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:

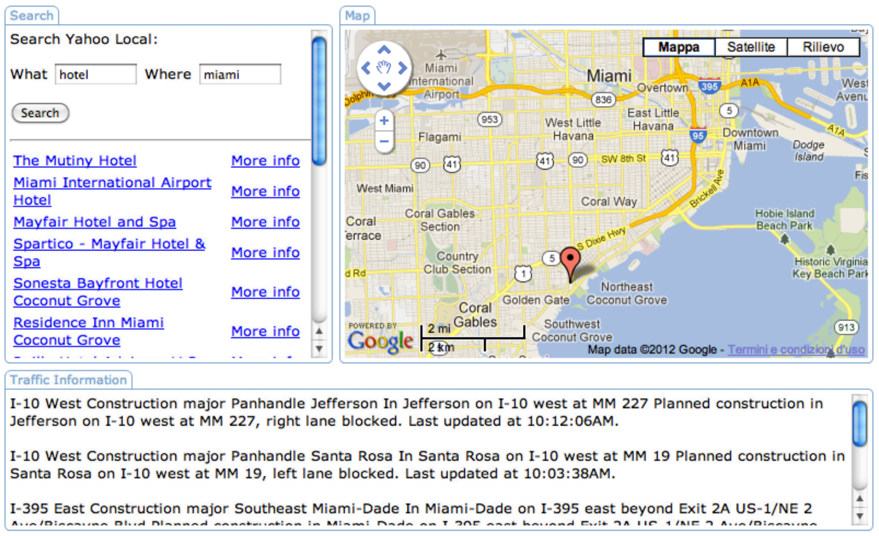
- We provide a comprehensive conceptualization of the most important *composition features* that characterize todays most prominent composition languages.
- We derive a *generic, extensible composition language meta-model*, which expresses how the identified features can be used together for the construction 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



Search Yahoo Local:

What hotel Where miami

Search

The Mutiny Hotel More info
Miami International Airport Hotel More info
Mayfair Hotel and Spa More info
Spartico - Mayfair Hotel & Spa More info
Sonesta Bayfront Hotel Coconut Grove More info
Residence Inn Miami Coconut Grove More info

I-10 West Construction major Panhandle Jefferson In Jefferson on I-10 west at MM 227 Planned construction in Jefferson on I-10 west at MM 227, right lane blocked. Last updated at 10:12:06AM.

I-10 West Construction major Panhandle Santa Rosa In Santa Rosa on I-10 west at MM 19 Planned construction in Santa Rosa on I-10 west at MM 19, left lane blocked. Last updated at 10:03:38AM.

I-395 East Construction major Southeast Miami-Dade In Miami-Dade on I-395 east beyond Exit 2A US-1/NE 2

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

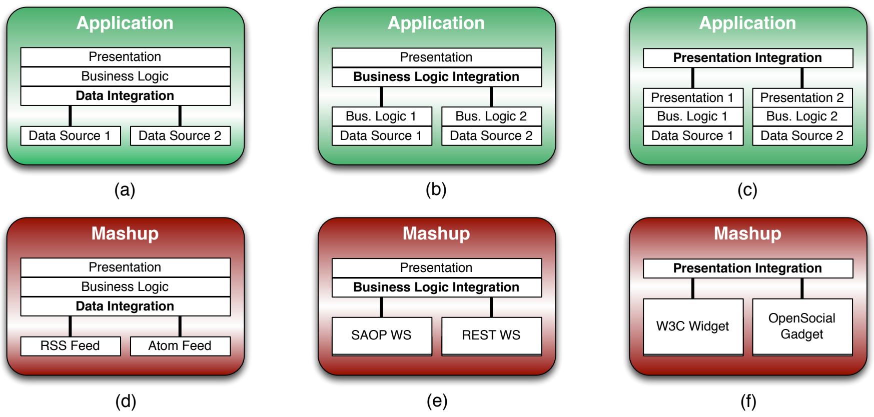


Fig. 2 The different levels of integration in general and in the specific context of web mashups.

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

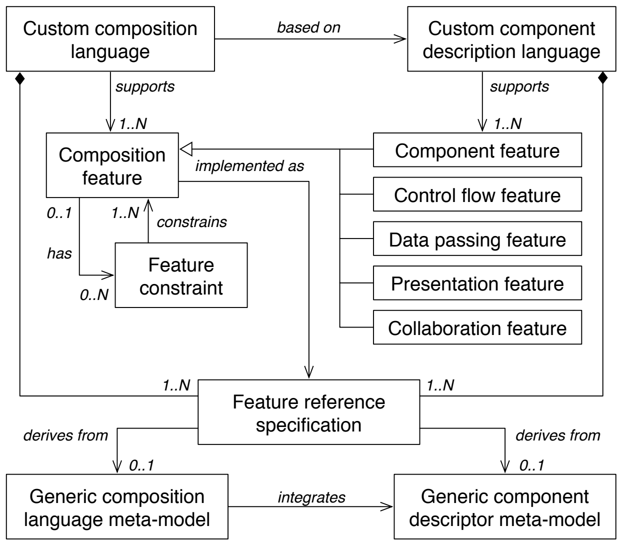


Fig. 3 Conceptual approach to the development of custom composition languages

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.

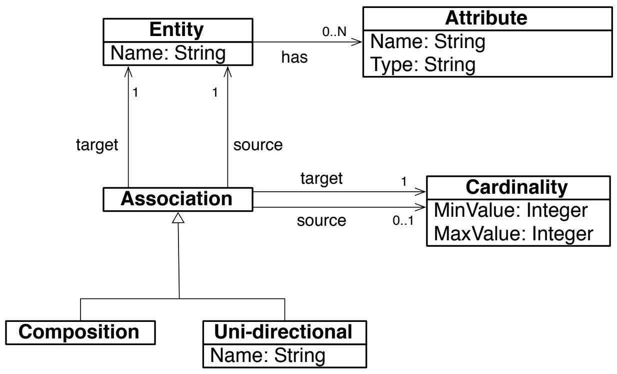


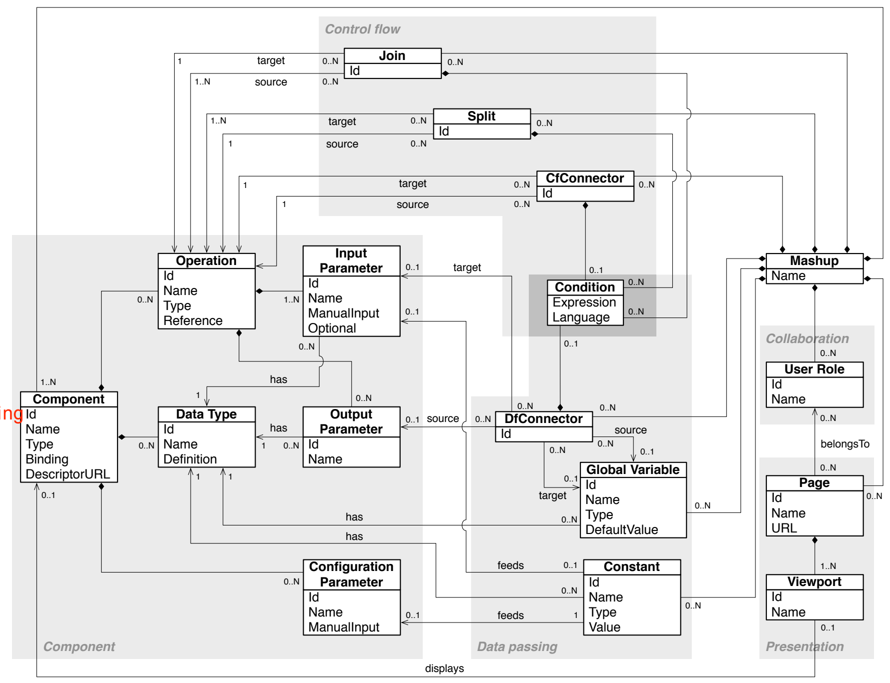
Fig. 4 Composition language meta-meta-model

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



SupportReferencePassing
Class
Syntax

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 (*Df-Connector*), 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.

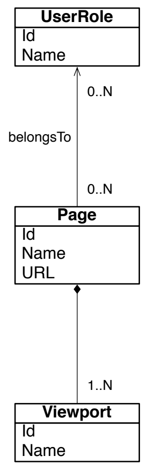
The above features and examples show that developing a good generic meta-model is a *trade-off* between the simplicity and usability of the final language (the fewer individual constructs the better) and the ease of mapping features onto the meta-model (the more constructs the better; in the extreme 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 specification 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  $Fsel = \{f_j \mid f_j \in F, f_j.name \in FnameSel\}$; //load sel. features from knowledge base F
5  for each $fb \in FB$ //set values of Boolean variables in FB
6     fb.val = ($fb.name \in FnameSel$) ? true : false;
7  for each $f \in Fsel$ //construct set of constraints to be checked
8     $CONSTR = CONSTR \cup f.Constr$;
9  if (checkSoundness(CONSTR, FB) == false) then //check soundness
10    return null; //interrupt processing if constraint conflicts occur
11 for each $f \in Fsel$
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`

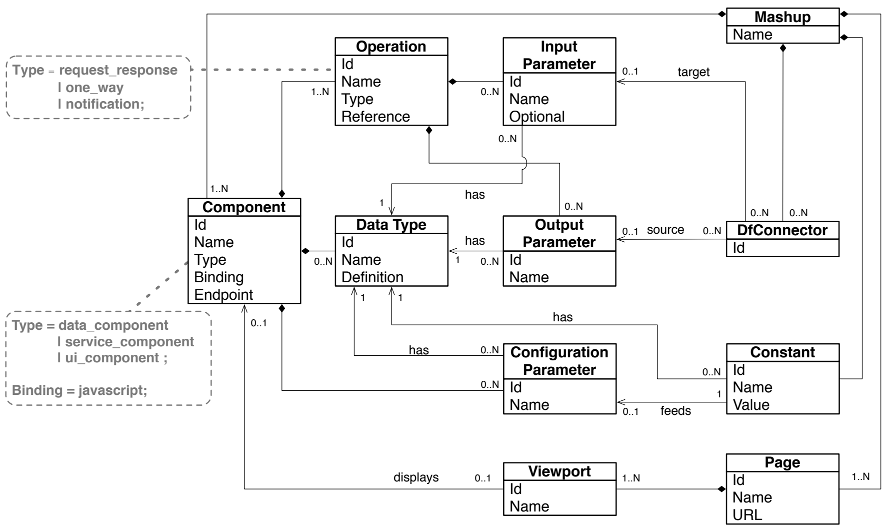


Fig. 7 A composition language meta-model supporting the discussed features set

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">
```

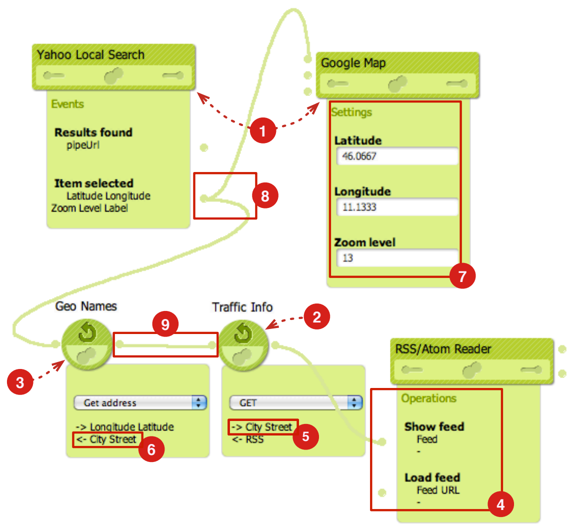
```
         <output id="O2-1" name="Latitute" dataType="double"/>
         <output id="O3-1" name="Longitue" dataType="double"/>
         <output id="O4-1" name="Zoom Level" dataType="int"/>
         <output id="O5-1" name="Label" dataType="string"/>
      </operation>
   </component>

   <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.

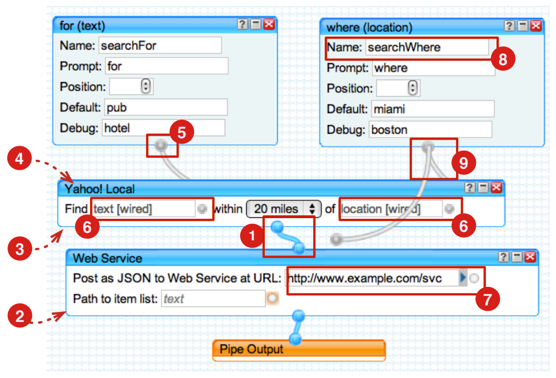


Latitude: 46.0667
Longitude: 11.1333
Zoom level: 13
Get address
-> Longitude Latitude
<- City Street
GET
-> City Street
<- RSS

Selected feature names *FnameSel*

1	ui_component one_way_for_ui notification_for_ui javascript_for_ui
2	service_component request_respone_for_service one_way_for_service notification_for_service javascript_for_service
3	data_component request_response_for_data javascript_for_data
4	max_N_operation_per_component
5	max_N_input_param_per_operation
6	max_N_output_param_per_operation
7	configuration_param
8	branch merge
9	data_flow
●	user_interface single_page

Fig. 8 mashArt example composition model and the set of respective language features



Name: searchFor
Prompt: for
Position:
Default: pub
Debug: hotel

Name: searchWhere
Prompt: where
Position:
Default: miami
Debug: boston

Find text [wired] within 20 miles of location [wired]

Post as JSON to Web Service at URL: http://www.example.com/svc
Path to item list: text

Selected feature names *FnameSel*

1	data_flow
2	service_component REST_for_service request_response_for_service
3	data_component RSS_for_data atom_for_data request_response_for_data
4	max_1_operation_per_component
5	max_1_output_param_per_operation
6	max_N_intput_param_per_operation
7	manual_input
8	configuration_param
9	branch

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

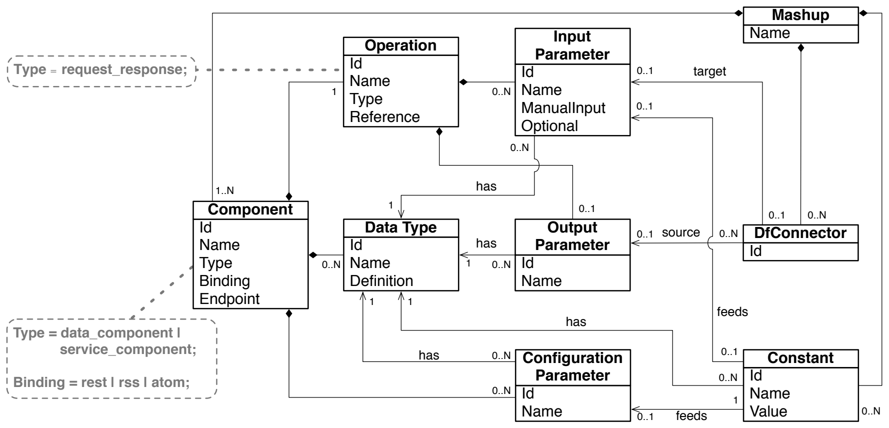


Fig. 10 A composition language meta-model supporting the discussed features set

(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 integration 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.