Composition: Models, Languages and Infrastructure in mashArt. *ER'09*, pp. 428-443.

Appendix F

Distributed User Interface Orchestration: On the Composition of Multi-User (Search) Applications

Florian Daniel, Stefano Soi, and Fabio Casati
University of Trento, 38123 Povo (TN), Italy
{daniel,soi,casati}@disi.unitn.it

Abstract. While mashups may integrate into a new web application data, application logic, and user interfaces sourced from the Web – a highly intricate and complex task – they typically come in the form of simple applications (e.g., composed of only one web page) for individual users. In this chapter, we introduce the idea of *distributed user interface orchestration*, a mashup-like development paradigm that, in addition to the above features, also provides support for the coordination of *multiple users* inside one shared application or process. We describe the concepts and models underlying the approach and introduce the *MarcoFlow* system, a platform for the assisted development of distributed user interface orchestrations. As a concrete development example, we show how the system can be profitably used for the development of an advanced, *collaborative search application*.

1 Introduction

After workflow management (which supports the automation of business processes and human tasks) and service orchestration (which focuses on web services at the application layer), web mashups [1] feature a significant innovation: **integration at the UI level**. Besides web services or data feeds, mashups indeed reuse pieces of UIs (e.g., content extracted from web pages or JavaScript UI widgets) and integrate them into new web pages or applications. While mashups therefore manifest the need for reuse in UI development and for suitable UI component technologies, so far they produced rather simple applications consisting of one web page only.

We argue that there is a huge spectrum of applications that demand for development approaches that are similar to those of mashups but that go far beyond single page applications and, in fact, support multiple pages, multiple actors, complex navigation structures, and – more importantly – process-based application logic or navigation flows. We call this type of applications **distributed UI orchestrations** [2], as (i) both components and the application itself may be distributed over the Web and operated by different actors, (ii) in addition to traditional web services we also integrate novel JavaScript UI components, and (iii) services and UIs are orchestrated in a homogeneous fashion.

Developing distributed UI orchestrations therefore raises the need for the coordination of individual actors and the development of a distributed user interface *and* service orchestration logic. Doing so requires:

- Understanding how to *componentize UIs and compose them*;

- Defining a logic that is able to *orchestrate both UIs and web services*;
- Providing a language and tool for *specifying distributed UI compositions*; and
- Developing a runtime environment that is able to *execute distributed UI and service compositions*.

In this chapter, we describe how the above challenges have been solved in the context of the MarcoFlow project [2] and how the resulting approach can be leveraged for the development of a distributed search computing application that requires the coordination of services, UIs, and people.

This chapter is organized as follows: Next, we describe the search application that raises the need for distributed UI orchestration. In Section 3, we look at how existing techniques and technologies may support the development of such kind of application, while in Section 4 we introduce the distributed UI orchestration approach in order to fill the gaps. In Section 5, we describe our current development prototype, and in Section 6 we conclude the paper.

2 A Search Scenario

Let us consider the following collaborative search scenario illustrated in Figure 1, an extension of the single-user scenario discussed in [3].

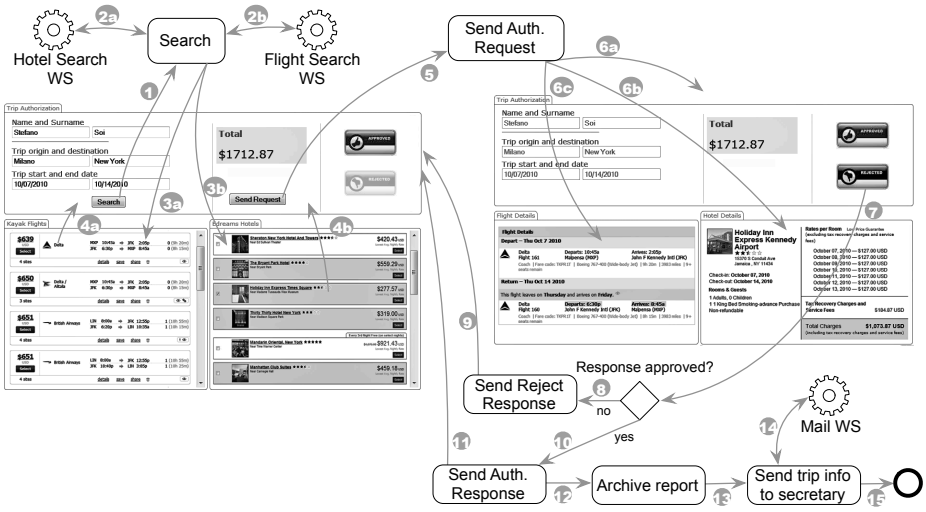


Fig. 1 Trip authorization distributed application. The gray arrows indicate synchronization or orchestration points; the number labels indicate their order in time.

This time we want to assist an employee that needs to request the authorization of a business trip to his superior. The employee can enter the relevant information about the trip (the origin and destination, and the start and end dates) and search for related flights and accommodations. He/she can select his/her preferred choices from the list of flights and hotels and send the request to the superior. The superior can inspect the

request, along with its details, and send a response (accept or redo) to the employee, together with some optional comment. If the request is approved, the response will be sent to the employee and the procedure will end with archiving and mailing operations. If the request is rejected the previous steps (all or a part of them) can be repeated until the authorization request is approved.

If we analyze the scenario, we see that the envisioned application (as a whole) is *distributed* over the Web. The application includes, besides the process logic, two mashup-like, web-based control consoles for the employee and the superior that are themselves part of the orchestration and need to interact with the underlying process logic. The UIs for the actors participating in the application are composed of UI components, which can be components developed in-house (like the *Trip Authorization* component) or sourced from the Web (like the *Hotel Search* and the *Kayak Flights Search* component); service orchestrations are based on web services. In our case, the latter two UI components serve only to render content, which still needs to be queried from the Web via dedicated web services. The *Trip Authorization* component, instead, collects data from the user (via its input fields) and from the other two UI components (via suitable synchronization operations). Finally, the two applications for the employee and the superior are instantiated in different web browsers, contributing to the distribution of the overall UI and raising the need for synchronization.

3 Distributed UI Orchestration

The **key idea** to approach the coordination of (i) UI components inside web pages, (ii) web services providing data or application logic, and (iii) individual pages (as well as the people interacting with them) is to split the coordination problem into two layers: *intra-page UI synchronization* and *distributed UI synchronization and web service orchestration*.

UIs are typically event-based (e.g., user clicks or key strokes), while service invocations are coordinated via control flows. In [2] we show how to describe UI components (as also introduced in [4]) in terms of standard WSDL descriptors, how to bind them to JavaScript, and how to extend the standard BPEL language in order to support the two above composition layers. We call this extended language **BPEL4UI**. Fig. 2 shows the simplified meta-model of the language and details all the new modeling constructs necessary to specify UI orchestrations (gray-shaded), omitting details of the standard BPEL language, which are reused as is by BPEL4UI. The model in Fig. 2 exclusively focuses on the composition aspects, while the events and operations of UI components are defined in their WSDL descriptors [2].

In terms of standard BPEL [5], a UI orchestration is a *process* that is composed of a set of associated *activities* (e.g., sequence, flow, if, assign, validate, or similar), *variables* (to store intermediate processing results), *message exchanges*, *correlation sets* (to correlate messages in conversations), and *fault handlers*. The services or UI components integrated by a process are declared by means of so-called *partner links*, while *partner link types* define the roles played by each of the services or UI components in the conversation and the *port types* specifying the operations and messages supported by each service or component.

Modeling UI-specific aspects requires instead introducing a set of **new constructs** that are not yet supported by BPEL. The constructs, illustrated in Fig. 2, are: *UI type* (the partner link type for UI components), *page* (the web pages over which we distribute the UI of the application), *place holder* (the name of the place holders in which we can render UI components), *UI component* (the partner link for UI components), *property* (the constructor parameters of UI components), and *actor* (the human actors we associate with web pages).

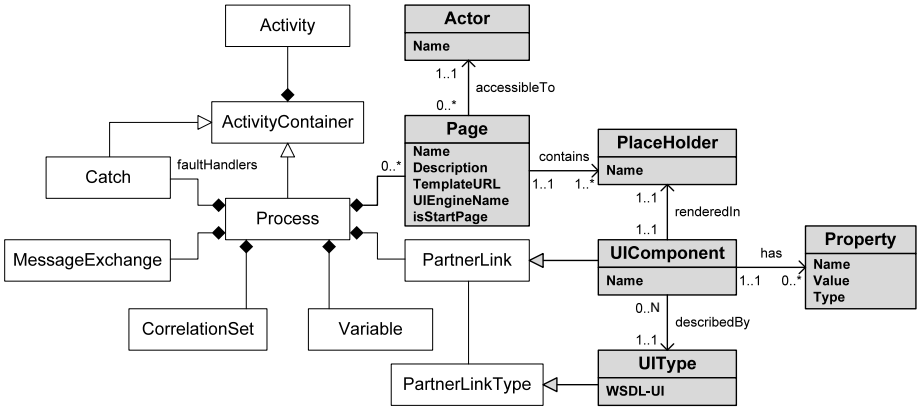


Fig. 2 Simplified BPEL4UI meta-model in UML. White classes correspond to standard BPEL constructs; gray classes correspond to constructs for UI and user management.

It is important to note that although syntactically there is no difference between web services and UI components (the new JavaScript binding introduced into WSDL to map abstract operations to concrete JavaScript functions comes into play only at runtime), it is important to distinguish between services and UI components as their *semantics* and, hence, their usage in the model will be different. A detailed description of the new constructs and their usage can be found in [2], while in Figure 3 we illustrate the BPEL4UI model of our example search application (shown in Figure Fig. 1) as modeled in our extended Eclipse BPEL editor.

The BPEL4UI model is structured into four main blocks: one *repeat-until* and three *sequences* (sub-types of the *Activity* entity in Fig. 2). The repeat-until block at the left manages the flights and hotels search operations. The processing of the block starts upon the reception of the relevant data about the trip from the employee's console, then it invokes the external search web services and, finally, sends the respective results to the UI components rendering the flight and hotel offers. This block of operations can be repeated an arbitrary number of times (e.g., in case the employee want to input new search criteria or a trip request has been rejected and needs to be redone), until the authorization is accepted. Once the search results are rendered in their UI components, the employee can choose a flight and a hotel combination by clicking on the respective choices. This allows the employee to compose his trip request summarized in the *Trip Authorization* component. The two sequence blocks (*Flight Selection* and *Hotel Selection*) in the middle of the model implement the operations that are necessary to synchronize the *Trip Authorization* UI-component, which is then in

charge of storing the combination and computing the total cost of the trip. These communications, involving only UI-components belonging to the same page, are completely managed inside the employee's web browser. Once all the trip data are available, the *Send Request* button in the employee console is activated and can be used to forward the authorization request to the superior. Receiving the authorization request starts the right block in the model (*Authorization Request and Response*), which waits for the trip request data and then forwards them to the *Trip Authorization*, *Hotel* and *Flight* UI-components of the superior's console. Now the superior can inspect the request and send a response that is forwarded to the employee's console. If the superior approves the request, two web services are invoked, respectively for archiving and mailing, and, finally, the process is terminated. If the response is a reject, the whole block of operations can be repeated, allowing the employee to modify his request. The right block of service orchestration hence requires the coordination of the two actors, i.e., employee and superior, and the distributed orchestration of UI components and web services. Doing so requires the help from the BPEL engine and the setting of a suitable BPEL correlation set.

Fig. 3 BPEL4UI modeling example for the Trip Authorization application.

As for the **layout** of distributed UI orchestrations, defining web pages and associating UI partner links with placeholders requires implementing suitable HTML templates that are able to host the UI components of the orchestration at runtime. For the design of layout templates, we do not propose any new development instrument and rather allow the developer to use his/her preferred development tool (from simple text editors to model-driven design tools). The only requirement the templates must satisfy is that they provide place holders in form of HTML DIV elements that can be indexed via standard HTML identifiers following a predefined naming convention, i.e., * … *. For instance, all the activities with a "[UI]" prefix in Figure 3 are associated to placeholders, in order to fill the two pages composing our reference scenario.

As this discussion shows, the main **methodological goals** in implementing our UI orchestration approach were (i) relying as much as possible on existing *standards*, (ii) providing the developer with only *few and simple new concepts*, and (iii) implementing a runtime architecture that associates each concern to the *right level of abstraction and software tool* (e.g., UI synchronization is handled in the browser, while service orchestration is delegated to the BPEL engine). These decisions, for instance, allow us to reuse BPEL's internal exception handling mechanisms to manage also exceptions in distributed UI orchestrations.

4 The MarcoFlow Environment

Fig. 4 shows the (simplified) architecture of the MarcoFlow environment, which aids the development and execution of distributed UI orchestrations. The architecture is partitioned into design time, deployment time, and runtime components, according to the three phases of the software development lifecycle supported by MarcoFlow.

The **design** part comprises the*BPEL4UI editor* that supports the full BPEL4UI language as defined in [2]. The editor is an extended Eclipse BPEL editor with (i) a panel for the specification of the pages in which UI components can be rendered and (ii) a property panel that allows the developer to configure the web pages, to set the properties of UI partner links, and to associate them to place holders in the layout.

The **deployment** of a UI orchestration requires translating the BPEL4UI specification into executable components: (i) a set of *communication channels* that mediate between the UI components in the client browser and the BPEL engine; (ii) a *standard BPEL specification* containing the distributed UI synchronization and web service orchestration logic; and (iii) a set of *UI compositions* (one for each page of the application) containing the intra-page UI synchronizations. This task is achieved by the *BPEL4UI compiler*, which also manages the deployment of the generated artifacts in the respective runtime environments.

The **execution** of a UI orchestration requires the setup and coordination of three independent runtime environments: (i) the interaction with users and intra-page UI synchronization is managed in the client browser by an *event-based JavaScript runtime framework*; (ii) a so-called *UI engine server* runs the web services implementing the communication channels; and (iii) a *standard BPEL engine* manages the distributed UI synchronization and web service orchestration.

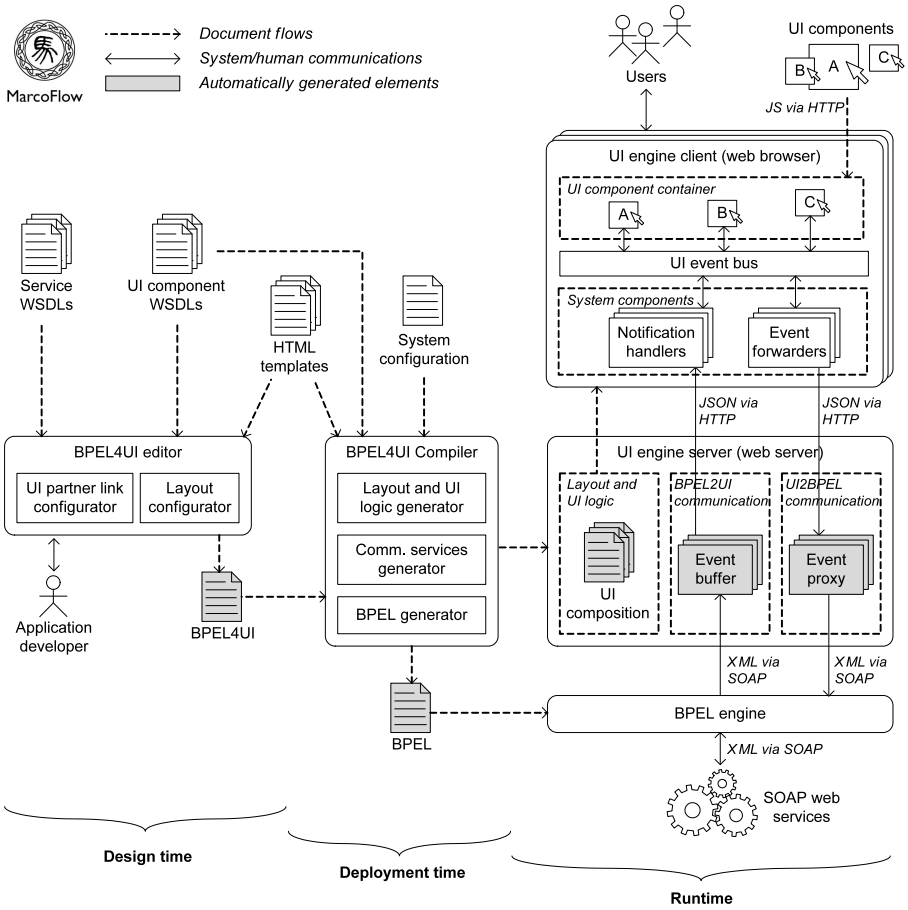


Fig. 4 From design time to runtime: overall system architecture of MarcoFlow

In order for the superior and the employee to manage their trip authorizations, MarcoFlow also comes with a simple **task manager** (not detailed in Fig. 4), which allows them to start new trip authorizations (the employee) and to participate in running instances of the application (the manager). Each new request requires a new instantiation of the process. All running instances are shown to both actors in their personalized lists. An instance terminates upon successful approval of the trip.

The MarcoFlow system shown in Fig. 4 is fully implemented and running. A patent application for parts of the system has been filed. A detailed demonstration of how MarcoFlow can be used for the development of distributed UI orchestration is available at *http://mashart.org/marcoflow/demo.htm*.

5 Related Work

In most **service orchestration** approaches, such as BPEL [5], there is no support for UI design. Many variations of BPEL have been developed, e.g., aiming at the invoca-

tion of REST services [6] or at exposing BPEL processes as REST services [7]. IBM's Sharable Code platform [8] follows a slightly different strategy in the composition of REST and SOAP services and also allows the integration of user interfaces for the Web; UIs are however not provided as components but as ad-hoc Ruby on Rails HTML templates.

BPEL4People [9] is an extension of BPEL that introduces the concept of people task as first-class citizen into the orchestration of web services. The extension is tightly coupled with the **WS-HumanTask** [10] specification, which focuses on the definition of human tasks, including their properties, behavior and operations used to manipulate them. BPEL4People supports people activities in form of inline tasks (defined in BPEL4People) or standalone human tasks accessible as web services. In order to control the life cycle of service-enabled human tasks in an interoperable manner, WS-HumanTask also comes with a suitable coordination protocol for human tasks, which is supported by BPEL4People. The two specifications focus on the coordination logic only and do not support the design of the UIs for task execution.

The systematic development of web interfaces and applications has typically been addressed by the web engineering community by means of **model-driven web design approaches**. Among the most notable and advanced model-driven web engineering tools we find, for instance, WebRatio [11] and VisualWade [12]. The former is based on a web-specific visual modeling language (WebML), the latter on an object-oriented modeling notation (OO-H). Similar, but less advanced, modeling tools are also available for web modeling languages/methods like Hera, OOHDM, and UWE. These tools provide expert web programmers with modeling abstractions and automated code generation capabilities for complex web applications based on a hyperlink-based navigation paradigm. WebML has also been extended toward web services [13] and process-based web applications [14]; reuse is however limited to web services and UIs are generated out of HTML templates for individual components.

A first approach to component-based UI development is represented by **portals and portlets** [15], which explicitly distinguish between UI components (the portlets) and composite applications (the portals). Portlets are full-fledged, pluggable Web application components that generate document markup fragments (e.g., (X)HTML) that can however only be reached through the URL of the portal page. A portal server typically allows users to customize composite pages (e.g., to rearrange or show/hide portlets) and provides single sign-on and role-based personalization, but there is no possibility to specify process flows or web service interactions (the new WSRP [16] specification only provides support for accessing remote portlets as web services). Also **JavaServer Faces** [17] feature a component model for reusable UI components and support the definition of navigation flows; the technology is however hardly reusable in non-Java based web applications, navigation flows do not support flow controls, and there is no support for service orchestration and UI distribution.

Finally, the web mashup [1] phenomenon produced a set of so-called **mashup tools**, which aim at assisting mashup development by means of easy-to-use graphical user interfaces targeted also at non-professional programmers. For instance, Yahoo! Pipes (http://pipes.yahoo.com) focuses on data integration via RSS or Atom feeds via a data-flow composition language; UI integration is not supported. Microsoft Popfly (http://www.popfly.ms; discontinued since August 2009) provided a graphical user interface for the composition of both data access applications and UI components;

service orchestration was not supported. JackBe Presto (http://www.jackbe.com) adopts a Pipes-like approach for data mashups and allows a portal-like aggregation of UI widgets (so-called mashlets) visualizing the output of such mashups; there is no synchronization of UI widgets or process logic.IBM QEDWiki (http://services.alphaworks.ibm.com/qedwiki) provides a wiki-based (collaborative) mechanism to glue together JavaScript or PHP-based widgets; service composition is not supported. Intel Mash Maker (http://mashmaker.intel.com) features a browser plug-in which interprets annotations inside web pages allowing the personalization of web pages with UI widgets; service composition is outside the scope of Mash Maker.

In the mashArt [4] project, we worked on a so-called universal integration approach for UI components and data and application logic services. MashArt comes with a simple editor and a lightweight runtime environment running in the client browser and targets skilled web users. MashArt aims at simplicity: orchestration of distributed (i.e., multi-browser) applications, multiple actors, and complex features like transactions or exception handling are outside its scope. The CRUISe project [18] has similarities with mashArt, especially regarding the componentization of UIs. Yet, is does not support the seamless integration of UI components with service orchestration, i.e., there is no support for complex process logic. CRUISe rather focuses on adaptivity and context-awareness. Finally, the ServFace project [19] aims at supporting even unskilled web users in composing web services that come with an annotated WSDL description. Annotations are used to automatically generate form-like interfaces for the services, which can be placed onto one or more web pages and used to graphically specify data flows among the form fields. The result is a simple, user-driven web service orchestration. None of these projects, however, supports the coordination of multiple different actors inside a same process, and none of the approaches discussed supports the distribution of UIs over multiple browsers.

6 Conclusion and Future Works

In this chapter, we addressed the problem of designing and orchestrating *component-based web applications* that are distributed over *multiple web browsers* and that involve *multiple different actors*. We particularly discussed the case of a search computing application that leverages on a collaborative search and browsing approach, an application feature whose development with traditional techniques would be everything but trivial. In fact, while the integration of UIs and web services is, for instance, also supported by current mashup platforms, the coordination of the actors involved in the application and the synchronization of their respective UIs would still require manual intervention. The MarcoFlow platform introduced in this chapter, instead, supports the seamless integration of services, UIs, and people in one and the same development environment, sensibly speeding up the development of process-based, mashup-like web applications.

The basic idea of MarcoFlow, i.e., the component-based development of applications is inspired by current web mashup practices, which in many cases aim at enabling also the *less skilled developer* (or even unskilled end users) to compose own applications. Given the complexity of the applications supported by MarcoFlow, it is

however important to note that MarcoFlow rather targets skilled developers (e.g., developers that are familiar with composite web service development in BPEL).

One of the challenges to be addressed in our future work is therefore lowering the complexity of the design environment for distributed UI orchestrations, hiding BPEL4UI behind an easier to learn, graphical modeling language. Also, we would like to extend the approach toward streaming web services, for example to support the design of continuous queries over sensor networks.

References

1. J. Yu, B. Benatallah, F. Casati, F. Daniel. Understanding Mashup Development and its Differences with Traditional Integration. *IEEE Internet Computing*, Vol. 12, No. 5, September-October 2008, pp. 44-52.
2. F. Daniel, S. Soi, S. Tranquillini, F. Casati, C. Heng, L. Yan. From People to Services to UI: Distributed Orchestration of User Interfaces. *Proceedings of BPM'10*, pp. 310-326.
3. F. Daniel, S. Soi, F. Casati. From Mashup Technologies to Universal Integration: Search Computing the Imperative Way. *Search Computing - Challenges and Directions*, June 2009, pp. 72-93.
4. F. Daniel, F. Casati, B. Benatallah, M.-C. Shan. Hosted Universal Composition: Models, Languages and Infrastructure in mashArt. *ER'09*, pp. 428-443.
5. OASIS. Web Services Business Process Execution Language Version 2.0, April 2007. [Online]. *http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html*
6. C. Pautasso. BPEL for REST. *BPM'08*, Milano, pp. 278-293.
7. T. van Lessen, F. Leymann, R. Mietzner, J. Nitzsche, D. Schleicher. A Management Framework for WS-BPEL, *ECoWS'08*, Dublin, pp. 187-196.
8. E. M. Maximilien, A. Ranabahu, K. Gomadam. An Online Platform for Web APIs and Service Mashups, *Internet Computing*, vol. 12, no. 5, pp. 32-43, Sep. 2008.
9. Active Endpoints, Adobe, BEA, IBM, Oracle, SAP. WS-BPEL Extension for People (BPEL4People), Version 1.0. June 2007.
10. Active Endpoints, Adobe, BEA, IBM, Oracle, SAP. Web Services Human Task (WS-HumanTask), Version 1.0. June 2007.
11. R. Acerbis, A. Bongio, M. Brambilla, S. Butti, S. Ceri, P. Fraternali. Web Applications Design and Development with WebML and WebRatio 5.0. *TOOLS'08*, pp. 392-411.
12. J. Gómez, A. Bia, A. Parraga. Tool Support for Model-Driven Development of Web Applications, *WISE'05*, pp. 721-730.
13. I. Manolescu, M. Brambilla, S. Ceri, S. Comai, P. Fraternali. Model-Driven Design and Deployment of Service-Enabled Web Applications. *ACM Trans. Internet Technol.*, Vol. 5, No. 3, August 2005, pp. 439-479.
14. M. Brambilla, S. Ceri, P. Fraternali, I. Manolescu. Process Modeling in Web Applications. *ACM Trans. Softw. Eng. Methodol.*, Vol. 15, No. 4, October 2006, pp. 360-409.
15. Sun Microsystems. JSR-000168 Portlet Specification, October 2003. [Online]. *http://jcp.org/aboutJava/communityprocess/final/jsr168/*
16. OASIS. Web Services for Remote Portlets, August2003. [Online]. *www.oasis-open.org/committees/wsrp*
17. Oracle. JavaServer Faces Technology. [Online] *http://java.sun.com/javaee/javaserverfaces/*
18. S. Pietschmann, M. Voigt, A. Rümpel, K. Meissner. CRUISe: Composition of Rich User Interface Services. *ICWE'09*, pp. 473-476.
19. M. Feldmann, T. Nestler, U. Jugel, K. Muthmann, G. Hübsch, A. Schill. Overview of an end user enabled model-driven development approach for interactive applications based on annotated services. *WEWST'09*, pp. 19-28.

Appendix G

Domain-specific Mashups: From All to All You Need

Stefano Soi and Marcos Baez
Dipartimento di Ingegneria e Scienza dell'Informazione
University of Trento
Via Sommarive, 14 38123
Trento, Italy
{soi,baez}@disi.unitn.it

Abstract. Last years, aside the proliferation of Web 2.0, we assisted to the drastic growth of the mashup market. An increasing number of different mashup solutions and platforms emerged, some focusing on data integration (a la Yahoo! Pipes), others on user interface (UI) integration and some trying to integrate both UI and data. Most of proposed solutions have a common characteristic: they aim at providing non-programmers with a flexible and intuitive general-purpose development environment. While these generic environments could be useful for web users to develop simple applications, they are often too generic to address domain-specific needs and to allow users to develop real-life complex applications. In particular, proposed mashup mechanisms do not reflect those specific concepts that are proper of a given domain, which domain-experts are familiar with and could autonomously manage. We argue the need for domain-specific mashup architectures, also going beyond today's enterprise platforms, in which standard mashup mechanisms and components are driven by an underlying domain-specific layer. This layer will provide a service and component ecosystem built upon a shared and uniform conceptual model specific for the given domain. This way, domain experts will be provided with mashup components and mechanisms, following those well-known concepts and rules proper of the domain they belong to, that they are able to understand, use and, finally, profitably compose. In this paper, we will show the necessity of such an architecture through a real-life use case in the context of scientific publications and journals.

Keywords: domain specific mashups, vertical mashups, end-user centric mashups

1 Introduction

During last decade a vast amount of functionalities have been made available as online services, in form of Web Services, APIs, RSS/Atom feeds and so on. While these services can also be used independently from one other, putting them together to create a value-adding combination could lead to much more fruitful results, as described in [1]. This is exactly what mashup solutions try to achieve. In addition,

most of available mashup platforms aim at giving the possibility to develop such composite applications to domain experts, i.e. users with very limited programming skills but deep knowledge of the domain being the context of the problem to be solved. A number of studies (e.g., [2], [3]) discuss about benefits of moving the development of this kind of composite applications from IT-experts to non-programmers. This would be a radical paradigm shift bringing two main advantages, that is, first, avoiding requirement transfer from domain-experts to IT-experts and, moreover, allowing to face the development of situational applications[1], that is applications addressing transient or very specific needs for which the standard development lifecycle is not adequate since it would not be time- and cost-effective.

Current mashup building tools have the - non trivial - target of enabling domain-experts to develop such mashed up applications without - almost - any programming skill or any intervention of expert developers. This is often the main claim of available mashup platforms but, from our studies and experience in the mashup field, we found that this claim is only partially fulfilled. In particular, available mashup solutions provide easy mechanisms, suited for non-programmers, allowing them to produce very simple applications (often just "toy applications") or covering only some aspects of the integration needs of the users [4]. When users' needs go beyond this complexity level, available solutions show up their limitations.

In our opinion, main reasons underlying difficulties in overcoming these complexity limits are related to the fact that current mashup platforms (like [5],[6] and similar) have the ambitious goal of "integrating all the Web". In other words, they are not targeted at a specific domain but aim to give the possibility to make interacting user interfaces (UIs) and services coming from completely different domains and producers. For the time being, we think that "integrating all the Web" is a too ambitious goal. The lack of widely adopted - official or de facto - standards in this area make the integration of highly heterogeneous components a complex task that, at the end, is not sufficiently supported with mechanisms well-suited for non-programmers.

Starting from these considerations, we argue that there is the need for domain-specific mashup solutions. In particular, we propose to place the mashup system upon a layer defining concepts and policies of the given domain. We will see that this layer should include all the knowledge about the specific domain that could be then used to ease the mashup development, making actually possible to move the mashup application development from IT-experts to domain-experts.

Summarizing, the main contributions of this work are:

- stating the need for moving from horizontal general-purpose mashup solutions to vertical domain-specific ones, allowing domain-experts to autonomously compose their applications playing in their well-know playground
- proposing a modular architecture that makes a net separation between mashup layer and – pluggable – domain layer
- giving a first characterization of the “domain” concept, in terms of domain entities and rules, and first proposals on how the mashup platform should adapt to these.

[1] For details and references about the concept of situational application we refer to the Wikipedia page: http://en.wikipedia.org/wiki/Situational_application

To make our proposal clearer, we will explain the proposed solutions with the help of a use case, taken from the scientific publications context. In particular, we will make reference to a project our group is working on, called *LiquidPub* (http://liquidpub.org/), and we will show how the solution we propose could be profitably applied in that context. This will be the domain we will try to characterize and on which verticalize. We will base our example on a mashup tool coming from another project of our group, called *mashArt* (http://mashart.org/), which provides a complete mashup platform allowing for *Universal Integration* [7], that is seamless integration of data, services and user interfaces (UIs), targeted to non-programmer web users.

The rest of this paper is structured as follows. Section 2 will introduce and describe the motivating scenario, with particular reference to the LiquidPub project. In Section 3 we will see in more detail how generic mashup platforms work and what are their limitations. Section 4 will propose an architecture trying to overcome the issues presented in the previous section. In Section 6 will be presented the final conclusions and future work.

2 Motivating Scenario: Knowledge Dissemination

The Web has pushed forward technological and social changes in different areas and the scientific domain has not been the exception. It has opened a brand new world of possibilities for how the scientific knowledge can be consumed, produced, shared and disseminated. This has motivated an extensive research on how to exploit these opportunities, leading to novel *models*, new forms of *scientific contributions*, *metrics*, *services* and *sources of information*. Having a virtually infinite number of possibilities also implies that there could be different ways of consuming /disseminating /evaluating the scientific research work. The selection of the right configuration in terms of type of content (peer-reviewed papers, preprints, blogs, datasets...), the metrics (h-index, citation count, pagerank,..), sources (reputed publishers, open archives or the whole web) and the actual process will probably depend on the final usage scenario and the believes of the community. Implementing a particular dissemination model would normally require programming knowledge to produce the required code (e.g., in java, perl, ruby…), following a particular development process. Considering that everybody in the scientific domain has different thoughts on how to do this, it will be unnecessarily limiting to restrict this to programmers. It is clearly something end-users, or domain experts, should be able to do, and not only programmers.

Mashups provide the foundations for supporting such a scenario. However, current mashup platforms provide rather generic components, and so, at a level the scientist (non-programmer) cannot manage. For example, if a scientist strongly believes that i) blogs and open archives are valid dissemination venues, ii) sharing and consumption should be the main goal of a dissemination model, and therefore iii) sharing data should be used as base to evaluate researchers; presenting her a component that connects to a web service as primary tool does not help her in the composition of the dissemination model she believes in. Configuring and wiring components would become extremely complex (e.g., setting up a web service connection, or even

selecting the right web service), and the user would be required to provide the mapping in terms of I/O among heterogeneous web services (when the whole concept of "mapping" is probably obscure to her), to reflect a flow and a process at such low level. Maintaining and reasoning over such a composition would also be a complex task, not to mention that there is no way to ensure that the final outcome is actually a dissemination model. Hence, in this context, defining the desired dissemination model would not only be complex but it would require programming skills to specify the mashup, regardless how fancy the user interface might be.

Thus, domain concepts and processes (e.g., notions of publication, review, paper, venue,…) should be exploited in order to really assist her in the composition. Taking this as reference scenario, we discuss the limitations and required extensions to current mashup platforms, in the following sections.

3 Understanding Current Mashups and their Limitations

As introduced in Section 1, the mashup approach should provide domain-experts with no programming skills with suitable mechanisms allowing them to develop autonomously their situational applications, leading to the above mentioned advantages in terms of responsiveness and effectiveness of solutions.

What makes possible moving application development from IT-experts to domain-experts, is probably the complete separation of roles, and thus of required skills, among component and composition developer, as depicted in Figure 1.

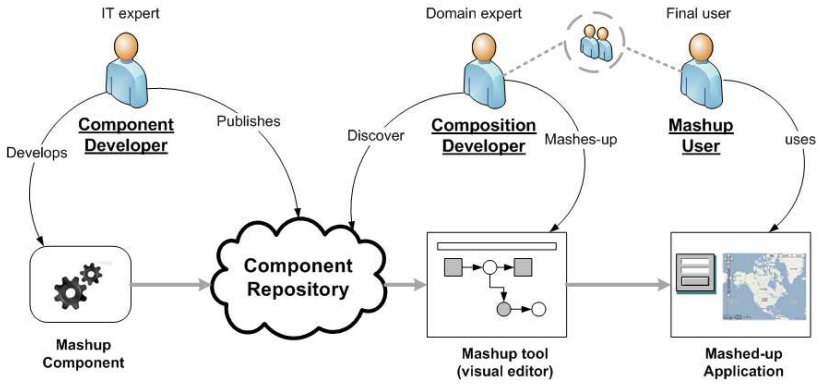


Figure. 1. Separation of roles among component and composition developer. Dotted representation among composition developer and mashup user, indicates that both roles can be covered by the same person, as typically happens in the context of situational applications. The figure is an adaptation of the one presented in [15].

The former is responsible to create and publish the building blocks that will be glued together to realize the final composite application. These components will implement or wrap some services or will represent a user interface. Complexity is primarily pushed into components, leaving compositions simpler and lightweight. It is clear that component development requires specific programming skills, in particular

in the web-programming field, since mashup platforms are typically offered as web applications, following the Software as a Service (SaaS) approach [8]. To this end, a number of web tools have been proposed to help and simplify the creation of services and components (e.g., data extraction from web pages, web clipping) [9]. Some examples are OpenKapow and Dapper. Both this tools provide simple mechanisms to grab contents from web pages and expose extracted data as web services or RSS feeds. However, this kind of tools still requires a not negligible knowledge of programming concepts to be effectively used.

Assuming that components have been developed and made available, composition developers, now, only need to define the business logic addressing their needs, connecting available components, usually through simple visual mechanism (e.g., drag and drop). This operation should not need any particular programming knowledge or complex operation and should be tackled by advanced web users that have a deep knowledge in their domain but no skills in programming. Typically, in the context of situational applications development, the domain-expert plays both the mashup developer and mashup consumer role (as indicated by dotted representation in Figure 1), since she develops compositions to automate processes covering her situational needs.

Providing autonomy in mashups composition to advanced web users is the big claim of most mashup platforms, but our experience and studies showed that it is not actually fulfilled in general. Proposed solutions allow domain-experts to compose their applications without the need to write programming code, but this does not mean that the composition process is easily and intuitively affordable by non-programmers [10]. In the example of Section 2, this would the case for the scientist trying to select publication venues but that find herself with a web service connector. Analyzing available mashup tools, we concluded that they are often confusing for domain-experts, starting from the components selection that, given the vast amount of available possibilities, could be time-consuming and error prone. For example, if we analyze two popular visual environments for "Consumer mashup" composition, Yahoo! Pipes and Microsoft PopFly[2], we can see that they provide users with about 50 components for Yahoo! solution and more than 300 for Microsoft one. Moreover, when the needs require more complex mashup solutions many tools either are no more sufficient or start requiring to the composition developer (domain expert) a deeper and deeper understanding of programming concepts. In fact, a significant part of offered components provide functionalities that can be exploited only by those users that have good programming knowledge (e.g., regular expressions, loops). Another important lack of currently available solutions is that almost none of them provide *universal integration*, that is, as discussed in [7], the seamless integration of data, application, and user interface (UI) components, characteristic that we consider necessary to actually enable end-users to develop their situational applications. For instance, Yahoo! Pipes is mainly oriented toward data integration while Intel MashMaker mainly focuses on UI integration, but to build real-life complex applications both ingredients are needed. In the field of the "Enterprise mashup" tools many efforts is being done to address some business-critical issues, like security,

[2] Microsoft PopFly was discontinued on August 2009, but still remain an important mashup platform example that attracted thousands of developers.

privacy, reliability and accountability. From the point of view of domain-experts usability, enterprise solutions suffer of the same problems of consumer ones. In particular, although there exist powerful and complete mashup solutions, they are usually targeted at programming-skilled users. A noticeable example is the Tibco[3] suite, providing users with a vast amount of different components and mechanisms, covering every possible need, but often strictly related to programming concept that domain-experts could completely ignore or, however, difficultly manage.

All the generic components and mechanisms that available mashup solutions, both consumer and enterprise, provide and their programming-nature limit the possibilities of composition of domain-experts to "toy applications".

We argue that the main reasons for these limitations regarding most of the available tools need to be searched in their aim to be generic-application building tools. This general purpose attitude make it difficult for domain-experts to get familiar with components, functionalities and mechanisms representing concepts and entities they are not acquainted - and which they are not interested in. Moreover, such an approach aims at integrating components from different sources belonging to different domains, so, very often, making possible the communication among different components is a complex task still requiring specific programming efforts. Back to our example of Section 2, this would be the case for the scientist who wants to aggregate, according to her, valid venues of scientific resources (e.g., publishers, blogs, eprints) to incorporate them in her model. This would lead to complex mappings requiring, most probably, some programming skills.

Overcoming these issues requires a different mashup platform architecture allowing domain experts to work in their natural playground, where they are familiar with concepts and issues, so that they can tackle the development of their situational applications. To the best of our knowledge, there is no related work actually exploring the concept of domain-specific mashup. Next section will describe our proposed solution and architecture, aiming to enable real-life application development for domain-experts.

4 Domain-specific Mashups

Domain-specific mashups is our proposal to exploit domain concepts at the mashup composition level in order to put domain-experts at the center by providing an environment that can really assist them in the composition of domain-specific mashups. So, we need specific solutions pushing domain concepts up to the composition editor level, so that users can play in their well-known conceptual environment. To achieve such a system, we propose a modular architecture including two main layers, as depicted in Figure 2.

[3] http://www.tibco.com

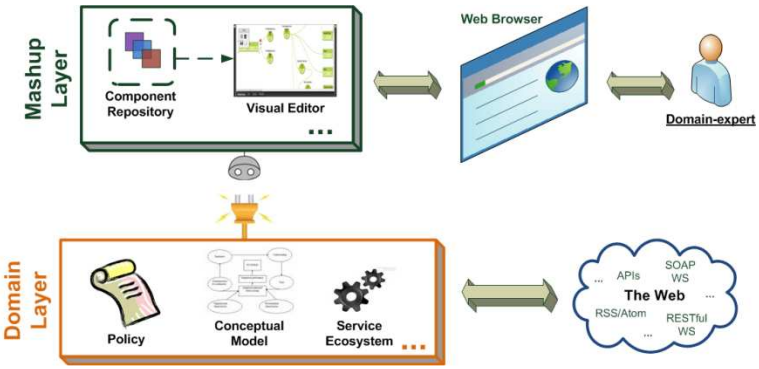


Figure 2. High-level architecture of a domain-specific mashup platform.

The upper layer is the actual *Mashup Layer*, that is, a mashup tool similar to some available today. In particular, this layer should - at least - include a composition visual editor, providing end-user friendly mechanisms for composition development through a common web browser, and a component repository, providing all the components that could be useful for building compositions in a given domain. In addition, other parts should be included at this level, like a runtime environment able to run the produced composition and other implementation-specific components, but those go beyond the scope of this work that is trying to focus on the composition-development phase seen from the domain-experts point of view. Our proposal to actually help and enable domain-experts to autonomously create their composite applications is to transform our generic mashup tool into a domain-specific one injecting domain related concepts into the development environment. Aiming at this, we create a *Domain Layer* that will be then plugged into the *Mashup Layer*. This lower layer is responsible for the domain characterization. In other words, it will define all the concepts and entities proper of the domain, their representation and general rules regulating the interactions among them. Furthermore, this layer will provide the domain related ecosystem of services, either implemented inside the layer or wrapping web-sourced services.

In this section we provide the two aspects covered by our proposal: modeling and characterizing the domain and its projection to the mashup platform.

4.1 Characterizing the Domain: Domain Layer

In order to leverage domain-experts knowledge in the composition, we need to understand the concepts, properties, rules and processes that make the domain. The definition of these elements is key to the selection of the right level of abstraction for users. To this end, we rely on the definition of the conceptual model, the business level operations and the domain rules.

Conceptual model. In the context of a particular domain, there are concepts and relations among these concepts that are known in the domain and familiar to domain experts. These concepts are commonly represented in a conceptual model. For

instance, in Figure 3, we show a possible conceptual model for the example introduced in Section 2.

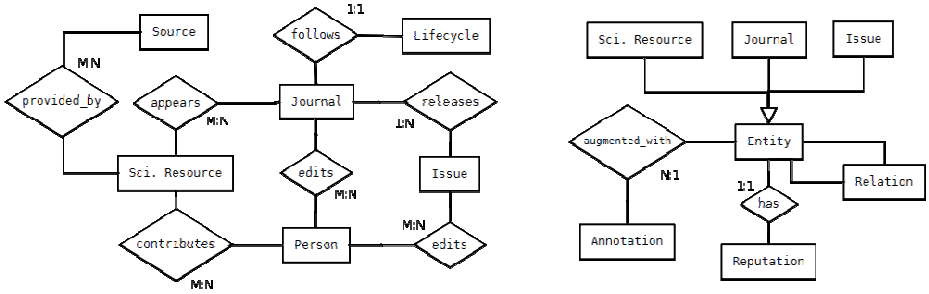


Figure 3. Liquid journals conceptual model

The Figure above captures the concepts in the knowledge dissemination domain. It is based on the liquid journal model, which represents a family of models from the traditional ones to the ones more social and web-aware [11]. In this model we can see that *journal* is a first-class entity composed of *scientific resources* (papers, blogs, datasets, ...), organized in *journal issues*, driven by *editors* and that follows a certain lifecycle. We can also see that *scientific resources* belong to certain sources (venues).

Business-level operations. Operations and processes that affect the shared concepts are highly relevant. These relate to users' every-day life and as such provide the level at which the expert can better reason. Following our example, these are the operations which are meaningful in the domain such as *publish, evaluate, review, submit*, and other more social such as *share, annotate, search*, etc.

Business rules. Business rules are well known by domain experts. They are very important as they give shape to the business logic and processes. In our example, we could establish as a business rule that whatever publication model we follow, we need to first select/review a paper before publishing it in journal.

The information we have described above is present in the domain but not exploited in the mashup composition. Mashup composition environments strongly rely on the domain expert to build application from usually low-level components, and so they do little or nothing to assist users. To inject these elements into the composition environment we propose to build a *Domain Layer* as a way of hiding the unnecessary complexity which is currently exposed to users. Although we are convinced that the complexity could be pushed into the component design, having such a platform will make component development much easier and would ensure consistency and, finally, smooth composition. Another good reason for having such a layer is the increasing existence of domain specific ecosystems. Thus, in practical terms, concepts and operations are captured typically by a platform exposing an API. In our example, the *liquid journal* platform provides the services and key entities via

RESTful services[4]. This platform builds on existing sources of information (e.g., Google Scholar, eprints, DBLP) and allows upper layer to access them using the common conceptual model in Figure 3. Solving the heterogeneity at this level has the advantage of not only providing homogeneous programmatic access, but also of avoiding taking complex mappings to the mashup environment by preparing the components according to the shared concepts. Of course, taking these and all the domain elements described here requires some work to make it happen. We discuss these and other related issues in the next subsection.

4.2 Taking the Domain into the Mashup Platform

Injecting the domain information provided by the ecosystem into the composition environment requires some work on both the component development and the mashup platform. More precisely, it requires (i) a proper design of the components in what regards the level of abstraction and composition, (ii) making the composition environment aware (to some extent) of the business rules by defining some composition rules, and thus taking to the mashup environment what is already on the backend, and (iii) making the environment aware of the coupling among the components (and concepts managed by the components) in order to assist users in the component selection.

In what follows we describe our approach using mashArt as reference platform, albeit the ideas described here could be applied to other mashup platforms.

Building domain components. Good quality components are key to the success of any mashup platform and derived applications, and so is the case for domain-specific components. Thus, in addition to known practices for component development (e.g., [12]), building domain components puts some extra usability requirements. To reach users, we must select the right level of abstraction for the components and composition. It should be the one at which users find in the environment only concepts they know, expressed with the same terminology, i.e., components should be meaningful to the domain expert and be related to the business-level *concepts and operations*. In practice, we pass from components that represent just technology (e.g., a component connecting to a service) to components that have a precise semantic that is familiar to the domain expert (e.g., the component that publishes a paper). This is a conceptual shift.

Components should also be designed for smooth composition. Composing components should be straightforward to domain experts and complex mappings avoided to the possible extent. To this end, components events and operations I/O should be presented in terms of the domain concepts. In practical terms, this means that a domain-specific *namespace* should be made available to the component definition. Additionally, to ease and, at the same time, check the component development process, the platform could possibly provide a domain-specific component editor able to guide developers in the generation of new components

[4]http://docs.google.com/Doc?docid=0ARoLwpXLTjBGZGt6Mng0cl8yZG00N3Y4Y24&hl=en

(particularly for their descriptors), based on the knowledge coming from the *Domain Layer* (e.g., available entities and their representations).

In Listing 1, we illustrate the definition of a component designed taking into account a domain-level operation for providing familiar functionality, and domain-concepts in the I/O to ease the composition.

```
<?xml version="1.0" encoding="utf-8" ?>
<mdl version="0.1" xmlns:lj=" http://liquidjournal.org/schema/liquidjournal.xsd">
 <component name="Publish" binding="component/UI" stateful="yes"
  url="http://mashart.org/registry/X/Publish/">
  <event name="Paper published" ref="onPublish">
    <output name="Published Entity" type="lj:entity"></output>
  </event>
  <operation name="Publish paper" ref="doPublish">
    <input name="Entity" type="lj:entity"></input>
  </operation>
 </component>
</mdl>
```

Listing 1. MDL of the component Publish

As seen in the *Publish* component I/O refers to the type *entity*, an abstraction introduced in the conceptual model. A strong point of this approach is that it does not introduce any change into the MDL (mashArt Description Language, used to define each component of the mashArt platform) but it introduces higher level types based on the conceptual model. In Listing 2 we show part of the definition of the XSD used to make the mashup platform aware of the conceptual model[5].

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema ... xmlns:tns="http://liquidjournal.org/schema/liquidjournal.xsd">
...
 <xs:complexType name="entity">
  <xs:choice>
   <xs:element ref="tns:liquidjournal" />
   <xs:element ref="tns:issue" />
   <xs:element ref="tns:sciResource" />
  </xs:choice>
 </xs:complexType>
 ...
</xs:schema>
```

Listing 2. XSD Definition of domain-level concepts

Business-level composition policies. As mentioned before, business rules are important and usually enforced at the backend. In order to ensure some patterns in the output and help users in the definition of consistent mashups we believe it is important to abstract these rules and take them to the level of composition. Of course, domain rules can be very complex to be completely pushed to the composition, so we target composition policies as a tool for "assistance" rather than "enforcement". There

[5] The complete XSD can be found here https://dev.liquidpub.org/svn/liquidpub/prototype/ljdemo/server/resources/meta/liquidjournal.xsd

could be different ways of defining such policies, but in this paper we consider syntactic constraints based on category of components as simple starting strategy (that will be then extended in future). Components providing some common functionality could be categorized and category-level policies defined on them. Thus, policies allowing/denying cross-category couplings could be defined. Taking our scenario as an example, we could define categories such as *review processes*, *selection modes*, *publication modes*, *sources and venues*, *people, metrics* and *entities*, and on these define policies such as publication cannot be performed before the selection. Note that categorization is not mandatory and so "free" components not regulated by the policies are perfectly allowed. Implementation-wise, policies can be defined using XML and the mashup editor can check and guide the user during the composition based on the rules regulating that domain.

Mashup composition environment. The information about the domain components and policies should finally be reflected on the mashup composition environment. Domain components provide the opportunity to make meaningful suggestions based on components coupling (I/O matching) and so introduce a basic yet useful proactive behavior in the component selection. In its simplest, we could see composition as a domino where the domain-expert select components among the compatible ones. Domain policies provide even richer information. Policies will have higher priority over coupling based suggestions, filtering out and ranking eligible components. In addition to this, the mashup composition environment should provide intuitive UI representation for components and the connections (e.g., meaningful icons for components) to ease the selection.

Finally, having composition information at this level will enable further improvements in the composition environment. It would make it easier to extract usage information that could be used to improve the selection process (by reusing past experiences), since all components are defined at business level, making possible to extract the semantics of the compositions and usage.

5 Conclusion

In this paper we have introduced domain-specific mashups as a way to inject domain knowledge into the mashup composition, with the ultimate goal of providing domain-experts with the tools to compose mashup applications from familiar domain concepts. Our approach rather than proposing a technological change, proposes a paradigm shift in going from generic platforms with mainly low level (and technological-oriented) components to domain-specific vertical extensions. To this end, we have introduced a layered architecture in which we distinguish the mashup layer from a domain layer that can be plugged in. The definition of this domain layer is what allows us to describe the level of abstraction familiar to the user. In addition, the separation among the two layers allows the same mashup tool to be reused for different domain verticals, simply replacing the underlying domain layer.

As immediate future work we need to investigate more on how to define and model the domain characteristics (entities and rules) such that they are at the same

time useful to help the composition phase but also not too rigid, guaranteeing the needed flexibility. Then, we plan to go from the conceptual modeling to the actual implementation of the extensions to the mashup environment. In particular, we plan to do that working on the mashArt platform and on the domain of scientific publishing, as introduced in this paper.

6 References

[1] A. Jhingran. Enterprise information mashups: integrating information, simply. In VLDB '06, pages 3–4. VLDB Endowment, 2006.
[2] I. Floyd, M. Jones, D. Rathi, M. Twidale. Web Mash-ups and Patchwork Prototyping: User-driven technological innovation with Web 2.0 and Open Source Software. Proc. Of HICCS '07
[3] S. Bitzer, M. Schumann. Mashups: An Approach to Overcoming the Business/IT Gap in Service-Oriented Architectures. Proc. of AMCIS 2009.
[4] J. Wong and J. I. Hong. Making mashups with marmite: towards end-user programming for the web. Proc. of the SIGCHI '07, pag. 1435-1444, 2007.
[5] Yahoo! Pipes project. [Online] http://pipes.yahoo.com/.
[6] Intel MashMaker project. [Online] http://mashmaker.intel.com/.
[7] F. Daniel, F. Casati, B. Benatallah, M. Shan. Hosted Universal Composition: Models, Languages and Infrastructure in mashArt. Proc. of ER'09, Pages 428-443.
[8] M. Shan. Software as a Service(SaaS) The challenges of application service hosting, Proc. of Int. Conference on Web Engineering, Como, Italy, July 2007.
[9] F. Daniel, S. Soi, F. Casati. Search Computing - Challenges and Directions, edited by S. Ceri and M. Brambilla, LNCS, Volume 5950, March 2010, Springer, Pages 72-93.
[10] R. Tuchinda, P. Szekely, C. A. Knoblock. Building Mashups by example. In Proc. of the 13th international Conference on intelligent User interfaces IUI '08, p. 139-148
[11] M. Baez, F. Casati, A. Birukou, M. Marchese. Liquid journals: Knowledge dissemination in the Web Era. http://eprints.biblio.unitn.it/archive/00001814/
[12] Daniel, F., Matera, M.: Turning Web applications into mashup components: issues, models and solutions. Proc. of ICWE'2009.

Appendix H

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

Florian Daniel, Antonella De Angeli, Muhammad Imran, Stefano Soi, Chritopher 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 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 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 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.

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

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.

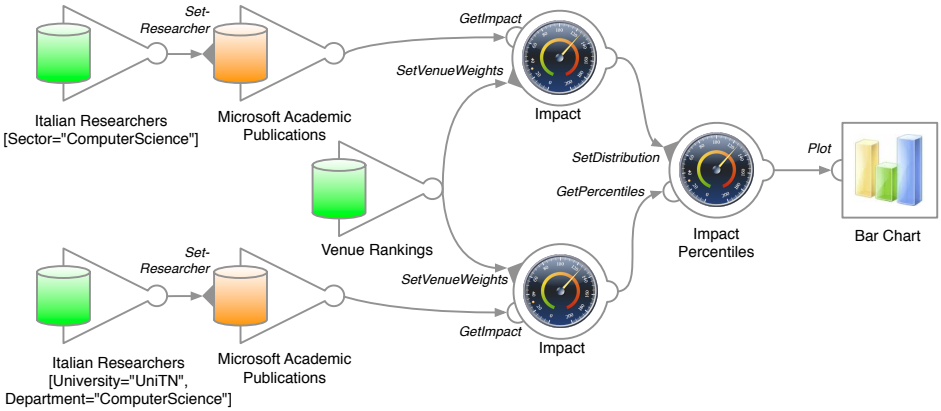


Figure. 1. University of Trento's internal researcher evaluation procedure.

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 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 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 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.

[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.

(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 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 concep-

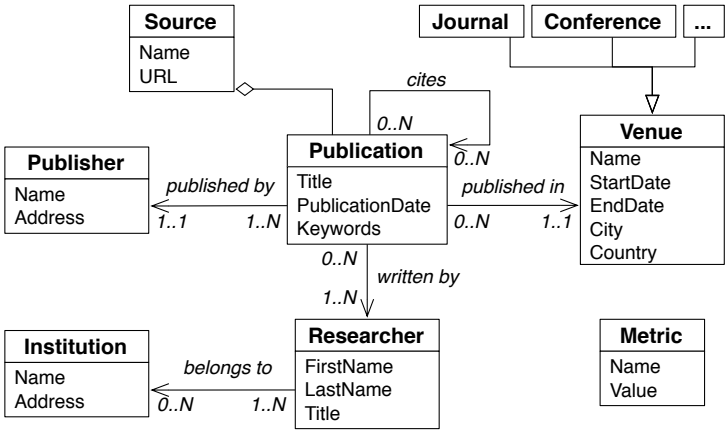


Figure. 2. Domain concept model for the research evaluation scenario

tual 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 re-

search 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

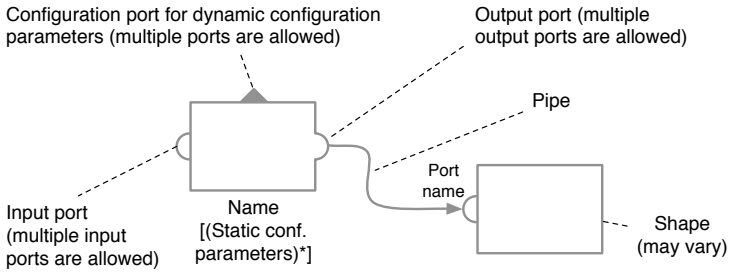


Figure. 3. Basic syntax for the concepts in the mashup meta-model.

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.

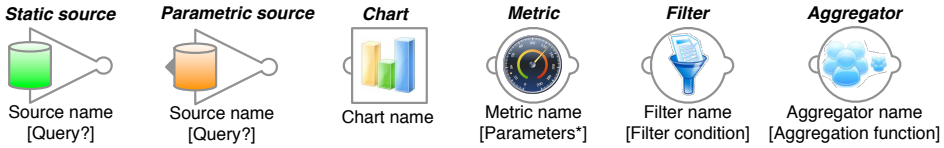


Figure. 4. Domain-specific syntax for the concepts in the domain-specific meta-model extension

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 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 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.

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 (`http://open.reseval.org/`). ***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 stated in Section 3, the implementation of ResEval Mash is based on specific design principles. These principles have not been adopted in existing mashup tools, leaving unsolved some usability issues and domain-specific needs, that we have to address in our domain. The following points present our solutions to them which are mainly driven by our past experience, careful analysis of the domain and preliminary user study's results.

(1) ***Hidden data mappings.*** As identified in [Namoun et al. 2010b], dealing with data mapping is a complex task for the users. In order to prevent them from defining data mappings, the mashup component used in the platform are all able to understand and manipulate 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 (e.g., we may need to extract and process the publications of the Italian researchers, i.e., publications of around sixtyone thousound researchers). Data processing should therefore be kept at the server side (we achieve this via dedicated RESTful web services running on the mashup server). In fact,

loading 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.

(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 (`http://pipes.yahoo.com/pipes/`).

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 typically 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 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.

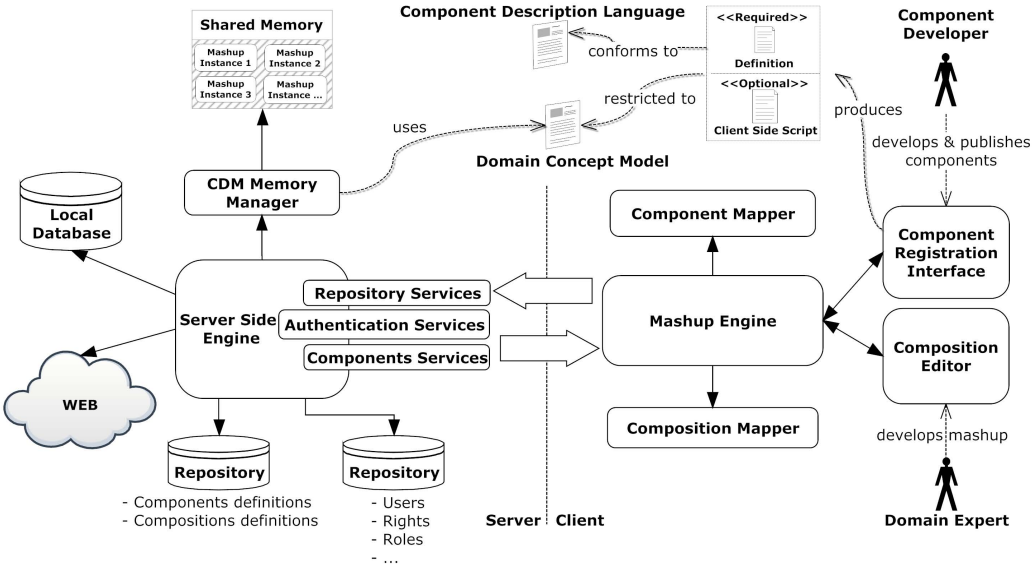


Figure. 5. Mashup Platform Architecture

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. As explained above, CDM interacts with a *shared memory* that provides a space for each mashup execution instance. As *shared memory* we decided to use a server's physical memory (RAM) area, allowing for high performance also when dealing with data-intensive processes. 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.

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 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

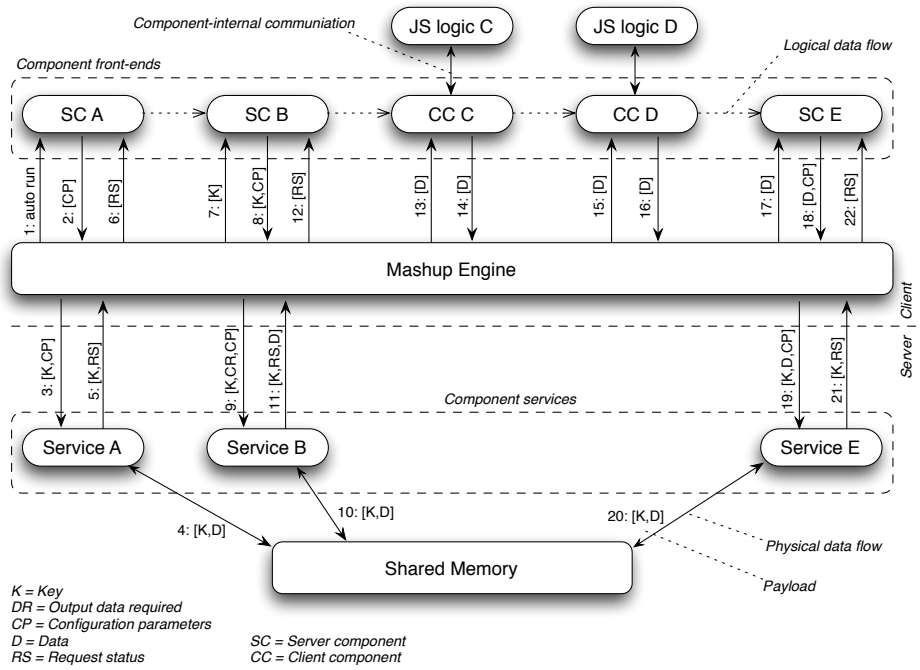


Figure. 6. ResEval Mash's internal data passing logic.

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.

(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

```
<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>
```

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

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://dev.liquidjournal.org:8081/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.

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

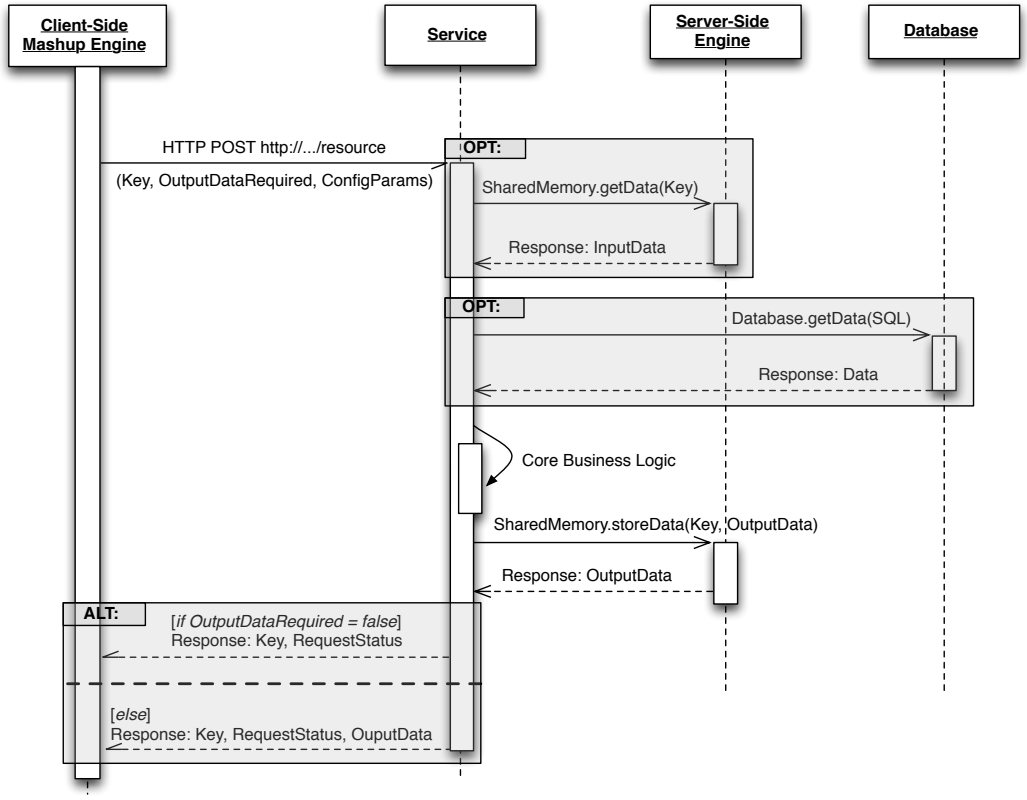


Figure. 8. Platform-specific interaction protocol each service must comly with

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

Figure. 9. Mashup composition to compute G-Index

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).

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 10.

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 completion. The interview focused on interactional difficulties experienced, the evolution of partici-

Figure. 10. Mashup composition to compute publication count

pants' 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 at . 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 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 (`http://pipes.yahoo.com`) provides an intuitive visual editor that allows the design of data processing logics. Support for UI integration is missing, and support for service integration is still poor. Pipes operators provide only generic programming features (e.g., feed manipulation, looping) and typically require basic programming knowledge. The CRUISe project [Pietschmann et al. 2009] specifically focuses on composability and context-aware presentation of UIs, but does not support the seamless integration of UI components with web services. The ServFace project (`http://www.servface.eu`), instead, aims to support 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 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 might 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 might 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.

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.

Appendix I

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

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

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

Abstract. The recent emergence of mashup tools has refueled research on *end user development*, i.e., on enabling end-users without programming skills to compose their own applications. Yet, similar to what happened with analogous promises in web service composition and business process management, research has mostly focused on technology and, as a consequence, has failed its objective. In this paper, we propose a *domain-specific* approach to mashups 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 domain (research evaluation) and describe the respective user study. The results of the user study confirm that domain-specific mashup tools indeed lower the entry barrier to mashup development.

1 Introduction

Mashups are typically simple web applications that, rather than being coded from scratch, are developed by integrating and reusing available data, functionalities, or pieces of user interfaces accessible over the Web. ***Mashup tools***, i.e., online development and runtime environments for mashups, ambitiously aim at enabling non-programmers to develop their own applications. The mashup platforms developed so far either expose too much functionality and too many technicalities, so that they are powerful and flexible but suitable only for programmers, or only allow compositions that are so simple to be of little use for most practical applications. Yet, being amenable to non-programmers is increasingly important, as the opportunity given by the wide range of applications available online and the increased flexibility that is required in both businesses and personal life management raise the need for situational applications.

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. 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.

As an example, in this paper we report on a mashup platform we specifically developed for the domain of research evaluation, that is, for the assessment of the performance of researchers, groups of researchers, departments, universities, and similar. There are no commonly accepted criteria for performing such analysis in general, and evaluation is highly subjective. Computing evaluation metrics that go beyond the commonly adopted h-index is still a complex, manual task that is not adequately supported by software instruments. In fact, computing an own metric may require extracting, combining, and processing data from multiple sources, implementing new algorithms, visually representing the results, and similar. In addition, the people involved in research evaluation are not necessarily IT experts and, hence, they may not be able to perform such IT-intensive tasks without help. In fact, we may need to extract, combine, and process data from multiple sources and render the information via visual components, a task that has all the characteristics of a data mashup.

In this paper, we 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. Specifically, (1) we provide a *methodology* for the development of domain-specific mashup tools, defining the necessary concepts and design artifacts; (2) we detail and exemplify all *design artifacts* that are necessary to implement a domain-specific mashup tool; (3) we apply the methodology in the context of an *example mashup platform* that aims to support research evaluation, (4) we perform a *user study* in order to assess the viability of the developed platform.

Next we outline the methodology we follow to implement the domain-specific mashup tool. In Section 3 we briefly describe the actual implementation of our prototype tool, and in Section 4 we report on our preliminary user study. In Section 5, we review related works. We conclude the paper in Section 6.

2 Methodology

Our development of a specific mashup platform for research evaluation has allowed us to conceptualize the necessary tasks and to structure them into the following ***methodology*** steps:

1. Definition of a *domain concept model* (CM) to express domain data and relationships. The domain concepts tell the mashup platform what kind of *data objects* it must support. This is different from generic mashup platforms, which provide support for generic data formats, not specific data objects.
2. Identification of a generic *mashup meta-model* (MM) that suits the composition needs of the domain. A variety of different mashup approaches, i.e.,

meta-models, have emerged over the last years and before focusing about domain-specific features, it is important to identify a meta-model that accommodates the domain processes to be mashed up.

3. Definition of a *domain-specific mashup meta-model*. Given a generic MM, the next step is understanding how to inject the domain into it. We approach this by specifying and developing:
   (a) A *domain process model* (PM) that expresses classes of domain activities and, possibly, ready processes. Domain activities and processes represent the dynamic aspect of the domain.
   (b) A *domain syntax* that provides each concept in the domain-specific mashup meta-model (the union of MM and PM) with its own symbol. Domain concepts and activities must be represented by visual metaphores conveying their meaning to domain experts.
   (c) A set of *instances of domain-specific components*. This is the step in which the reusable domain-knowledge is encoded, in order to enable domain experts to mash it up into new applications.
4. *Implementation* of the domain-specific mashup tool (DMT) as a tool whose expressive power is that of the domain-specific mashup meta-model and that is able to host and integrate the domain-specific activities and processes.

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

2.1 The Domain Concept Model

The ***domain concept model (CM)*** is obtained via interactions between an IT expert and a domain expert. We represent it as ER diagram or XSD schema. It describes the *conceptual entities* and the *relationships* among them, which, together, constitute the domain knowledge. For example in the chosen domain we have *researchers, publications, conferences, metrics, etc.* 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, and printed by a *publisher*. It is written by one or more *researchers* belonging to an *institution*.

2.2 The Generic Mashup Meta-Model

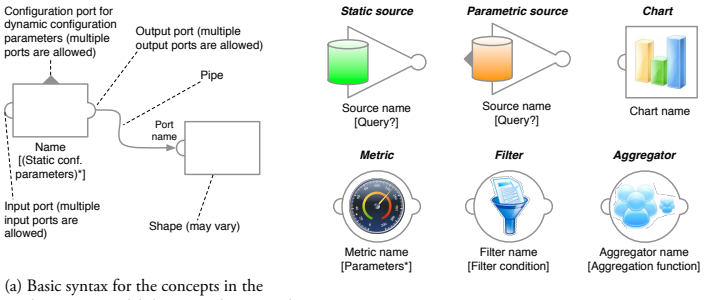
We first define a generic mashup meta-model, which may fit a variety of different domains, then we show how to define the domain-specific mashup meta-model, which will allow us to draw domain-specific mashup models. Specifically, the generic ***mashup meta-model (MM)*** specifies a *class* of mashups and, thereby, the *expressive power*, i.e., the concepts and composition paradigms, a mashup platform must know in order to support the development of that class of mashups. Thus the MM implicitly specifies the expressive power of the mashup platform class. Identifying the right features of the mashups that fit a given domain is therefore crucial. For our domain, we start from a very simple MM, both in terms of notation and execution semantics, which enables end-users to model their own mashups. Indeed, it can be fully specified in one page:

- A ***mashup*** $m = \langle C, P, VP, L \rangle$, consists of a set of *components* C, a set of data *pipes* P, a set of view ports VP that can host and render components with own UI, and a *layout* L that specifies the graphical arrangement of components.
- A ***component*** $c = \langle IPT, OPT, CPT, type, desc \rangle$, where $c \in C$, is like a task that performs some data, application, or UI action. Components have *ports* through which pipes are connected. Ports can be divided in *input* (IPT) and *output* ports (OPT), where input ports carry data into the component, while output ports carry data generated 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*.*Configuration* ports (CPT) are used to configure the components. They are typically used to configure filters 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, or *application components*, which either fetch or process information. Components can also have a description *desc* at an arbitrary level of formalization, whose purpose is to inform the user about the data the components handle and produce.
- A ***pipe*** $p \in P$ carries data (e.g., XML documents) between the ports of two components, implementing a data flow logic. So, $p \in IPT \times (OPT \cup CPT)$.
- A ***view port*** $vp \in VP$ identifies a place holder, e.g., a DIV element or an IFRAME, inside the HTML template that gives the mashup its graphical identity. Typically, a template has multiple place holders.
- Finally, the ***layout*** L defines which component with own UI is to be rendered in which view port of the template. Therefore $l \in C \times VP$.

In the model above there are *no variables* and *no data mappings*. This is at the heart of enabling end-user development as this is where much of the complexity resides. It is unrealistic to ask end-users to perform data mapping operations. Because there is a CM, each component is required to be able to process any document that conforms to the model.

The ***operational semantics*** of the MM is as follows: execution of the mashup is *initiated* by the user. All the 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. All ready components are *executed*. They process the data in input ports, consuming the respective data items form the input feed, and generate output on their output ports. The execution proceeds by identifying ready components and executing them, until there are no components to be executed left.

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



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

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

Fig. 1. Generic and domain-specific syntax for research evaluation

MM. In Figure 1(a) we map the above MM to a basic set of generic graphical symbols and composition rules. In the next section, we show how to configure domain-specific symbols.

2.3 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 into different domains. Thus, it is not yet tailored to a specific domain. Now we want to push the domain into the mashup meta-model. 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. What we obtain from this is a *domain-specific mashup meta-model*. Each domain-specific meta-model is a specialization of the mashup meta-model along three dimensions: (i) domain-specific activities and processes, (ii) domain-specific syntax, and (iii) domain instances.

The ***domain process model (PM)*** describes the classes of *processes* or *activities* that the domain expert may want to mash up to implement composite, domain-specific processes. Operatively, the PM is again derived by specializing the generic meta-model based on interactions with domain experts. 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: *source extraction*, *metric computation*, *filtering*, and *aggregation* activities.

A possible ***domain-specific syntax*** for the classes in the PM is shown in Figure 1(b). Its semantic is the one described by the MM in Section 2.2.

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

3 The ResEval Mash Tool

The ResEval Mash platform is composed of two parts, i.e., client side and server side. The heart of the platform is the *mashup execution engine* on the client side, which support client-side processing, that is, it controls data processing on the server from the client. The engine is responsible for running a mashup composition, triggering the component's actions and managing the communication between client and server. The client side *composition editor* (shown in Figure 2) provides the mashup canvas and a list of components 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*. The platform also comes with a *component registration interface* for developers to set up and configure new components for the platform. On the server side, we have a set of RESTful web services, i.e., the *components services*, *authentication services*, *components and composition repository services*, and *shared memory services*. Components services allow the invocation of those components whose business logic is implemented as a server-side web service. These web services, together with the client-side components, implement the *domain process model*. Authentication services are used for user authentication and authorization. Components and composition repository services enable CRUD operations for components and compositions. Shared memory services provide an interface for external web services (i.e., services which are not deployed on our platform) to use the shared memory. The *shared memory manager* provides and manages a space for each mashup execution instance on the server side. The *common data model (CDM) module* implements the *domain concept model* (CM) and supports the checking of data types in the system. CDM configures itself using an XSD (i.e., an XML schema representing domain concept model). All services are managed by a *server side engine*, which fulfills all requests coming from the client side. A demo of ResEval Mash is described in [3] and a prototype is available online at `http://open.reseval.org/`.

4 User Study and Evaluation

In order to evaluate our domain-specific mashup approach, we conducted a user study with 10 users. Participants covering a broad range of domain and technical expertise were invited to use ResEval Mash. At the beginning participants were asked to fill in a questionnaire reporting their computing skills and to watch a video tutorial followed by a set of tasks to complete.

Overall, the tool was deemed to be usable and the participants were comfortable using it. 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 was a different level of confidence in task execution. IT experts appeared to be more confident during the test. The results of our study indicate real potential for the domain-specific mashup approach to allow people with no computing skills to create their own applications. The definition of the

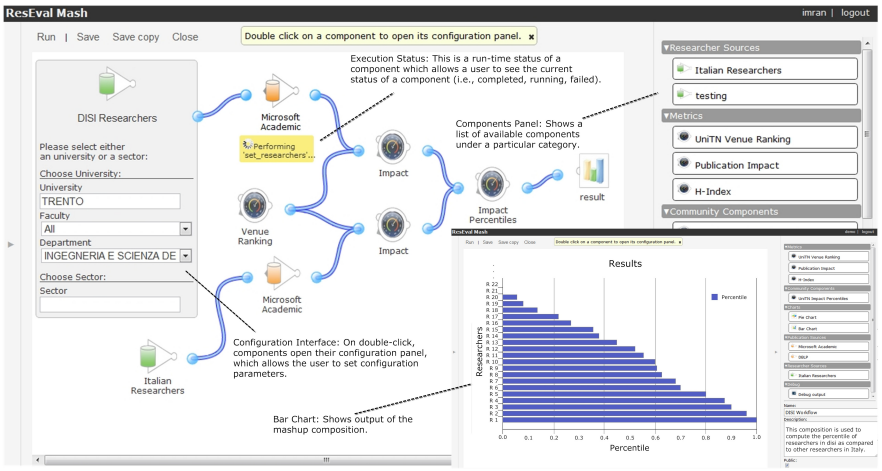


Fig. 2. Composition editor and example mashup output.

mappings among the components, which is a well-acknowledged problem known form several user studies of EUD tools [6], did not occur at all in the our study. This preliminary study suggests that ResEval Mash is a successful tool appealing to both expert programmers and end-users with no computing skills.

5 Related Work

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 [5, 1]. 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. 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 [4]). 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 [1] and [5].

Web mashups [8] have emerged as an approach to provide easier ways to connect together services and data sources available on the Web [2], together with the claim to target non-programmers. Yahoo! Pipes (`http://pipes.yahoo.com`), for instance, 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 while it provides only generic programming features (e.g., feed manipulation, looping) and typically require basic programming knowledge. The CRUISe project [7] specifically focuses on composability and context-aware presentation of UIs, but does not support the seamless integration of UI components with web services. The ServFace project (`http://www.servface.eu`), instead, aims to support normal web users in composing semantically annotated web services. The result is a simple, user-driven web service orchestration tool, but UI integration and process logic definitions are rather limited and again basic programming knowledge is still required.

6 Status and Lessons Learned

The work described in this paper resulted from actual needs within our university and within the context of an EU project, which were not yet met by current technology. It also resulted from the observation that in general composition technologies failed to a large extent to strike the right balance between ease of use and expressive power. They define seemingly useful abstractions and tools, but in the end developers still prefer to use (textual) programming languages, and, at the same time, domain experts are not able to understand and use them. What we have pursued in our work is, in essence, to constrain the language to the domain (but not in general in terms of expressive power) and to provide a domain-specific notation so that it becomes easier to use and in particular does not require users to deal with one of the most complex aspect of process modeling (at least for end-users), that of data mappings.

References

1. R. France and B. Rumpe. Domain specific modeling. *Software and Systems Modeling*, 4:1–3, 2005.
2. B. Hartmann, S. Doorley, and S. Klemmer. Hacking, Mashing, Gluing: A Study of Opportunistic Design and Development. *Pervasive Computing*, 7(3):46–54, 2006.
3. M. Imran, F. Kling, S. Soi, F. Daniel, F. Casati, and M. Marchese. ResEval Mash: A Mashup Tool for Advanced Research Evaluation. In *Proceedings of WWW 2012*, pages 361–364.
4. E. M. Maximilien, H. Wilkinson, N. Desai, and S. Tai. A domain-specific language for web apis and services mashups. In *ICSOC*, pages 13–26, 2007.
5. M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.
6. A. Namoun, T. Nestler, and A. De Angeli. Service Composition for Non Programmers: Prospects, Problems, and Design Recommendations. In *Proceedings of ECOWS*, pages 123–130. IEEE, 2010.
7. S. Pietschmann, M. Voigt, A. Rümpel, and K. Meißner. Cruise: Composition of rich user interface services. In *Proceedings of ICWE'09*, pages 473–476. 2009.
8. J. Yu, B. Benatallah, F. Casati, and F. Daniel. Understanding Mashup Development. *IEEE Internet Computing*, 12:44–52, 2008.

Appendix J

ResEval Mash: A Mashup Tool for Advanced Research Evaluation

Muhammad Imran, Felix Kling, Stefano Soi, Florian Daniel,
Fabio Casati and Maurizio Marchese,
University of Trento, Via Sommarive 5, 38123, Trento, Italy
lastname@disi.unitn.it

ABSTRACT

In this demonstration, we present ResEval Mash, a mashup platform for *research evaluation*, i.e., for the assessment of the productivity or quality of researchers, teams, institutions, journals, and the like – a topic most of us are acquainted with. 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. Our preliminary user study shows that ResEval Mash indeed has the power to enable *domain experts* to develop own mashups (research evaluation metrics); other mashup platforms rather support skilled developers. The reason for this success is ResEval Mash's domain-specificity.

Categories and Subject Descriptors

H.m [**Information Systems**]: Miscellaneous; D.1 [**Software**]: Programming Techniques

Keywords

Mashup, Domain-Specific Mashup, End-User Development, Research Evaluation

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. ***Mashup tools***, i.e., online development and runtime environments for mashups, typically aim to enable also non-programmers to develop own applications. Yet, similar to what happened in web service composition, the mashup platforms developed so far either expose too much functionality and too many technicalities so that they are powerful and flexible but suitable only for programmers, or only allow compositions that are so simple to be of little use for most practical applications. For example, mashup tools typically come with SOAP services, RSS feeds, UI widgets, and the like. Non-programmers simply do not know how to use these and what to do with them.

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. Giving up generality means narrowing the focus of a design tool to a well-defined *domain* and tailoring its development paradigm, models, language and components to the specific needs of that domain only.

Domain-specific development instruments are traditionally the object of domain-specific modeling (DSM) [4] and domain-specific languages (DSLs) [5], yet they typically target developers, with only few exceptions. Costabile et al. [1], for instance, successfully implemented a DSM-based tool enabling end user development in the context of a specific company and technological framework. Given the huge technological diversity on the Web, however, mashup tools are still too complex, and non-programmers are not able to manipulate the provided compositional elements [6] (e.g., Yahoo! Pipes comes with web services, RSS feeds, regular expressions, and the like). Web service composition approaches like BPEL are completely out of reach.

In this paper we present ***ResEval Mash*** i.e., a domain-specific mashup tool, which we specifically developed for the domain of research evaluation. The development of complex evaluation metrics that go beyond the commonly adopted h-index is usually still a complex and manual task that is not adequately supported by software instruments. In fact, computing an own metric might mean extracting, combining, and processing data from multiple sources, implementing new algorithms, visually representing the results, and similar. The *Web of Science* (`http://scientific.thomson.com/products/wos/`) by Thomson Scientific, *Publish or Perish* (`http://www.harzing.com/pop.htm`), or *Google Scholar* do provide some basic metrics, such as the h-index or g-index; but they are not able to satisfy more complex evaluation logics. The goal of ResEval Mash is therefore to enable *domain experts* (typically non-programmers) to define and run their own evaluation metrics in a simple and rapid way, leveraging on suitably tailored *mashup technology*.

For explanation purpose, throughout this paper we will use a specific scenario, which we introduce in the next section. However, the tool is not restricted to one scenario only and rather aims at allowing users to develop their own

WWW 2012 Companion, April 16–20, 2012, Lyon, France.
ACM 978-1-4503-1230-1/12/04.

mashups addressing a variety of different scenarios in the research evaluation domain.

2. EXAMPLE SCENARIO

As an example of a concrete research evaluation task, let's look at the procedure used by the central administration of the University of Trento (UniTN) to assess the productivity of the researchers of its departments. The evaluation is used to allocate resources and funding for the university departments. In essence, the algorithm compares the scientific production of each researcher in a given department of UniTN with the average production of the researchers belonging to similar departments (i.e., departments in the same disciplinary sector) in all Italian universities. The comparison uses a procedure based on a simple bibliometric indicator, i.e., a weighted publication count metric:

1. A list of all researchers working in Italian universities is retrieved, and a reference sample with similar statistical features 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, or similar).
3. The obtained publications are weighted using a venue classification provided by a UniTN committee, which is split into three quality categories based on the ISI Journal Impact Factor. For each researcher, a weighted publication count is obtained with a simple weighted sum of his/her publications.
4. A statistical distribution – more specifically, a negative binomial distribution – of the weighted publication count metrics is then computed for the national researcher reference sample.
5. Each researcher of the selected department is then ranked based on his/her individual weighted publication count, estimating his/her percentile in the derived statistical distribution, i.e., the percentage of the researchers in the same disciplinary sector that have the same or lower values for the specific metric.

The percentile for each researcher in the selected department is used as the parameter that estimates the publishing profile of that researcher and is used for the 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.

Many factors can significantly impact on the results of this evaluation process (e.g., the data sources or the sampling criteria), and people (e.g., administrative employees and researchers) want to check the results of different possible metrics. If manually done, this would cost too much time and human resources. The task, however, has all the characteristics of a *mashup*, especially if the mashup logic comes from the users.

3. THE RESEVAL MASH TOOL

The above scenario, the domain, and our target user group, i.e., domain experts, pose a set of peculiar requirements to the development of the ResEval Mash tool. In the following we summarize the design principles that underlie ResEval Mash and where the domain specifics come into play. Then, we provide insights into its internals and implementation.

3.1 Principles and Requirements

ResEval Mash is based on the following principles and requirements:

1. ***Intuitive graphical 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 mappings.*** Our experience with prior mashup platforms, i.e., mashArt [2] and MarcoFlow [3], 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.
4. ***Runtime transparency.*** In order to convey to the user what is going on during the execution of a mashup especially when it takes several seconds, we provide transparency into the state of a running mashup. We identify two key points where transparency is important in the mashup model: components and processing state. At each instant of time during the execution, the runtime environment should allow the user to inspect the data processed and produced by each component. In addition, to convey the processing state of each component and thus the mashup model the environment should graphically show the state.
5. ***Data-intensive processes.*** Although apparently simple, the chosen domain is peculiar in that it may require the processing of large amounts of data (e.g., we need to extract all the publications of the Italian researchers' sample for a given scientific sector). While runtime transparency is important at the client side, data processing should however be kept at the server side. In fact, loading 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.

3.2 The Domain

Some of the above requirements 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 (`http://pipes.yahoo.com/pipes/`). We achieve domain-specificity as follows:

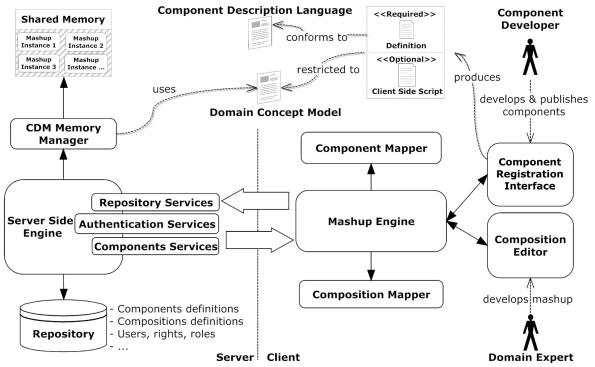


Figure 1: The ResEval Mash architecture

To provide users with a mashup environment that has an *intuitive graphical UI* we design first a ***domain syntax***, which provides each object in the composition environment with a visual metaphor that the domain expert is acquainted with and that visually convey the respective functionalities. For instance, ResEval Mash uses a gauge for metrics and the icons that resemble the chart types of graphical output components.

The core of the platform are the *functionalities* exposed to the domain expert in the form of *modeling constructs*. These must address the specific domain needs and cover as many as possible mashup scenarios inside the chosen domain. To design these constructs, a thorough analysis of the domain is needed, so as to produce a so-called ***domain process model***, which specifies the classes of domain activities and, possibly, ready processes that are needed (e.g., data sources and metrics). Next, a set of ***instances of domain activities*** (e.g., an h-index algorithm) must be implemented, which can be turned into concrete mashup components.

Finally, in order to relieve users from the definition of data mappings, ResEval Mash is based on an explicit ***domain concept model***, which expresses all domain concepts and their relationships. If all instances of domain activities understand this domain concept model and produce and consume data according to it, we can omit data mappings from the composition environment in that the respective components simply know how to interpret inputs.

3.3 Architecture and Implementation

Figure 1 shows the architecture of ResEval Mash, which is divided into two parts, i.e., client side and server side.

The ***mashup engine*** is the most important part of the platform. It is developed for client-side processing, that is, we control data processing on the server from the client. The engine is primarily responsible for running a mashup composition, triggering the component's actions, and managing the communication between client and server. The engine provides for data flow processing. The ***composition editor*** provides the mashup design canvas to the user. It shows a components list, from which users can drag and drop components onto the canvas in order to connect them. The editor implements the domain syntax. From the editor, it is also possible to launch the execution of a composition through a run button and hand the mashup over to the *mashup engine* for execution. The composition editor and its various parts are shown in Figure 3. ***Component and composition mappers*** parse component and composition descriptors to represent them in the *composition editor* at design time and to bind them in the *engine* at run time.

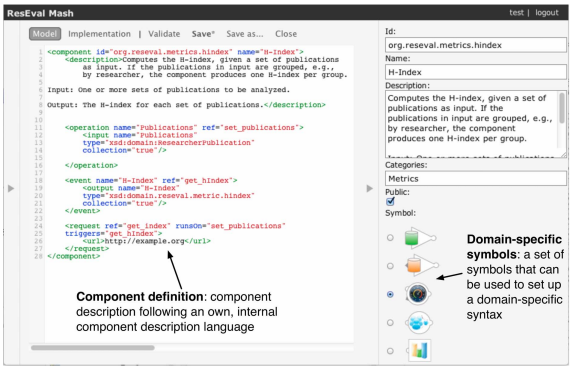


Figure 2: Component registration interface

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. As shown in Figure 2, 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 a web service, for server-side components, which is linked by the component definition.

The whole client-side part of ResEval Mash is developed in JavaScript, using the *Google Closure* and *WireIt* libraries.

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 web service. These web services, together with the client-side components, implement the *instances of domain activities* inside the *domain process model*. The ***common data model (CDM)*** implements the *domain concept model* and supports the checking of data types in the system. The CDM is a *shared memory* that provides a space for each mashup instance. All data processing services read and write to this shared memory. In order to configure the CDM, the *CDM memory manger* generates corresponding Java classes (e.g., in our case POJO classes, annotated with JAXB annotations) from an XSD that encodes the domain concept model. The ***server-side engine*** 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 fulfilled.

In Figure 3, we illustrate the final mashup model of our research evaluation scenario as developed with ResEval Mash. 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 the Computer Science disciplinary sector. The other computing the same "impact" metric for the researchers belonging to UniTN Computer Science department. The first branch defines the dis-

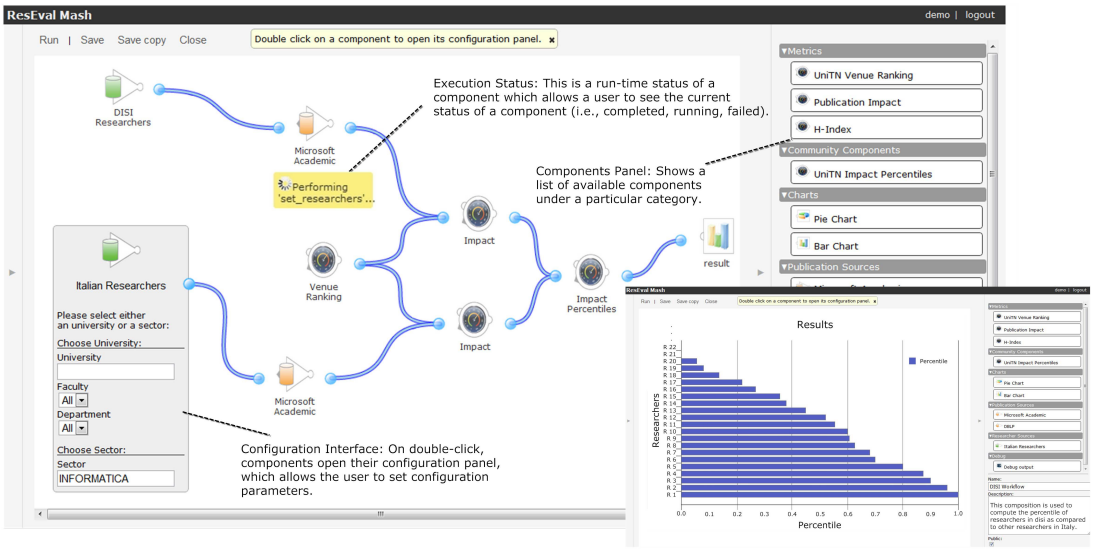


Figure 3: ResEval Mash in action: screen shots of the modeling canvas and the final mashup output

tribution of the Italian researchers for the Computer Science disciplinary sector, while the second branch is used to compute the impact value of UniTN's researchers and to determine their individual percentiles, which are finally visualized in a bar chart (clearly, we anonymized the respective data).

4. DEMONSTRATION STORYBOARD

The live demo will be presented starting from an introduction of the reference domain (i.e., research evaluation) and the motivation behind the implementation of ResEval Mash. A guided walk-through the tool will be presented to introduce the modeling paradigm of the tool. We will compose a few example scenarios and describe the various features provided by the tool. After this, we will ask the audience to try the tool and to develop their own simple research evaluation mashups. Finally, the platform architecture will be presented to highlight the various aspects of the tool.

A screencast and a continuously updated prototype of ResEval Mash is available at `http://open.reseval.org/`.

5. EVALUATION AND LESSONS LEARNED

ResEval Mash stems from the actual needs in our university and from our own needs in term of research evaluation. It also results 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 domain experts are still not able to understand and use them. What we have pursued in the development of ResEval Mash, in essence, is to constrain the language to the domain and to provide a domain-specific notation so that it becomes easier to use and in particular does not require users to deal with one of the most complex aspects of process modeling (at least for end users), that of data mappings.

We have performed a *user study* of ResEval Mash with 10 users (5 with and 5 without IT skills and with different domain expertise). Participants were asked to fill in a questionnaire about their computing and research evaluation skills before the test, to watch a video tutorial about ResEval Mash, and to use the tool, while being filmed.

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. A major finding is related to the ease with which our sample understood that components had to be linked together so that information could flow between different services. This is a well-acknowledged problem evinced in several user studies of EUD tools (e.g., [6]), which did not occur at all in the current study. To a large extent, this result can be achieved thanks to the fact that ResEval Mash relieves users from the definition of data mappings.

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

6. REFERENCES

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

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

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

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

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

[6] A. Namoun, T. Nestler, and A. De Angeli. Service Composition for Non Programmers: Prospects, Problems, and Design Recommendations. In *Proceedings of ECOWS*, pages 123–130. IEEE, 2010.

Appendix K

Developing Domain-Specific Mashup Tools for End Users

Florian Daniel, Muhammad Imran, Felix Kling, Stefano Soi,
Fabio Casati and Maurizio Marchese
University of Trento, Via Sommarive 5, 38123, Trento, Italy
lastname@disi.unitn.it

ABSTRACT

The recent emergence of mashup tools has refueled research on *end user development*, i.e., on enabling end users without programming skills to compose 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.

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 apply the approach implementing a mashup tool for the research evaluation domain. Our user study confirms that domain-specific mashup tools indeed lower the entry barrier to mashup development.

Categories and Subject Descriptors

H.m [**Information Systems**]: Miscellaneous; D.1 [**Software**]: Programming Techniques

Keywords

Domain-Specific Mashups, End-User Development

1. INTRODUCTION

Mashups are typically simple web applications that, instead of being coded from scratch, are developed by integrating and reusing available data, functionalities, or pieces of user interfaces accessible over the Web. ***Mashup tools*** aim at enabling non-programmers (web users) to develop own applications. Yet, similar to what happened in service composition, the mashup platforms developed so far either expose too much functionality and too many technicalities so that they are powerful and flexible but suitable only for programmers, or they only allow compositions that are so simple to be of little use for most practical applications.

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. 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, therefore creating a ***domain-specific mashup tool***.

Domain-specific development instruments are traditionally the object of domain-specific modeling (DSM) [2] and domain-specific languages (DSLs) [4], yet they typically target developers, with only few exceptions. Costabile et al. [1], for instance, successfully implemented a DSM-based tool enabling end user development in the context of a specific company and technological framework. Given the huge technological diversity on the Web, however, mashup tools are still too complex, and non-programmers are not able to manipulate the provided compositional elements [5] (e.g., Yahoo! Pipes comes with web services, RSS feeds, regular expressions, and the like). Web service composition approaches like BPEL are completely out of reach.

In this poster, we describe a ***methodology*** for the development of domain-specific mashup tools, defining the necessary concepts and design artifacts. The methodology targets expert developers, who implement mashup tools. We show how we used the methodology to implement a ***mashup platform*** for research evaluation. The platform targets domain experts (e.g., scientists). Finally, we report on our ***user study***, which confirms the viability of the developed platform and of the respective development methodology.

2. METHODOLOGY

Reverse-engineering our experience with the implementation of the mashup platform described in the next section, developing a domain-specific mashup platform requires:

1. Definition of a ***domain concept model*** (CM) to express domain data and relationships, which allow 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*** (MM) that suits the composition needs of the domain. A variety of different mashup approaches, i.e., meta-models, have emerged over the last years, (e.g., data, user interface and process mashups).

WWW 2012 Companion, April 16–20, 2012, Lyon, France.
ACM 978-1-4503-1230-1/12/04.

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 ***domain process model*** (PM) that expresses classes of domain activities and, possibly, ready processes (which we can map to reusable components of the platform). Domain activities and processes represent the dynamic aspect of the domain. They operate on and manipulate the domain concepts.

   A ***domain syntax*** that provides each concept in the domain-specific mashup meta-model (the union of MM and PM) with an own symbol that conveys the respective functionality to domain experts.

   A set of ***instances of domain-specific components***. This is the step in which the reusable domain-knowledge is encoded in the form of components in order to enable domain experts to mash it up into new applications.

4. ***Implementation*** of the domain-specific mashup tool (DMT) whose expressive power is that of the domain-specific mashup meta-model.

3. THE RESEVAL MASH TOOL

ResEval Mash [3] is a mashup platform (a DMT) 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 needs 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 running inside a common web browser.

Developing ResEval Mash required addressing the specific requirements coming from the research evaluation domain. The first step to characterize this domain was the definition of a suitable *domain concept model* (CM). Research evaluation deals with publications, researchers, conferences, journals, metrics (e.g., h-index or citation counts), and so on. We encoded a respective CM in a suitable XML schema.

Next, composing the above concepts into a new, complex evaluation logic in essence means processing data (next to visualizing the output graphically). As *generic mashup meta-model* we therefore chose a data flow based meta-model, which focuses the attention of the user to the passing of data (e.g., publications) from one computing step to another.

Turning this meta-model into a *domain-specific mashup meta-model* then required selecting a set of abstract domain activities, i.e., defining the *domain process model*. Here we have identified data source extraction activities (e.g., for Google Scholar or Scopus), metric computation activities (e.g., h-index, g-index), aggregation and filtering activities, and finally visualization activities (e.g., UI widgets). After that, we implemented a set of *instances of domain-specific components* for the identified domain activities. For instance, we developed a Google Scholar and a Microsoft Academic data component, a h-index component, a citation count component, a filter component, a bar chart component for the visualization of metrics, and others. We then equipped these components with a *domain syntax* that clearly distinguished between data sources, metrics, filters and visualization components. Figure 1 shows an example mashup in ResEval Mash.

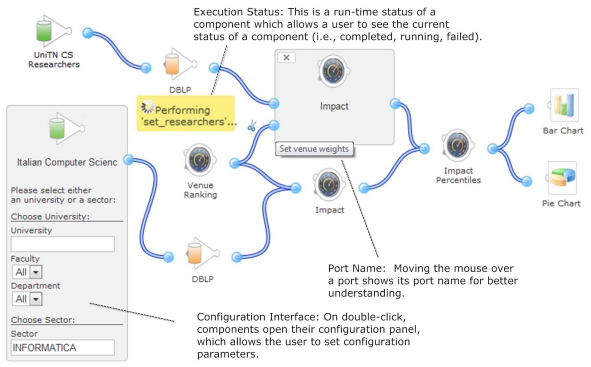


Figure 1: ResEval Mash in action

4. EVALUATION AND LESSON LEARNED

We have performed a *user study* of ResEval Mash with 10 users (5 with and 5 without IT skills and with different domain expertise). Participants were asked to fill in a questionnaire about their computing and research evaluation skills before the test, to watch a video tutorial about ResEval Mash, and to use the tool. This interaction was filmed, as was the interview that followed task completion. The results of the user study show that end users indeed 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 skills step by step by simply "playing" with ResEval Mash.

With ResEval Mash, we constrain the mashup language to a single domain and the mashup components to the domain's concept model. While this might be an additional burden on the component developer, it allows us to shield the user from one of the most complex aspects of mashups, i.e., data mappings. Users only need to think about the data flow, then the components know themselves which data to use. This is a very simple, but powerful simplification.

5. REFERENCES

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

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

[3] M. Imran, F. Kling, S. Soi, F. Daniel, F. Casati, and M. Marchese. ResEval Mash: Advanced Research Evaluation for Domain Experts. In *WWW'12*, 2012.

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

[5] A. Namoun, T. Nestler, and A. De Angeli. Service Composition for Non Programmers: Prospects, Problems, and Design Recommendations. In *Proceedings of ECOWS*, pages 123–130. IEEE, 2010.

Appendix L

From Mashups to Telco Mashups: A Survey

Hendrik Gebhardt, Martin Gaedke
Chemnitz University of Technology, Germany

Florian Daniel, Stefano Soi, Fabio Casati
University of Trento, Italy

Carlos A. Iglesias
Universidad Politecnica de Madrid, Spain

Scott Wilson
University of Bolton, UK

Abstract. In this article, we aim at understanding a type of mashups that has been gaining attention recently, i.e., telco mashups, and that, given its novel requirements and characteristics, deserves an analysis that goes beyond what has been done so far for mashups in general. We cluster telco services into different types, analyze their features, derive a telco mashup reference architecture, and survey to which extent existing mashup tools are able to respond to the novel needs, an exercise that finally allows us to identify a set of open and challenging research questions.

Introduction

Web mashups are web applications developed by integrating data, application logic, and/or pieces of user interfaces (UIs) sourced from the Web [1]. A typical example is the housingmaps.com application (http://www.housingmaps.com), a mashup that integrates housing offers from craigslist.com with a Google map. Although mashups are mostly coded manually, so-called *mashup tools* or *platforms* aim at mashup development paradigms that do not require programming skills and, hence, also target end users. Yet, the scope of the instruments conceived so far is typically very broad and technology- centric, which limits their capability to cater for *domain-specific* features and needs when it comes to the development of concrete applications.

For instance, interconnecting people, possibly via a variety of different channels, such as voice, video, or instant messaging in both fixed and mobile settings, is still a hard and time-consuming endeavor - if feasible at all. In fact, the peculiarities of the *telecommunication* (telco) domain, which specifically focuses on the transmission of data to enable *communication* and *collaboration* among people, have not percolated into existing mashup tools. Features like multi-device deployment, audio/video streaming, distributed session management, live collaboration, and similar are, for example, not supported in an integrated fashion and, hence, available for the general public. The same holds for the key non-functional requirement in telco, i.e., *quality of service* (QoS).

We believe that one of the main reasons for this weak support for telco features in mashups is the general *lack of understanding* of what telco mashups actually are and of how they can be developed. In order to foster research in this area and advance current mashups toward the telco domain, in this article we (i) introduce the necessary concepts and terminology, (ii) review the state of the art in telco services, (iii) derive a reference architecture for mashup platforms, (iv) compare it with existing

mashup platforms, and (v) identify a set of challenges and open research questions, specifically addressing those aspects that are proper of the telco domain.

Scenario and Challenges

For a better understanding of how a telco mashup could look like, let's consider the following application scenario. Several consultants from a multinational consultancy firm are discussing the technical architecture for a project proposal they are elaborating. They use a corporate collaborative environment consisting of a multi-channel web application that integrates the necessary telco functionalities and a shared whiteboard. All participants are connected via different clients: Marco via a smart phone using a mobile web browser, Steve via a desktop web browser, Jürgen via a tablet using a mobile web browser, and Maria via a traditional mobile phone using the phone's built-in capabilities.
The first three consultants are using web-based instant messaging while the latter is using SMS messaging. The collaborative environment provides a telco mashup for combining these two communication channels with the whiteboard. After a while, they decide to switch to voice, using a facility of the collaborative environment based on a dedicated Voice over IP service. Maria can be called by the environment or dial-in to the ongoing session. Since Maria cannot draw with her phone, she sketches her ideas on paper and sends a photo taken with her phone's built-in camera via MMS to the telco mashup, which renders it to the rest of the consultants.

The described scenario is rather complex, and supporting it via a dedicated mashup requires support from a mashup platform that is telco-ready. Devising such a platform is non-trivial and requires a thorough understanding of the nature of both *telco services/APIs* and *telco mashups*.

Understanding Telco Services & Device APIs

Analyzing the scenario, we see that some of the features require interacting with remote *software services* (functionalities accessed via the Internet using message exchanges complying with a protocol) providing telco support (e.g., the voice over IP service), while others require the ability to use local *device capabilities* (e.g., access to the phone's camera). We call the former telco services (software services that provide communication and collaboration support) and the latter device APIs.

Telco capabilities

Telco companies, such as Orange[1], Telefonica[2] or Deutsche Telekom[3], have invested in Service Delivery Platforms (SDPs) that expose network capabilities to third parties, in order to enable user-generated, value-adding services. The core of these platforms is the Telecommunication Application Server, based on technologies such as SIP Servlets, JAIN SLEE, Parlay-X or IMS. While these telco services are evolving only slowly, non-telco companies, such as Google, Yahoo, Twilio or Tropo already provide their own telco services for managing calls, messaging or presence.

We distinguish three types of *telco services*, depending on the used networks and their purpose,

[1] http://www.orangepartner.com/site/enuk/access_orange_apis/p_access.jsp
[2] https://bluevia.com/en/
[3] http://www.developergarden.com/apis/

- *Internet telco services* are services that operate exclusively in the Internet, using it as communication infrastructure. Voice over IP (VoIP) or instant messaging are examples of Internet telco services.
- *Converged services* operate across the Internet and operator networks, mediating between different networks and communication protocols. A VoIP call to a mobile phone or fixed line phone is an example of converged service.
- *Signaling services* provide access to a network operator's signaling infrastructure. Notifying a mobile phone about an incoming call or negotiating QoS parameters are examples of signaling operations.

We analyzed a set of readily available telco services and APIs, in order to better understand the tasks of a mashup developer who wants to integrate telco services or APIs into his own mashup: Given a service or API, first of all he needs to understand what kind of service it actually provides (we say, what kind of *communication paradigm* it supports). Once the developer has identified a candidate, he typically wants to know how he can use, i.e., interact with, it in his mashup (we call this the *interaction paradigm* of the service). Finally, the typical question is at what cost or service levels a candidate service is delivered (we speak about *Service Level Agreements - SLAs*), in order to further discriminate services based on non-functional properties. These three activities inspire three dimensions (communication paradigm, interaction paradigm and SLA) that we can use to analyse telco services.

The *communication paradigm* describes the direction of the communication channel and the number of involved parties. In fact, contrary to the commonly used REST APIs, the cardinality plays a decisive role in the communication with a telco service. It depends on the service, if we are able to use a one-to-one or one-to-many communication. In both cases we have one service as a sender, but a different number of receivers. Another important property of telco services is their synchronicity. While voice and video communications require a synchronous communication (co-presence of participants is required), messaging is asynchronous (participants can operate them at different times).

The *interaction paradigm* looks at how the interaction with a service/API is handled. A sub-dimension, binding, describes how content is transferred, i.e., voice and video services are based on streaming data, since delays in synchronous communications are not desirable or even prohibited (e.g., in real-time streaming). Another sub-dimension is the internal state management of services, which is responsible for the instantiation and management of resources and communication channels. For instance, establishing a GSM phone call implies acting first on a control channel to obtain a separate stream for the actual communication. In mashups, where we might have multiple parallel communication connections open at the same time, this demands for suitable stream state management. All these aspects differ from common Web mashups where we call, for instance, REST APIs that instead are stateless.

Finally, *Service Level Agreements* look at QoS, cost, security and related aspects. One advantage of common Web mashups is the availability of a wide range of free services on the Web. In the telco domain, services are potentially subject to tarification by the network operator, based on different options (pay-per-use, subscription model, prepaid / postpaid billing models, discount plans, etc.). Thus, telco services are usually executed in a controlled environment where QoS, security, and tarification can be guaranteed.

Device capabilities

Modern mobile phones have evolved into full-fledged computing devices that are able to run mashups inside mobile browsers, also enabling mashups to leverage on advanced device capabilities,

such as the built-in camera or SMS texting. Telco mashups should therefore be capable of processing incoming telco events (e.g., phone calls or SMS messages) and accessing phone facilities (e.g., initiate phone calls or consult the agenda). These features can be done through *device APIs*, which allow the access to embedded cameras/webcams, location services, SMS and MMS messaging interfaces, and the like via web-ready interfaces (e.g., in JavaScript). For instance, cross-device standards, such as the W3C Device APIs (http://www.w3.org/2009/dap/) or the Widget Handset APIs developed by the Wholesale Applications Community (WAC, http://www.wacapps.net), provide APIs accessible from within regular web applications and cover a wide range of capabilities including position, accelerometer, messaging, system information, camera and microphone. For example, an application can capture an image using the following JavaScript code using WAC:

```
camera.captureImage(onCaptureImageSuccess,onCaptureImageError,{destinationFil
ename:"images/a.jpg", highRes:true});
```

Or in HTML5 using the W3C DAP Media Capture specification:

```
<input type="file" accept="image/*" id="capture">
```

By themselves, device APIs offer nothing especially new. The challenge for telco mashups is to seamlessly bring together device APIs and telco services with web mashups in a way that is not tied to any specific phone model, operating system, or service operator.

SIDEBOX: On the Role of Gateways in Telco Services

Telco services like converged services and signaling services are possible thanks to communication networks that actually predate the Internet, constitute its backbone, and evolved independently. For instance, fixed access is provided via the Public Switched Telephone Network (PSTN, also called POTS for Plain Old Telephony System), and mobile access via UMTS, GPRS, and GSM networks.

Each network uses its own protocols (e.g., GSM MAP) and signaling conventions (e.g., SS7), which are different from the Internet's TCP/IP stack. As a consequence, it is not possible to straightaway implement a web-based telco service that directly interoperates, for instance, with a GSM voice call. Implementing such kind of service requires therefore the ability to bridge between the two network types and to mediate between their respective protocols. This functionality is provided by telco operators in the form of *network gateways*, which can be reached from the Internet via standard Web protocols, such as REST/HTTP or SOAP, and that expose some network capabilities of the operator network (e.g., the GSM voice call).

Specifically, network gateways allow access to telephony infrastructure like the one in Figure 1. A telephony network essentially handles two different pieces of information[1]: (i) the *content* that is transmitted (e.g., voice or data) and (ii) *control signals* that instruct the network how to transmit content and provide for the allocation of the needed resources. In the past, control signals used *in-band* signaling techniques, i.e., signals were transmitted together with voice or data in a same channel. Due to its intrinsic bandwidth efficiency problems, this technique was soon replaced by *out-of-the-band* control channels and dedicated signaling protocols. The most popular out-of-the-band signaling protocol is SS7. As we can see in the figure, there are *circuit-switched* technologies (like PSTN or GSM) and *packet-switched* technologies (like UMTS PSD or VoIP). In circuit-switched technologies, a dedicated circuit path is established (via suitable SS7 control signals) before the content is transmitted. In package-switched technologies, content is fragmented into packages that can be transmitted through different paths and re-assembled by the destination. Package-switched technologies, therefore, require establishing a session between the caller and the receiver, which is usually done with the Session Instantiation Protocol (SIP). *Media gateways* provide for the conversion between circuit-switched and package-switched technologies, while *signaling gateways* do the same for control signals.

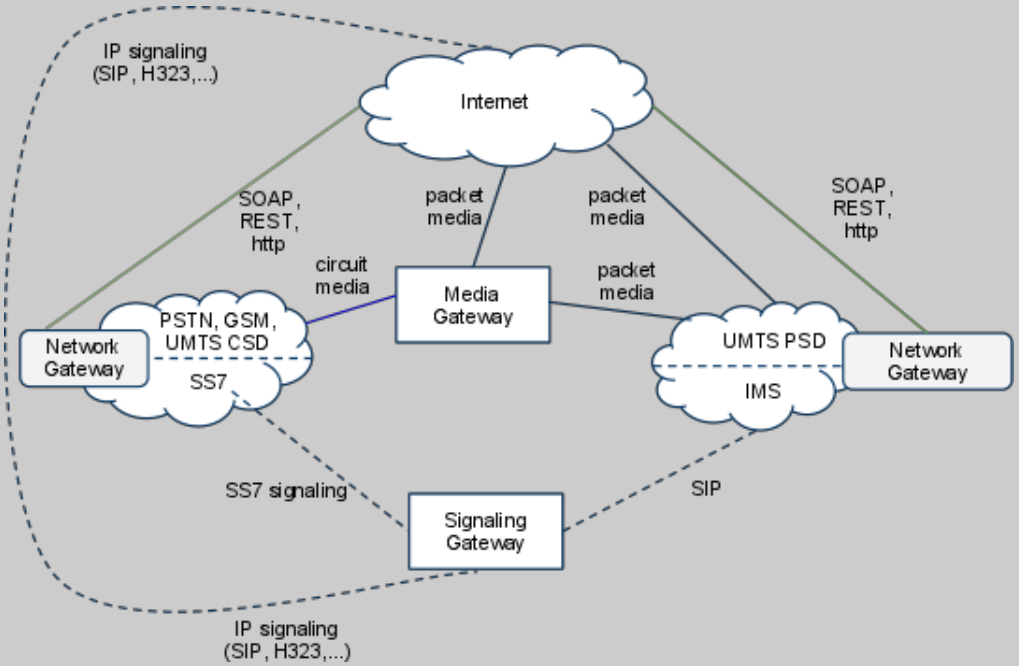


Figure 1. The typical topology of today's telecommunication infrastructure. Solid lines represent content flow; dashed lines represent control signals.

Given their crucial role in bridging the Web and the telco world, recently network gateways have been the subject of several standardization activities, such as Parlay-X[2], OneAPI (http://www.gsmworld.com/oneapi), and WAC (http://www.wacapps.net). In order to further reduce the complexity and costs of operating heterogeneous networks, so-called *next generation networks* (NGNs), such as the IP multimedia subsystem (IMS[3]), propose to use just one set of protocols for all kinds of networks, i.e., directly the Internet protocols.

[1] R. Bates and D. Gregory. *Voice & Data Communications Handbook*, Fifth Edition, McGraw-Hill Communication Series, 2007.
[2] 3rd Generation Partnership Project. Open Service Access (OSA); Parlay X web services; Part 1: Common. *3GPP TS 29.199-1*.
[3] M. Poikselka, G. Mayer, H. Khartabil. *The IMS Multimedia Concepts and Services*. Wiley, third edition, 2008.

A Telco-Specific Mashup Platform

Now, we define a *telco mashup* as a *web mashup* that, in addition to optional data, application logic, and UIs, also integrates *telco services* and/or *device APIs*, in order to support communication and collaboration among multiple users (e.g., our reference scenario) or to provide them with individual telco features (e.g., an advanced GPS navigation mashup).
If we want to aid the development of our reference scenario by means of a suitable telco mashup platform, the above analysis shows that our example scenario poses some *novel requirements* that are not yet supported by existing mashup platforms, such as:

- manage *streaming media* involving multiple users;
- integrate *device APIs* running inside client devices;
- manage *quality of service and billing;*
- *multi-channel access* to support different device types;
- *multi-modal access* to support different interaction paradigms;
- *multi-user access* to enable communication and collaboration.

Streaming audio/video conferencing is different from just streaming a video or audio file from a web server. In the latter case, if the stream breaks, it is enough to start the stream again; no special support from the web server is needed. If the stream of any of the participants breaks during the phone conversation in the case study, the platform must be able to reconnect the user to the ongoing live conference by keeping track of which user is involved in which conversation. Therefore, if a telco mashup uses multiple collaborative streams, it must be able to manage the *state* of each individual stream at the client side. This may require suitable browser extensions, client-side state management logic, or server-side logic, depending on the nature of each specific telco mashup. The use of *device APIs* does not directly impact the logic of the platform. Yet, if device APIs are used to communicate with other participants of a telco mashup, the platform must provide for the necessary client-server data communication channel (e.g., to broadcast Maria's photo). Both telco services (e.g., streaming services) and device APIs may require the monitoring and tracking of *QoS*. More importantly, especially the use of converged and signaling services inside a mashup requires the management of *billing* information, taking into account different contract options.
Multi-channel access requires the platform to deliver its mashups via different communication networks and protocols, such as the Internet or conventional telco networks. *Multi-modal access* requires support for different interaction paradigms, such as voice for Maria and traditional hypermedia for Jürgen, Marco and Steve. *Multi-user access* not only requires proper user identity management and authentication, but also the capability to allow multiple users to navigate (co-browse) the same mashup (e.g., to collaboratively draw the architecture picture), i.e., to work on one and the same mashup instance [2]. This is different from providing each user with an own, independent mashup instance, as it is customary in today's mashup platforms.

Understanding these subtleties of telco mashups is paramount for the development of a *telco mashup platform* that is indeed able to adequately support real-life telco mashup scenarios. In Figure 2, we illustrate our *reference architecture* for telco mashups; we specifically focus on the runtime architecture and the telco-specific features.
The architecture highlights how to deliver telco mashups via multiple channels. Maria's phone uses a traditional operator network (e.g., GSM), while Steve's PC, Marco's smart phone and Jürgen's tablet use the Internet. To enable the execution of the necessary converged and signaling services and to mediate between the Internet and the operator networks, either a *network gateway* (typically provided by the operator or upstart telco service providers such as Twillio) or a dedicated *telco application server* [3] (e.g., inside the *communications manager*) is needed.

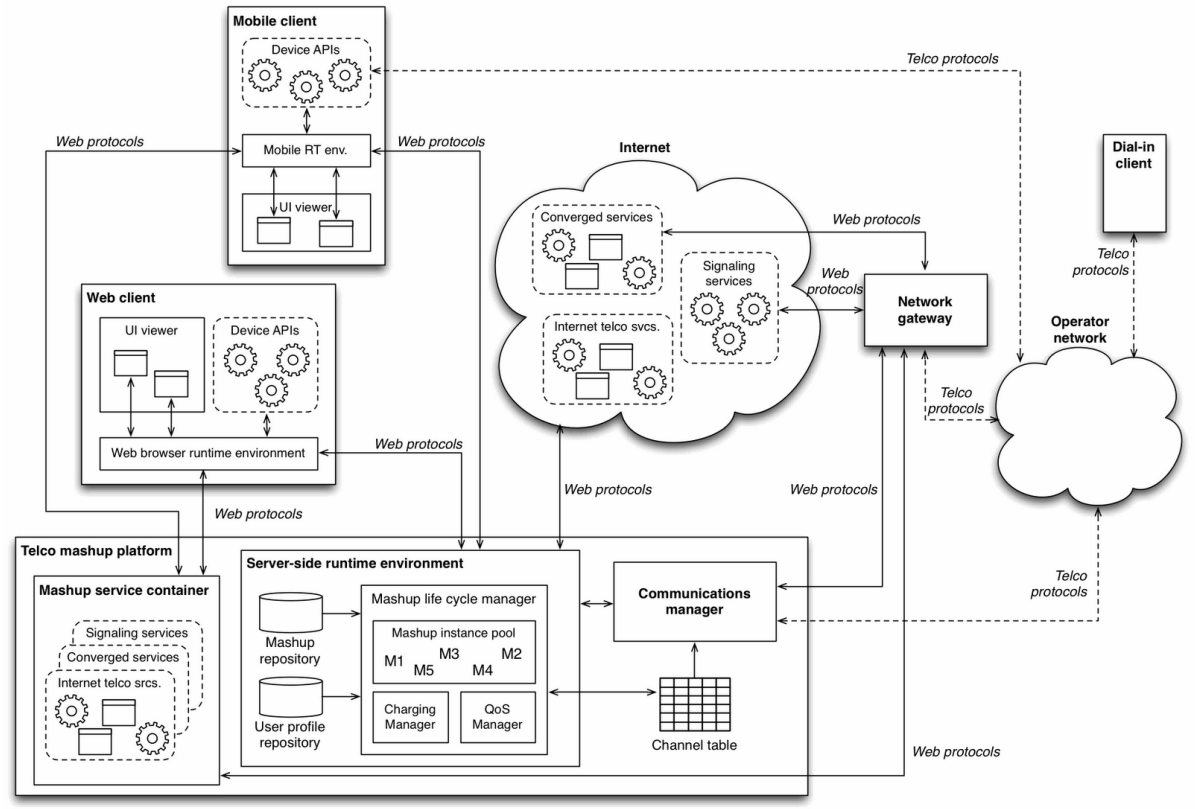


Figure 2. Telco mashup reference runtime architecture. UI components are represented as rectangles; services without UI as cogwheels.

Multi-modal access can be provided by the communications manager, e.g., allowing Maria to instantiate the mashup from her phone, even without the presence of Marco, Steve or Jürgen. Like in phone conferences, multi-user access requires then the creation of a shared resource everybody can connect to and the sharing of a respective identifier. In the architecture, this resource is represented by the *mashup instances* managed by the *mashup instance pool* which maintains the necessary correlation and life cycle information for the mashups' user interfaces running inside the client devices.

To assist client devices in the management of *streams* (both incoming/outgoing calls and web-based streams), a *channel table* correlates users with their streams and channels and the respective mashup instances; the channel table also allows one to book a telephone channel for audio/video conferences (scheduling) and to route asynchronous messages. With the channel table's help, the communications manager knows that Maria's photo is to be routed to Steve, Marco and Jürgen and not to other users of the platform. The use of device APIs impacts less the capabilities of the *server-side runtime environment* and more those of the client-side runtime environments. These must be able to provide access to device capabilities in a way that is compatible with standard web technologies, e.g., by means of web browsers that implement the respective W3C or WAC APIs or via suitable browser plug-ins. The communication among these APIs and the server-side platform then occurs via standard web protocols. QoS and billing are managed by a dedicated *QoS manager* and a *charging manager*.

In addition to these telco-specific features, a telco mashup platform will typically be able to host services (third-party and own components) in an own *mashup service container* and ready mashups

in a dedicated *mashup repository* (in either executable or interpretable format). For instance, the repository may cater for the voice call service used in our scenario, while the shared whiteboard may be sourced from the Internet. Upon request, the *mashup life cycle manager* (part of the *server-side runtime environment*) must be able to instantiate a mashup from the mashup repository, causing the instantiation of one or more *client-side runtime environments*, which host the actual UI of the mashup. These runtime environments may be native mobile applications, regular web applications, or JavaScript libraries running inside the web browser.

Analysis of Current Mashup Platforms

In order to understand which of the requirements discussed above are already supported by state-of-the-art mashup platforms, we have analyzed Yahoo! Pipes (http://pipes.yahoo.com/), Intel Mash Maker (http://mashmaker.intel.com/web), JackBe Presto (http://www.jackbe.com/), IBM Mashup Center (http://www-01.ibm.com/software/info/mashup-center/), WSO2 Mashup Server (http://wso2.com/products/mashup-server/), MyCocktail [4], ServFace [5], Karma [6], CRUISe [7], MashArt [8], Mashlight [9], OPUCE [10], SPICE [11], and SOA4All [12]. In the following we summarize the main findings of our analysis.

None of the analyzed platforms provides support for *multi-user* mashups. This shortcoming also impacts on other important requirements of telco mashups, i.e., the capability to *manage streaming media* involving *multiple users*. Streaming media management in single-user mashup tools is relatively simple, and supported by most of the available tools (e.g., embedded YouTube videos), but multi-user streaming management is impossible due to the lack of support for multi-user mashups. Regarding the access to mashup instances via *different channels*, i.e., the Internet and operator networks, only OPUCE and SPICE support bidirectional network integration, as envisioned by the communications manager in Figure 2. For example, OPUCE uses a JAIN SLEE server for integration with telco protocols; however, the preferred solution so far is delegating all interactions with operator networks to dedicated converged services. We have seen that the possibility to interact with a system via different channels also opens up the way for new *multi-modal* interaction. Instantiating a mashup only from a voice device running in an operator network is not possible so far; non-web clients can only be included into a running instance of a mashup, e.g., by calling it from the mashup. Also in this case dealing with multi-user mashups requires a different management of bidirectional channel integration that is not supported by any platform (e.g., the broadcasting of an incoming MMS or voice call stream to multiple mashup participants). Only OPUCE and SPICE support the integration with operator networks; but they have very limited support for interaction paradigms alternative to classic hypermedia and, hence, are not suitable for regular phones.

Another requirement discussed above is the integration of *device APIs*. Most of the analyzed platforms are able to generate web-based mashup applications; some also allow the creation of native device apps (e.g., Mashlight). Therefore, although they could exploit standard interfaces to access device APIs (e.g., W3C or WAC interfaces), device APIs are not yet commonly supported.

Finally, another important requirement for telco mashups is the mangement of *QoS and billing*. Only the telco-specific tools partly address this aspect. OPUCE adds annotations with pricing and QoS parameters to service descriptors, yet so far these annotations are not used at runtime. SPICE comes with a dedicated component for SLA management and the billing of services (based on the IMS and 3GPP).

Discussion and Outlook

Our aim with this paper was to approach the telco domain from the *Internet perspective*. We specifically looked at how web mashups can integrate with telco network and device capabilities. Our analysis shows that a minimum level of telco support is already there in some of the analyzed mashup platforms, yet advanced telco features still need to be implemented by hand.
The following *research challenges* seem crucial for the success of telco mashups:

- Telco service providers must develop *web-ready streaming and signaling services* that are easy to use and manage. For instance, setting up a video conference using the public Skype API still requires the programmer to master the Skype telephony protocol, which is complex and vendor-specific. Exposing this level of complexity toward a mashup environment is like not exposing the API at all. Although some authors have proposed a framework based on state machines [14] or communication hyperlinks [13], there is still a lack of a shared telco service model.
- Browser vendors must implement *full support for device APIs*. The W3C and WAC proposals to interface device capabilities are reasonable and easy to use. Their concrete support even inside the latest browser versions is still weak and partly browser-specific, which hinders adoption.
- Network operators and the Web community need to agree on standard, cross-operator APIs for the negotiation of *quality of service* and for *payment*, as well as respective *monitoring and charging* infrastructures. As of today, the market is fragmented, each operator adopts own policies and technologies, QoS is not adequately tracked, and each telco service requires an own payment logic.
- Strictly related with the previous point, the two communities need to develop *cross-network user identification and authentication* protocols to enable a seamless integration of networks. Suitable single sign-on and federation protocols seem of paramount importance.
- We must design mashups that are able to manage *intermittent connectivity*. Especially in the mobile Web, network disconnection is the rule, not the exception. Yet, we are typically still in the presence of services that are not able to work without the Internet, and we don't have robust solutions to deal with connectivity problems at the application level.
- Similarly, we need to be able to design mashups that are *adaptive*, e.g., that are able to autonomously fall back to lower-quality services if a higher-quality service is not available or to switch to a different service if we cross a border and operate in international roaming. Telco services are typically country-specific, and using them in roaming may cause huge costs.

Luckily, some of the open challenges are already on the research agenda of academia and industry, and most network operators open APIs to the public. For instance, GSM Association's OneAPI initiative (http://www.gsmworld.com/oneapi/) aims to devise cross-operator, lightweight web APIs toward typical telco network capabilities. The Web-telco convergence so far therefore mostly moves from traditional telco networks toward the Web, which means that the number and variety of telco services available on the Web is destined to grow significantly. This, on the other hand, requires the web community to better understand, master, and suitably interface the telco world.

Acknowledgements: This work was supported by funds from the European Commission (project Omelette, contract no. 257635).

References

[1] J. Yu, B. Benatallah, F. Casati, F. Daniel. Understanding Mashup Development. *Internet Computing*, vol. 12, no. 5, Sept-Oct 2008, IEEE Press, pp. 44-52.

[2] F. Daniel, A. Koschmider, T. Nestler, M. Roy, A. Namoun. Toward Process Mashups: Key Ingredients and Open Research Challenges. In *Proc. of Mashups'10*, Dec. 2010, ACM.

[3] OpenCloud Limited. OpenCloud Rhino Telecom Application Server™. OpenCloud DataSheet, 2010. www.opencloud.com

[4] C.A. Iglesias, J.I. Fernández-Villamor, D. del Pozo, L. Garulli, B. García. Combining Domain-Driven Design and Mashups for Service Development. In *Service Engineering*, Springer, 2010.

[5] M. Feldmann, T. Nestler, U. Jugel, K. Muthmann, G. Hübsch, A. Schill. Overview of an end user enabled model-driven development approach for interactive applications based on annotated services. In *Proc. of WEWST'09*, Nov. 2009, ACM.

[6] R. Tuchinda, P. Szekely, C. A. Knoblock. Building Mashups by example. In *Proc. of IUI '08*, Jan. 2008, ACM.

[7] S. Pietschmann, M. Voigt, A. Rümpel, K. Meißner. CRUISe: Composition of Rich User Interface Services. In *Proc. of ICWE'09*, June 2009, Springer.

[8] F. Daniel, F. Casati, B. Benatallah, M.C. Shan. Hosted universal composition: Models, languages and infrastructure in mashArt. In *Proc. of ER'09*, Nov. 2009, Springer.

[9] L. Baresi, S. Guinea. Consumer Mashups with Mashlight. In *Proc. of ServiceWave'10*, Dec. 2010, Springer.

[10] J. Sienel, A. León, C. Baladrón, L. W. Goix, A. Martínez, B. Carro. OPUCE: A telco-driven service mash-up approach, *Bell Labs Tech. J.* 14, 1, May 2009, pp. 203-218.

[11] O. Droegehorn, I. Konig, G. Le-Jeune, J. Cupillard, M. Belaunde, E. Kovacs. Professional and end-user-driven service creation in the SPICE platform, In *Proc. of WoWMoM'08*, June 2008, IEEE Press.

[12] M. Zuccalà. SOA4All in Action: Enabling a Web of Billions of Services. In *Proc. of ServiceWave'10*, Dec. 2010, Springer.

[13] V. Verdot, G. Burnside, and N. Bouché. An adaptable and personalized web telecommunication model, vol. 16, issue 1, 2011, *Bell Labs Tech. J.*, June 2011, pp. 3-17.

[14] R. Arlein, D. Dams, R. Hull, J. Letourneau and K. Namjoshi. Telco meets the Web: Programming shared-experience services, *Bell Labs Tech. J.*, 2009, pp. 167–185.

Appendix M

Orchestrated User Interface Mashups Using W3C Widgets

Scott Wilson[1], Florian Daniel[2], Uwe Jugel[3] and Stefano Soi[2]
[1] University of Bolton, United Kingdom
scott.bradley.wilson@gmail.com
[2] University of Trento, Povo (TN), Italy
{daniel,soi}@disi.unitn.it
[3] SAP AG, SAP Research Dresden, Germany
uwe.jugel@sap.com

Abstract. One of the key innovations introduced by web mashups into the integration landscape (basically focusing on data and application integration) is *integration at the UI layer*. Yet, despite several years of mashup research, no commonly agreed on component technology for UIs has emerged so far. We believe *W3C's widgets* are a good starting point for componentizing UIs and a good candidate for reaching such an agreement. Recognizing, however, their shortcomings in terms of inter-widget communication – a crucial ingredient in the development of interactive mashups – in this paper we (i) first discuss the *nature of UI mashups* and then (ii) propose an *extension of the widget model* that aims at supporting a variety of inter-widget communication patterns.

Keywords. UI Mashups, W3C widgets, Inter-widget communication

1 Introduction

If we analyze the state of the art in mashups today, we recognize that basically two different approaches have reached the necessary critical mass to survive: data mashups and UI (user interface) mashups. ***Data mashups*** particularly focus on the integration and processing of data sources from the Web, e.g., in the form of RSS or Atom feeds, XML files, or other simple data formats; mashup platforms like Yahoo! Pipes (http://pipes.yahoo.com/pipes/), JackBe Presto (http://www.jackbe.com/), or IBM's Damia [1] are examples of online tools that aim at facilitating data mashup development. ***UI mashups***, instead, rather focus on the integration of pieces of user interfaces sourced from the Web, e.g., in the form of Ajax APIs or HTML markup scrapped from other web sites; Intel Mash Maker [2] or mashArt [3] both support the integration of UI components, but most of the times these mashups are still coded by hand (e.g., essentially all of the mashups on programmableweb.com are of this type).

The mashup platforms focusing on data mashups typically come with very similar features in terms of supported data sources, operators, filters, and the like. RSS, Atom, or CSV are well-known and commonly accepted data formats, and there are not many different ways to process them. Unfortunately, this is not what happens in the context of UI mashups. In fact, there are still many different ways to look at the

problem and, hence, each tool or programmer uses its own way of componentizing UIs (both in JavaScript inside the browser and in other languages in the web server) and of integrating them into the overall layout of the mashup. As a consequence, UI components are not compatible among mashup tools, and we are far from common concepts and approaches when it comes to UI mashups.

Given for granted that UI components are able to encapsulate and deliver pieces of UIs that can be embedded into a mashup and operated by its users, the key ingredient for UI componentization we identify is the component's ability to ***interoperate*** with its surroundings, i.e., with other UI components and the hosting mashup logic. Interoperability is needed to enable components to synchronize upon state changes, e.g., in response to user interactions or internal logics. While technically this is not a huge challenge, conceptually it is not trivial to understand which communication paradigm to adopt, which distribution logic to support, or which data format to choose, maximizing at the same time the reusability of UI components across different mashup platforms, also fostering interoperability among mashups themselves.

In this paper, we approach these challenges by leveraging on a UI componentization technology that we believe will have a major impact in the near future, i.e., W3C's Widgets [4]. This choice is motivated, firstly, by the comprehensiveness of W3C's Widgets specifications family which tries to cover models and functionalities proper of the most used widget technologies existing so far, e.g., Google gadgets, Yahoo widgets and, in particular, Open Social gadgets. Moreover, the W3C consortium is a leading actor in web standards creation and its proposal already attracted important vendors that are implementing W3C's Widget compliant tools (e.g., Apache Wookie and Rave).

Specifically, in this paper, we provide the following ***contributions***:

- We discuss three types of *mashup logics for widgets* and identify a set of requirements the widgets should satisfy, in order for them to be mashed up.
- We propose an *extension of the W3C widget model* expressed in terms of an API extension and set of expected behaviours.
- We report on our experience with the *implementation* of a UI mashup following one of the described mashup logics and the extended widget model.

Before going into the details of our proposal, in the next section we briefly summarize the logic of and technologies used in the implementation of W3C widgets. Then, in Section 3, we investigate the basic mashup types for widgets. In Section 4 we specifically look at one type of mashups and derive a set of requirements for widgets. In Section 5 we propose an according extension of the W3C widget model, also providing concrete implementation examples. Finally, in Section 6 we discuss related works, in order to conclude the paper in Section 7.

2 W3C Widgets

The World Wide Web Consortium (W3C) provides a set of specifications collectively known as the *Widget family of specifications*. A Widget is defined by W3C (http://dev.w3.org/2006/waf/widgets-land/) as "an end-user's conceptualization of an

interactive single purpose application for displaying and/or updating local data or data on the Web, packaged in a way to allow a single download and installation on a user's machine or mobile device."

Widgets are made available to users by a *widget runtime* (also known as a *widget engine*). A *widget runtime* is an application that can import a widget that has been packaged according to the *W3C Widgets: Packaging and Configuration* specification [4]; the runtime may also make available at runtime any script objects required by the widget, for example the *W3C Widget Interface* [5] (the API a widget exposes to provide access to the widget's metadata and to persistently store data) or *W3C Device APIs* [6] (client-side APIs that enable the development of widgets that interact with device services like calendar, contacts, or camera). Widget runtimes are available on mobile devices, as desktop applications, or for embedding widgets in websites.

The Packaging and Configuration specification defines the metadata terms used to describe the widget (such as name, author and description) and to enable the configuration of the widget runtime. Configuration information includes the <feature> element, which can be used by the widget author to request that the widget runtime makes additional features available when the widget is running; examples of features include JavaScript APIs, libraries, and video codecs.

Within the W3C Widget family of specifications, widgets are largely conceptualized as operating independently, communicating with the widget runtime using the Widget Interface and with the client environment using standard browser features such as the Document Object Model and related JavaScript APIs.

While a widget runtime may render multiple widgets to the user simultaneously – for example, on the Home screen of a mobile device, or as part of the layout of a portal or social networking site – there are no mechanisms specified by the W3C Widget family of specifications by which the widgets communicate with each other as members of a mashup.

3 User Interface Mashups

Given a set of widgets that comply with the W3C Widget family of specifications, the question is therefore how a mashup of widgets could look like. Considering the state of the art in which widgets do not support inter-widget communications, we define a ***basic UI mashup***, as a tuple $m = \langle L, W, VA \rangle$ with:

- $L = \langle l, V \rangle$ being the *layout* of the mashup, of which l is the layout *template* (typically the template consists of an HTML page, a set of JavaScript and image files, and one or more CSS style sheets) and $V = \{v_i\}$ is the set of *viewports* inside l that can be used for the rendering of the widgets (e.g., iframes or div elements);
- $W = \{w_j\}$ being the set of *widgets* in the mashup, where each widget is of type $w_j = \langle id_j, name_j, Pref_j, version_j, height_j, width_j \rangle$ with $Pref_j$ being a set of configuration *preferences* (typically, name-value pairs); and
- $VA = \{va_k | va_k \in W \times V\}$ being the set of *widget-viewport associations* needed for placing and rendering the widgets inside the mashup.

This model focuses on the layout only and is clearly not able to represent UI mashups like most of the ones that can be found on programmableweb.com. In fact, UI mashups typically are able to synchronize their widgets or UI elements upon user interactions, a feature that is missing in mashups of type m above.

Assuming now that widgets are able to communicate, in the following subsections we define three UI mashup models that are able to deal with inter-widget communications and to support widget synchronization:

- *Orchestrated UI mashups*, where the interactions between the widgets in the mashup are defined using a central control logic;
- *Choreographed UI mashups*, where the interactions between the widgets in the mashup are not defined, but instead emerge in a distributed fashion from the internal capabilities of the widgets;
- *Hybrid UI mashups*, where the emerging behaviour of a choreographed UI mashup is modified by inhibiting individual behaviours, practically constraining the ad-hoc nature of choreographed UI mashups.

We define each of these mashup types in the following, while in the rest of this paper we will specifically focus on orchestrated UI mashups, which can be considered the basis also for the development of the other two types of UI mashups.

3.1 Orchestrated UI Mashups

We define an ***orchestrated UI mashup*** as a tuple $m^o = \langle L, W, VA, C \rangle$ with:

- L being the layout as defined before;
- $W = \{w_j | w_j = \langle id_j, name_j, Pref_j, version_j, height_j, width_j, E_j, O_j \rangle\}$ being the set of widgets with $E_j = \{e_{jl} | e_{jl} = \langle name_{jl}, P_{jl} \rangle\}$ being the set of events the widget can generate, $O_j = \{o_{jm} | o_{jm} = \langle name_{jm}, P_{jm} \rangle\}$ being the set of operations supported by the widget, and P_{jl} and P_{jm}, respectively, being the sets of output and input parameters;
- $VA = \{va_k | va_k \in W \times V\}$ being the set of widget-viewport associations; and
- $C = \{c_n | c_n \in E \times O, E = \bigcup_j E_j, O = \bigcup_j O_j\}$ being the set of direct inter-widget communications, i.e., message flows between two widgets connecting an event of the source widget with an operation of the target widget.

This definition of UI mashup implies that the mashup (and, therefore, the mashup developer) knows which events are to be mapped to which operations and that it is able to propagate the respective data items on behalf of the user of the mashup. This is common practice, e.g., in web service composition languages like BPEL, and does not require the widgets to know about each other.

The strength of this model is that mashups behave as they are expected to, that is, as specified in the mashup specification. A drawback is that this central mashup logic must be specified in advance, i.e., before runtime, which require a good knowledge of the used widgets by the mashup developer.

Note that in the above definition and in the following we intentionally do not introduce complex data mappings (e.g., requiring data transformation logics) or service components (e.g., requiring to follow web servie protocols), in order to keep

the model simple and focused. We however assume each inter-widget communication c_n also contains the necessary mapping of event outputs to operation inputs.

We believe UI mashups are good candidates for end user development and that data transformations or web services are not intuitive enough to them in order to profitably use them inside a mashup. Possible complex data transformations or service composition logics can always be developed by more skilled developers and plugged in in the form of dedicated widgets.

3.2 Choreographed UI Mashups

We define a ***choreographed UI mashup*** as a tuple $m^c = \langle L, T, W, VA \rangle$ with:

- L being the layout of the mashup;
- $T = \{t_n | t_n = \langle name_n, P_n \rangle\}$ being the reference topic ontology for events and operations, i.e., the set of concepts and associated parameters P_n the widgets in the mashup can consume as input or produce as output;
- $W = \{w_j | w_j = \langle id_j, name_j, Pref_j, version_j, height_j, width_j, E_j, O_j \rangle\}$ being the set of widgets with $E_j = \{e_{jl} | e_{jl} = \langle name_{jl}, Topic_{jl} \rangle\}$ being the set of events the widget can generate, $O_j = \{o_{jm} | o_{jm} = \langle name_{jm}, Topic_{jm} \rangle\}$ being the set of operations supported by the widget, and $Topic_{jl}, Topic_{jm} \subseteq T$, respectively, being the set of topics an event sends data to and an operation reacts to; and
- $VA = \{va_k | va_k \in W \times V\}$ being the of widget-viewport associations.

In contrast to orchestrated UI mashups, choreographed UI mashups do not have an explicitly defined set of mappings of operations and events. Instead, each widget is capable of sending and receiving communications and of acting on them independently. Interoperability is achieved in that each widget complies with the reference topic ontology T, which provides a reference terminology and semantics each widget is able to understand. The behaviour of a choreographed UI mashup, therefore, is not modelled centrally by the mashup developer and rather emerges in a distributed way by placing one widget after the other into the mashup. That is, only placing a widget into the mashup allows the developer to understand how it behaves in the mashups and which features it supports.

The strength of this approach is that there is no need for explicit design of interactions: a developer simply drops widgets into his mashup and they autonomously interact. One weakness is that the reference topic ontology must be "standardized" (or, at least, understood by all widgets), in order for any meaningful communication to occur. This may reduce the overall richness of communication possible to a small number of fairly primitive topics – for example, location, dates and unstructured text. Another weakness is that with no predefined "plan" of the mashup, there could be the risk of the emergent behaviour of the widgets being pathological – for example, self-reinforcing loops or hunting. This could be a serious problem where the mashup components have real-world consequences, such as SMS-sending widgets or similar.

3.3 Hybrid UI Mashups

We define a ***hybrid UI mashup*** as a tuple $m^h = \langle L, T, W, VA, C \rangle$ with:

- L being the layout of the mashup;
- $T = \{t_n | t_n = \langle name_n, P_n \rangle\}$ being the reference topic ontology;
- $W = \{w_j | w_j = \langle id_j, name_j, Pref_j, version_j, height_j, width_j, E_j, O_j \rangle\}$ being the set of widgets with $E_j = \{e_{jl} | e_{jl} = \langle name_{jl}, Topic_{jl} \rangle\}$ being the set of events the widget can generate and $O_j = \{o_{jm} | o_{jm} = \langle name_{jm}, Topic_{jm} \rangle\}$ being the set of operations supported by the widget;
- $VA = \{va_k | va_k \in W \times V\}$ being the set of widget-viewport associations; and
- $C = \{c_n | c_n \in T \times O, O = \bigcup_j O_j\}$ being a set of constraints preventing operations from reacting to the publication of an event referring to a given topic.

In hybrid UI mashups, integration is achieved in a bottom-up fashion by the widgets themselves, while there is still the possibility for the mashup developer to control the interaction logic of the overall mashup in a top-down fashion by inhibiting interactions and, hence, application features that are not necessary for the implementation of his mashup idea.

The strength of this approach is that it brings together the benefits of both orchestrated and choreographed UI mashups, that is, simplicity of development and control of the behaviour. On the downside, the overall mashup logic is buried inside two opposite composition logics: the implicit capabilities of the widgets and the explicit constraints by the developer. This may be perceived as non-intuitive by less skilled developers or end users.

4 A W3C Widget Extension for Orchestrated UI Mashups

As a first step toward supporting the above UI mashup types, in this paper we aim at enabling the development of *orchestrated UI mashups*, a task that is already not possible with the W3C widget model as is. From the definition of m^o above we can, in fact, derive a set of extension requirements for W3C widgets, without which the implementation of interactive UI mashups is not possible:

1. Widgets must be able to communicate internal state changes via *events* to the outside world, i.e., the mashup or other widgets in the mashup. That is, while the users interacts with the widget, the widget must implement an internal logic that tells the widget when it should raise an event, in order to allow other widgets in a same mashup to synchronize.
2. Widgets must be able to accept inputs via *operations*, in order to allow the outside world to enact widget-internal state changes. The enacting of an operation is the natural counterpart of an event being raised. Typically, the operation implements the necessary logic to synchronize the state of the widget (e.g., the content rendered in the widget's viewport) with the event.
3. The *data formats* for the data exchanged among widgets should be kept as

simple as possible (we propose simple name-value pairs), in order to ease inter-widget communication. Considering that synchronizing widgets based on user interactions or internal state changes typically will require only the transfer of one or two parameters [3], e.g., an object identifier upon a selection operated by the user, this assumption seems reasonable. Remember that here we do not focus on web service orchestration or data processing.

We approach each of these requirements in the following sections and show how so extended widgets can be mashed up into UI mashups.

5 A Prototype Implementation

In order to better explain our ideas, in the following we adopt a by-example approach and contextualize them in our prototype implementation, finally also showing how the extended widget model can be successfully used for the implementation of orchestrated UI mashups.

5.1 Widget configuration

The W3C Widgets: Packaging and Configuration specification supports the run-time loading of extensions using the <feature> element of the widget's config.xml file. This requires that the widget runtime environment can resolve the URI of the feature to an installed capability. For example, given the feature URI *http://example.org/rpc* a runtime may install an implementation specific to that runtime environment, or a generic one if the functionality is relatively simple. If the URI is not recognized, the runtime will reject the installation of the widget if the required attribute is set to "true", but will proceed (optionally warning the user) if it is set to "false".

However, it is also possible for a W3C Widget to load capabilities dynamically while running, using <script src> elements in the HTML start file or using lazy loading techniques to dynamically insert new <script> elements based on the current context. Therefore for an orchestration interface we have to make a decision as to which approach to take in loading the required capabilities. Each has its advantages and disadvantages.

An advantage of using <feature> loading is that it gives the runtime environment the option to use server-side capabilities or augmented functionality. For example, to load an API in the widget that then talks to a high-performance server-side messaging service. The disadvantage is that if the runtime does not support the feature, then the widget is either not able to be installed, or is installed without necessary functionality. The advantage of using HTML-based script loading is that it should work in any widget runtime environment; however it is not able to take advantage of any special capabilities of the runtime. A compromise solution is to use the <feature> declaration but to set the required attribute to "false", and provide a dynamic <script> tag loader as a fallback. This enables the widget to take advantage of native runtime implementations, but has a fallback option if none is provided. This can be implemented using a fairly simple script in the widget, as illustrated in Figure 1.

```
If (widget.intercom && typeof(widget.intercom)==function){
  // the runtime has provided the intercom API
} else {
  // load the fallback library – in this case PMRPC
  widget.intercom = loader.load("pmrpc.js");
}
```

Figure 1. Widget-internal JavaScript logic to decide whether to load a fallback library or not.

5.2 Widget interface

We enable widgets to participate in orchestrated UI mashups through the specification of a so-called ***Intercom*** interface as an extension of the W3C Widget Interface. An implementation of the Intercom object must have the following three capabilities:

- It must be able to execute *operations* on the widget;
- It must be able to raise *events*; and
- It must be able to expose *metadata* about the operations and events supported by the widget.

The implementation of the Intercom interface may be made available at runtime through the use of a <*feature*> element in the widget configuration document or as a direct extension to the W3C Widget Interface specification implemented by the widget runtime.

The Intercom does not specify any orchestration configuration, but the capabilities of the orchestration participants and an interface to access the inter-widget communication features of the Intercom implementation. Therefore, we propose to introduce an attribute `intercom` to the W3C Widget Interface (see Figure 2).

```
[NoInterfaceObject]
interface Widget {
    readonly attribute DOMString     author;
    readonly attribute DOMString     authorEmail;
    readonly attribute DOMString     authorHref;
    readonly attribute DOMString     description;
    readonly attribute DOMString     id;
    readonly attribute DOMString     name;
    readonly attribute DOMString     shortName;
    readonly attribute Storage       preferences;
    readonly attribute DOMString     version;
    readonly attribute unsigned long height;
    readonly attribute unsigned long width;
    readonly attribute Intercom      intercom;
};
```

Figure 2 Widget interface extended with intercom attribute

The Intercom interface itself is defined as described in Figure 3: Inspecting the `metadata` attribute of the Intercom interface allows the widget runtime environment to obtain the list of events and operations implemented by the widget, along with their respective output/input parameters. The two functions `raise` and `call` can then be used to generate an event and to enact an operation, respectively.

```
interface Intercom {
    void raise(in DOMString operationName, in optional DOMString param1, ... );
    void call(in DOMString operationName, in optional DOMString param1, ... );
    readonly attribute IntercomMetaData metadata;
}
interface IntercomMetaData {
    readonly attribute sequence<IntercomSignature> events;
    readonly attribute sequence<IntercomSignature> operations;
}
interface IntercomSignature {
    readonly attribute DOMString name;
    readonly attribute sequence<IntercomArgument> parameters;
}
interface IntercomArgument {
    readonly attribute DOMString name;
}
```

Figure 3. A possible Intercom interface, including access functions and metadata structures.

For instance, Figure 4 exemplifies how a widget can use its Intercom to raise the events "eventName", and how an external RPC module (e.g., the one used by the specific Intercom implementation) can use the widgets' intercoms to call operations.

```
//called from widget
this.intercom.raise("eventName", arg1, arg2);

//called from communication module
widget.intercom.call("operationName", arg1, arg2);
```

Figure 4. Using the intercom object.

With the help of the Intercom interface, an automatic composition component or a composition tool can use the `metadata` attribute of several widgets to learn about the composition capabilities that the widget supports.

To keep the Intercom interface as simple as possible, we do not support operation return types or complex parameter types.

5.3 Widget implementation and behaviour

In Figure 5 we provide a possible implementation of the Intercom interface, which makes use of the external communication infrastructure (`SOMERPC`) declared as required <feature> in the widget configuration.

```
var SOMERPC = {/* some rpc module required by this Intercom implementation */};

var Intercom = function( widget ) {
    var w = widget,
        rpcmodule = SOMERPC,
        operations = {},

        // reads the meta data from a config file, xml, etc.
        metadata = rpcmodule.getMetaData( w.name ),
        raise = function( eventName ){       //init public raiseEvent method
            var args = Array.prototype.splice.apply(arguments, 1,
                                                arguments.length-1);
```

```
            rpcmodule.raiseEvent( w, eventName, args );
        },
        call = function( opName ){
            var args = Array.prototype.splice.apply(arguments, 1,
                                                arguments.length-1);
            //call widget operation if it is in the public operations
            if(operations[opName]) {
                operations[opName].apply( w, args );
            }
        },
        i = 0;

    //setup the private operations list for faster access when 'call' is executed
    for(i = 0; i < metadata.operations.length; i += 1) {
        operations[metadata.operation[i].name] = w[metadata.operation[i]];
    }

    this.raise = raise;
    this.call = call;
    this.metadata = metadata;

    //register this intercom at the rpc module
    rpcmodule.register( this );
};
```

Figure 5. A basic implementation of the Intercom interface.

The Intercom of a widget should be initialized in the widget constructor to prevent modifications from the outside:

```
// called from the widget contructor
this.intercom = new Intercom( this );
```

After the intercom is set up, a widget can start raising events via its own Intercom, and all modules that have access to the widget or the widget's Intercom can call operations on the widget.

5.4 UI mashup implementation

Using the formalization introduced in Section 3, we are able to model a variety of mashups involving multiple widgets. The specification does not include any additional runtime aspects, such as message delivery time, message buffering, or similar technical aspects. Thereby, it is flexible enough to also accomodate mashups with more complex characteristics, such as mashups involving multiple windows or multiple origins, and it is agnostic as to whether communication is purely within the browser (e.g., using HTML 5 PostMessage) or also involving the server side.

Implementing a UI mashups can be achieved relatively simply through the use of publish-subscribe services propagating events from one widget to others. In orchestrated UI mashups of type $m^o = \langle L, W, VA, C \rangle$, it is the inter-widget communication logic C that subscribes widgets, i.e., their operations, to events. In choreographed UI mashups of type $m^c = \langle L, T, W, VA \rangle$, each widget publishes its events to the topics in T and subscribes to the topics it understands. In hybrid UI mashups $m^h = \langle L, T, W, VA, C \rangle$, the bottom-up subscriptions by the widgets can be fine-tuned via the constraints C. All this can implemented using a range of existing

mature software technologies, for example, client-side using OpenAjax Hub[1] or server-side using solutions such as Faye[2] or ActiveMQ[3].

6 Related Work

In our former work [8], we developed an approach to the componentization and intercommunication of UI components. The approach is different from the one proposed in this paper, in that it aims to wrap full-fledged web applications developed with traditional, server-side web technologies. The wrapping logic requires the presence of simple event annotations inside the application's HTML markup in order to intercept events and a descriptor for the enacting of operations on the wrapped web app. Widgets, instead, are pure client-side apps.

In the context of widgets, Sire et al. [7] proposed an idea that is similar to what we propose in this paper, also advocating the use of events and event listeners (the equivalent of our operations). The widget decides whether an event is distributed in a unicast (one receiver), multicast (multiple receivers), or broadcast (all possible receivers) fashion. This design choice, however, leads to tightly coupled widgets, in that a widget must know in advance with which other and how many widgets it will communicate, a limitation we do not have in our proposal. In fact, in our case it is the mashup logic (which, for choreographed UI mashups, may be missing) that manages the inter-widget communication, and widgets are unaware of their neighbours.

The Java Portlet Specification 2.0 [9] proposes inter-widget communication for web portals. Portlets may communicate via events, but interactions occur on the server-side, a strong limitation in a UI-intensive Web 2.0 context. So far, the adoption of this technique is relatively low, also because its limitation to the Java world.

Communicating across technical boundaries, as proposed in this paper, is required in many networked computing domains. Especially for web browsers, the communication across domains and across browser windows (including iframes) is an important issue. Therefore, the HTML 5 standard defines a messaging API [10], which is, for example, used by the "pmrpc" project [11]. This project provides a Javascript module that adds a *pmrpc* object to a running website *window* object. All scripts running inside this window may access *pmrpc* to register own operations, or make calls to other windows/frames [12].

Our investigation of these and similar RPC approaches showed that different projects use different interface syntax and mainly focus on cross-window communication. In comparison to that, our proposed interface extension does not specify any cross-domain/window aspects. A single *widget*, in our case, is similar to a *window* in these related approaches, but there can be many *widgets* in many *windows* that constitute a *mashup*. All widgets will use their *intercom* transparently. Cross-domain issues must be solved internally by the Intercom implementation, which may of course use, e.g., *pmrpc* internally for this aspect.

[1] http://www.openajax.org/whitepapers/Introducing%20OpenAjax%20Hub%202.0%20and%20Secure%20Mashups.php

[2] http://faye.jcoglan.com/

[3] http://activemq.apache.org/

7 Conclusion and Future Work

In this paper, we addressed a relevant issue in UI-based mashup development, i.e., the intercommunication of W3C widgets. Mashups are typically heavily UI-based, but so far no standard for how to componentize UIs and how to get them into communication has emerged. We believe W3C widgets have the potential to represent this agreement and that they will gain importance in the near future in both desktop and mobile computing environments.

The aim of our research in this context is to come up with an inter-widget communication interface and respective widget behaviours, which – thanks to our involvement in the standardization of the widget technology – we would like to propose to the W3C for standardization. This is an effort we carry on in the context of the European project Omelette (http://www.ict-omelette.eu).

In order to obtain a first feedback from the community regarding the proposed communication interface, in this paper we focused on inter-widget communication at the level of events and operations. In the future, we also aim to identify and propose a standard format for the exchange of data among widgets, e.g., based on the OData protocol or similar initiatives.

Acknowledgements: This work was supported by funds from the European Commission (project OMELETTE, contract no. 257635).

References

1. M. Altinel, P. Brown, S. Cline, R. Kartha, E. Louie, V. Markl, L. Mau, Y.-H. Ng, D. Simmen, and A. Singh. Damia: a data mashup fabric for intranet applications. *VLDB'07*, September 2007, VLDB Endowment, pp. 1370-1373.
2. R. Ennals, E. Brewer, M. Garofalakis, M. Shadle, P. Gandhi. Intel Mash Maker: join the web. *SIGMOD Rec*. 36, 4, December 2007, pp. 27-33.
3. F. Daniel, F. Casati, B. Benatallah, M.-C. Shan. Hosted Universal Composition: Models, Languages and Infrastructure in mashArt. *ER'09*, November 2009, Springer, pp. 428-443.
4. W3C. Widget Packaging and Configuration. *W3C Working Draft*, March 2011, http://www.w3.org/TR/widgets/
5. W3C. The Widget Interface. *W3C Working Draft*, September 2010, http://www.w3.org/TR/widgets-apis/
6. W3C. Device APIs and Policy Working Group Charter. http://www.w3.org/2009/05/DeviceAPICharter
7. S. Sire, M. Paquier, A. Vagner, J. Bogaerts. A Messaging API for Inter-Widgets Communication. *WWW'09*, April 2009, ACM, pp. 1115-1116.
8. F. Daniel and M. Matera. Turning Web Applications into Mashup Components: Issues, Models, and Solutions. *ICWE'09*, June 2009, Springer, pp. 45-60.
9. S. Hepper. Java(TM) Portlet Specification Version 2.0. Proposed Final Draft, Rev. 29. http://jcp.org/aboutJava/communityprocess/pfd/jsr286/index.html
10. WHATWG. HTML Living Standard, Communication. WHATWG specification. Website, April 2011: http://www.whatwg.org/specs/web-apps/current-work/multipage/comms.html
11. I. Kovic and I. Zuzak. Pmrpc, HTML5 inter-window and web workers RPC and pubsub communication library. Project website, April 2011: http://code.google.com/p/pmrpc/.
12. I. Kovic and I. Zuzak. List of system that enable inter-window or web worker communication. Website, April 2011: http://code.google.com/p/pmrpc/wiki/IWCProjects.

Appendix N

Conceptual Design of Sound, Custom Composition Languages

Stefano Soi, Florian Daniel, Fabio Casati

Abstract Service composition, web mashups, and business process modeling are based on the composition and reuse of existing functionalities, user interfaces, or tasks. Composition tools typically come with their own, purposely built composition languages, based on composition techniques like data flow or control flow, and only with minor distinguishing features - besides the different syntax. Yet, all these composition languages are developed from scratch, without reference specifications (e.g., XML schemas), and by reasoning in terms of low-level language constructs. That is, there is neither reuse nor design support in the development of custom composition languages.
We propose a conceptual design technique for the construction of custom composition languages that is based on a generic composition reference model and that fosters reuse. The approach is based on the abstraction of common composition techniques into high-level language features, a set of reference specifications for each feature, and the assembling of features into custom languages by guaranteeing their soundness. We specifically focus on mashup languages.

1 Introduction

The proliferation of composition instruments like mashup platforms or web service composition environments, which allow one to integrate Web-accessible APIs and data into value-adding, composite applications or services, also led to the prolifer-

Stefano Soi
University of Trento, Via Sommarive 5, 38123 Trento - Italy e-mail: soi@disi.unitn.it

Florian Daniel
University of Trento, Via Sommarive 5, 38123 Trento - Italy e-mail: daniel@disi.unitn.it

Fabio Casati
University of Trento, Via Sommarive 5, 38123 Trento - Italy e-mail: casati@disi.unitn.it

ation of respective *composition languages*. Depending on the type of API or data source (we call them collectively components), the type of application or service (e.g., data mashup vs. UI mashup vs. service composition, and similar), and the target user of the application or service, composition languages differ in the features they offer to the developer - not only in their syntax. While in many cases language differences among tools actually don't seem to be necessary, in other cases these differences may indeed "make the difference". This is, for instance, the case of domain-specific mashup platforms [1], which aim to provide more effective development support (compared to generic tools) by tailoring their composition language to a specific domain and its very own needs. That is, despite the existence of standard languages like BPEL, there are good reasons for having different languages for different uses and different users.

Designing a composition language is however *not an easy task*. There are lots of conceptual and technological choices to be made, such as (i) which *components* to support (e.g., SOAP services, RESTful services, UI widgets, or proprietary component technologies); (ii) which *composition logic* to adopt (e.g., event-based, control flow, data flow, blackboard-like data exchange, and so on); (iii) which *data integration* capabilities to support (e.g., parameter mapping, template-based transformations, scripts, etc.); and (iv) which *presentation* features to provide, if any (e.g., UI templates, UI widgets, single pages, multiple pages). All these choices do not only affect the structure of the composition language, but eventually they determine the complexity and viability of the composition platform built on top. A careless selection of features and constructs inevitably results in inconsistent languages and tools. Even worse, oftentimes developers are not even aware of which choices need to be made and which options are available, or they do not understand which implications an individual choice has on another choice. For example, it does not make sense to support both control flow and data flow based composition logics in one and a same language, as both paradigms specify the order in which component operations are to be invoked. The former explicitly defines this order independently of how data is passed from one component to another; the latter defines the order implicitly focusing instead on how data is passed among components. Having both together could thus lead to duplicate - possibly inconsistent - definitions of the operations' invocation order.

Recognizing this difficulty, which we experience ourselves in the development of our mashup tools, with this paper we would like to lay the foundation for the *conceptual design of custom composition languages* for mashup tools, an approach that aims to modularize and reuse language construction knowledge. The idea is to enable a developer to reason at a high level of abstraction about the composition language he would like to obtain and to allow him to interactively construct his language by specifying the set of composition features that characterize his target language - everything by guaranteeing the soundness, i.e., consistency, of the final result. With the help of a hosted design tool, we would like to provide custom composition language design *as a service* and equip the design tool with an according, hosted runtime environment (an execution engine) that is able to execute compositions/mashups expressed in any of the languages constructed with the tool. The final

objective is very ambitious. The approach is to start with a set of core functionalities and to extend this set over time as new requirements emerge. The *contributions* we provide in this paper are:

- We provide a comprehensive conceptualization of the most important *composition features* that characterize todays most prominent composition languages.
- We derive a *generic, extensible composition language meta-model*, which expresses how the identified features can be used together for the construction of custom composition languages.
- We modularize the identified composition features into *reusable language patterns*, and equip the patterns with a simple logic-based language to express feature composition constraints and to guarantee consistency.
- We generate *custom composition languages* and according custom component description languages from the developer's selection of composition features.

The *structure* of the remainder of the paper is as follows. Next, we provide an example scenario and some background knowledge on composition language features. Then, in Section 4, we describe key requirements and our problem statement. In Section 5, we outline our approach. In Section 6, we describe our generic composition language meta-model, and in Section 7 we describe the structure of composition features. In Section 8 we show two composition language definition examples, in Section 9 we discuss related works and in Section 10 we conclude.

2 Scenario

Let's assume we need to develop a custom composition language with specific properties. Specifically, let's assume we want to develop a mashup language presenting the same characteristics of the language used by the mashArt mashup platform [4], which we developed from scratch in the context of the mashArt project. A simple example of a composition instance that the language must be able to support is the one presented in Figure 1: we want to allow any user to search for a given - user-selected - object in a specific - user-selected - geographical area and to get a list of results. Then, by selecting one of the results the user will see its location displayed on a map and will be provided with the traffic information related to the geographical area around this location. For example, a user must be able to look for hotels in Miami, get a list of hotels in the city and, when selecting one of them, visualize its location on a map and have the traffic information regarding the area around the selected hotel. This example shows the need for the integration and synchronization of data, business logic and user interfaces.

Concretely, we need a mashup language allowing one to integrate data, application logic (e.g., through Web services) and graphical UI components. This is what we called *universal integration* in the context of the mashArt project. Moreover, as shown in Figure 1, the language has to support the presentation of the UI components inside a single Web page, manage their synchronization (considering the

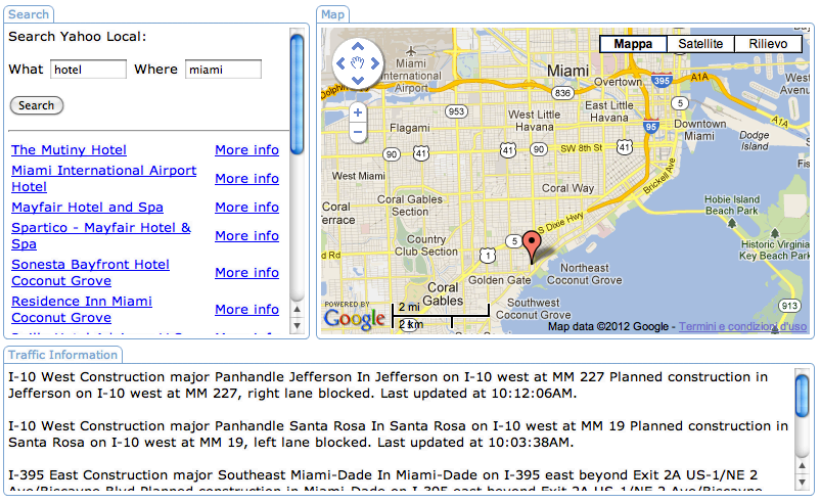


Fig. 1 Example of mashup application the mashArt language must support

event-based nature of UIs), and allow for the explicit definition of the data flow schema enabling components to exchange data. Propagating data among components may require conditional execution of flows, as well as branching and merging of parallel flows. UI components, which are implemented in JavaScript, can possibly have parameters for their configuration and one or more operations including an arbitrary number of input and output parameters. Web services are typically SOAP-based or RESTful. The resulting mashups are accessible to any user in a single-user fashion; thus, no user management or collaboration support by the language is needed.

3 Background: Software Composition

The scenario shows that mashup development is an intricate software integration and composition endeavor. As highlighted in [1], next to the integration of data and application logic, mashups also feature integration of user interface, i.e., UI integration. Figure 2 graphically illustrates the situation from a conceptual point of view and contextualizes the three integration layers in the domain of the Web with its very own component technologies

Data level integration. When the focus is on the integration of data, we have specific needs to address. Typically, solutions for retrieving, combining, splitting and transforming data are needed. In addition, when more than one entity is involved in the data integration process data exchange among the involved parties may be

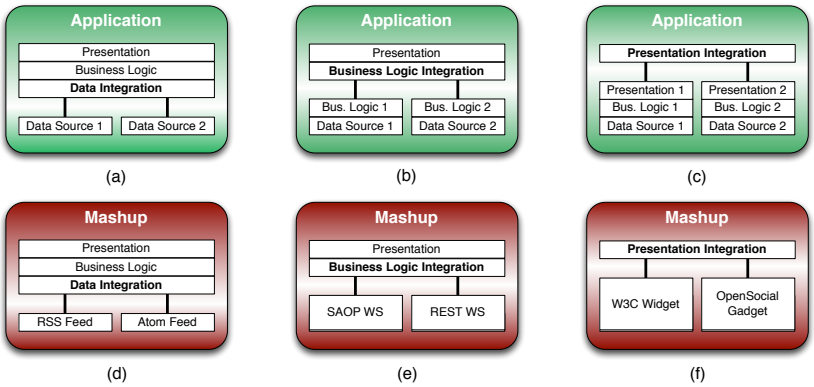


Fig. 2 The different levels of integration in general and in the specific context of web mashups.

needed. In the context of Web mashups, we have specific conditions and constraints. Data sources are typically not fully accessible, i.e., the standard way of retrieving data on the Web is through Web services or Web APIs. This means that we can only access the data provided by the service and we cannot make arbitrary complex, free queries over the data source, as we could do with conventional databases. The key problem of data integration is understanding which data items are semantically similar to which other data items and solving possible formatting differences. Mashups aren't any different. They usually integrate data coming from completely independent sources, which were not designed to work together; thus, data format and structure mismatches must be solved. Mechanisms to address these kinds of problems span from simple data mapping solutions, allowing one, e.g., to map part of the output of one service onto (part of) the input of another service, to more powerful solutions supporting data transformation languages and processors (like, e.g., XSLT). On the other side, though, on the Web there are official and de-facto standards that are oftentimes adopted (e.g., RSS and Atom feeds, XML and JSON formats), which simplify data integration in that they standardize the syntax and partly also the semantics of data (e.g., RSS and Atom).

In the mashup context, considering also the usual intent to keep the tools' complexity as low as possible, a well-known and widely adopted paradigm for data integration is *data flow* integration. Specifying a data flow among components means explicitly expressing (e.g., visually modeling) how data flows from one component to one or more other components, thereby also stating an order of invocation of components (the flow) and respective activation conditions (the availability of input data). In other words, a data flow based composition logic implies also a control flow logic, i.e., an execution order of components. With the term component we specifically refer to software artifacts (e.g., Web services) exposing public functions (also called operations) providing for data provisioning or processing. Data travelling along a flow are visible only to the component involved in the flow. Data flows allow the easy implementation of data mappings, e.g., by creating separate data flow

connections for each communicating output-input pair. Features like data aggregation, splitting or transformation can be supported by the composition language or through dedicated components offering these kinds of functionalities as a service.

The data flow paradigm is, for instance, the solution adopted also by Yahoo! Pipes (http://pipes.yahoo.com/pipes/), a popular example of data mashup tool. Pipes allows users to mash up components retrieving and processing data (typically structured as data feeds) and to set up data flows (so-called pipes), allowing the produced data to flow through the composition.

Business logic level integration. When the main target is instead the integration at the business logic level, the key requirement is orchestrating the services implementing the different pieces of business logic to be integrated. In concrete terms, the developer must be able to explicitly define the order in which component operations are to be triggered. The most suitable composition paradigm supporting these features is the *control flow* paradigm. Specifying a control flow means specifying when to enact which component inside a composition. Doing so may require the definition of conditional flows, of flow branching (i.e., parallel flows) and flow merging (i.e., parallel flows synchronization).

Examples of pure control flow based compositions can be developed, e.g., in BPMN, which offers many control flow related constructs including conditions, loops, parallel flows and so forth. Although the focus of the control flow paradigm is on the order of tasks or components, executing them usually requires complementary data passing mechanisms to feed them with the necessary inputs. In combination with the control flow paradigm, the *blackboard approach*, i.e., global variables holding data produce and consumed at runtime, is typically used for this (note that the "data flow" constructs of BPMN do not express a data flow based composition logic, but rather the writing and reading of business data). This scheme is also used in the BPEL language, where the main target is the integration of SOAP-based Web services.

Presentation level integration. As mentioned, in other cases the main focus is on the integration of user interfaces at the presentation layer. In this case the composition language must support the graphical representation of UI components with suitable constructs. Also in this case, our focus on Web mashups sets specific constraints. UI presentation takes place inside the browser, normally in standard HTML pages. As shown by the example of Figure 1, typically a Web page may contain one or more UI components. UI components are software artifacts that have two main functions: show a graphical user interface and provide users with a point of direct interaction with the composition through their interfaces. UI components usually require synchronization, in order to have them show related content. Typically the interaction mechanism implementing UI synchronization is event-based, since UI development is intrinsically event-based and it is just not possible to predict when and in which order user interactions will take place (which makes asynchronous events a good instrument to manage communication among components). Support for data passing among UI components may also be needed and can be implemented following either the data flow or the blackboard paradigms.

Concretely, in the mashup world, languages supporting presentation features typically include two additional concepts to lay out UI components: *pages* and *viewports*. A viewport is a placeholder where a UI component is hosted and rendered (e.g., a `div` or `iframe` element contained in an HTML page). A page can contain one or more viewports, allowing for the presentation of integrated user interfaces. These concepts are present in the models of several mashup tools, e.g., mashArt and JackBe Presto, as well as in the W3C Widgets family of specifications (where the term viewport itself comes from).

Having user interfaces oriented toward human users opens to the introduction of other composition features, such as user authorization and management mechanisms in the case of mashups with multiple pages. Individual pages may be assigned to specific user roles, allowing for the definition of multi-user, collaborative mashup applications where several users can work on a shared mashup instance acting on the pages they have access to. This is, for instance, one of the main features in the MarcoFlow platform [5].

4 Requirements and problem statement

What does it now mean to develop a *custom* composition language for mashup design and to support its execution? In order to answer this question, first of all we define a *custom composition language* as a composition language that is specifically tailored to a given combination of component types and a target application/service type (mashup type). We represent a language (we use the terms *language* and *composition language* interchangeably) by means of its meta-model or XSD schema. Standard languages like BPEL [7] or BPMN [8] are very focused languages that are generally not able to satisfy the requirements of a mashup platform, since mashups typically go much beyond the orchestration of SOAP web services or human tasks.

In order to develop a custom language, we generally have different *design options* that allow us to achieve the desired expressive power:

- *Development from scratch*: This is the current practice that we want to prevent. Developing a language from scratch means designing the language without any reference by looking at the composition problem to be solved and by deriving suitable, ad-hoc composition constructs. This task is more complex than it looks like and often leads to poorly designed, inconsistent languages, which can only be run by specifically tailored runtime environments.
- *Selection of off-the-shelf language*: This is the other, ideal extreme, in which for each component and mashup type combination we have a pre-defined language that supports all features of the given combination. Implementing all these languages is not feasible, in that the number of potential languages (and execution engines) grows combinatorially with the number of component types and features of the target mashups. Also, the introduction of a new component type or feature would require the update of the whole languages library.

- *Extension of existing language*: A practice that works in many situations is to take an existing language, e.g., BPEL, and to extend it with new constructs and semantics, so as to support custom features. Starting from a known language eases the adoption of the extended language, but it is typically hard to identify a suitable language, and changes to the original language may involuntarily introduce inconsistencies into the custom extension. Even with small extensions, the language's own engine can usually no longer be used for execution.
- *Customization of reference language*: Another option is to provide a set of reference languages with predefined extension mechanisms. For instance, we could have reference languages for data-flow-based, control-flow-based, UI-based mashups, and combinations thereof. Yet, it is hard to predict all possible customization requirements and to maintain the library of reference languages and execution engines up to date with changing technologies and applications.
- *Modular composition of language*: Finally, we can provide a set of basic language features, such as control flow, data flow, UI synchronization, and the like and allow the developer to compose his own language. Newly emerging features can be added to the feature library without invalidating prior language specifications. Given a library of language features, it suffices to implement only one execution engine that is able to understand all the features, in order to be able to execute a large set of custom mashups.

In this paper we specifically focus on the problem of developing custom languages, while our vision is also to provide runtime support for custom languages; the modular composition approach seems therefore most suitable. But which is a good granularity for *reusable language modules*? We again have several options:

- *Individual language constructs* (with the term *construct* we generically refer to both meta-model and XSD constructs): Constructs like components, pages, ports, inputs, outputs, connectors, and similar are the basic ingredients for every language. Yet, constructs represent the lowest level of granularity of a language. It is therefore hard to encode reusable language construction knowledge, if not in the form of a library of typical composition constructs. How to use each construct, in which combination with other constructs, for which typical modeling situation, and so on can however not be expressed.
- *Composite constructs*: Modules may express composite constructs, such as the structured elements sequence, parallel flow, and loop, typically used for the construction of well-formed models. This technique aids the development of composition languages that are sound, but it is still very syntactic and does not support reuse of more complex language construction knowledge.
- *Language patterns*: Modules may also express more complex usage patterns of constructs that represent semantically meaningful composition language properties, such as control flow, data flow, UI synchronization, component types, asynchronous vs. synchronous communications, etc. If such patterns are further equipped with suitable language composition constraints, it is also possible to guarantee their sound composition.

Given our experience with the reuse of modeling knowledge [2], we advocate the use of semantically meaningful language patterns to represent reusable language composition knowledge. We call these patterns *language features*, since they allow us to represent composition features in an abstract fashion. The question that remains to be answered is therefore which language features must be provided, so as to support the construction of a reasonably wide set of possible languages. Looking at set of existing mashup approaches [3][4][6] and standard composition languages [7][8] and without trying to crack the whole problem at once, we identify five key aspects (groups of features) that influence the expressive power of a composition language:

1. *Component types*: First and foremost, the *object* of the composition, i.e., the types of components, influences the whole logic of the language most prominently. There are many possible component technologies to take into account, such as SOAP web services, RESTful services, UI widgets, JavaScript classes, plain XML or CSV data sources, and similar. Composing UI widgets is, for example, fundamentally different from orchestrating web services.
2. *Control flow logic*: Next, it is important to define how the *computation* of a composite application or service is enacted, that is, how and when individual components are processed. Components may be enacted in parallel (e.g., in the case of simple UI widgets placed in a web page), they may be executed sequentially, their execution may be subject to conditions, and so on. The possibility to integrate heterogeneous component technologies (e.g., UI widgets and web services) further increases the number of available control flow options, if the control flow paradigm is required at all.
3. *Data passing logic*: In addition to the control flow logic, the language must be able to express how data is *propagated* among components. While data flow paradigms typically bring together aspects of both control flow and data passing, other paradigms like pure control flow or UI synchronization may rather adopt a blackboard approach with global variables.
4. *Presentation logic*: One of the distinguishing features of mashups is that they also feature *integration of user interfaces*, not only services and data sources. This however asks for specific techniques to lay out and render UI elements. For instance, we may make use of HTML templates with placeholders or we may have automatic arrangements of UI widgets, there might be the need of special visualization components for data sources, and so on.
5. *Collaboration support*: Finally, mashups can be much more than simple, one-page applications. We can have mashups that implement *collaborative* business processes with different actors per task, or we can have mashups that support the *concurrent* use of individual pages by multiple users. Supporting these features requires the possibility to express at least roles of users and to assign them to pages, while more complex logics can be envisioned.

The *problem* we want to solve in this paper is to *enable developers to design custom composition languages in an abstract, conceptual fashion*, supporting the five above feature types and guaranteeing that the final languages come without

internal inconsistencies, i.e., that they are *sound*. Our focus is on imperative mashup languages that can be executed by a mashup engine.

5 Approach

Figure 3 graphically illustrates how we decompose the problem into artifacts and how we finally obtain a custom language. The idea is to express a *custom composition language* as a set of *composition features* that give the language its expressive power. Features come with a set of *feature constraints*, which express feature compatibilities, conflicts, and subsumptions. For each of the five types of composition features discussed above, we provide a set of concrete features (we discuss them next). Each feature has a *reference specification*, i.e., a pattern of language constructs, which implements the feature and represents reusable language composition knowledge. Patterns are based on a *generic composition language meta-model*. The meta-model does not yet represent an executable language. It syntactically puts composition constructs and features in relation with each other, but it also contains constructs and features that are not compatible with each other (e.g., control flow and data flow constructs). The meta-model determines which features are supported and how they are syntactically integrated; the sensible design of feature constraints provides for soundness. Hence, given a set of non-conflicting composition features, the custom composition language is represented by the *union* of the respective reference specifications. Similarly, we derive a *custom component description language*, which can be used as guide for the implementation of components and to describe their external interfaces.

In the following, we first construct the generic meta-model, then we describe how we define composition features on top using patterns and constraints and how patterns can be used and integrated for the development of custom languages.

6 The generic composition meta-model

Before going into the details of the language meta-model, we introduce the meta-meta-model it complies with, as such is also the basis for the final code generation.

6.1 Language meta-meta-model

To design the meta-model for the composition languages, we use a notation and modeling language derived from the UML Class Diagram with some peculiarities. Specifically, we impose some constraints on the allowed types of modeling constructs, tailoring them to the expressive power required by our modeling needs. As

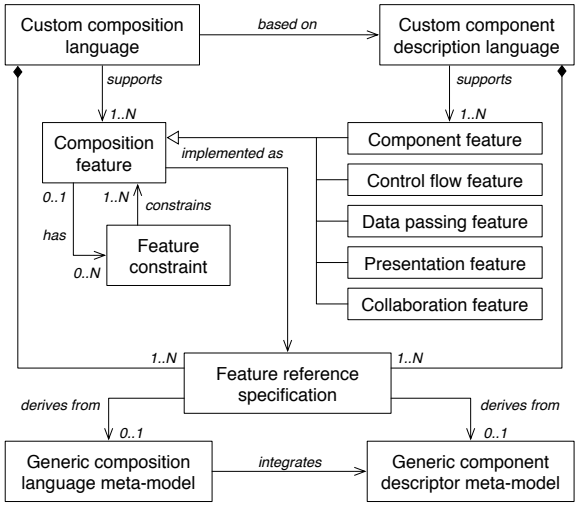


Fig. 3 Conceptual approach to the development of custom composition languages

detailed in Section 6.3, applying these constraints allows for an unambiguous translation of the meta-model into a formal - and machine-readable - language schema definition, which is then needed for the definition of other artifacts of the system. In addition, using this constrained modeling language also opens to future extensions of the meta-model by third parties, making them aware of the implications of each model extension or modification on the resulting language definition (since deterministic translation rules are defined). Concretely, as defined by the meta-meta-model depicted in Figure 4, the meta-model may consist of:

- *Entities*. Represent main constructs of the composition language. They are identified by a name.
- *Attributes*. Each entity can have a set of related attributes characterizing it. Attributes have a name and a type. The type can be stated through its name or can be explicitly defined in form of enumeration of possible values. To be noticed, each entity in our meta-model must contain an attribute named *id*, representing a unique identifier for the instances of the entity used to reference them.
- *Associations*. Relations among the entities are expressed through associations. Only two possible types of associations are needed: *composition* and *uni-directional* association. The composition is used to state that an entity is contained in another one, while the uni-directional association states that an entity simply refers to another entity, but it is not contained in it.
- *Cardinalities*. Represent associations' multiplicities. The target cardinality represents the multiplicity of the association when reading it following the specified association direction, while the source cardinality represents the multiplicity when reading the association in the opposite direction.

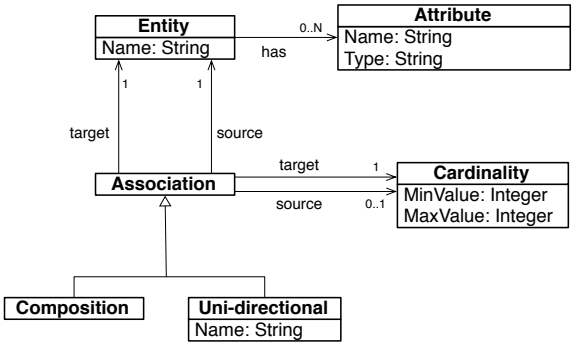


Fig. 4 Composition language meta-meta-model

6.2 The generic meta-model

In essence, our approach is to *compose* composition languages out of composition features represented as language patterns. Just like in any other composition approach, the core problem is therefore the identification and formalization of the "components" to work with. In our case, these components are *language patterns* (e.g., XSD fragments). However, these patterns have a distinctive feature that makes our problem very different form generic component-based development (next to the fact that we do not handle software modules but document/model fragments): unlike, for example, web services, *language patterns are not independent*. That is, the reference specifications of different composition features may overlap (e.g., interacting with a *SOAP service* is very similar to interacting with a *RESTful service*), include other features (e.g., the *data flow* paradigm generally subsumes the presence of *data source components*), or exclude others (e.g., the *data flow* paradigm does not make use of *variables*). This asks for a thorough design of the language patterns and their mutual interaction points, a task that we achieve by mapping each composition feature into the *generic composition meta-model* (see Figure 5), which (i) integrates all basic language constructs syntactically, (ii) allows us to define composition features as language fragments on top, and (iii) guarantees that fragments are compatible by design.

We have identified several dozens of composition features that can be used to describe the expressive power of mashup languages. In the following paragraphs, we overview the features and provide some examples. For space reasons, however, we refer the reader to an online resource (http://goo.gl/hfkLO) for the list of supported features and respective details. The list of identified features comes without the claim of completeness and is meant to grow over time; however, as we will see in Section 8, we are already able to express a fairly complex set of mashup languages.

Component features. They specify which kinds of components - in terms of technologies and communication patterns - the language should support. For instance, a

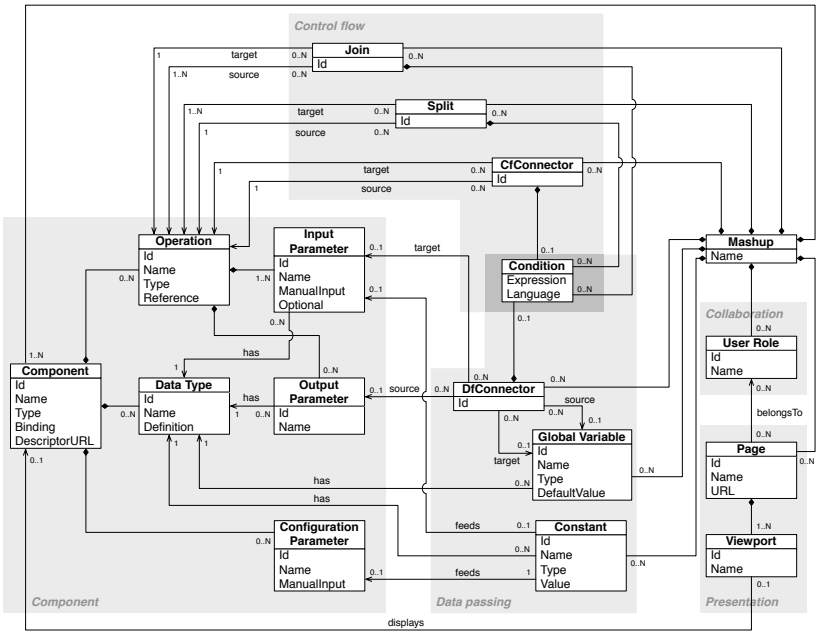


Fig. 5 The generic composition meta-model for custom languages. Gray boxes group entities into feature types. The Component group is also used to derive component descriptor languages

SOAP web service may come with message-based operations of four different types (request-response, solicit-response, one-way, notification), custom data formats for each input and output message, a service endpoint, and a protocol binding (e.g., SOAP). We represent such a service in the meta-model as a *component* that has a set of *operations* with different input/output parameter patterns (implementing the four different operation types), only single *input/output parameters* per operation to represent input/output messages, an own *data type* for each parameter, and respective *binding* and *endpoint* attribute values. Similarly, a W3C UI widget [9] can be seen as a *component* with some *configuration parameters* but without operations, which can be displayed in a *viewport* of a *page* of the mashup.

Analogously, the meta-model so far conciliates the following *technologies*, which are the basis of many types of mashups and, as such, widely used and accepted (component technologies are tracked by the *type* attribute of the *component* entity):

- Data source components: RSS feeds, Atom feeds, RESTful data components, SOAP data components, JavaScript data components.
- Web service components: Atom services, RESTful services, SOAP services, JavaScript components.
- UI components: W3C UI widgets [9], JavaScript UI components [4] (our own).

For each of these component technologies, it is then important to specify which exact *communication patterns* the language should support. For instance, the language could support only synchronous communications (operations with input *and* output parameters), only asynchronous communications (operations with *either* input *or* output parameters), or both. It might be necessary to limit the number of operations per component (e.g., in Yahoo! Pipes each component corresponds to one operation) or the number of parameters per operation (like for SOAP services as described above). All these options can be represented via patterns that suitably set the relationship cardinalities in the meta-model.

Control flow features. They specify whether the language is control-flow-based (e.g., BPMN) or not and, if yes, which control flow constructs to support. Sequential execution can be expressed by connecting operations using control flow connectors (*CfConnetors*). Parallel executions are supported via *split* and *join* constructs. Each of these constructs can have one or more *conditions*, which constrain the control flow along connectors and, for instance, allow the implementation of conditional control flow constructs like conditions, conditional split, and conditional joins. Loops can be implemented by means of conditions and joins. Events for event-based mashups (e.g., our mashArt platform [4]) are operations with only outputs. Each of these features can be added to the language by including the respective entities in the meta-model.

Data passing features. They specify whether the language is data-flow-based or not and how data is propagated among components. In data-flow-based languages (e.g., Yahoo! Pipes) it suffices to connect two operations using a data flow connector (*DfConnector*), in order to propagate the output of the first operation as input to the second operation. Implicitly, data flow connectors also determine how components are enacted and, hence, do not require any additional control flow construct. Data flows may however be subject to conditional execution. Control-flow-based languages, instead, require additional constructs to specify how data are passed among components. The most common technique is to write/read *global variables* (blackboard feature), which are accessible during the execution of a composition (e.g., as in BPEL). The meta-model represents the writing/reading operations with a data flow connector between the variable and its target/source parameter. UI-based mashups, such as widget portals, typically run all widgets in parallel, and data is passed via global variables or events (operation with only outputs). Configuration parameters are instead typically set once at the startup of a component (e.g., the background color of a UI widget); we support this by means of *constants*. Data passing may also require mapping output parameters to input parameters, a feature that can be achieved by specifying data flow connectors between parameters instead of between operations.

Presentation features. They specify whether the language is UI-based or not and how UI widgets are laid out into web pages. Unlike service compositions, mashups typically also come with an own user interface that renders UI components and data from UI-less components. The minimum support required to express this capability in the meta-model is represented by the *page* and *viewport* entities, which allow the

ordering of UI components into pages (HTML web pages) and their rendering in selected areas inside these pages (typically `div` or `iframe` HTML elements). We assume the HTML pages are given and already linked to each other as necessary.

Collaboration features. They specify whether the language describes single-user or multi-user mashups and how user roles collaborate. Single-user mashups (the most common type of mashups) do not require any user management. Multi-user mashups, instead, may restrict the visibility of individual *pages* to selected *user roles* only. Users may have different views on a mashup (e.g., via different pages) or they may have the same view (e.g., via the concurrent use of a same page). For the time being, we start with a simple, role-based user management logic and do not say anything about how such is implemented, as this is a runtime choice.

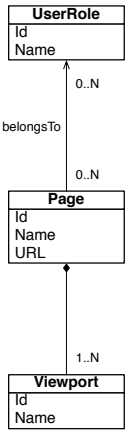
The above features and examples show that developing a good generic meta-model is a *trade-off* between the simplicity and usability of the final language (the fewer individual constructs the better) and the ease of mapping features onto the meta-model (the more constructs the better; in the extreme case, each feature could have its own construct). The challenge we try to solve in this paper is exactly that of identifying the right balance between the two, so as to be able to map all relevant features and to do so in an as elegant as possible fashion from the resulting language point of view.

6.3 Mapping the generic meta-model to XSD

The information represented by the generic meta-model constitutes the basis for the definition of the feature reference specifications (see Section 7.1) and is required by the language generation algorithm (see Section 7.3). Therefore, we need to serialize the generic meta-model in a machine-readable format. To this aim, also considering the context where mashup languages are used (i.e., the Web), we map the meta-model onto an equivalent XSD definition. As introduced in Section 6.1, we impose some simple conventions and constraints to the admitted modeling constructs for the meta-model so that we can define a set of rules which guarantees an unambiguous translation of the model.

Figure 6 exemplifies how the generic meta-model is translated into an equivalent XSD definition applying the following translation rules:

- Entities (e.g., *page*) are translated as XSD elements having the same name of the entity.
- Entity attributes (e.g., a page's *URL*) are translated as XSD attributes of the related element having the same name of the entity's attribute.
- Composition associations (e.g., the one having *viewport* as source and *page* as target) are translated defining within the element associated to the target entity an XSD child element (with zero or more possible occurrences depending on the specified cardinality) having the name of the source entity (e.g., the element *page*




```
<xs:element name="userRole">
      <xs:complexType>
            <xs:attribute name="id" type="xs:string" use="required" />
            <xs:attribute name="name" type="xs:string" use="required" />
      </xs:complexType>
</xs:element>

<xs:element name="page">
      <xs:complexType>
            <xs:sequence>
                  <xs:element name="viewport" minOccurs="1" maxOccurs="unbounded">
                        <xs:complexType>
                              <xs:attribute name="id" type="xs:string" use="required" />
                              <xs:attribute name="name" type="xs:string" use="required" />
                        </xs:complexType>
                  </xs:element>
                  <xs:element name="belongsTo_userRole" minOccurs="0" maxOccurs="unbounded">
                        <xs:complexType>
                              <xs:attribute name="ref" type="xs:string" use="required" />
                        </xs:complexType>
                  </xs:element>
            </xs:sequence>

            <xs:attribute name="id" type="xs:string" use="required" />
            <xs:attribute name="name" type="xs:string" use="required" />
            <xs:attribute name="URL" type="xs:string" use="required" />
      </xs:complexType>
</xs:complexType>
```

Fig. 6 Example of translation of a meta-model fragment into XSD

has 1 to N child elements *viewport*). As shown in the example in the figure, the child elements are contained and defined within the parent element.

- Uni-directional associations (e.g., the one having *page* as source and *userRole* as target) are translated defining within the element associated to the source entity an XSD child element (with zero or more possible occurrences depending on the specified cardinality) having the name of the form "associationName_targetEntityName" and including an attribute *ref* designed to contain a reference (i.e., the ID) to a target entity instance (e.g., the element *page* may have 0 to N child elements *belongsTo_userRole*). The child elements only refer to the target entity and do not define it.

Applying the above translation rules to the meta-model presented in Figure 5 we obtain an equivalent XSD definition that we use as base for the production of the artifacts and algorithm presented in the next section. The complete schema definition can be inspected at http://goo.gl/hfkLO.

7 Representing and assembling composition features

The meta-model in Figure 5 solves the problem of integrating the composition language constructs needed to specify a varied set of composition features. Designing the meta-model required both the analysis of the features to be supported and knowledge about their implementation in terms of language constructs. We aim to abstract away from low-level language constructs and represent concrete composition features on top of the generic meta-model so as to allow the language developer to focus on the selection of features only, in order to design his custom language.

We define a composition feature as $f = \langle name, label, desc, spec, Constr \rangle$, where *name* is a text label that uniquely identifies the feature (e.g., `data_flow`); label briefly describes the feature and expresses its semantics; *desc* is a natural language verbose description of the feature for human consumption; *spec* is the reference specification of the feature; and $Constr = constr_i$ is a set of feature constraints.

```
<feature name="condition" label="Conditions">

   <description> Conditions can be set for each connector to define the
        possible flows of the composition. Conditions are supported both for
        control flow and data flow composition paradigms.
   </description>

   <specification>

      <include fragments="conditionForCf" if="control_flow"/>
      <include fragments="conditionForDf" if="data_flow"/>
      <include fragments="conditionForSplit" if="split"/>
      <include fragments="conditionForJoin" if="join"/>

   </specification>

   <constraints>
      (control_flow AND blackboard) OR data_flow
   </constraints>

</feature>
```

Listing 1 XML reference specification of the condition composition feature

The XML code in Listing 1 shows an example of how we serialize, for instance, the `condition` feature in our *feature knowledge base* of the form $F = f_j$. The example shows the two core ingredients that allow us to collapse the assembling of features into a simple selection of feature names: First, the *reference specification* of the feature expresses which specific language constructs - out of all those represented in the generic meta-model - are needed to implement the feature. From the XSD representation of the generic meta-model (see Section 6.3), we identify given subsets of the schema definition representing semantically meaningful parts of it. An ID uniquely identifies each of these fragments in the XSD. Second, the *feature constraints* state feature compatibilities or incompatibilities. They are simple Boolean conditions. We detail these two aspects in the following.

7.1 Feature specification language

In order for feature specifications to be composable, we adopt a constructive approach that starts with an empty language specification (we call it the *base language*), which contains only the basic XSD structure (e.g., name space definitions and types) for the language to be generated, and then incrementally adds new constructs based on the specifications the of selected features. Since a given feature may span multiple constructs of the meta-model, a feature reference specification generally requires multiple language fragments (identified through manually assigned IDs) to be included in the final custom language definition. For instance, the specifi-

cation of the `blackboard` feature requires several fragments to be included, e.g., those related to the specification of the *Global Variable* construct and those related to the specification of the *DfConnector* construct used to connect variables and parameters. The syntax to require the *inclusion of the fragments* referenced by a given feature is as follows:

```
<include
        fragments="[comma separated list of fragments IDs]"
        if="[condition]" />
```

Each feature specification contains one or more `include` elements that are composed by an attribute `fragments` listing the fragments needed to implement the feature in the custom language XSD definition. The referenced IDs relate to XSD fragments defining elements, attributes, enumerations and similar. In addition, the `include` element may optionally contain an attribute `if` that can be used to require a conditional inclusion of the referenced fragment(s). In particular, the condition can require the selection or non-selection of other features for the inclusion to be performed (as exemplified in Listing 1). The fragments come with default values for cardinalities (i.e., values for the `minOccur` and `maxOccur` XSD attributes), as specified in the meta-model in Figure 5. Some features, such as the `max_1_operation_per_component` or the `single_page` features, may need to modify them. In order to change cardinality values, we provide a dedicated cardinality setting function with the following syntax:

```
<setCardinality
        element="[elementID]"
        minOccurs="[value]"
        maxOccurs="[value]" />
```

The function has three attributes, which allow us to select which XSD `element` in the current language specification to modify and which `minOccurs` and/or `maxOccurs` values to assign to the element. It can be noticed that an association's cardinality setting involves only one XSD element. This is because, according to our translation rules, associations are translated in one element that is nested into the associated element and, therefore, the cardinality setting needs only to set the number of possible occurrences of one element, i.e., the nested one.

7.2 Feature constraints language

Feature constraints are *Boolean conditions* that check (i) whether all features *required* by a given selection of features are contained in the selection and (ii) whether the selection contains *conflicting* features. Feature constraints therefore guarantee for the semantic soundness of a selection of features. Feature constraints are of the form: $constr ::= fbool \mid \neg\ constr \mid constr\ op\ constr$.
$fbool \in FB$ is a Boolean variable representing the selection (or not) of a feature, $FB = \{fb_j | fb_j = \langle name, val \rangle, name = f_j.name, f_j \in F, val = true | false\}$ is the set of Boolean variables representing all features, and $op \in \{\wedge, \vee, \oplus\}$ is one of the logical AND, OR, and XOR operators.

For example, in Listing 1 we have the constraint `(control_flow AND blackboard) OR data_flow`, since for the definition of conditions it is required the presence of some data passing mechanism in the mashup model. This is an example of constraint assessing the presence of the features required for the selected one. An example of constraint preventing conflicting features is the one associated to the feature `max_0_operation_per_component` (e.g., used for simple UI widget portals), which may state: `NOT(data_flow OR control_flow)`. It would not make sense to support any of these paradigms in a language that by definition does not allow communication among components.

In addition to assigning constraints to individual features, we assign a set of base constraints to the base language, in order to enforce global constraints that guarantee the integrity of the overall language. For instance, the constraint `(control_flow XOR data_flow) OR user_interface` asks for the selection of at least one basic mashup paradigm (e.g., a simple state machine or UI widget portal).

7.3 Language generation algorithm

Algorithm 1 summarizes the language generation logic. It takes as input a set of feature names and produces as output either an according combination of composition and component description languages or *null* (in case of constraint conflicts). After initializing the variables holding the language to be generated and the constraints to be evaluated (lines 2-3), the algorithm loads the complete feature specifications of each feature in input from the feature knowledge base (line 4) and sets the respective Boolean variables to *true* and all the remaining variables (those associated to non-selected features) to *false* (lines 5-6). This enables the processing of the `checkSoundness` function, which checks whether all the constraints associated to the selected features are satisfied. For this purpose, the function evaluates the Boolean formula contained in *CONSTR* based on the variable values assigned in lines 5-6. If the evaluation returns *false*, the function stops processing and returns *null* (lines 7-10). Otherwise, the algorithm constructs the list of IDs of all the fragments required by the selected features and the set of *setCardinality* instructions needed to update the default cardinalities (lines 11-13). Based on these sets the algorithm constructs the actual output composition language including all the fragments in the *FRAGMENTS* set and then updates the cardinalities of the elements of the resulting composition language based on the instructions contained in the *SETCARDINS* set (lines 14-15). Finally the algorithm returns the composition language definition and the component description language definition, which is extracted by the former (line 17).

Our current prototype of the language generator comes as a simple command line tool, which takes as input a text file with the list of desired language features and, if successful, produces as output two XSD files for the composition and component description languages. The feature knowledge base *F* is a plain XML file, which can easily be extended with new features.

Algorithm 1. generateLanguage

Data: Set of selected feature names *FnameSel*

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

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

8 Examples

In the following sections, we apply the conceptual design approach introduced above to two concrete examples with different requirements.

8.1 mashArt

In Section 2, we stated a set of requirements for the mashArt composition language. In the following, starting from these requirements, we derive the set of features (emphasized in `Courier` font in the following paragraphs) to be given as input to our generation algorithm to produce a mashup language supporting our scenario.

As said, mashArt aims at integrating data, business logic and user interfaces. Therefore, `data_component`, `service_component` and `ui_component` features are required to support all the different types of needed components. All the components must be implemented through JavaScript, therefore the features `javascript_for_data`, `javascript_for_service` and `javascript_for_ui` have to be included. In particular, data components must support only `request_response_for_data` operations, service components both `request_response_for_service`, `one_way_for_service`

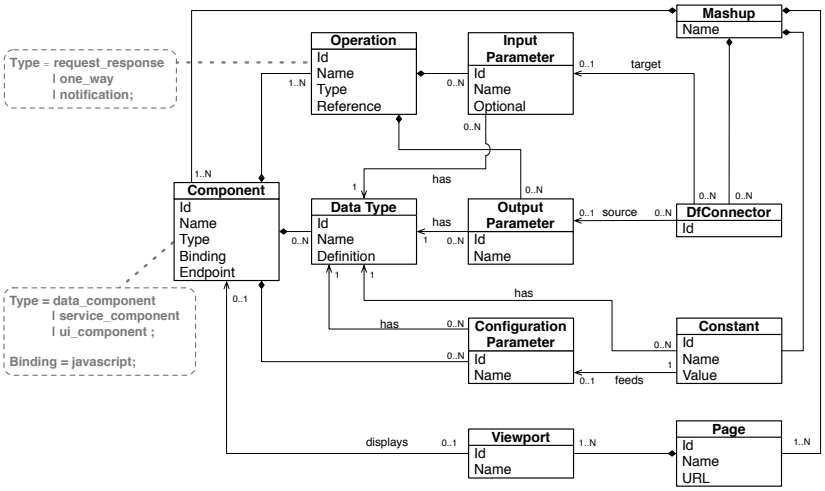


Fig. 7 A composition language meta-model supporting the discussed features set

and `notification_for_service` operations and UI components only `one_way_for_ui` and `notification_for_ui` operations. The requirements do not include isolated UI components (i.e., widgets), so all components will have minimum one operation, while no maximum number of operations per component is required (`max_N_operation_per_component`). Also the number of input and output parameters per operation should not be constrained to any limit (`max_N_input_param_per_oper` and `max_N_output_param_per_operation`). Clearly, it is also required to support the display and layout of UI components, which is fulfilled by the `user_interface` feature. In particular, we require compositions to be constituted by a `single_page`. The components' intercommunication, according to the requirements, must be supported through the `data_flow` mechanisms. In addition, `merge` and `branch` features are explicitly required.

The above paragraph provides the list of features supporting our scenario (the only design artifact to be produced) to be given as input to the language generation algorithm shown in Algorithm 1. Doing so produces an XSD specification for the composition language that is equivalent to the meta-model illustrated in Figure 7.

For space reasons we cannot include the whole XSD specification, which can be inspected at http://goo.gl/hfkLO. Listing 2, though, provides an excerpt of the XML definition - compliant to this specification - representing the example scenario introduced in Section 2 (i.e., geo-localized search with traffic information).

```
<mashup name="GeoLocalSearchWithTraffic">
   <component id="C1" name="Yahoo Local Search" type="ui" binding="javascript"
              endpoint="http://...">
      [...]
      <operation id="OP2-1" name="Item Selected" type="notification"
                 reference="itemSelected">
```

```
        <output id="O2-1" name="Latitute" dataType="double"/>
        <output id="O3-1" name="Longitue" dataType="double"/>
        <output id="O4-1" name="Zoom Level" dataType="int"/>
        <output id="O5-1" name="Label" dataType="string"/>
      </operation>
   </component>

   <component id="C2" name="Google Map" type="ui" binding="javascript"
              endpoint="http://...">
      [...]
      <configurationParameter id="CP1-2" name="latitude" dataType="double"
                              manualInput="yes"/>
      <configurationParameter id="CP2-2" name="longitude" dataType="double"
                              manualInput="yes"/>
      <configurationParameter id="CP3-2" name="zoomLevel" dataType="int"
           manualInput="yes"/>
      [...]
      <operation id="OP1-2" name="Show Point" type="one-way" reference="
           showPoint">
         <input id="I1-2" name="longitude" dataType="double" optional="no" />
         <input id="I2-2" name="latitude" dataType="double" optional="no" />
      </operation>
   </component>

   <component id="C3" name="Geo Names" type="service" binding="javascript"
              endpoint="http>//...">
      [...]
      <operation id="OP1-3" name="Get address" type="request-response"
                 reference="getAddress">
         <input id="I1-3" name="longitude" dataType="double" optional="no" />
         <input id="I2-3" name="latitude" dataType="double" optional="no" />
         <output id="O1-3" name="city" dataType="string"/>
         <output id="O2-3" name="street" dataType="string"/>
      </operation>

   </component>

   [...]
   <constant id="CNST1" name="Latitude" dataType="double" value="46.0667"
             feeds_configurationParameter="CP1-2"/>
   <constant id="CNST2" name="Longitude" dataType="double" value="11.1333"
             feeds_configurationParameter="CP2-2"/>
   <constant id="CNST3" name="Zoom Level" dataType="int" value="13"
             feeds_configurationParameter="CP3-2"/>
   [...]

   <dfConnector id="DF1" source_output="O2-1" target_input="I1-2" />
   <dfConnector id="DF2" source_output="O3-1" target_input="I2-2" />
   <dfConnector id="DF3" source_output="O1-1" target_input="I1-3" />
   <dfConnector id="DF4" source_output="O2-1" target_input="I2-3" />
</mashup>
```

Listing 2 XML definition of the example mashup application presented in Section 2

Figure 8 shows how the example scenario can be modeled using the graphical syntax we adopt in the mashArt editor. It can be noticed that all the main composition features supported by the existing editor are also supported by the language produced by our system, which are summarized on the right side of this figure.

8.2 Yahoo! Pipes

In the following, we derive part of the mashup language underlying the popular mashup platform Yahoo! Pipes from an example modeled in its graphical editor.

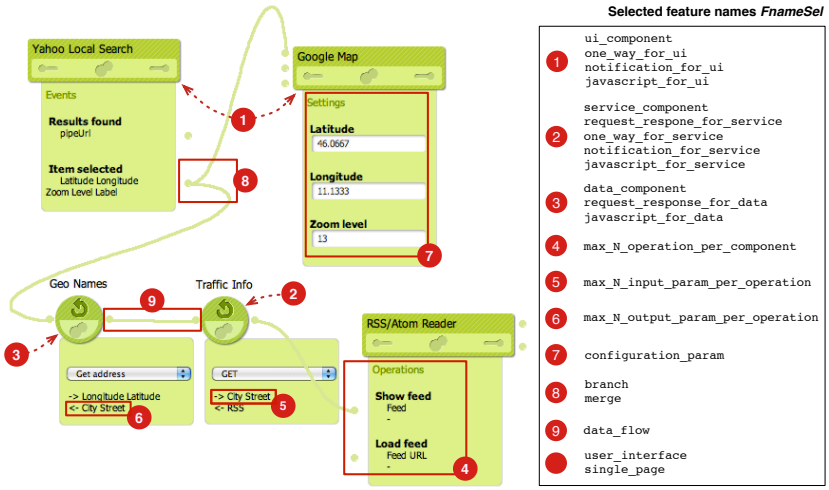


Fig. 8 mashArt example composition model and the set of respective language features

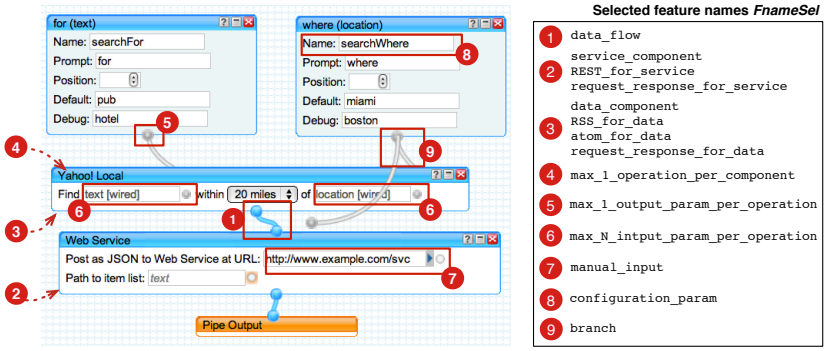


Fig. 9 Yahoo Pipes example composition and set of respective language features

Pipes is a data mashup tool for the retrieval and processing of web data feeds. Figure 9 shows an example Pipes model, which we use to analyze Pipes' language features.

Pipes is based on the `data_flow` paradigm. It supports `data_component` and `service_component` types to retrieve and process data, respectively. Specifically, data source components types are `RSS_for_data` or `atoms_for_data`, while the only supported service component type is `REST_for_service`. Each component in Pipes provides exactly one function, that is, each component represents one single operation. Therefore `max_1_operation_per_component`. All operations are of type request-response (`request_response_for_data` and `request_response_for_service`). Each operation may have one or more inputs (`max_N_input_param_per_operation`) but one and only one output

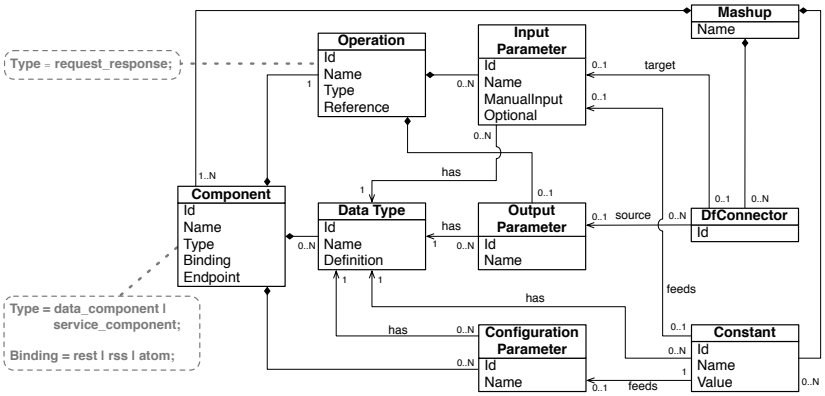


Fig. 10 A composition language meta-model supporting the discussed features set

(`max_1_output_param_per_operation`). Manual inputs (`manual_input`) are used to fill the values of input fields, i.e., of `configuration_parameter`(s). Some inputs can be fed with both an input pipe and a manually set constant value. Also in this example, the output of a component can be the source for an arbitrary number of dataflow connectors, allowing one to `branch` the data flow into parallel flows. Input parameters, instead, have at most one input pipe; so, there is not need for any `merge`.

The language produced by the language generation algorithm (defined in Algorithm 1) giving as input to it the described features is equivalent to the meta-model illustrated in Figure 10. The respective language XSD specifications and the XML model of the scenario can be inspected online at http://goo.gl/hfkLO.

9 Realated work

The problem we aim to solve in this paper, i.e., supporting the design of custom mashup/composition languages, has not been addressed before. Most contributions in the area of mashup and service-oriented computing focus on the design of specific languages taking into account, for example, quality of service [10], adaptivity or context-awareness [11], energy efficiency [12], and similar. We instead propose a language (the composition features) for the design of languages - *a model weaving approach* (at the meta-model level) for *black-box composition* languages (e.g., mashups), in the terminology of Heidenreich et al. [13]. The problem is very complex, but our analysis of a large set of mashup tools and practices has shown that the design space for non-mission-critical mashups (without fault handling, compensations, transactions, etc.) is limited and manageable, up to the point where we can provide mashup execution as a service for a large class of custom languages.

If we compare the meta-model in Figure 5 with, for example, that of BPEL [7] (see also http://www.ebpml.org/wsper/wsper/ws-bpel20b.png) or XPDL, we notice a bias toward *simplicity*. The reason for this is that mashup platforms (our target) aim to simplify composition, typically moving complexity from the composition to the components. For instance, it is common practice to have a dedicated *data filter* component, instead of a filter construct at language level (see, for example, Yahoo! Pipes). The meta-model we propose in this paper shares this interpretation for both the component model and the composition model. Also Saeed and Pautasso [14] have a similar perspective, but they focus on the design of a generic mashup component description language only and do not elaborate on their composition. Their model contains technology aspects (e.g., component wrappers), which are instead a runtime aspect. We only propose the use of component types and bindings.

A proposal toward the standardization of a generic mashup language, covering as many different uses as possible, is represented by the Open Mashup Alliance's EMML (Enterprise Mashup Markup Language) specification [15]. The target of the initiative is however different: data mashups. In our view the key novelty mashups brought to software integration is integration at the UI layer. Hence, the focus on data mashups only is too narrow, yet the language has already grown very complex and has not been adopted so far by vendors outside the Alliance itself.

However, especially with the growing importance of cloud computing and composition as a service providers (such as mashup platforms or scientific workflows [16]), we expect the importance of customization of composition languages - as a means of diversification - to grow. Also Trummer and Faltings [17] work toward composition as a service; yet, instead of focusing on custom language design, they approach the problem from the provider side and study the optimal selection of service composition algorithms - a task that could be eased if customers were allowed to tailor the composition language to be executed to their very specific needs.

10 Conclusion and future work

Component-based development and composition tools, such as mashup tools, are an increasingly important reality in today's software development landscape. With this paper, we aim to lower the barriers to the development of *good* composition tools by approaching a relevant and central aspect of composition, i.e., the design of *composition languages*. We specifically focus on the problem of developing *custom* mashup languages and show that a sensible design of suitable abstractions and reference specifications enables a *conceptual* development paradigm for mashup languages that is based on the assisted selection of desired composition features and allows developers to neglect low-level details. The paradigm improves *awareness* of design choices and fosters *reuse* of language design knowledge.

In approaching this methodological problem, we also solve a relevant, non-conventional composition problem per se, i.e., the composition of components (the language patterns) that are *not independent* of each other and that require an inte-

gration that is much tighter than that of traditional component technologies, such as web services, already *before* composing them. The key to solve this problem is mapping composition features to a generic language meta-model, an artifact that aim to refine and evolve *collectively* with the help of the scientific community.

The idea is to make the meta-model, the feature reference specifications, and the language generator open source and to share it with the community. In this context, we also want to equip the language design paradigm with an interactive language design tool and a hosted execution engine that is able to run compositions developed with any variation of language developed on top of the common meta-model. The final goal is to provide mashup execution *as a service*. This will eventually lower the barriers to the development of custom mashup platforms.

References

1. Daniel, F., Yu, J., Benatallah, B., Casati, F., Matera, M., Saint-Paul, R.: Understanding UI Integration: A survey of problems, technologies and opportunities. Internet Computing, Volume 11, Number 3, May/June 2007, IEEE, Pages 59-66.
2. Daniel, F., Rodriguez, C., Roy Chowdhury, S., Motahari Nezhad, H.R., Casati, F.: Discovery and Reuse of Composition Knowledge for Assisted Mashup Development. WWW 2012 Companion, pp. 493-494.
3. Daniel, F., Imran, M., Kling, F., Soi, S., Casati, F., Marchese, M.: Developing Domain-Specific Mashup Tools for End Users. WWW 2012 Companion, pp. 491-492.
4. Daniel, F., Casati, F., Benatallah, B., Shan, M.C.: Hosted Universal Composition: Models, Languages and Infrastructure in mashArt. ER 2009, pp. 428-443.
5. Daniel, F. and Soi, S. and Tranquillini, S. and Casati, F. and Heng, C. and Yan, L. Distributed orchestration of user interfaces. Information Systems, Volume 37, Number 6, September 2012, Elsevier, Pages 539556.
6. Baresi, L., Guinea, S.: Mashups with Mashlight. ICSOC 2010, pp. 711-712.
7. OASIS: Web Services Business Process Execution Language, Version 2.0, April 2007. [Online] http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
8. OMG: Business Process Model and Notation, Version 2.0, January 2011. [Online] http://www.omg.org/spec/BPMN/2.0/
9. W3C. Widget Packaging and Configuration. W3C Working Draft, March 2011. [Online] http://www.w3.org/TR/widgets/
10. Mohabbati, B., Gasevic, D., Hatala, M., Asadi, M., Bagheri, E., Boskovic, M.: A Quality Aggregation Model for Service-Oriented Software Product Lines Based on Variability and Composition Patterns. ICSOC 2011, pp. 436-451.
11. Hermosillo, G., Seinturier, L., Duchien, L.: Creating Context-Adaptive Business Processes. ICSOC 2010, pp. 228-242.
12. Hoesch-Klohe, K., Ghose, A.K.: Carbon-Aware Business Process Design in Abnoba. ICSOC 2010, pp. 551-556.
13. Heidenreich, F., Johannes, J., Aβmann, U., Zschaler, S.: A Close Look at Composition Languages. ACoM 2008.
14. Saeed, A. and Pautasso, C.: The mashup component description language. iiWAS 2011, pp. 311-316
15. Open Mashup Alliance: Enterprise Mashup Markup Language (EMML), May 2012. [Online] http://www.openmashup.org/omadocs/v1.0/index.html
16. Blake, M.B., Tan, W., Rosenberg, F.: Composition as a Service. IEEE Internet Computing 14(1), 2010, pp. 78 - 82.
17. Trummer, I., Faltings, B.: Dynamically Selecting Composition Algorithms for Economical Composition as a Service. ICSOC 2011, pp. 513-522.

Appendix O

A

Domain-Specific Mashup Platforms as a Service: A Conceptual Development Approach

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

Despite the common claim by *mashup platforms* that they enable end users to develop software, in practice end users still don't develop own mashups, as the either highly technical or inexistent user bases of existing mashup platforms testify. Based on more than five years of own experience in mashup development and on several user studies, we identify the key shortcoming of current platforms in their *general purpose* nature, which privileges expressive power over intuitiveness. In our prior work, in fact, we have demonstrated that a *domain-specific* mashup approach, which privileges intuitiveness over expressive power, has much more potential to lower the burden on its users and to effectively enable *end user development* (EUD). The problem is that developing a domain-specific mashup platform is *complex* and *time-consuming*, which is no different from generic mashup platforms. Yet, domain-specific mashup platforms, by their very nature, target only a small user basis, i.e., the experts of the target domain, which makes their development not sustainable if it is not adequately supported and automated.

In this article, we describe a *conceptual design approach* for the development of *custom, domain-specific mashup platforms* (comprising custom mashup languages, component descriptors, design time and runtime environments) and show how we *automatically generate* mashup platforms from conceptual designs and provision them *as a service*. We implement a platform *design, generation, and hosting environment* and equip it with a suitable *methodology* for the conceptual development of mashup platforms. The application of the approach in the development of a mashup platform for research evaluation demonstrates the benefits.

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

General Terms: Design, Languages

Additional Key Words and Phrases: Mashups, Domain-Specific Mashups, Mashup Platforms, Conceptual Software Development, Meta-Design, Mashup Platforms as a Service

1. INTRODUCTION

During the last decade both industry and academia proposed a variety of different mashup tools, such as Yahoo! Pipes[1], JackBe Presto[2], ServFace Builder [], Karma [], CRUISe [], MashLight [], and similar. ***Mashup tools*** (or platforms) are integrated development and runtime environments that are typically accessed on-line as a hosted service and allow their users (ranging from target users with low software development skills to experienced programmers) to develop and run own mashups in an graphical, most of the times model-driven fashion. ***Mashups*** in this respect are composite web applications integrating data (e.g., RSS or Atom feeds), application logic (e.g., via SOAP or RESTful web services) and/or user interfaces (e.g., graphical widgets) accessible over the Web. That is, mashups are not completely built from scratch; their development is based on the re-use and integration of existing resources.

In our own research, we worked ourselves on mashup paradigms and tools targeting both end users (e.g., in the MashArt project[3] [Daniel et al. 2009]) and skilled developers (e.g., in the MarcoFlow project[4] [Daniel et al. 2011]). One of the lessons we learned in

[1] `http://pipes.yahoo.com/pipes/`
[2] `http://www.jackbe.com/products`
[3] `https://sites.google.com/site/mashtn/industrial-projects/mashart`
[4] `https://sites.google.com/site/mashtn/industrial-projects/marcoflow`

this context is that mashup tools (including our own) are *too technical* to end users and, as a consequence, they are not able to (i) understand what exactly they can do with the tool and (ii) how to do it. This observation is backed by Namoun et al. [2010b], who performed a set of user studies to understand how end users perceive mashups and conclude that they generally lack an understanding of what, for instance, web services are, and of how these can be integrated to form new logics (e.g., via data or control flows). Yet, they also identified a positive attitude of end uses toward these new kinds of software development tools and the necessary curiosity to start "playing" with them, which we consider a necessary requirement for our work.

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

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

As a proof of concept, we developed an own domain-specific mashup platform for a domain we are well acquainted with ourselves, i.e., research evaluation, but that also involves people without software development skills (e.g., administrative staff and research staff from non-IT departments). More and more are we evaluating ourselves, others, institutions, groups, conferences, and so on, all activities that require *integrating data* (e.g., about conferences, workshops, journals, institutions, people), *computing metrics* (e.g., an h-index or g-index), and *visualizing results* for comparison. The tool that provides these tasks in the form of a dedicated mashup platform is called ResEval Mash [Daniel et al. 2012], and the user study we performed with our target domain experts supports our hypothesis that domain-specific mashup platforms are another step forward toward EUD. Test groups with and without IT skills performed similarly well, and both were able to implement a set of test research evaluations.

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

With this article, we conceptualize all the major aspects that characterize the development of mashup platforms, be them generic or domain-specific, and we describe a meta-design approach to the development of domain-specific mashup platforms that (i) features a conceptual platform development approach and (ii) a complete generative

platform architecture able to automatically produce suitable design time and runtime mashup environments. Specifically, we provide the following ***contributions***:

(1) We thoroughly analyze the current mashup ecosystem, characterize the most used types of mashups (e.g., data vs. user interface mashups), and derive a large set of *composition features* (e.g., the types of components supported or the way data is propagated among components) that mashup platforms may have to support.
(2) We define a *unified mashup meta-model* that brings together all the identified features in one consistent meta-model and express composition features as meta-model patterns and feature constraints.
(3) We develop a *conceptual approach* to the development of *custom mashup languages*, based on the selection and configuration of composition features and the automatic resolution of conflicts.
(4) We describe the implementation of a *mashup runtime environment* that is able to execute mashups expressed in any of the conceptually designed mashup languages in a hosted fashion. We equip the runtime environment with a prototype *mashup editor* that can be tailored to a given domain.

The body of this article is structured as follows: In the next section, we elaborate better on the background and motivations of this work. In Section 3 we state the requirements and design principles that drive the development of the meta-design platform and outline our approach. Then, in Section 4, we specifically focus on mashup language features, the design of the unified meta-model, and the conceptual development of custom mashup languages. In Section 5, we clarify the respective operational semantics by explaining our mashup runtime environment, and we describe the customizable mashup editor. In Section 6, we report on the implementation of the whole meta-design platform and generative architecture. We apply the approach to a use case in Section 7, and conclude the paper with a critical evaluation of the work (Section 8), a discussion of related work (Section 9), and the lessons we learned (Section 10).

2. RATIONALE AND BACKGROUND

Since mashups are a conceptually and technologically complex topic of their own, next we discuss the major types of mashups and mashup tools in use today, describe our domain-specific mashup scenario, and define our problem space and background work.

2.1. Mashups

Understanding which mashup types, i.e., which types of applications, exist and their characteristics is very important for the design of any mashup platform, since they provide the requirements for the development of mashup tools. Aiming at a high diversity of requirements, for the purpose of this work we identify the following types of mashups:

— ***Simple data mashups***. One of the first and most popular mashup types aims at the *integration of data*, typically coming in the form of structured data like RSS or Atom feeds, JSON or other XML data resources. Data mashup tools are typically based on a data flow paradigm, where data are fetched from the resources and processed via different operators by passing data from one operator to the other. Common operators are filter, split, merge, truncate, and so on. The result of data mashups is generally a data resource itself, e.g., an RSS feed that can be accessed from the Web. A well known representative of mashup tool for this category is Yahoo! Pipes.
— ***Data-intensive data mashups***. This type of mashup is a specialization of data mashups, which is characterized by *large volumes of data* to be processed. Data-intensive mashups may work with data amounts that cannot be easily transferred

over the Web, which is instead the preferred solution of generic, hosted data mashup solutions based on data flows. A solution to this problem is, for example, the use of components and a mashup environment that pass data by reference, instead of by value. Scientific workflow tools are examples that adopt this paradigm.

— ***User interface (UI) mashups***. UI mashups integrate *UI components* into a shared layout so that the user can directly interact with them, provide inputs, and get visual outputs. A key characteristic of UI mashups is the event-based synchronization of UI components (e.g., through a publish/subscribe infrastructure), which allows the user to control multiple components by interactive with only one of them. Web portals or widget containers (e.g., based on W3C widgets or OpenSocial gadgets) fall into this category of mashup tools.

— ***Simple service mashups***. This kind of mashups focuses on *web service compositions* where web services are integrated and orchestrated in a process-like fashion. The integration logic is typically expressed as a control flow specifying when which service is to be invoked. Data is usually passed among services via intermediate variables that host the output of one service until the invocation of one or more other services (the so-called blackboard approach). Two key aspects of service mashups are long-running processes and asynchronous communications, the former requiring an execution environment that is independent of the client starting the service mashup, the latter requiring message correlation. One example of service mashup tool is ServFace Builder [], but also simple BPEL compositions fall into this category.

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

A domain-specific mashup may be of any of these basic types of mashups and include some domain-specific features, such as specific components, terminology, composition rules, and similar.

2.2. Research evaluation mashups

A concrete domain for mashups we have been working on is *research evaluation*. We identify with this term all those procedures aimed at providing a quantitative value representing the quality of a researcher, a research artifact, a group of researchers (e.g., a department), a group of research artifacts (e.g., conference proceedings) and similar entities. Producing and having access to these kinds of data is very important in different contexts, ranging from self-assessment to the hiring of new personnel and the distribution of money in departments. For example, in Figure 1 we provide a graphical representation of the evaluation process adopted in the University of Trento (UniTN) for distributing resources and research funds among departments based on their research productivity.

In essence, the process compares the quality of the scientific production of each researcher in a given department of UniTN with the average quality of the research production of researchers belonging to similar departments (i.e., departments in the same disciplinary sector) in all Italian universities. The comparison is based on the weighted publication count.

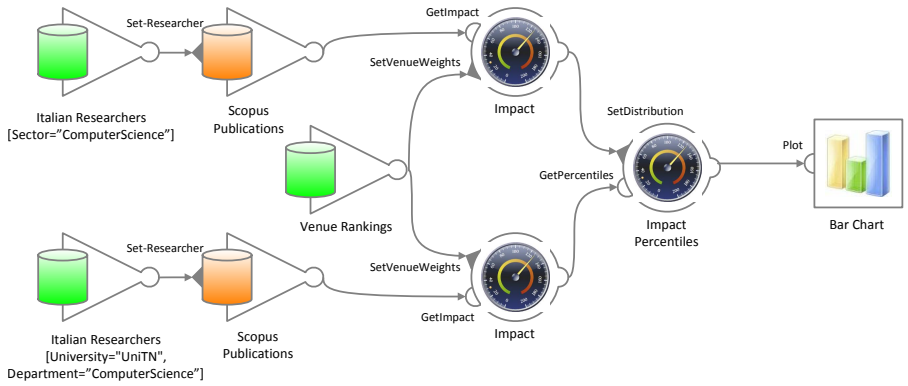


Fig. 1. Model of University of Trento's internal department evaluation procedure.

Today the computation of research evaluation processes is done manually by, for example, the university administrative employees, which use fixed criteria for the selection of, e.g., bibliographic information sources or evaluation metrics. Evaluating research, though, is not so easy and straightforward. Many aspects can heavily impact the evaluation results like, for instance, the publication venues ranking, the source of the bibliometric information, the algorithms used for the metrics, and the like. Fervid discussion on the suitability of the selected criteria often arise, as people would like to understand how the results would differ changing these criteria. This would require, though, the computation of many variations of the process (each one adopting different criteria). Doing so would cost huge human resources and time. It is clear, therefore, that developing all these process variations through manual implementation (as it happens nowadays) or through the standard software development lifecycle are not viable solutions.

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, retrieve and filter resources involved in the evaluation (e.g., publications, people, institutions), and visually analyze the results. Doing so requires to extract, combine, and process large amounts of data through services from multiple sources available on the Web or locally and to graphically visualize computed results through suitable visual components.

This task has all the characteristics of a *mashup*, especially if the mashup logic comes from the *users* themselves. Developing a mashup tool that effectively enables admins and generic faculty members to run evaluation logics as the on in Figure 1, is challenging and requires (i) supporting all technical intricacies of the scenario (e.g., the integration of data, web services and UIs or data transformations), while (ii) hiding them to the user (e.g., Figure 1 does not present any technical aspects besides source, metrics, and visualization components and data flows). End users not only do not know about technicalities, they also don't want to know about technicalities.

2.3. Domain-specific mashup tools

As anticipated in the Introduction, in order to identify the level of abstraction end users are comfortable with we developed *ResEval Mash*[5], a mashup platform for research evaluation. The development of ResEval Mash can be divided into two phases: the formalization and analysis of the above highlighted *requirements* of research evaluation and the actual design and *implementation* of the platform.

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

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

Abstracting from the concrete case of research evaluation, i.e., the *domain* of our mashup tool, we define a ***domain*** as a delimited sphere of concepts and processes. ***Domain concepts*** express the static aspects of a domain, i.e., they tell which elements (e.g., researchers and publications) and relationships (e.g., publishes) thereof characterize the domain and introduce the necessary domain-specific terminology. ***Domain processes*** express the dynamic aspects of the domain, i.e., they tell how domain concepts are operated and manipulated (e.g., the computation of an h-index from a set of publications). Domain processes can be atomic (activities) or composite (processes integrating multiple activities). A domain-specific mashup is an example of composite domain process. More precisely, a ***domain-specific mashup*** (DSM) is a mashup that implements a composite domain process manipulating domain concepts via domain processes. Accordingly, a ***domain-specific mashup tool***[6] (DMT) is a development and execution environment that allows ***domain experts*** (the end users of the target domain) to develop all possible types of domain-specific mashups (and nothing more). In order to do so, a DMT may feature a domain-specific *composition language* and a domain-specific *syntax* that enable the domain expert to development own mashups in an as familiar as possible environment. The research evaluation process shown in Figure 1 is an example of domain-specific mashup expressed in a domain-specific, graphical modeling notation (note the terminology used and the three types of components: data sources, metrics, and charts).

Aiming at assisting mashup platform designers in the development of mashup platforms specifically targeted to a given domain, we abstracted the lessons learned during the development of ResEval Mash and designed a *methodology* [Daniel et al. 2012]

[5]ResEval Mash project page: `http://open.reseval.org/`

[6]Throughout this paper, we use the terms *tool* and *platform* interchangeably, and we specifically focus on mashup tools that produce mashup specifications that can be interpreted and executed by a suitable engine.

focussing on the domain analysis and formalization phase. This methodology guides platform designers through this phase defining a set of artifacts to be produced that describe and formalize different aspects of the target domain. These artifacts will be then used during the DSM platform design and implementation.

Next we summarize the steps and artifacts constituting the methodology:

(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*[7] (MM) that suits the composition needs of the domain and the selected scenarios. As discussed in Section 2.3, 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 hybrid 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. Beside the definition of the actual meta-model constructs and of their relations, also the MM operational semantics (i.e., the basic rules driving the execution of mashups) must be specified.
(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. As said, domain activities and processes represent the dynamic aspect of the 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 or classes of components.
    (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. Specific syntax can be assigned also at the level of component instances. 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 into mashup components, in order to enable domain experts to mash it up into new applications. Components must be described stating their functionalities and interface (i.e., available operations along with their expected inputs and outputs).

Driven by the outcomes of the domain analysis encoded in the artifacts just discussed, platform designers must then design and implement an according DMT. The ***problem*** we address in this article is how to ease the development of DMTs, both technically and methodologically.

3. CONCEPTUAL DESIGN OF DSM PLATFORMS

After the domain analysis phase is concluded — following the described methodology — the DSM platform must be actually implemented. In this paper we propose a *DSM*

[7]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.

platform generation framework that is able to ease this phase providing DSM platform developers with tools assisting and automating to the possible extent the design and implementation steps.

3.1. Requirements

We identify four core requirements for the DSM platform generation framework, which are detailed later in this section:

(1) support a large variety of features and mashup types to cover as many domains domains as possible,
(2) provide a conceptual development approach to design sound DSM platforms more easily,
(3) automate the actual implementation of the DSM platforms,
(4) provide DSM platforms as a service.

Feature support. We focus on the support of the features of both the commonly adopted mashup types identified in Section 2.3 and on other less common features that allow us to expand the coverage of our framework to a broader range of domains, as we will discuss next. Next we identify and describe the mashup features we want to support in our DSM platform generation framework (highlighted in bold font). We can group them in five categories:

— **Component features**. A first important characteristic associated to the mashup components relates to the integration layers that mashups can cover. In other words, a mashup can integrate ***data components***, ***application logic (service) components*** and/or ***UI components***. We want to support a seamless integration of all these three layers (i.e., component types), that is the so-called *universal integration*[Daniel et al. 2009]. When tools do not provide universal integration, produced mashups may require a manual, expert intervention for the development and integration of the application parts related to the missing layer (e.g., implement suitable UIs to be integrated with a service composition).
Mashup platforms are also characterized by the patterns they support to interact with components. We want to support the four standard interaction patterns: ***request-response*** (for *synchronous* interaction), ***solicit-response***, ***one-way*** and ***notification*** (for *asynchronous interaction*). Different types of components can provide different types of interaction patterns, e.g., for UI components only one-way and notification patterns apply, while for service components all the four patterns are possible. The first type does not need specific architectural components, while the others typically require suitable server-side solutions to listen to asynchronous service responses and manage them correctly with respect to the process logic (e.g., managing the correlation among request and responses). Asynchronous service interaction patterns are typically supported in case of *long-running processes*. Differently from *short-living processes*, they require the process logic to be executed on the server, since running the process on the client would stop its execution on browser closing. Both process types must be supported to accommodate different domain needs.
Another important point from the technological perspective is the selection of which components implementation technologies to support in a mashup platform. We want to support components implementation through standard technologies. For this reason we select ***RSS*** and ***Atom*** formats, ***REST*** and ***SOAP*** services and ***W3C Widgets*** and ***JavaScript*** components, which altogether allow the implementation of data, service and UI components. Different technologies have different communication

protocols, formats and semantics and, thus, to support them we must implement specific resources to allow a correct interaction between them and the platform.
Mashup components can also include ***configuration parameters***, used to set up the component when it is loaded, and so-called ***manual inputs***, i.e., input parameters that can be fed by the output of some other component or can be manually set by the mashup designer during the mashup development.
Finally, it could also be necessary to limit the minimum and maximum number of operations per component or the number of input/output parameters per operation. Examples of this kind of features are ***maximum one operation per component*** or ***maximum N output parameters per operation*** (supporting these two features, e.g., would be needed to build a mashup tool like Yahoo Pipes).

— **Control flow and data passing features**. Control flow and data passing paradigms are strictly related to each other, therefore it is sensible to discuss them together. We identify two paradigms defining the flow of control: ***data flow*** and ***control flow***. When adopting a data flow paradigm, once the mashup is started, the mashup execution is driven by the flowing of the data produced by the mashup components. In other words, the control flow is implicitly defined by (and corresponds to) the data flow (i.e., how the data pass from one component's outputs to another component's inputs. In the control flow paradigm, instead, it is explicitly defined, stating which operations must be executed and in which order, where are possible synchronization points and so on. The data passing logic in this case is decoupled from the control flow and can be defined through the adoption of a ***blackboard*** schema, where global variables are used to let data pass among different components.
We also want to support the possibility to have, beside simple *sequential flows*, also *parallel flows*, possibly requiring specific ***split/branch*** and ***join/synchronization*** constructs. Having parallel flows requires a precise definition of the semantics of branching or joining flows. We define that in case of branching flows the semantics is that from a single flow we generate two or more flows proceeding in parallel. This semantics is valid in the case of both data flow and control flow. In the case of two joining flows we may have different semantics, also depending on the paradigm being used. If the paradigm is data flow, control flow constructs like synchronization are not present at all, so the only semantic for two joining flows implements a logic OR, that is, any time there is a message on a flow attached to an input parameter it activates the operation the parameter belongs to, independently of other flows connected to the same parameter. In this case a flow join/synchronization can still be realized, but this is demanded to specific mashup components that will implement it within their internal business logic, which must manage both the flow synchronization and the data merging (requiring some additional predefined or user-defined semantics). In case of control flow, instead, we may want to implement either a logic AND or a logic OR. In the first case there will be a specific synchronization construct taking as input two or more flows and producing a single output flow only when all the input ones have been activated. In the second case, we can directly connect all the input flows to the target operation and anytime one of these flows is activated it triggers the operation, independently of the other flows.
Another functionality we want to support is the possibility to set ***conditions*** to manage the process execution flow. Conditions can work over variable values (in case of blackboard approach) or over the data flowing through connectors (in case of data flow).
We also want to support features to effectively deal with data-intensive processing. In this case, beside the standard *data passing by value* mechanism, we also want to support a *data passing by reference* one (***data passing mode*** feature), which

significantly improves the performance of the system. However, this choice must be carefully evaluated since it affects the whole mashup platform, requiring shared memory structures that suitably developed mashup components must use to read and write their input and output parameters based on the data references.

— **Presentation features**. As said, we want to allow the integration of UI components in the mashups. In this context, we want to support ***single page*** (all UI components are laid out on one page) or ***multi page*** mashups (UI components are distributed over multiple, possibly linked, pages).

— **Collaboration features**. The ***collaboration*** of multiple users at runtime, i.e., during the execution of a given mashup instance, is another interesting peculiarity of some mashup platforms that we want to support. Collaboration can either happen among different users that, based on their role, have access to different pages of the mashup each containing different UI components (***role based access***), or among users acting in parallel over the same instances of UI components (***any user)***.

All the features listed and discussed above are our requirements in term of functionalities to be possibly supported by the DSM platforms produced by our generation framework.

Conceptual development approach. The second core requirement for our system is to provide a conceptual development approach for the design of DSM platforms. We want to let platform designers to abstract from low level design details and focus on higher level design choices to address their specific domain needs. To address this requirement, we need to allow designers to ***work at the level of the conceptual features*** identified above, letting them forget about lower level implementation issues and architectural choices. This means that they must be able to completely design the platform through the selection of a set of features. Then, we need to ***define a clear mapping among features and related software artifacts***, that is, it must be possible to translate the features into according DSM language constructs, architectural components, tools' properties and so on, supporting the selected features. In addition, we have to ***assist designers in the feature selection***, since a wrong selection may lead to inconsistent languages and platforms. There are incompatibility and dependency relations among the features that must be taken into account to assist designers in their selection. For example, if control flow and blackboard features are selected it does not make sense to also select the data flow or, on the other side, the choice of having multiple pages depends on the support for UI components and UI synchronization. This kind of constraints must be identified and checked to assist the platform design phase, so that to be able to limit to the possible extent inconsistencies at the language and tools levels.

Implementation automation. After the design phase, the platform must be actually implemented requiring many efforts. We want to ***relieve developers of the platform implementation*** and automate it to the possible extent. In particular, the system must be able to ***automate the implementation*** of the software artifacts discussed in Section 4 and 5, i.e., the ***DSM language***, the ***runtime environment*** able to run mashup models defined through this language, the ***DSM editor*** facilitating DSM composition design. Clearly, it is not possible to automate the ***DSM components*** implementation (due to their logic variability), but we want to provide some mechanism to allow an easy integration of the components developed through the supported technologies (that we defined above).

In addition, beside the list of features selected by the platform designers, the DSM generation framework must also base the generation process on the artifacts devel-

oped following the methodology steps discussed in Section 2.3. In other words, the framework must provide solutions to ***inject into the DSM platforms the domain knowledge*** encoded into these artifacts.

DSM platforms as a service. Finally, we want to provide the DSM platforms as a service. Concretely, we want to allow platform designers to easily ***select a set of feature*** and validate this set (checking that feature constraints are respected). After this simple steps designers must be provided with a ***complete and ready-to-run DSM platform*** including a DSM language, a DSM runtime environment and a DSM editor supporting the selected features.

3.2. Approach

One of the main requirements for our system, as defined in Section 3.1, is to let developers design their DSM platforms thinking in terms of conceptual features and relieving them from struggling with low-level details and implementation issues. Aiming at this, we want provide developers with a tool allowing them to design the platform simply selecting a set of features they require to be supported in their DSM platform. Then, we want to supply developers with a DSM platform generation framework that, based on a set of selected features, provides according *DSM composition language*, *runtime environment* and *DSM editor*, that is, a complete DSM platform only missing the mashup components, which must be implemented by the developers.

Our approach to generate the *DSM languages* is to *compose* them out of composition features represented as language patterns, i.e., patterns including several language constructs and their relations. Just like in any other composition approach, the core problem is therefore the identification and formalization of the "components" to work with. In our case, these components are *language patterns*. However, these patterns have a distinctive feature that makes our problem very different from generic component-based development (next to the fact that we do not handle software modules but model fragments): unlike, for example, web services, *language patterns are not independent* and may present incompatibility and dependency relations, as already mentioned in the previous section. More precisely, the reference specifications of different composition features may overlap (e.g., interacting with a *SOAP service* is very similar to interacting with a *RESTful service*), include other features (e.g., the *data flow* paradigm generally subsumes the presence of *data source components*), or exclude others (e.g., the *data flow* paradigm does not make use of *variables*). This asks for a thorough design of the language patterns and their mutual interaction points, a task that we achieve by mapping each composition feature into the *unified mashup meta-model* presented in Figure 3 and detailed in Section 4.1. This meta-model (i) integrates all basic language constructs syntactically, (ii) allows us to define composition features as language fragments on top, and (iii) guarantees that fragments are compatible by design. Based on the unified mashup meta-model, therefore, we are able to generate the first artifacts to be produced by our DSM platform generation framework, i.e., the DSM languages.

The just discussed meta-model and approach is key also for the realization of the runtime environment and mashup editor to be provided by our framework. By design, all the DSM languages derived by our meta-model will contain a subset of its constructs. This property allows us to build *single runtime environment* that knows and supports all and only the constructs in the meta-model and the related operational semantics. The runtime environment, by definition, is consequently able to execute mashups defined in any of the languages our system can generate based on the meta-model. The runtime environment only needs to be aware of the set of features the DSM language is based on to correctly interpret the mashup definition and execute

it accordingly. For example, it must know whether to expect control flow or data flow constructs, how data passing is defined and so on.

Similarly, also the realization of the mashup editor depends on the above approach. The editor must provide a graphical representation of the constructs of the DSM language it must support, which, as said, are a subset of the ones contained in the unified mashup meta-model. Also in this case, therefore, we can develop a *single mashup editor* able to adapt its behavior and the construct and functionalities it exposes based on the set of features selected by the platform developers during the design phase. For instance, if developers choose to adopt a blackboard paradigm (i.e., use global variables for the data passing), the editor will provide the domain expert using it with suitable graphical constructs to define global variables and connect them to components' input and output parameters, which would not be shown when the blackboard paradigm is not selected.

Putting together the methodology described in [Daniel et al. 2012] (summarized in Section 2.3), which helps developers to formalize the target domain requirements and characteristics, and the conceptual design approach and DSM generation framework proposed in this article, which guide the design of and semi-automatically builds a custom and ready-to-run mashup platform, we are able to relieve developers of most of the efforts needed to design and implement sound and functional DSM platforms. Clearly, there is still a lot of effort to be provided from developers, which must be spent on these three main areas: (i) domain analysis and formalization (however, under the guidance of our methodology), (ii) components development (supported by our service "componentization" approach - explained later) and (iii) possible extensions or modifications to the DSM editor (in case more specific customization is required).

In general, we do not argue we can build effective platforms for any domain, but we argue that our methodology and system can be applied to many domains allowing a faster and more affordable development of DSM platforms for them.

3.3. Architecture

The functional architecture of the DSM generation framework introduced in the previous section is shown in Figure 2.

The *language and platform design and generator tool* is constituted by two main parts: the design tool and the generator. The first, is a visual interface allowing *platform designers* to design the target DSM platform by selecting the set of conceptual features they require to support the specific needs of their target domain, also following the indications encoded in the *mashup meta-model*. The design tool also checks the validity of the set of selected features, verifying the presence of possible incompatibilities among them. In addition, it takes as input the *domain syntax* artifact that is then used to customize the DSM editor, as explained later. Based on the set of selected features the generator creates a *DSM platform configuration package*, which is stored in a repository and identified by a unique URL (which will be then used to retrieve it). This package includes:

— the *configuration* document, which contains the list of the features selected by the *platform designer*, the reference to the *domain syntax* artifact (provided by the *platform designer*), and the references to the components and compositions repositories;
— the schema definitions of the *DSM composition language* and *DSM component descriptor language*, which *DSM composition definitions* and *DSM component descriptors*, respectively, must comply to. The languages defined by these documents are generated using the *unified mashup meta-model* as base and include all the constructs needed to support the set of selected features.

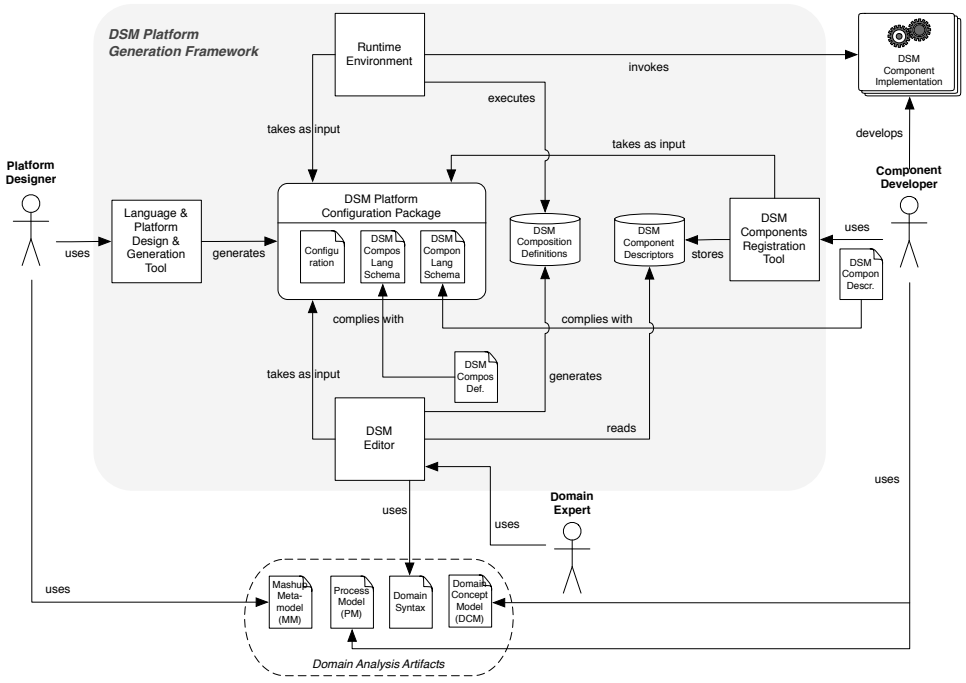


Fig. 2. Functional system architecture.

In addition, the framework provides a *runtime environment* and a *DSM editor* that are able to adapt their functionalities and behavior based on the information included in a configuration package they take as input.

The *DSM editor* is used by the *domain experts* to graphically design the mashups. To show the available DSM components to domain experts, the editor reads the *DSM component descriptors* from the component descriptors repository. The editor also uses the *domain syntax* artifact (referenced in the same document), which is a document listing the language constructs having an own domain-specific graphical syntax and providing a reference to the associated image files to be injected in the editor replacing the editors' generic graphical syntax. Finally, the editor generates representations of designed mashups compliant with the DSM composition language defined within the configuration package taken as input, and stores them in the compositions repository.

The *runtime environment* executes the *DSM composition definitions* stored in the composition repository. Composition definitions include a reference to the *DSM platform configuration package* associated to the definition, so that the engine can retrieve it and set up its behavior according to the information included in the configuration package. During mashups execution the *runtime environment* interacts with the *DSM components* invoking their operations.

DSM components must be previously implemented by a *component developer*, which implements them based on the *process model* and the *domain concept model* defined during the domain analysis. The latter formally defines the schema the possible domain-specific data types that DSM components can consume, produce and exchange. Having such a shared formal data model is very important to make a DSM platform where components can effectively work together and can be composed more easily (e.g.,

this allows for automating data mappings between components out and input parameters based on their types). As already mentioned, components are accompanied by a descriptor defining the component's main properties and interface. This descriptor must comply with the *DSM component language schema*. Developers can integrate the DSM components in a given DSM platform using the *DSM Components Registration Tool*, which must be provided with the descriptors of the components to be integrated.

Indeed, our DSM platform generation framework generates "logical platforms". In fact, it only generates DSM platform configuration packages and not the runtime engine and mashup editor program code. Passing a configuration package as input to the runtime engine and to DSM editor (which are physically deployed on our server) instructs them to work with the languages, features, components and compositions associated to the specific DSM logical platform described by the given configuration package. This approach allows us to provide hosted DSM platforms as a service, which are already deployed and ready to work.

Alternatively, the generation framework could generate a package containing the whole "physical platform" to be deployed on a different machine (including the generated languages and also a copy of the editor and of the runtime engine). This solution could be useful in case a developer wants to modify the tools' code to extend them. Although it would not be technically difficult to realize, currently we do not support this solution. In general, we consider more convenient to provide the platforms in a hosted fashion, relieving developers of software installation and deployment problems. We instead plan to address the above mentioned needs making the whole project open source.

The conceptual aspects of the main modules of this architecture are discussed in Section 4, 5.1 and 5.2, while implementation details are presented in Section 6.

4. CONCEPTUAL DESIGN OF CUSTOM MASHUP LANGUAGES

Starting from the requirements and features identified in Section 3.1, in the following sections, we describe how the unified mashup meta-model we designed is able to support these features and how it is used to produce DSM composition and component descriptor languages. The set of features we identify comes without the claim of completeness and is meant to grow over time; however, we are already able to express a fairly complex set of mashup languages.

4.1. The meta-model

The meta-model must support all the features identified in Section 3.1. Next we overview how these features are mapped onto the constructs of the unified mashup meta-model (shown in Figure 3) and how these constructs relate to each other. The gray boxes in Figure 3 group the meta-model constructs based on the feature category they relate to.

4.2. Mashup language features

Component features. They specify which kinds of components — in terms of technologies and communication patterns — the target domain-specific language should support. For instance, a Web service may come with message-based operations of four different types (request-response, solicit-response, one-way, notification), custom data formats for each input and output message, a service endpoint, and a protocol binding (e.g., SOAP). We represent such a service in the meta-model as a *component* that has a set of *operations* with different input/output parameter patterns (implementing the four different operation types), only single *input*/*output parameters* per operation to represent input/output messages, an own *data type* for each parameter, and respective *binding* and *endpoint* attribute values. Similarly, a W3C UI widget [W3C 2011] can

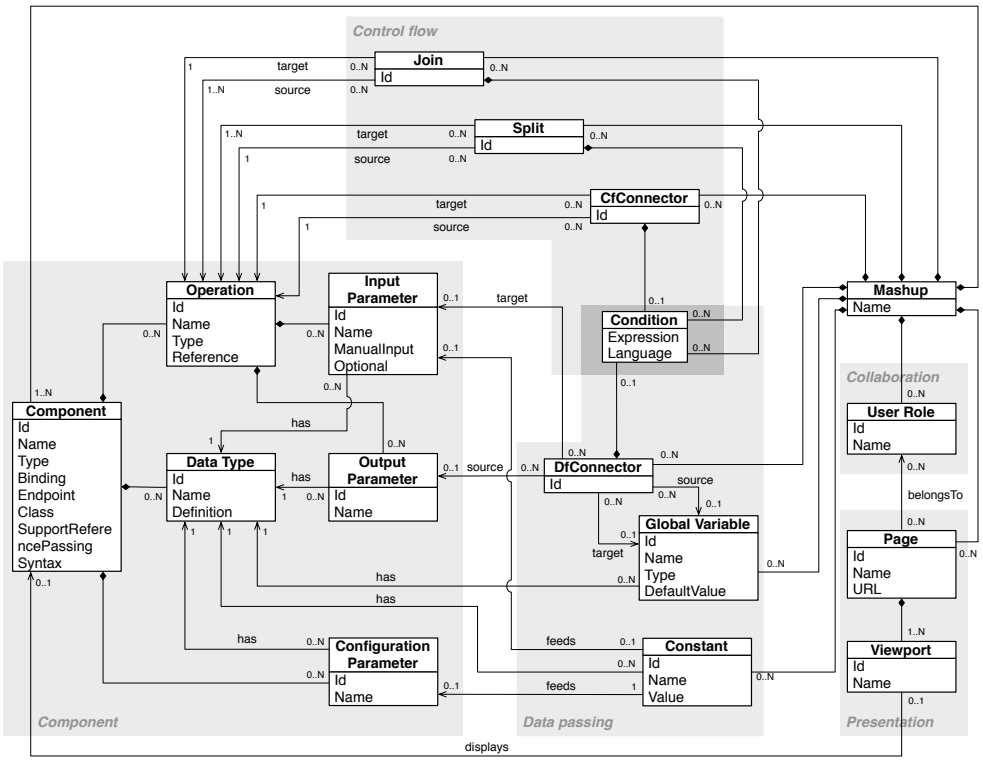


Fig. 3. The unified mashup meta-model. Gray boxes group entities into feature types. The Component group is also used to derive component descriptor languages

be seen as a *component* with some *configuration parameters* but without operations, which can be displayed in a *viewport* of a *page* of the mashup.

Analogously, the meta-model conciliates the three component types already introduced (data, service and UI), which can be implemented through the following *technologies*, which are the basis of many types of mashups and, as such, widely used and accepted (component types are tracked by the *type* attribute of the *component* entity while implementation technologies by the *binding* one. The valid co-occurrences of the type and binding values are defined as OCL constraints expressing the following rules:

— Data source components: RSS feeds, Atom feeds, RESTful data components, SOAP data components, JavaScript data components.
— Web service components: Atom services, RESTful services, SOAP services, JavaScript components.
— UI components: W3C UI widgets [W3C 2011], JavaScript UI components [Daniel et al. 2009] (i.e., our own technology to implement intercommunication-enabled UI widgets, since there are no standard widget technologies supporting inter-widget communication).

For each of these component technologies, it is then important to specify which exact *communication patterns* the language should support. For instance, the language could support only synchronous communications (operations with input *and* output pa-

rameters), only asynchronous communications (operations with *either* input *or* output parameters), or both. The communication paradigm that each operation implements is encoded in the *type* property of the *operation* entity. Components can be grouped in logical categories based on the content of the process model (PM) described in Section 2.3. The category of component is defined by the *class* property of the *component* entity. Similarly, the *syntax* property associates a domain-specific graphical syntax to components.

The many components' properties discussed above are represented by a number of features in our framework. The **data component**, **service comonent** and **UI component** features state whether related components are supported or not. For each of them, a set of features defining the characteristics of these component types are present, i.e., to define the allowed component's implementation technologies (e.g., **RSS for data components** or **SOAP for service components**), to define the allowed operation types (e.g., notification for **UI components** or **request-response for service components**) and to define whether components may have configuration parameters (**configuration parameters**).

It might also be necessary to limit the number of operations per component (e.g., in Yahoo! Pipes each component corresponds to one operation) or the number of parameters per operation (like for SOAP services as described above). All these options can be represented via patterns that suitably set the relationship cardinalities in the meta-model. Also in this case suitable features represent these options, e.g., **maximum 1 operation per component** or **no maximum number of output parameters per operation**.

Control flow features. They specify whether the language is control-flow-based (e.g., BPMN) or data-flow-based (**control flow** and **data flow** features) and which control flow constructs to support. In the former case, sequential execution can be expressed by connecting operations using control flow connectors (*CfConnetors*). Parallel executions are supported via *split* and *join* constructs (represented by the **split** and **join** features). In the latter case, instead, the flow of control corresponds to the flow of data that, as described below, is defined through data flow connectors (*DfConnector*). In this case, no split or join constructs can be used.

Each of the mentioned constructs can have one or more *conditions*, which constrain the control flow along connectors (**conditions** feature) and, for instance, allow the implementation of conditional control flow constructs like conditional connectors, conditional splits, and conditional joins. Loops can be implemented by means of conditions and joins.

Data passing features. They specify how data is propagated among components. In data-flow-based languages (e.g., Yahoo! Pipes) the data passing logic is defined mapping output parameters to input parameters, a feature that can be achieved by specifying data flow connectors (*DfConnector*) between parameters instead of between operations.

Control-flow-based languages require specific constructs to specify how data are passed among components. The most common technique is to write/read *global variables* (**blackboard** feature), which are accessible during the execution of a composition (e.g., as in BPEL). The meta-model represents the writing/reading operations with a data flow connector between the variable and its target/source parameter.

UI-based mashups, such as widget portals, typically run all widgets in parallel, and data is passed via global variables or events (operation with only outputs). Configuration parameters are instead typically set once at the startup of a component (e.g., the background color of a UI widget); we support this by means of *constants*.

An additional feature (**reference passing mode**) related to data passing allows enabling the data passing by reference in the runtime engine that, as discussed in Section 3.1, can be used for domains characterized by data-intensive processes. This feature requires component definitions stating whether a given mashup component supports the data passing by reference. This information is encoded in the *supportReferencePassing* attribute of the *component* construct.

Presentation features. They specify whether UI components and related constructs (e.g., *page* and *viewport*) are supported or not (**user interface** feature). Unlike service compositions, mashups typically also come with an own user interface that renders UI components and data from UI-less components. The minimum support required to express this capability in the meta-model is represented by the *page* and *viewport* entities, which allow the ordering of UI components into pages (HTML web pages) and their rendering in selected areas inside these pages (typically `div` or `iframe` HTML elements). The HTML pages hosting the UI components can be given and already linked to each other as necessary or can also be generated automatically. Mashups can include maximum one page (**single page** feature) or multiple pages (**multi page** feature).

Collaboration features. They specify whether collaboration among different users of a mashup is supported (**collaboration** feature) and how users collaborate. Single-user mashups do not require any user management. Multi-user mashups, instead, may restrict the visibility of individual *pages* to selected *user roles* only. Users may have different views on a mashup, e.g., via different pages, (**role-based access** feature) or they may have the same view, e.g., via the concurrent use of a same page (**any user** feature). For the time being, we only support the former type of collaboration.

The above features and examples show that developing a good unified mashup meta-model is a *trade-off* between the simplicity and usability of the final language (the fewer individual constructs the better) and the ease of mapping features onto the meta-model (the more constructs the better; in the extreme case, each feature could have its own construct). The challenge we faced is exactly that of identifying the right balance between the two, so as to be able to map all relevant features and to do so in an as elegant as possible fashion from the resulting language point of view.

4.3. Feature-driven design of mashup languages

The model discussed in the previous section cannot be directly used to create executable mashup definitions. This is due to the fact that the model includes construct supporting features potentially incompatible. For example, the model includes the constructs to support both the data flow and the blackboard paradigms. These two features, though, cannot coexist in the same executable model since there could be conflicts in the data passing logic. For this reason we need to extract consistent, concrete models from the unified meta-model.

Therefore, we need to process the unified mashup meta-model to extract consistent, concrete mashup models including all and only the model constructs needed to support a valid set of required features specified by the DSM platform designer. A set of features to be valid must respect incompatibility constraints among features (e.g., data flow and blackboard features, as described above, are incompatible) and dependency constraints among features (e.g., collaboration features, as described in the previous section, depend on UI-related features).

To process the unified mashup meta-model, we first translate it into a *unified mashup language*, i.e., a representation of the meta-model into a more sim-

ply processable format. Then, we formally represent each composition feature as $f = \langle name, label, description, specification, constraints \rangle$, where

— *name* is a text label that uniquely identifies the feature (e.g., `data_flow`);
— *label* briefly describes the feature and expresses its semantics;
— *description* is a natural language verbose description of the feature for human consumption;
— *specification* is the reference specification of the feature and lists a set of associated unified mashup language fragments;
— *constraints* is a set of feature compatibility and dependency constraints expressed as logical formulas.

All features definitions are stored in the *feature base*, i.e., a document containing features definitions following the just described format.

At this point, assuming a set of selected features is provided, we can generate the DSM composition language supporting these features as follows.

First, we check the set validity using an algorithm that associates a boolean variable to each feature present in the feature base and then sets to `true` all the variables associated to the selected features and to `false` all the remaining. Based on these values it computes the validity response as the result of the logical formula constituted by the logical conjunction of all the constraints associated to the selected features (described in the features definitions).

Once the set validity is assessed, we generate the target DSM composition language including all the unified mashup language constructs needed to support the selected features. Operatively, the DSM composition language is generated including all the unified mashup language fragments referenced in the specification field of the selected features. This algorithm guarantees that the generated languages, by design, support all the selected features (since they include all the construct needed to support them) and that the languages are consistent (thanks to the feature set validity check performed before the generation).

The DSM component descriptor language is generated following the very same algorithm. The only difference is that instead of using the full unified mashup language it is used a subset of this language only including the constructs needed to describe the mashup components (i.e., the ones within "component gray-box" in Figure 3).

5. CUSTOM RUNTIME AND DESIGN TIME MASHUP ENVIRONMENTS

5.1. Operational semantics: the runtime environment

The runtime environment (or engine) is in charge of executing the mashup compositions. Compositions can be expressed in any language that the framework can produce. The frameworks provides a single implementation of the engine that is able to adapt to the specific language and features associated to the composition being executed which are specified within a configuration package taken as input (referenced in the composition definition itself). The overall picture of the runtime environment architecture is shown in Figure 4.

The engine is split into two parts: the *server-side engine* (SS-Engine) and the *client-side engine* (CS-Engine). The core of the engine's business logic is located at the server side. A *DSM composition definition* is executed issuing a request to the *server*, containing a reference to the composition definition itself (which is stored in the internal *DSM composition definition* repository).

When an execution request is issued, the *server* instantiates an *SS-Engine* to manage the execution of the given composition.

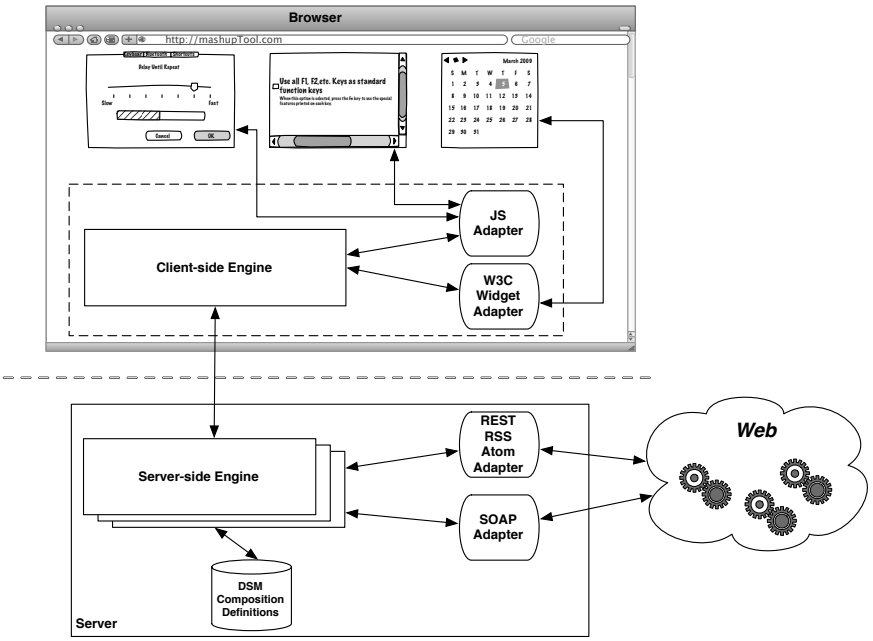


Fig. 4. Runtime environment overall architecture

The main responsibility of the *SS-Engine* is the management the flow of control and the data passing. This means that the SS-Engine is in charge of invoking components' operations following the order defined by the given DSM composition definition, taking into account, e.g., whether the definition is based on the control or data flow paradigm, whether there are branching or merging flows in the composition, whether components expect actual data or data references (in case of data passing by value or by reference, respectively) and where to retrieve these data from (operations' output parameters or global variables), and so on. In other words, the engine must be able to correctly interpret and manage all the constructs defined in the unified mashup meta-model.

On the client side we have a lighter *client-side engine*, which mainly instantiates the client-side mashup components (e.g., JavaScript components) and allows the SS-Engine to interact with them, through the adapters.

The engine delegates the communications with the mashup components to a set of external *adapter* modules that act as mediation gateways among the engine and the external services, which may be arbitrarily distributed on the Web. Each module is responsible for managing the interaction with components implemented through a specific technology, requiring specific communication protocols. We provide adapters for the main standard technologies identified in Section 4.1, that is REST, RSS, Atom, SOAP, W3C Widgets[8] and JavaScript components. Having the mashup components interaction managed by independent adapter modules allows one to simply extend the engine to support new component implementation technologies by adding a new suitable adapter.

[8]The integration of W3C Widgets, although conceptually supported by the meta-models, is not implemented yet.

The implementation of the architectural modules described above must encode all the *operational semantics* of the possible languages we can derive from the unified mashup meta-model, so as the engine to be able to execute mashup compositions expressed in any language generated by our framework. The operational semantics driving the execution of a mashup heavily depend on the features supported by the DSM composition language a given composition definition is expressed with. For instance, the correct management of parallel flows during the execution of a mashup depends first on the adopted paradigm, i.e., control or data flow. Then, in case of control flow paradigm, it also depends on whether specific control flow-related features are supported or not (i.e., flow synchronization and branching, which are available only when adopting the control flow paradigm).

For this reason it is difficult to provide a complete report of all the possible semantics deriving from different combinations of features. Following we give the general semantics ruling over compositions execution:

(1) The execution of the mashup compositions is initiated by the user. In case of UI compositions the UI components themselves can be used as starting trigger, otherwise, in case of service compositions or hybrid compositions including both UI and non-UI components, a suitable interface will be available for starting the compositions and providing possible initial inputs.
(2) (a) In case of control flow-based compositions, the first operation of each flow (i.e., a sequence of operations connected either through control flow or data flow connectors) of the mashup is triggered.
(b) In case of data flow-based compositions, all the request-response or one-way operations which are ready to be triggered (i.e., which are provided with all the needed input parameters) are invoked in parallel.
(3) Once an operation is triggered, it is executed, possibly processing some input. Possible outputs produced are used to feed the next operations in the flow that are connected to the current one.
(4) Step 3 is repeated until all the operations in all the flows of the composition are triggered.
(5) The execution is stopped either right after step 4 (automatically), in case no asynchronous notification operations are present in any flow, or with an explicit command by the user, in the opposite case.

The important point to be remarked is that the runtime environment is able to understand the syntax and implement the operational semantics of any mashup language our system can produce deriving it from the unified mashup meta-model based on the selection of a valid set of feature, as described above.

5.2. The mashup editor

The platform we proposed in this article also include a mashup development environment (in short, editor) to be used by domain experts to graphically design their mashup compositions through simple, visual interaction paradigms[9]. The editor supports any DSM language generated by our framework. This means, first, that for each construct of the unified mashup language it provides an associated graphical construct visually representing the language construct in the editor. Second, it means that the editor is able to translate the mashups graphically designed by the domain experts into representations compliant with the DSM composition languages generated by our framework.

[9]The mashup editor is currently under development. Some features are already supported while others are not. We plan to complete the editor's development in the next months

```
<domainSyntax>

  <construct id="component" syntax="http://.../genericComponent.png" />
  <construct id="globalVariable" syntax="http://.../globalVar.png" />
  <construct id="split" syntax="http://.../split.png" />
  <construct id="join" syntax="http://.../join.png" />

</domainSyntax>
```

Fig. 5. An example of domain syntax descriptor

To be used, the editor must be provided with a DSM platform configuration package (or, more precisely, with a URL referencing it) that the editor uses to adapt its functionalities and user interface based on the set of features to be supported listed in the configuration document included in the package. For example, only if the blackboard paradigm feature is selected the editor exposes the functionalities to create new global variables and use them to define the data passing logic of the mashups.

In addition, the configuration package also includes the reference to the domain syntax definition document, i.e., a document listing the language constructs (e.g., the global variable or join constructs) having an own domain-specific graphical syntax and providing a reference to the associated image files to be injected in the editor replacing the editors generic graphical syntax (Figure 5 shows a domain syntax descriptor example). Each mashup component can have an own domain-specific, graphical syntax, replacing the general component syntax specified in the domain syntax descriptor or editor's generic one; in this case the reference to the image file to be used to graphically represent the component is defined in the component descriptor.

Figure 6 shows the editor when configured to support different features. Figure 6(a) presents the editor when configured to support, e.g., the data flow paradigm (wires are defined among parameters), and mashup components possibly having multiple operations and configuration parameters (draggable from the left toolbar of the editor to the design canvas). The editor shown in Figure 6(b), instead, supports the control flow paradigm (wires are defined among operations) and related constructs, e.g., global variables, join and split, which are shown in the left toolbar of the editor. In addition, in this case components can have only one operation and cannot include configuration parameters. In both cases, all the components are represented by an own, domain-specific graphical syntax.

What we provide is a basic editor able to support all the DSM languages and features provided by our DSM platform generation framework and able to adapt its functionalities and interface based on a configuration package given as input. The tool can be used off-the-shelf to allow domain experts to design their compositions. However, if platform developers have specific requirements in terms of user interface or advanced tool's functionalities, they can develop their own mashup editor building upon the one we provide, which represent a solid base for the development and already provides most of the needed functionalities.

6. IMPLEMENTATION

While in Section 4, 5.1 and 5.2 we described the main system modules from a conceptual and functional point of view, in this section we provide some details about their implementation aspects. In particular, first we shortly describe the tool allowing the developers to design their target platform and generating it for them, then we provide the details related to the main modules constituting the generated platforms, i.e., languages, mashup editor, runtime engine and mashup components.

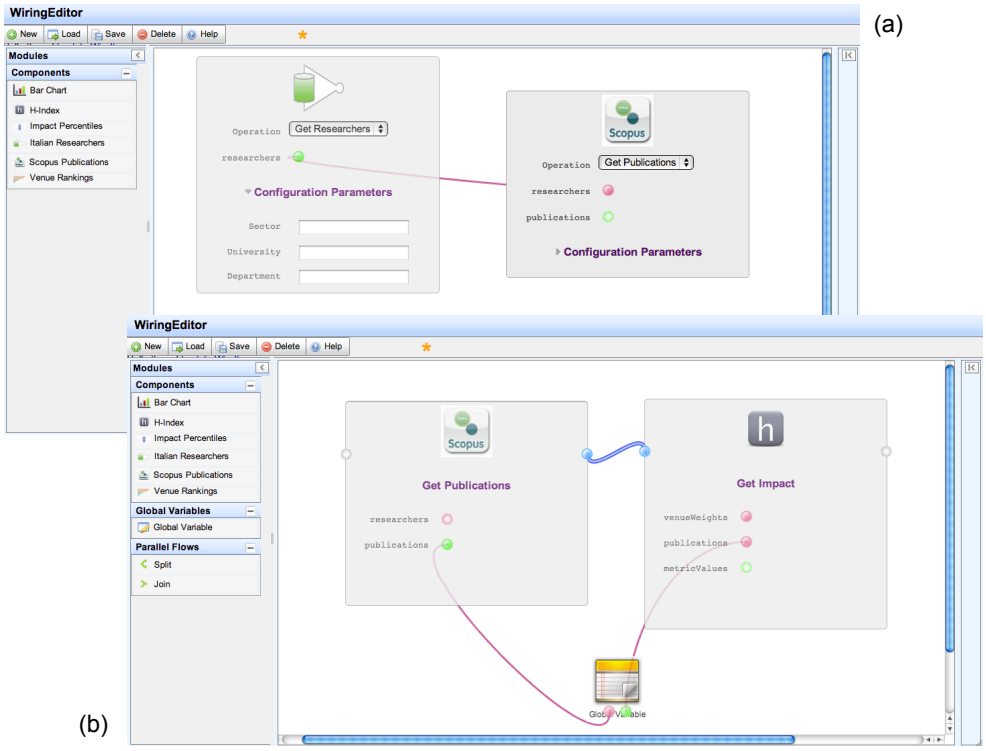


Fig. 6. The customizable mashup development environment. Different screenshots represent how the environment changes depending on the features it is configured to support

6.1. The platform design and generation tool

One of the most powerful aspects characterizing the work proposed in this article is that platform designers can simply reason staying at the feature level and forget low-level implementation issues. The platform design tool is basically constituted of a list of checkboxes, along with related labels and descriptions, each one representing a conceptual feature and allowing designers to select a set of features to be supported in their DSM platform. The only additional field that can be filled in by the designer is used to upload a domain syntax descriptor file, that is an XML descriptor which associates a given domain-specific graphical syntax (i.e., an image) to the mashup language constructs to be used inside the mashup editor. Figure 5 presents an example of domain syntax descriptor while Figure 7 shows the prototype interface of the tool.

The designers, clearly, must first pass through the domain analysis and formalization steps, as described in the methodology in Section 2.3, but after this effort they only have to select the required features according to the analysis outcomes.

The tool interface allows the developers to read all the available features and shows them a short description for each feature on the mouse-over. Once the designer has selected the required features he/she must check the feature set validity clicking on the *check constraints* button, which invokes the already introduced associated algorithm (detailed in the next section).

If the validity check is successful, the last action the designer has to do is clicking on the *generate* button, which triggers the platform generation algorithm (implemented

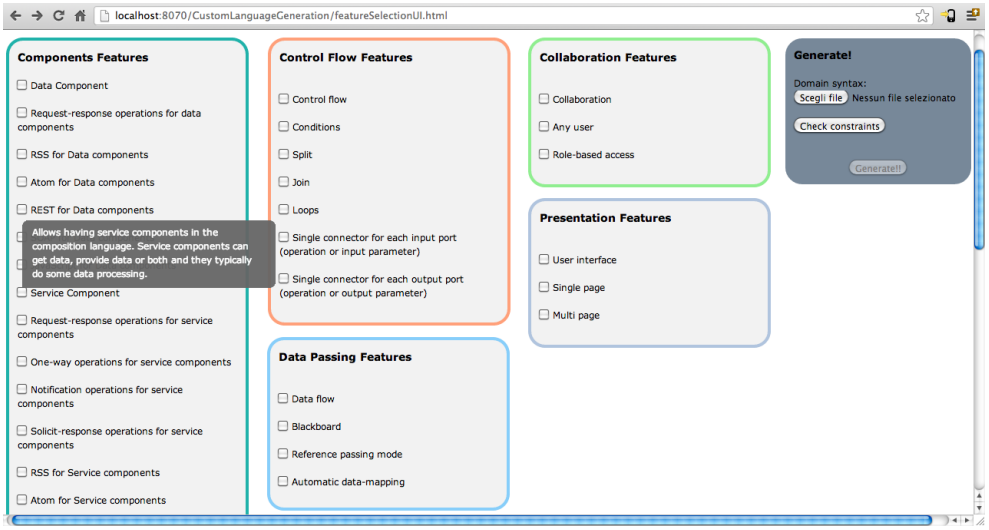


Fig. 7. Features selection user interface used by the platform designer

as a Java servlet), providing it with the list of selected features. This algorithm first invokes the language generation algorithm (detailed in next section), which generates the DSM composition and component descriptor languages, then it creates and stores a suitable DSM platform configuration package (described in Section 3.3) on our server and returns the package URL identifier to the designer.

The mashup editor and the runtime environment will then retrieve the configuration package based on its URL and will use it to adapt their behavior and functionalities based on the information contained in it.

For the complete features definitions, for the unified mashup language XSD and for more examples of DSM languages we refer the reader to an online resource (`http://goo.gl/hfkLO`).

6.2. The DSM languages creation

For creating the DSM languages through the generation process described in Section 4.3, we first need to select a machine-processable format allowing us to represent and process the mashup models and languages described in that section. We selected for this purpose the XML family of specifications. Therefore, we use XML schema definitions (XSDs) to formally define the unified mashup language and the DSM languages we generate.

Also the features are defined as XML documents, following the same structure presented in Section 4.3. Figure 8 shows an example of feature definition.

As shown in Figure 8 the features' specification element contains a list of identifiers. These identifiers refer to well-defined fragments of the unified mashup language XSD. The generation algorithm includes in the final DSM language XSD all the unified mashup language XSD fragments specified in the definitions of the set of selected features and excludes all those associated to non-selected features, which would be incompatible or unnecessary. A feature specification element may also include a *set-cardinality* definition; these are used to set the XSD elements' occurrence bounds. Before the language generation to be performed, the set of selected features is validated

```
<feature name="condition" label="Conditions">

   <description> Conditions can be set for each connector to define the
        possible flows of the composition. Conditions are supported both for
        control flow and data flow composition paradigms.
   </description>

   <specification>

      <include fragments="conditionForCf" if="control_flow"/>
      <include fragments="conditionForDf" if="data_flow"/>
      <include fragments="conditionForSplit" if="split"/>
      <include fragments="conditionForJoin" if="join"/>

   </specification>

   <constraints>
      (control_flow AND blackboard) OR data_flow
   </constraints>

</feature>
```

Fig. 8. An example of composition feature definition representing the *condition* feature

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

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

  <specification>

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

  </specification>

  <constraints></constraints>
</feature>
```

Fig. 9. The data flow composition feature definition

by the constraint verification algorithm, which checks that the constraints specified in the selected features' definitions are respected.

For example, let us assume that in the set of selected features are present the data flow and condition features. The definitions of these two features are presented in Figure 9 and 8, respectively.

These two features are compatible since the data flow one does not specify any constraint while the condition feature's constraints are satisfied by the presence of the data flow feature.

Therefore, the language generation algorithm can be executed. The algorithm will include in the output DSM language the fragments of the unified mashup language XSD whose *fragmentID* is listed in the specification element of the two features definitions. The unified mashup language XSD fragments associated to the data flow and condition features are shown in Figure 10(a). An example of XML mashup definition excerpt compliant with the defined part of DSM language XSD is shown in Figure 10(b).

The constraint verification algorithm is implemented as a client-side JavaScript script, since it is integrated in the platform design tool described in Section 6.1, while the language generation one is implemented in Java within the platform generation servlet. Both the algorithms take as input the list of identifiers of the selected features and use the feature base (containing the definition of all the features) and the unified mashup language XSD. The constraint verification and the generation algorithms are

```
<xs:complexType name="dfConnectorType" fragmentId="dfConnectorType">
  <xs:sequence>

    <xs:element name="condition" minOccurs="0" maxOccurs="1" fragmentId="
        conditionForDf">
      <xs:complexType mixed="true">
        <xs:attribute name="language" type="xs:string" use="optional" />
      </xs:complexType>
    </xs:element>

    <xs:element name="source_outputParameter" minOccurs="0" maxOccurs="1"
        fragmentId="dfSourceOutputParameter">
      <xs:complexType>
        <xs:attribute name="ref" type="xs:string" use="required" />
      </xs:complexType>
    </xs:element>

    <xs:element name="target_inputParameter" minOccurs="0" maxOccurs="1"
        fragmentId="dfTargetInputParameter">
      <xs:complexType>
        <xs:attribute name="ref" type="xs:string" use="required" />
      </xs:complexType>
    </xs:element>

    <xs:element name="source_globalVariable" minOccurs="0" maxOccurs="1"
        fragmentId="dfSourceGlobalVariable">
      <xs:complexType>
        <xs:attribute name="ref" type="xs:string" use="required" />
      </xs:complexType>
    </xs:element>

    <xs:element name="target_globalVariable" minOccurs="0" maxOccurs="1"
        fragmentId="dfTargetGlobalVariable">
      <xs:complexType>
        <xs:attribute name="ref" type="xs:string" use="required" />
      </xs:complexType>
    </xs:element>

  </xs:sequence>

  <xs:attribute name="id" type="xs:string" use="required" />

</xs:complexType>
```

(a)

```
<dfConnector id="DF5">
  <source_outputParameter ref="O1-4" />
  <target_inputParameter ref="I2-10" />
</dfConnector>

<dfConnector id="DF6">
  <source_outputParameter ref="O2-4" />
  <target_inputParameter ref="I1-9" />

  <condition language="javascript">
    text!= 35
  </condition>
</dfConnector>
```

(b)

Fig. 10. The unified mashup language XSD fragments associated to the data flow and condition features (a) and a compliant XML mashup definition excerpt (b)

triggered by the platform designers by pressing the “Check constraints” and “Generate” buttons, respectively.

6.3. The runtime environment

As introduced in Section 5.1, the runtime environment is distributed among the client and the server.

Due to the event-driven nature of mashups (which often include UI components or asynchronous services) and to the high performance associated to this technical choice, we implemented the server as a Node.js[10] server and the SS-engine as a set of Node.js modules, therefore, implemented in JavaScript. Thanks to the architecture modularity, the adapter modules on the server-side (i.e., those in charge of interacting with RSS, Atom, REST and SOAP services) can be implemented using any technology, also different from NodeJs. Indeed, we decided to implement them as RESTful services implemented in Java and running on an, Apache Tomcat server ((installed on the same machine of the server hosting the SS-Engine, to avoid any network latency or issue).

Both the CS-Engine and the client-side adapters (i.e., for JavaScript components and W3C Widgets) are completely implemented in JavaScript. In addition, the CS-Engine also acts as interface through which the mashup end user can start the mashup execution. Concretely, the end user will always be provided with an HTML page to interact with the mashup process. The page, beside showing the possible UI components included in the compositions (which may react to users action or show process results),

[10]Node.js is a framework for writing scalable JavaScript web applications and servers on the server-side. More information at: `http://nodejs.org/`

provides end users with a starting interface, that is, a UI allowing the user to possibly feed non-UI components with some manual input (if any) and to launch the composition execution, that will continue following the operational semantics introduced in Section 5.1.

To run a mashup composition, first, the CS-Engine connects to the server through an HTTP GET message, passing some parameters and, in particular, the DSM composition definition to be executed, which also includes a reference to the configuration package needed by the engine to retrieve the list of features to be supported. When the server receives this message it creates a new SS-Engine instance that will manage this mashup execution and associates to it a WebSocket [W3C 2013] server, whose address is sent back to the CS-Engine. All the following communications among the CS-Engine and the SS-Engine will be direct messages over the WebSocket protocol. This allows the engine to easily and very efficiently support long-running processes, asynchronous interactions and real-time applications.

6.4. The mashup editor

For the implementation of the mashup editor we build upon the WireIt library[11]. It is a JavaScript library providing a basic Web tool and a visual language for the development of mashup-like compositions. It comes with a readily available basic editor already including generic compositional elements like components, input and output ports, component configuration forms, wires and the like.

We extended this editor to support all the language constructs of the unified mashup language. For example, we extended it adding a new construct to represent global variables and adding different types of wires, e.g., one to be used to define the compositions' control flow and another for designing the data passing logic.

When the editor is loaded it retrieves a configuration package using the URL passed as HTTP parameter. From this configuration package extracts the list of features to be supported and adapts its interface and functionalities accordingly. In addition, it also extracts the URL of the component descriptors repository that is used to retrieve the descriptors of all the mashup components available for the DSM platform described by the given configuration package and represent them in the component toolbar within the editor. Similarly, the composition definitions repository is accessed to load previously stored compositions and to save the new ones created in the editor.

6.5. The mashup components

The mashup components implementation cannot be significantly automatized and must still be mainly developed by programmers. Components should comply with (i) the guidelines encoded in the analysis outcome artifacts and (ii) the DSM component description language produced by the platform generator. In particular, components should belong to one of the classes defined in the PM and should consume and produce parameters complying to the data types defined in the DCM (i.e., to their XSD definition). In addition, to improve the domain specificity and usability of the mashup editor, a component can be accompanied by an own specific graphical syntax, which overrides the basic editor's generic syntax.

An important aspect to be noticed is that we want to allow a simple integration of already existing services and components. For this reason, as already introduced, we support by design the usage of widely spread standard technologies. The integration of components developed through these technologies is typically very straightforward since only requires the description of the component following the component description language format. This descriptor includes all the needed information to connect

[11] WireIt library homepage: `http://neyric.github.com/wireit/docs/`

to and interact with the actual service/component, which will be then used by the mediator adapters described in Section 6.3. For example, to "componentize" an existing SOAP service, it is sufficient to create an XML descriptor similar to the one shown in Figure 11, which describes the H-index component needed for the realization of the scenario presented in Figure 1.

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

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

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

</component>
```

Fig. 11. An example of composition feature definition

This example descriptor specifies the main properties and interface of a SOAP service for the computation of an H-index value based on a list of publications and a set of venue weights passed as input. The definition of a similar descriptor is all we need for making the runtime engine able to use and interact with the service.

Once the components are implemented and according descriptors are defined, they must be registered to be used within a given DSM platform. Component registration is done through a simple web interface. This interface requires to upload the component descriptor and the DSM platform configuration package of the DSM platform the component must be associated to. The registration interface simply stores the component descriptor in the specific DSM components repository of the given DSM platform, using the repository's URL specified within the configuration package.

Summarizing, the component development approach we propose has two main benefits: (i) allows developers to implement new components exploiting well known techniques and tools they are used to and (ii) allows the integration of already existing services available within a specific target domain at a negligible cost.

7. USE CASE: RESEVAL MASH

To let the reader better understand how our framework works and the steps and efforts that platform designers should go through during the development of a DSM platform using it, in this section we provide a concrete use case. We show how a DSM platform similar to ResEval Mash can be implemented using our framework. We select ResEval Mash as reference platform since we already introduced it in Section 2.2 and, furthermore, to take the research evaluation as target domain, which readers are already acquainted with and can find easier to understand.

7.1. Scenario and requirements

For the selected scenario we want to build a mashup platform for the development of processes for the evaluation of researchers and research production. This tool is useful in various situations, e.g., researchers hiring or research production evaluation for

funds distribution, and has to be directly used by (typically university) administrative employees (i.e., our domain experts).

The research evaluation processes are data-driven processes which are characterized by the need for managing and processing large amounts of data, coming from different bibliographic web sources and processed by specific services. These data-processing compositions typically also include some user interface, used only to present the process result to the domain experts.

The description given above represents only the main requirements for the target domain language and platform and is meant to let the reader understand the context and the core needs of the target domain. The complete requirements definition must be done following the methodology summarized in Section 2.3, which guides the platform designer through the domain analysis and formalization phase. Here we do not further expand on this phase, since it is not the main focus of this paper. For the details about this phase and its resulting output artifacts in the context of the research evaluation domain, we refer to [Daniel et al. 2012]. The next section shows how our system can be used to generate our target platform, assuming the domain analysis has already been done and its output artifact are available.

7.2. Building ResEval Mash, simply

Once the domain analysis phase has been completed following our methodology, the platform designers have a clear idea about the specific requirements of their target platform and have produced the set of domain formalization artifacts described earlier in the paper (e.g., DCM, MM, PM, domain syntax).

At this point the platform designers can start the platform generation procedure. As discussed in Section 6.1, the only action required to them is to configure the DSM language and the platform through the simple interface shown in Figure 7. Considering the requirements of our research evaluation scenario, the designer should select a set of features like the one shown in Figure 12.

The figure shows the whole list of required features for our use case. In particular for the components configuration we have specified that both data, service and UI components are needed, and for each type the supported communication patterns (e.g., UIs are used only to show processing results, therefore, only the one-way pattern is selected) and implementation technologies (e.g., service components can be implemented either as SAOP or REST services).

Then, since our processes are mainly data-driven, we define as base composition paradigm the data flow and we also state that compositions will have their UI counterpart rendered on a single page.

Finally, we require the enabling of the reference passing mode in the runtime environment (significantly limiting the processing time due to latency and network bottlenecks in data-intensive processing scenarios) and the automatic data-mapping functionality in the mashup editor (allowing the wiring definition at operations level and, therefore, relieving the domain expert of the setting of low-level, parameter-to-parameter data-mappings, which are automatically defined based on parameters types matching).

Once the features have been selected the last step is the upload of the domain syntax descriptor, which has been produced during the domain formalization phase. After checking features selection validity pressing the related button (which in this case confirms its validity as shown in the figure), the last step is press the generate button, which will trigger the platform generation procedure, as described above in Section 6.1.

The generation process will first produce the DSM platform configuration package including the generated DSM languages. In Figure 13 shows some excerpts of an ex-

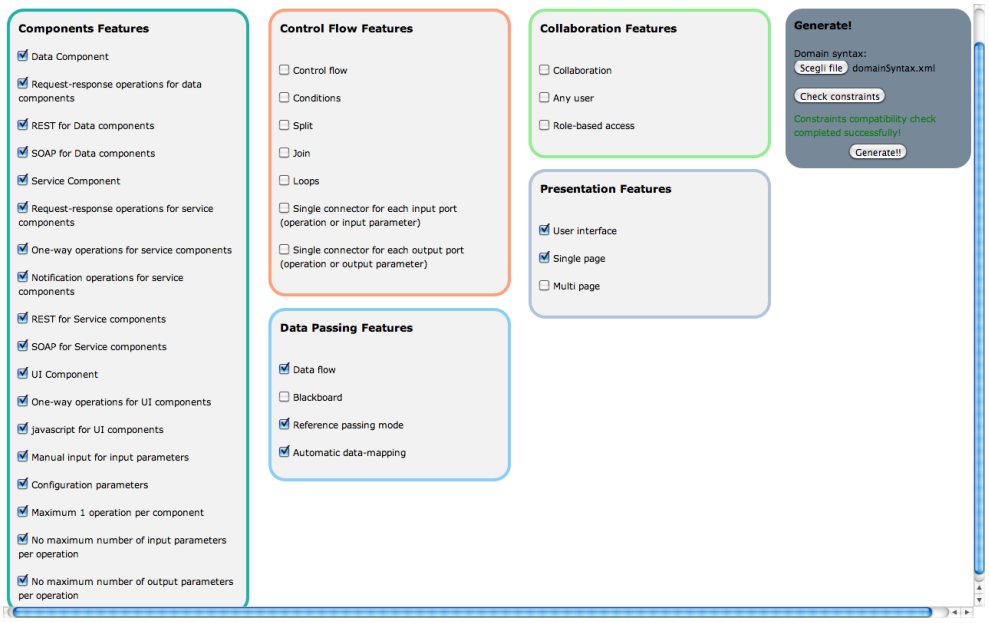


Fig. 12. The set of features selected for the generation of the research evaluation DSM platform. Note: here non-selected features of the Component category have been removed from the UI for visualization purposes.

ample composition XML definition (implementing the scenario presented in Figure 1), which is compliant with the generate DSM composition language.

Using this configuration package the framework provides a hosted mashup editor customized to fit the generated DSM language, along with its constructs, to support - all and only - the selected features and to show the domain specific syntax provided during the platform design and generation steps. Figure 14 shows the DSM editor customized based on the features discussed above. The figure includes, as composition example, the mashup implementing the scenario presented in Figure 1. The editor provides only the functionalities supporting the selected features and, for instance, control flow constructs (e.g., split or join) or global variables are not present at all. In addition, each component has an own domain-specific graphical syntax.

Some more tuning could be needed in case of particular user interface or functional requirements. For example, the actual ResEval editor provides some functionalities we still do not provide, like the “live development environment”, where mashup execution is performed live during its development. The provided editor, thus, can be used as it is or can be used as solid and valuable base for building a - even more customized - DSM editor.

The provided runtime environment, instead, is provided “off-the-shelf”: it does not need any modification (unless new language constructs are introduced) and is ready to run any mashup defined through the generated DSM language.

Clearly, developers must also develop the domain-specific components. However, as already discussed in Section 6.5, already existing services can be integrated in the platform very easily, typically just providing an XML descriptor defining their main attributes and interface.

This section shows how simple is for a platform designer to get an almost-complete (still components must be implemented by developers) DSM platform with a negligible

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

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

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

  [...]

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

  </component>

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

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

Fig. 13. Partial mashup XML definition implementing the scenario presented in Figure 1

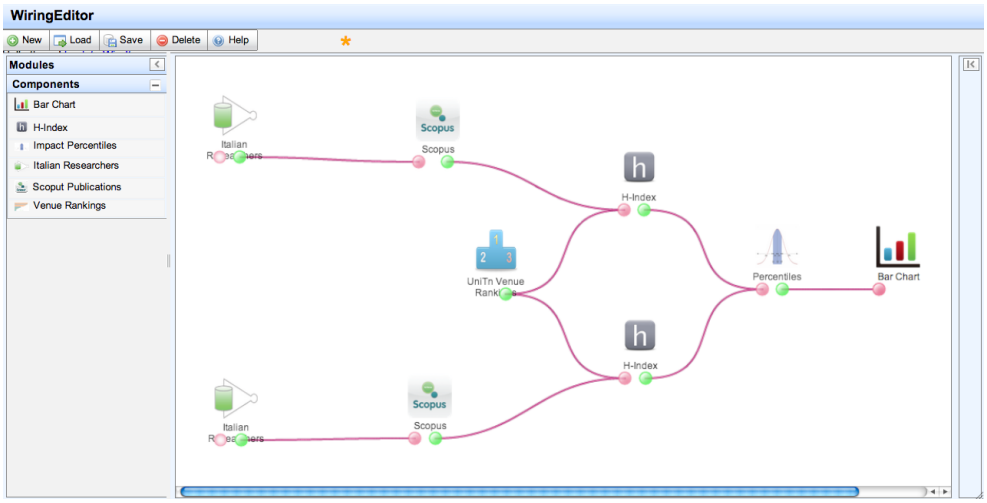


Fig. 14. The mashup implementing the scenario presented in Figure 1, designed within our the DSM platform

effort, clearly, excluding the domain analysis phase, that must be manually performed in any case. In other words, most of the actual development effort and cost is absorbed by the DSM platform generation framework we provide.

8. EVALUATING THE SYSTEM

In the previous sections we described the concepts, approaches and tools underlying the DSM platform generation framework. In this section, we describe their limitations and how we validated them to provide a clearer picture of the value of this work.

8.1. Validation

The main goal of this work is to enable EUD. We propose to address this goal providing users (domain experts) with domain-specific mashup solutions, as we describe in this paper.

We validated the domain-specific mashup approach through the user studies ran on ResEval Mash [Daniel et al. 2012], which have shown that domain experts are able to manage composition tasks when provided with domain-specific mashup tools recalling concepts and semantics of the domain they work in, therefore, validating the DSM approach as effective EUD enabler.

Moreover, in [Soi et al. 2013] we have shown that the set of features we support allow the generation of DSM languages supporting the requirements of different types of mashup tools, thus, providing a validation for our conceptual language generation approach and for its coverage in terms of requirements it can support. For example, we generated mashup languages having the same expressive power of the languages underlying the YahooPipes, mashArt and ResEval Mash tools, which present very different requirements, all supported by the languages we generated.

Validating the DSM platform generation framework itself is extremely complex. To do it, we should find a number of developers able to use our framework (i.e., fully aware of mashup technologies and able to design a mashup platform), which are also experts in a given application domain and that are willing to work with us for a relatively long period (in the order of several weeks), since they should go through the methodology steps (to analyze the domain and produce the relative artifacts) and through the platform generation process using our framework. In addition, a test group developing similar platforms through the standard, manual development approach would be also needed. Being able to perform a similar study would be clearly useful to evaluate and collect feedbacks about the usability and the effectiveness of our DSM platform generation framework, but it is evident that this would be extremely costly in terms of resources (i.e., the mashup platform developers) and, thus, almost impossible to realize.

However, in this paper, we describe how the generation framework is able to create DSM platforms including languages and tools supporting the set of identified features. Since, as we have shown, supporting these features allows us to cover a wide range of mashup platform requirements, this let us state that the DSM platforms we generate are able to address the same wide range of requirements. This means that our generation framework can be used to produce DSM platforms for many applications domains having different requirements.

8.2. Limitations

The DSM platform generation framework already provides a fairly wide set of features allowing us to build languages and tools supporting many different domain requirements. Nonetheless, the list of identified features comes without the claim of completeness and it is meant to grow over time to increase the spectrum of requirements the framework can effectively support. The framework will be available as open-source project to let the community further expand it, e.g., including the support for new features or improving the current runtime or development environments.

Regarding the mashup editor, we still provide a basic editor whose implementation has still to be completed to better support all the identified features. In addition, depending on the specific needs developers may have, the editor may benefit from some extension related to the user interfaces or user experience. The runtime environment, instead, is fully working and supports the conceptual features we identified.

The framework do not provide any user management system. A similar system is needed, e.g., to associate the users (i.e., domain experts) to the different DSM platforms hosted on our server (so that they can access the mashup editor without providing the required DSM configuration packages reference) or to set up access restrictions to the mashup compositions developed by domain experts. User management systems, if needed, must be implemented as external modules that than suitably invoke or redirect to our tools.

Finally, in general we do not argue we can build effective platforms for any domain, but we argue that our methodology and system can be applied to many domains allowing a faster and more affordable development of DSM platforms for them.

9. RELATED WORK

The main goal of this work is to bring development to end users. Mashups are commonly seen as a viable solution to enable end user development (EUD), as described in [Grammel and Storey 2010]. Despite this, current mashup tools still fail in achieving this goal since they present concepts and compositional elements that end users cannot understand and compose [Namoun et al. 2010b; 2010a]. The solution we propose to address this problem is to provide end users with domain-specific mashup (DSM) tools. To foster and facilitate the adoption of this DSM solutions, we also propose a DSM platform generation framework making the DSM platform development simpler and faster. To the best of our knowledge, no other research works are focussing on domain-specific solution within the mashup context. Next, thus, we provide an overview of the state of the art of the two research areas most closely related to our approach, i.e., domain-specific development and mashups.

Domain-specific development. The idea of exploiting domain 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]. As mentioned, none of them is directly related to the mashup research field and, instead, most of them are related to the Domain Specific Languages (DSL) and Domain Specific Modeling (DM) research areas.

In DM, domain concepts, rules, and semantics are represented by one or more models, which are then translated into executable code. DM tools provide domain-specific programming instruments, allowing developers to abstract from low-level programming details, and powerful code generators that implement the application code on behalf of the developers. Studies performed using different DM tools (e.g., the commercial MetaEdit+ tool and the academic solution MIC [Lédeczi et al. 2001]) have shown that developers' productivity can be increased up to an order of magnitude. Managing these models, though, can be a complex task suited only to programmers. 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] a DSL for mashup development). Also here the introduction of the "domain" raises the abstraction level, but the typical textual nature of these languages makes them less intuitive, harder to manage and, thus, not suitable for end users. Benefits and limits of the DSM and DSL approaches are summarized in [France and Rumpe 2005] and [Mernik et al. 2005].

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]. Most mashup tools also claim to enable EUD targeting non-programmers. Fischer et al. [2009] analyzed a large set of mashup tools and categorized them based on the development paradigm (from manual to automatic development paradigms) and on the required skills to the user (from expert programmer to casual user). The outcomes of this survey state that none of the available tools is actually able to enable EUD and proposes a new automatic tool targeting this goal [Fischer et al. 2008]. The tool focuses on automatic service composition only.

In general, mashup tools can focus on data, service or user interface (UI) integration, or on a combination of them. Yahoo! Pipes (`http://pipes.yahoo.com`) is an example of mashup tool focussing on data integration. It 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 require some basic programming knowledge.

The ServFace project (`http://www.servface.eu`), instead, aims to support web users in composing semantically annotated web services. Annotations are used to automatically gener- ate form-like interfaces for the involved services, which can be placed onto one or more web pages that can be graphically linked to specify the data flow in the composition. The result is a simple, user-driven web service orchestration tool [Nestler et al. 2010], but UI integration and process logic definitions are rather limited and, yet, basic programming knowledge may still be required.

An example of mashup tool focussing on UI integration is Intel Mash Maker. It provides a completely different mashup approach: rather than taking inputs from structured data sources (e.g., RSS/Atom feeds), Mash Maker allows users to reuse entire web pages and, if suitably annotated, to extract data from them. Mash Maker focusses on data extraction and UI presentation, but the concept of service composition is completely missing. Its use, especially for advanced features, requires programming skills.

Our mashup tool, mashArt [Daniel et al. 2009], supports the integration both data, services and UIs in one single language and tool. This is what we called *universal integration*. Also this tool, although designed to target end users, failed in enabling EUD.

We developed also another mashup-like tool implementing the universal integration idea, MarcoFlow [Daniel et al. 2011]. This tool extends the standard service integration layer provided by BPEL with a presentation layer allowing to include UI components within standard service compositions, so that to integrate users in the compositions themselves. This tool, however, explicitly target programmers with advanced service composition skills.

The variability in the mashup tools' characteristics, functionalities and approaches is very high, as described in several works [Aghaee et al. 2012], [Koschmider et al. 2009], [Hoyer and Fischer 2008]. Aghaee et al. [2012] define a mashup design space based on different dimensions, represented by design issues. The article provides a clear idea of the mashup tools' variability and of the complexity of designing a mashup platform.

These reasons motivated us to the development of solutions proposed in this article, i.e., (i) the DSM approach (to address the lack of usability identified by Fischer et al. [2009] hampering EUD), (ii) the conceptual design approach (facilitating the complex mashup platform design phase) and (iii) the platform generation framework (which allows to automatically translate the design choices into a concrete mashup platform, fostering the adoption of our solutions).

10. LESSONS LEARNED

Our research activities were driven by the inspiring idea that mashup technologies can enable the radical paradigm shift making non-programmers the real designers and developers of their own applications, that is, enable end user development (EUD). Clearly, the development of, e.g., complex and/or large software systems will still require the work and expertise of software architects and professional developers. For the development of simpler applications (realizable through the lightweight composition of the huge variety of available data, services, APIs and UI widgets), though, this paradigm shift would have a huge impact, moving their development from IT departments directly to the final consumer (or *prosumer*, at this point) and enabling the development of situational applications that today cannot just be implemented, since they typically require too many resources to be developed following the standard software development lifecycle.

During our work we have learned that to make mashup tools more usable (to go towards EUD) should be the tools to adapt to users and to their mindset and habits, and not the opposite. This is the difference between DSM solutions and general-purpose ones. The former provide users with compositional elements resembling the concepts they face and manage in their everyday life, which, thus, they are able to understand and manage. The latter, instead, require users to map the concepts they know to lower-level compositional elements, which, suitably composed, can be able to represent those concepts. This abstraction and mapping exercise, though, is far from the possibilities of the non-IT skilled end users, since it typically require programming knowledge (to understand the low-level composition elements) and abstraction skills (to map them to the concepts familiar to the user) that end users do not have. These lessons motivated us to design and follow the DSM approach proposed in this work, which allows the creation of simpler mashup tools that are domain-specific and closer to domain experts' mindset and, finally, more usable for them.

We are convinced that the DSM approach is an important starting point to achieve our main goal, i.e., bring application development to end users, but it must be further developed and merged with other approaches and technologies. In our experience we have been involved in the design of other approaches going towards EUD and we realized that they can complement each other, leading to really effective solutions to enable a wider and wider range of users to develop their own applications. Our approach goes in the direction of simplifying to the possible extent the mashup tools. Other approaches follow different ways to enable EUD; for example, a promising research track focusses on the development of technologies to assist end users during the mashup development phase, e.g., recommending possible needed components or how to compose them. An example of this kind of tools is Baya [Chowdhury et al. 2012], which has been developed by our research group and is being applied in the context of the OMELETTE project. The convergence of the DSM approach with other technologies like the one just described is, in our vision, the right way to follow to enable EUD in real-life scenarios.

Another important lessons we learned, and that we try to apply to our solutions, is that to achieve EUD, the tools we provide to end users must comply with the definition of *gentle slope systems* [Myers et al. 1992]. This means that the tools must allow users to learn how to effectively use the tool step-by-step without steep learning curve. We have experienced this also during the user studies we did during our research, understanding that this is particularly true for end users. This class of users is not prepared to and is not even interested in learning a large amount of conceptual and technology-related notions to use a tool. Users must be able to learn by attempts and constantly perceive the results of their learning efforts. This is very important also in the mashup

context. The inherent complexity of modeling a composition, establishing the components' execution sequence and dealing with the data passing require an algorithmic and technical mindset that end users do not have, but that, as we have seen, they can "gently learn" when provided with DSM mashup tools [Daniel et al. 2012] (which are simpler to understand and work with) and, e.g., assisted during the composition development (making the learning slope even more gentle) [De Angeli et al. 2011].

As we have directly experienced during the development of several mashup tools (i.e., mashArt, MarcoFlow, OMELETTE Live mashup Environment, ResEval Mash) developing mashup platforms, in particular DSM ones, is definitely non-trivial and expensive. To foster and push the adoption of the DSM approach, which we consider a key ingredient towards the EUD goal, we designed supporting methodologies and tools for making their development simpler, faster and more affordable.

Finally, although the approaches and tools proposed in this paper focus on the development of DSM solutions targeting EUD, we recognize that the generation framework we provide can also be applied to other contexts. In general, through its conceptual design approach and platform generation algorithms, it can simplify the development of many types of mashup tools. For example, it can be used to rapidly build mashup platforms easing the work of developers (thus, not targeting EUD, for the development of scientific workflow systems (like myExperiments[12]) requiring to deal with data-intensive process or, in general, in any case a custom mashup platform may be useful.

In the next future we will work to expand the set of features supported by our generation framework and to make the tools it provides more stable and complete (in particular the editor). Then, we will make the whole generation framework available as open-source project, letting the community use it and participate to its evolution. In addition, we will also work on the integration of complementary technologies into the development environment to improve its usability level. In particular, we plan to integrate the above mentioned Baya system, which assists the end users during the composition tasks.

REFERENCES

AGHAEE, S., NOWAK, M., AND PAUTASSO, C. 2012. Reusable decision space for mashup tool design. In *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems*. ACM, 211–220.

CHOWDHURY, S. R., RODRÍGUEZ, C., DANIEL, F., AND CASATI, F. 2012. Baya: Assisted mashup development as a service. In *Proceedings of WWW 2012 Companion*. 409–412.

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. 2009. Hosted universal composition: Models, languages and infrastructure in mashart. In *Proceedings of ER'09*.

DANIEL, F., IMRAN, M., SOI, S., ANGELI, A. D., WILKINSON, C. R., CASATI, F., AND MARCHESE, M. 2012. Developing mashup tools for end-users: On the importance of the application domain. *International Journal of Next-Generation Computing (IJNGC) 3,* 2.

DANIEL, F., SOI, S., TRANQUILLINI, S., CASATI, F., HENG, C., AND YAN, L. 2011. Distributed orchestration of user interfaces. *Information Systems, Elsevier 37,* 6, 539–556.

DE ANGELI, A., BATTOCCHI, A., CHOWDHURY, S. R., RODRIGUEZ, C., DANIEL, F., AND CASATI, F. 2011. End-User Requirements for Wisdom-Aware EUD. In *Proceedings of IS-EUD*. 245–250.

FISCHER, T., BAKALOV, F., AND NAUERZ, A. 2008. Towards an automatic service composition for generation of user-sensitive mashups. In *Proceedings of the 16th Workshop on Adaptivity and User Modeling in Interactive Systems*.

FISCHER, T., BAKALOV, F., AND NAUERZ, A. 2009. An overview of current approaches to mashup generation. In *Proceedings of the International Workshop on Knowledge Services and Mashups (KSM)*.

[12]myExperiments homepage: `http://www.myexperiment.org/`

FRANCE, R. AND RUMPE, B. 2005. Domain specific modeling. *Software and Systems Modeling 4*, 1–3.

GRAMMEL, L. AND STOREY, M. A. 2010. *The Smart Internet*. LNCS Series, vol. 6400. Springer, Chapter A Survey of Mashup Development Environments, 137–151.

HARTMANN, B., DOORLEY, S., AND KLEMMER, S. 2006. Hacking, Mashing, Gluing: A Study of Opportunistic Design and Development. *Pervasive Computing 7,* 3, 46–54.

HOYER, V. AND FISCHER, M. 2008. Market overview of enterprise mashup tools. In *ICSOC*. 708–721.

KOSCHMIDER, A., TORRES, V., AND PELECHANO, V. 2009. Elucidating the mashup hype: Definitions, challenges, methodical guide and tools for mashups. In *Proceedings of Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web*.

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 *Proceedings of ICSOC*. 13–26.

MERNIK, M., HEERING, J., AND SLOANE, A. M. 2005. When and how to develop domain-specific languages. *ACM Computing Surveys 37,* 4, 316–344.

MYERS, B., SMITH, D. C., AND HORN, B. 1992. *Languages for Developing User Interfaces*. Jones and Bartlett, Boston, Chapter Report of the 'End-User Programming' Working Group, 343–366.

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: Prospects, Problems, and Design Recommendations. In *Proceedings of the 8th IEEE European Conference on Web Services (ECOWS)*. IEEE, 123 – 130.

NESTLER, T., FELDMANN, M., HÜBSCH, G., PREUSSNER, A., AND JUGEL, U. 2010. The servface builder - a wysiwyg approach for building service-based applications. In *ICWE*. 498–501.

SOI, S., DANIEL, F., AND CASATI, F. (in press) 2013. *Web Services Foundations*. Springer, Chapter Conceptual Design of Sound, Custom Composition Languages.

W3C. 2011. Widget Packaging and Configuration. W3C Working Draft.

W3C. 2013. The WebSocket API.

WILSON, S., DANIEL, F., JUGEL, U., AND SOI, S. 2011. Orchestrated user interface mashups using w3c widgets. In *Proceedings of ICWE Workshops*. Springer, 49–61.

YU, J., BENATALLAH, B., CASATI, C., AND F., D. 2008. Understanding mashup development. *IEEE Internet Computing 12,* 5, 44–52.