Modeling the *Mashup* Space

Serge Abiteboul
INRIA Saclay
Paris, France
<fname>.<lname>@inria.fr

Ohad Greenshpan
Tel-Aviv University &
IBM Research Labs
Tel-Aviv, Israel
ohadg@il.ibm.com

Tova Milo
Tel-Aviv University
Tel-Aviv, Israel
milo@tau.ac.il

ABSTRACT

We introduce a formal model for capturing the notion of *mashup* in its globality. The basic component in our model is the *mashlet*. A mashlet may query data sources, import other mashlets, use external Web services, and specify complex interaction patterns between its components. A mashlet state is modeled by a set of relations and its logic specified by datalog-style active rules. We are primarily concerned with changes in a mashlet state relations and rules. The interactions with users and other applications, as well as the consequent effects on the mashlets composition and behavior, are captured by streams of changes. The model facilitates dynamic mashlets composition, interaction and reuse, and captures the fundamental behavioral aspects of mashups.

Categories and Subject Descriptors

H.2.1 [**Database Management**]: Logical Design—*Data models*; H.1.0 [**Models and Principles**]: General

General Terms

Design, Languages

Keywords

Web, Mashups, Data, Model, Information, Integration

1. INTRODUCTION

A (music) mashup is a composition created from the combination of music from different songs. *Web mashups*, in a similar spirit, stem from the reuse of existing data sources or Web applications, the emphasis being on GUI and programmingless specification. As described in [22], the concept of mashups originated from the understanding that the number of applications available on the Web and the needs to combine them to meet user requirements, are growing very rapidly. However, these applications are often complex, provide access to large and heterogeneous data, varied functionalities and built-in GUIs, so that it becomes in many cases an impossible task for IT departments to build them in-house as rapidly as they are requested to. The role of mashups is to facilitate this rapid, on-demand, software development task. Furthermore, by enabling dense (Web 2.0-style) interactions between components, the mashup concept goes even further, illustrating the Gestalt principle that the whole is different than the sum of its parts.

From a scientific viewpoint, previous works have typically focused on *specific* aspects of mashups. Among those one can list in particular works that studied (semantic) data integration [10, 9, 7], works dealing with service composition [36, 33, 40], and works considering interaction between mashup components and interaction with users [15]. Each such aspect is clearly interesting in itself. However, we believe that it is also essential to understand the notion of mashup in its globality, and in particular the interaction between the various facets previously mentioned. For that there is a need for a formal model for mashups. *Such a model is still missing, and is the topic of the present paper.*

Figure 1: The `MedickIT` application

Before presenting our model, let us consider a simple example that highlights some of the main requirement from such a model. An example of a mashup is shown in Figure 1. It is from the `MedickIT` system, developed for the healthcare domain at IBM Research Labs in Haifa. It served both as a motivation for the present work and as a test bed for some of the ideas presented here. The mashup is composed of several components that we call *mashlets*. A mashlet can be GUI-based (e.g., a widget) or not (e.g., Web Service). To allow for modular application development, a mashlet can itself serve as a component in some other (more complex) mashlet. The screen shot here contains six mashlets. An Electronic Health Record (EHR) mashlet (at the top left corner), a map, a calendar, a medical search engine, an SMS mashlet, and a medical

WIDM'08, October 30, 2008, Napa Valley, California, USA.

data analyzer. Various interactions between the mashlets are possible. For instance, addresses of doctor appointments can be pulled from the *calendar* and then be fed into the *map* for display; The appointments time can be retrieved from the *calendar* and sent as *SMS* reminders to phone numbers taken from the patient *EHR*. This may be done automatically by the application (e.g. in response to some events), or on user demand (e.g. by drag and dropping a calendar entry on the SMS mashlet). In general, the set of mashlets presented to the user, as well as the interactions between them, may dynamically change in response to user actions or system events. For instance, in response to a user query on a particular illness, some dedicated patient forums that discuss it can be dynamically identified (by querying UDDI directories) and be plugged into the screen, as mashlets, replacing there some other (e.g. least recently used) components.

As previously mentioned, in our model, the basic component of a mashup is the *mashlet*. The model facilitates (dynamic) modular mashlets composition, interaction with users and other mashlets, and reuse, and captures the fundamental behavioral aspects of mashups. For simplicity, the state of a mashlet consists of a set of relations. Indeed, all data (including the data exchanged with users and other applications, as well as the data describing the mashup structure, state and components) is described by relations. The logic of a mashlet is specified using datalog-style active rules. The use of datalog enables taking advantage of advanced existing technology, notably query optimization [14] and is used to support other aspects such as monitoring [3]. The model is hierarchical in the sense that a mashlet can embed other mashlets, which can themselves embed other mashlets, and so on, recursively.

Since mashups are very dynamic by nature, we will be interested in changes (Δs) of the relations and the rules defining them. The interactions with users and other applications, are captured by streams of Δs. A mashlet may query data sources, import other mashlets, use external Web Services, and impose certain interaction patterns on its components. It can also be used by other mashlets and export some Web Services. All this can be specified statically (before the mashlet is activated) or dynamically (e.g., by incorporating a just-discovered mashlet in an active mashlet). So the program (set of rules) of a mashlet also evolves in time.

The use of a declarative language for specifying mashups provides a wide range of advantages. First, intuitive user interfaces, supporting the declarative language and not requiring to write complex code, can be used to specify mashups. Then, the semantics is precise and not implementation dependent, which protects against unexpected behavior. Also, it becomes possible to optimize mashups, in much the same way that relational system applications are optimized. Furthermore, the use of formal declarative specifications facilitates verifying properties of mashups, e.g., checking whether plugging a particular mashup in another one has any chance to succeed or whether it violates security requirements. Finally, such an approach facilitates mashup reuse and in particular searching for mashups, since we have clear ways of stating what specific mashups do.

The concept of mashup borrows many aspects from standard data integration technologies, such as databases, warehouses or portals, but poses requirements that are raising a number of new challenges. We present a brief review of the state of art in Section 2, highlighting central aspects and requirements that will be addressed by our contribution. Section 3 defines the mashlets, that are the basic components of our model. Section 4 illustrates, via a real-life example, how aspects discussed in Section 2 are addressed by our model. Future research directions are considered in Section 5.

2. RELATED WORK

To understand the importance of the problem, it suffices to browse the numerous Web sites dedicated to listing Web applications and those discussing their reuse. For instance, *Programmableweb.com* [30] is a Web site which lists thousands of mashups, organized in dozens of mashup classes. Statistics provided by the site show a constant increase of approximatively 100 mashups per month since September 2005. There is also a constant increase in the number of mashup classes (maps, music, etc.) offered on the Web site. Note that the mashups here undergo a filtering process by the site owners, so the real increase in the mashups number is clearly even higher.

Mashups integrate heterogeneous data provided by various sources and in that differ from classical databases and are more of a data warehouse [20] spirit. In contrast with data warehouses, they are typically specified and deployed very rapidly. They stress reactivity to changes in data sources, as in data warehouse maintenance [31]. Generally speaking, the technology developed for data integration is relevant for mashups, see, e.g., [24]. The management of information distributed between a large number of autonomous sources is also relevant. This has been recently the starting point of a number of works, e.g., dataspaces [17] and data ring [6]. Systems based on these principles can serve as underlying infrastructures for the management of data in mashups.

As an integration concept, the notion of mashup reminds in many aspects of a portal. The notion of portal has been discussed thoroughly and is being widely used (see e.g. Apache Jetspeed or IBM WebSphere Portal) but there is still no well-agreed definition in the literature. In [34], a portal is defined as "an infrastructure providing secure, customizable, integrated access to dynamic content from a variety of sources, in a variety of source formats." Although all this fits very well the definition of mashups, there are essential differences. The development of portlets, the portal components, typically requires programming expertise. Then, in most portals, the general layout and functionality of a portal are typically frozen once it has been developed. In others that enable dynamic selection of portlets and layout are limited to a pre-defined set of states and cannot be simply extended. Finally, there is usually limited interoperability between portlets in a portal, and the interactions between them have to be specified in advance. In contrast, the developer of a mashup, often also an end user, is typically not a programmer. The mashup designer is provided the means to define her own layout, with the relevant sets of components, and even define the level and way of interactions between them (in a Web 2.0 style). Another significant difference is that the components used in mashups present much more heterogeneity than in portals. In portals, even though the data may be stored elsewhere, the portlets are hosted in the same platform, share the same look-and-fill and support similar interfaces. In the mashup world, one can connect components running on different servers, with different user interfaces and displays.

The data used in mashups is by definition heterogeneous (i.e., comes with various formats and types). We expect that main standards for data streams, such as RSS and Atom, will be dominant. Previous work have dealt with data integration issues in mashups. Damia [7], being based on the abovementioned feeds, shows how to create such an integrator, and Mashmaker [15] presents how this integration can be queried and shared with other users. Enterprise information mashup fabric [22] was presented as a new class of integration technologies that will address such complex information composition tasks.

Mashups do not only integrate data, but also use Web Services provided by other components. Therefore, a mashup uses services provided by others and possibly provides services to be used by

other applications, notably mashups. It has been claimed [36] that most popular frameworks for Web Service composition, such as BPEL [26], do not meet the needs of mashups, in particular, from a data integration perspective. The recent proposal of mashup feeds [36] seems to be better adapted in that it deals both with data and control flows, so provides at least a partial solution. Models that are explicitly based on data, such as business artifacts [27] or Active XML [2], also seem better adapted to a mashup context than workflow approaches.

Mashups interact with both their components and users. Not surprisingly, with such deep interactions, mashup systems share a number of concerns with active databases [39] or event-driven architectures such as [16]. Interactions are captured in our model by streams of changes. This is in the spirit of Active XML [2, 25].

As already mentioned, our mashup model is strongly influenced by active databases and our language uses datalog-style rules. The particular variant that we employ here is specifically tailored to model the dynamic nature of mashups. In particular, we take a high-order dynamic approach where the active (datalog) rules themselves evolve over time. Since we embed mashlets into mashlets, we obtain a hierarchical data structure (with corresponding hierarchial rules). This is reminiscent of nested relations and complex objects. (References to works on these topics may be found in standard database textbooks, e.g. [37, 5]). Here again, for simplicity, we take from these models only the concepts essential for our context.

As mentioned in the introduction, our goal is to provide a design of a model that will allow to capture the many aspects of mashups, described above, in one uniform setting. In the next section, we introduce our model for mashups. The following one will detail an example.

3. THE MASHUP MODEL

As we will see, a *mashup* is a dynamic network of interacting *mashlets*, the basic components in our model. We introduce in this section, a formal model for mashlets. A particular mashlet may provide a graphical user interface (e.g., a widget) or not. The goal of the paper is to capture the essence of data management in mashups. We mostly ignore aspects related to visual interfaces or to the specifics of particular mashup systems.

For mathematical simplicity and elegance, the model we use is based on relations as in relational database systems. The state of a mashlet is maintained and represented by a set of relations. Some of them are *internal* relations, used to store internal data. Some are *input/output* relations, used for the interaction with users and other mashlets. Finally *service* relations (with binding patterns) are used for capturing Web Services that the mashlet imports or exports. There is a strong connection between the mashup model we introduce and the relational model, with mashlets as the analog of relational databases and mashlet components (internal, input/output and service relations) as the analogs of classical relations. A distinction however is that in the relational model, the component of a relation is a tuple of atomic values (first normal form). We relax this restriction here and, as we will see, our language will not be strictly first-order. In particular, mashlet relations may contain other relations and even entire mashlets.

Mashlets borrow from *object databases* [12], as well as *active databases* [39]. A mashlet can be seen as an object defined using a set of active rules such as:

$$\mathtt{R}_{out}(y,z,w) \text{ :- } \mathtt{R}_{in}(x), \mathtt{R}_{local}(x,y), \mathtt{WS}_{bff}(y,z,w)$$

Suppose that $\mathtt{R}_{in}$ is a relation that represents an editable window. The user enters values in it, say some illness name. The local relation $\mathtt{R}_{local}(x,y)$ provides the names of doctors specializing in such illness. Then a Web Service $\mathtt{WS}_{bff}$ (where the first parameter is bound and the last ones are free) provides, given the doctor names, their address and phone number. The relation $\mathtt{R}_{out}$, displayed in another window, describes the results.

Clearly, an essential aspect is defining when rules are invoked. The default is that a rule is invoked when some relation in its context (a relation occurring in the body) has changed. Our system also supports `changeAll` (all context relations have changed) or `onInit` (on mashlet initialization). Then the effect of a rule firing depends on the kind of assignment it uses. With ":-", the rule *defines* the value of the head relation. With "+:-", it *adds* new facts to the head relation, and with "−:-", it *deletes* some. Issues such as the execution order of simultaneously triggered rule, confluence and termination, are handled as in standard active databases [39].

To complete the picture, we consider in turn three essential aspects for mashlets, the first relating to the various ways mashlets may be used, the second about the dynamic nature of the mashlets interaction, and the third regarding the dynamic nature of the data that they manage.

Using mashlets. Given a mashlet specification $\mathcal{M}$, one can create a mashlet `M` that is an instance of $\mathcal{M}$. Once activated, the mashlet is by definition a Web component and as such it is associated with a URL. (The choice of this URL is beyond the scope of the present paper). This URL is available from inside the mashlet in a particular relation, `mashletURL(url)`. The mashlet runs for a *session*. Each of its relations may be defined *transient*, in which case its content is lost when the session ends; or *persistent*, in which case it is saved and reinstalled for the next session.

There are two ways of using such a mashlet instance. The first is in the spirit of Web Services technology. Each mashlet exports a Web-API, i.e., a list of methods (Web Services) that are exposed to be used by other software (possibly other mashlets). Each method comes with a signature that specifies how to use it. For instance, as in the previous example, one could export a service that, given some illness, provides the name, phone and address of doctors, e.g., $\mathtt{DocDir}_{bfff}$. Typically, a call to such a service causes a chain of rule invocations inside the particular mashlet, for computing and returning the result.

The second way to use a mashlet `M` is the essence of the mashups spirit. A mashlet exports a mashlet-API, namely the list of its input and output relations and their signatures. When a mashlet `M` is used alone, it interacts directly with the user via a GUI and the input/output relations. When `M` is used as a component of another mashlet, say $\widehat{\mathtt{M}}$, it may get its input from (and deliver its output to) other mashlet components of $\widehat{\mathtt{M}}$. This is achieved by including in the definition of $\widehat{\mathtt{M}}$, rules that direct the output of one mashlet component to the input of another one.

Consider again the previous example. We can import in the mashlet $\widehat{\mathtt{M}}$, a `MAP` mashlet. Then the following rule can be used to feed the doctor addresses, (retrieved when the user types in an illness name), as input to a `MAP` mashlet that displays them graphically:

$$\mathtt{MAP.Mrk}_{in}(x,y,name) \text{ :- } \mathtt{M.R}_{out}(name,phone,addr), \\ \mathtt{AddrToXY}_{bff}(addr,x,y)$$

The address is converted to x, y coordinates (using a Web service). Observe the naming convention: A relation `R` in a mashlet `M` is denoted `M.R`.

This inclusion of mashlets in other mashlets and the interconnection of mashlets may be viewed as a modularity mechanism. We do

not impose that the interacting mashlets be on the same machine. If we keep creating mashlets inside other mashlets, we reach the hierarchical organization that one typically sees in mashup system interfaces. However, we do not limit ourself to this hierarchical architecture. We allow interactions with external mashlets. One can use an output of $\texttt{M}_1$ in a rule of $\texttt{M}_2$ and conversely, in a recursive manner. This leads to a flexible framework for combining mashlets in mashlet *networks*. Such a *network of mashlets* is what we call a *mashup*.

Dynamic mashlets. The components of a given mashlet, as well as the interactions between them, may be specified statically, in the mashlet definition. More interestingly, they can also be added, changed or removed dynamically at run time, without interrupting the operation of the mashlet. For instance, depending on the type of illness specified by the user, one may want to find (e.g. by querying some UDDI-like directory) specific patient forums and dynamically include them in the mashup. Corresponding rules that connect the inputs/outputs of the new mashlets may also be dynamically added (e.g. to feed the illness name as input query to the various forums, and display the obtained results.)

This is achieved by the higher-order features of our model. In particular, each mashlet includes five relations called `Inputs, Outputs, Mashlets, MashletAPIs` and `Rules`, containing respectively, the names and signature of the mashlet input and output relations, the names of the component mashlets and mashlet-APIs, and the rules gluing their inputs and outputs. When a mashlet instance is created, they are initialized based on its specification. Then, at run time, they can be queried and updated, like any other relation, consequently changing the mashlet composition and behavior.

Dynamic Data. We distinguish between mashlet APIs (Web Services) that return a value (we call them *static*) and those that return an *update stream* (we call them *active*). The updates we consider are insertions of tuples in the result relation, and deletions of tuples satisfying a certain condition, e.g. of tuples with certain IDs. Similarly, the input/output relations of a mashlet may be specified as static or active. For instance, an input relation is static if once a value has been assigned to it (either by the user or by supplying it the output of another mashlet), it does not change. Otherwise it is active and may be changed (by the user or by using active output of another mashlet). To denote the fact that a service or a relation is active, we use an exclamation mark, e.g., `getLocation` might return our instantaneous location whereas `!getLocation` might monitor our moves until interrupted. In general, active services may provide data in a pull mode (as, e.g., in RSS feeds) or in push mode (as, e.g. in publish-subscribe systems). For simplicity we ignore the distinction here and consider the obtained stream of data items uniformly.

We call *mashup* such a (possibly dynamic) network of interacting mashlets. These are interacting Web components in the purest Web 2.0 style.

Remark We conclude this section with two remarks, one regarding the user interface and the second regarding data aggregation.

- A mashlet definition includes a specification of the layout of its components. This enables Internet browsers to present its content and support user interactions with the mashlet. Common properties which are included for the various components are coordinates on the screen and dimensions. We mostly ignore this important aspect here and focus on the management of data.
- Mashups that integrate (streams of) data items from several sources may wish to aggregate them along various dimensions [36]. This can be achieved by enriching our datalog-style language with aggregation functions, or alternatively, to keep the simplicity of the model, by abstracting the aggregates computation in a "black box" mashlet-API (Web Service).

4. MASHLETS AT WORK

We illustrate the above concepts with the example of a mashlet in Figure 2. This is a portion of the specification of the `MedickIT` application, providing a personal health information system for a patient. A screen shot of this mashlet (composed of other mashlets) was shown in Figure 1.

Recall first that each mashlet includes five built-in high-order relations, `Inputs, Outputs, Mashlets, MashletAPIs` and `Rules`, specifying respectively, the names and signatures of the input/output relations, the names of its mashlet components and their mashlet-APIs, and the rules gluing their inputs and outputs.

Some mashlet components are defined to be *visible* in which case they are displayed on the screen; others are *invisible* and are only used for computation purposes (to be detailed further). The input fields are the visible input relations of its visible mashlet components. The specification consists of four parts, as detailed in turn next.

Relations. `MedickIT` has two input relations. $\texttt{!Login}_{in}$ where the user inputs her name and password, and $\texttt{!Illness}_{in}$ where she inputs names of illness for which she wishes to obtain relevant information. Both relations are *active* as noted by the exclamation mark: The user can modify this information (e.g., in case of login error). The two output relations provide, respectively, the patient's scheduled doctor appointments ($\texttt{!DocAp}_{out}$) and the information gathered about the specified illness ($\texttt{!IllnessInfo}_{out}$). The output relations are also *active* and change in accordance to changes in the inputs. The application uses two internal relations: `UDDI`, a directory for mashlets dynamically included in the application (to be described further), and `Layout` for the details on the screen layout. `Layout` is defined as *persistent*, so layout adjustments performed by a user are reflected in the next session.

Mashlets and Mashlet-APIs. `MedickIT` includes, among others, four visible mashlet components. An `EHR` mashlet (for Electronic Health Record), storing patient health data, a CAL mashlet (for calendar), including, among others, the patient doctors list and their corresponding appointment dates, a `MAP` mashlet, and an `SMS` mashlet for sending SMS messages. As we will see further, some additional mashlets will be added dynamically at run time. In particular, depending on the illness input by the user, patient forums that discuss the illness will be plugged in into the system.

`MedickIT` also uses three external Web Services (some of them possibly supported by other mashlets not visible on the screem). `AddrToXY` is given a textual address and converts it to x, y coordinates (to be used for placing items on the map). `!MedicForums` returns a list of known medical forums (mashlets) and continuously updates this list with new information. (Its *active* nature is again denoted by the exclamation mark). Finally, `ForumsClassifier` (resp. `RelClassifier`) is given a forum id (relation name) and a topic and determines whether the forum discussions (relation content) are relevant to particular topics.

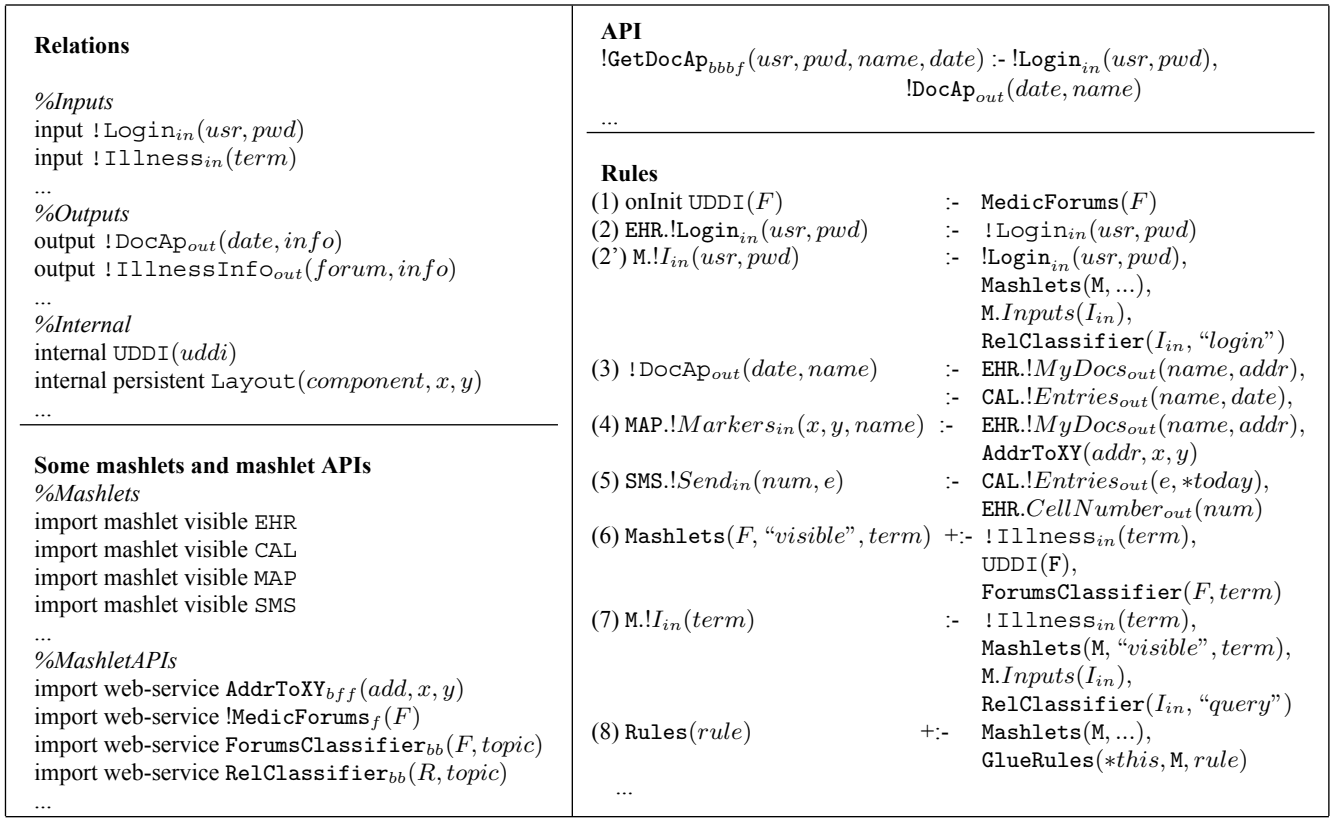

Figure 2: Part of the MedickIT **mashlet specification**

Rules. Datalog-style rules describe the logic of the mashlet and the interaction between its various components. We can see, in Rule 1, that when the session starts, the UDDI relation is initialized. It is set to contain a list of known medical forums retrieved by !MedicForums (and is continually updated by this service).

The first example of interaction between mashlets is given in Rule 2, where the user name and password typed by the MedickIT patient are automatically fed into the EHR mashlet, making her Electronic Health Record accessible with the relevant details. This rule reminds the single sign-on approach being used for access control. A more generic variant of this rule, that exploits the high-order nature of our model and allows to propagate the login information to all the relevant mashlet components, is depicted in Rule 2'. Rather than writing an individual rule for each mashlet component (as done for the EHR mashlet), we retrieve the mashlet names M from Mashlets relation of MedickIT. The names of the input relations of these mashlets are retrieved from the corresponding M.Inputs relations. The RelClassifier service (mashlet-API) is then used to determine to which input relations the login data should be fed into. (Further sophisticated variants, e.g. for scenarios where the same person uses distinct names/passwords for different applications, can be similarly defined.)

Now that the patient's EHR data is accessible, we use it, combined with the other mashlets to (i) extract the list of patient's doctor appointments, (ii) display the doctors office location on the map, and (iii) send SMS reminder messages to the patient. This is done by Rules 3, 4 and 5 respectively. Rule 3 retrieves the names of the patient doctors from the EHR output relation *MyDocs* and matches the names to the patient's Calendar entries to obtain the appointment dates. Rule 4 retrieves the addresses of the patient doctors from the *MyDocs* relation, converts them to x, y coordinates (using the AddrToXY service), and feeds them as markers to the MAP mashlet, which will display each doctor name at the corresponding location. Finally, Rule 5 obtains the patient cell phone number from her EHR, retrieves today's entries from her calendar, and feeds them into the SMS mashlet.

The rules so far use a given predefined set of mashlets. The dynamic nature of our model is illustrated by the following rules where new mashlets are dynamically included in the application. In Rule 6, we see that, whenever the user types in a new illness, the forum mashlets in the UDDI repository are checked, and those determined to be relevant are included in MedickIT (inserted to its Mashlets relation). Rule 7 then feeds (in a manner similar to Rule 2' above) the illness name as input (query) to these forums. The forum mashlets (including their response to the query) are *visible* (see the head of Rule 6) and thus are displayed to the user as part of MedickIT.

Note the use of +:- in Rule 6. It indicates that the new mashlets are *added* to the already existing ones. (This is in contrast to the use of :- in Rules 1-5, where a new content is assigned to the relation at the head of the rule, each time that the rules are triggered). A corresponding rule that removes from the mashup irrelevant mashlets (e.g. of forums relevant only to previously queried illnesses) is defined in an analogous manner. We omit this here.

The example so far captures the modular nature of mashup applications, where several mashlets are (dynamically) glued together to provide richer functionalities. Interestingly, the high-order nature of our model allows for *another level of modularity*, where the glue rules themselves can be defined in a modular manner and, depending on the context, be dynamically added to the application. To see an example, consider a Web Service (mashlet-API) GlueRules$_{bbf}(M_1, M_2, rule)$ that given a mashlet M_1 and one of

its component mashlets M_2, analyzes the semantic relationship between their input and output relations and returns a corresponding set of glue rules. Rule 8 shows how such a service is used to enrich the predefined set of rules of MedickIT.

API. Finally, MedickIT exports a Web-API, i.e., a list of methods (Web Services) that are exposed to be used by other software (possibly other mashlets). One such method, for retrieving doctor appointment dates, is depicted at the bottom of Figure 2. Given a user id, password, and a doctor name, !GetDocAp feeds the user id and password into the input relation of MedickIT, and retrieves, from the computed corresponding output relation !DocAp, the dates of the appointments with the given doctor. Note that the computation of the result involves a combination of capabilities from the component mashlets, as previously described.

Inheritance. We conclude this section with section by considering mashup reuse based on inheritance.

An analysis that we conducted on the extensive collection of mashups found in the Programmableweb.com [30] website, lead us to the understanding that a large number of mashups are similar to each other, in their components and in the functionality they offer to users. For example, at the time of our study, 1669 mashups (39% of all mashups) included maps provided by various vendors (Google, Yahoo!, etc.). Since their characteristics are often standard, it is possible to reuse the composition logic defined for one, for another one. Even if some of the functionalities may not be enabled, the core logic should be reusable.

Motivated by the previous observation, we introduce in our model an inheritance relationship among mashlets. More specifically, mashlet m' inherits from mashlet m if the interface of m' (input/output relations and their respective attributes) is a superset of the interface of m, (and similarly for the Web-API relations). This definition of inheritance implies that mashlet m can be used in any composition that employs an instance of mashlet m'. We note that inheritance can be achieved using explicit language means, e.g., by importing the code of a mashlet and refining it in subclasses. It can also be realized by simply "cloning" the interface of a mashlet.

To be consistent with the example given in Figure 2, Figure 3 shows the inheritance relationship (dotted lines) of the mashlets being placed on the mashup. In this example, the finer rule extracts the entries from the calendar (CAL) and the cell phone number from the Electronic Health Record (EHR) of the patient, and feeds it to the SMS mashlet in order to send them as an SMS reminder to the patient. A coarser rule depicted in Figure 3 takes current image presented in the UI mashlet (CAL's ancestor) and a number from the Data Mashlet (EHR's ancestor) and feeds them into the SMS mashlet. The coarser rule does not exploit all the potential of the more refined mashlet. It may still be sufficient to answer the user needs. This notion will be used to help a user improve her mashup [4], as further explained in the following section.

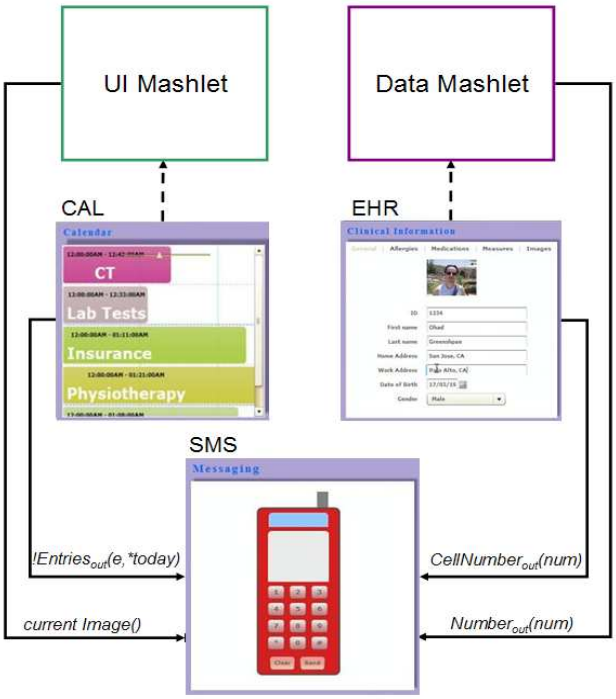

Figure 3: Mashlets Inheritance

5. CONCLUSION

We introduced a data model accompanied by a declarative language for mashup specification. Our approach facilitates and simplifies many issues related to mashups. Currently, there is no working group in W3C [38] promoting the mashup standards. However, some efforts leading development of standards for mashups have begun to arise. OpenSAM [29] is a comprehensive and sophisticated set of standards and documented practices for creating mashups. OpenAjax [28] is an organization of leading vendors, open source projects, and companies using Ajax that are dedicated to the successful adoption of open and interoperable Ajax-based Web technologies. IWidget is standard for mashup creation which is developed by IBM. DataPortability [13] is a set of open standards that enable interoperabilty and portability of data. We see our effort contributing in the future to the development of mashup standards being lead by such organizations.

We briefly consider some next, focusing in particular on some we already started investigating.

Semantics. The focus of the paper is on specifying the composition of mashups *syntactically*. Clearly, the semantic dimension with strong links to the semantic Web [9, 19, 18] is a key aspect. [41] presents a knowledge representation and interface system which combines the rule-based and object-oriented paradigms and designed for various tasks on the Semantic Web. We believe that a formal approach based on a small number of precise concepts as the one presented here, facilitates the integration of semantic Web services in mashups. This claim clearly has to be substantiated.

Security. The import of data and code from possibly unknown parties introduces major security risks. In many application, one has to guarantee some level of security. For instance, in the health care example, one would like to guarantee confidentiality and privacy for patients. Cross-domain communication mechanisms allowing efficient communication across entities without sacrificing security are clearly needed [21]. Although our model does not address the issue directly, the use of formal and precisely defined specification provides a clean basis for installing security. Adaptations of the model, similar to those presented in [23] will be needed in order to provide such capability.

Querying and monitoring. A mashlet may allow Web applications (with appropriate access rights) to query and monitor its data, i.e., its relations. This can be supported by Web Ser-

vices (mashlet-APIs) in the style we have seen in the previous section. Optimization techniques for distributed, stream-based, datalog evaluation [1] can be used here to speed up processing.

Finally, we mention a direction we started investigating.

Autocompletion: Search, adaptation and composition. The mashlet approach is relying on the reuse of existing services or mashlets to provide new needs. So, given a new problem, we would like the support of the system to find off-the-shelf solutions for different facets of the problem, adapt and compose them to obtain a general solution.

For instance, suppose we want to introduce in the health care mashup, `MedickIT`, a new functionality consisting in uploading medical data from home sensors. We would like the system to first help us search in some UDDI-like repository, for mashlets supporting such functionality. In our model, one can look *syntactically* for mashlets matching the needs, based on the names and types of their relations, and their signatures. Semantics should clearly play an essential role: the searchable profiles of mashlets should include semantic annotations (e.g., in RDF [11], RDFS or DAML-S [8]) about their content and behavior.

For composition, we can benefit from the works on Web Service composition, e.g., [32, 35]. Once a mashlet for supporting (some of) the needs has been discovered, it has to be adapted to the environment to be composed with others. Some simple example for this were given in Section 4 (Rules 6-8). The mashlets providing such adaptation will typically involve data restructuring or transformation (e.g., changing a medication name into a generic one) or filtering (e.g., filtering out medication based on adverse drug reaction).

We started developing a recommending system [4] offering (automatic or semiautomatic) support in this task of search-adaptation-composition. The central idea is *autocompletion* that we describe next. Typically, at some stage of the design, a user has already specified a partial mashlet. The idea is to benefit from the experience of other users for completing the task. The system searches for mashlets or services performing some missing functionality and for some "glue patterns" (other mashlets) adapting or composing the new components. In these different tasks, the recommendation engine takes into account the collective wisdom of previous users that have successfully built mashups to rank possible solutions.

6. REFERENCES

[1] S. Abiteboul, Z. Abrams, S. Haar, and T. Milo. Diagnosis of asynchronous discrete event systems: datalog to the rescue! In *PODS '05*, pages 358–367, 2005.

[2] S. Abiteboul, O. Benjelloun, and T. Milo. The active xml project. *To appear in VLDB J.*, 2008.

[3] S. Abiteboul, P. Bourhis, and B. Marinoiu. Axlog and view maintenance over active documents. *submitted*.

[4] S. Abiteboul, O. Greenshpan, T. Milo, and N. Polyzotis. Matchup: Autocompletion for mashups. *submitted*.

[5] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[6] S. Abiteboul and N. Polyzotis. The data ring: Community content sharing. In *CIDR*, pages 154–163, 2007.

[7] 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. In *VLDB '07*, pages 1370–1373, 2007.

[8] A. Ankolekar, M. Burstein, J. R. Hobbs, O. Lassila, D. L. Martin, S. A. McIlraith, S. Narayanan, M. Paolucci, T. Payne, K. Sycara, and H. Zeng. Daml-s: Semantic markup for web services. In *Proceedings of the International Semantic Web Workshop*, 2001.

[9] A. Ankolekar, M. Krötzsch, T. Tran, and D. Vrandečić. The two cultures: Mashing up web 2.0 and the semantic web. *Web Semant.*, 6(1):70–75, 2008.

[10] F. Belleau, M.-A. A. Nolin, N. Tourigny, P. Rigault, and J. Morissette. Bio2rdf: Towards a mashup to build bioinformatics knowledge systems. *Journal of biomedical informatics*, March 2008.

[11] K. S. Candan, H. Liu, and R. Suvarna. Resource description framework: metadata and its applications. *SIGKDD Explor. Newsl.*, 3(1):6–19, 2001.

[12] R. Cattell. *The Object Database Standard: ODMG-93*. Morgan Kaufmann Publishers, 1994.

[13] Dataportability. http://dataportability.org/.

[14] O. de Moor, D. Sereni, P. Avgustinov, and M. Verbaere. Type inference for datalog and its application to query optimisation. In *PODS '08*, pages 291–300, 2008.

[15] R. J. Ennals and M. N. Garofalakis. Mashmaker: mashups for the masses. In *SIGMOD '07*, pages 1116–1118, 2007.

[16] O. Etzion and G. Sharon. Event processing network: Model and implementation. *IBM Systems Journal*, pages 51–59, 2008 (in press).

[17] M. Franklin, A. Halevy, and D. Maier. From databases to dataspaces: a new abstraction for information management. *SIGMOD Rec.*, 34(4):27–33, 2005.

[18] T. Gruber. Collective knowledge systems: Where the social web meets the semantic web. *Web Semant.*, 6(1):4–13, 2008.

[19] T. Heath and E. Motta. Ease of interaction plus ease of integration: Combining web2.0 and the semantic web in a reviewing site. *Web Semant.*, 6(1):76–83, 2008.

[20] W. Inmon. *Building the data warehouse (2nd ed.)*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[21] C. Jackson and H. J. Wang. Subspace: secure cross-domain communication for web mashups. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 611–620, New York, NY, USA, 2007. ACM.

[22] A. Jhingran. Enterprise information mashups: integrating information, simply. In *VLDB '06*, pages 3–4. VLDB Endowment, 2006.

[23] F. D. Keukelaere, S. Bhola, M. Steiner, S. Chari, and S. Yoshihama. Smash: secure component model for cross-domain mashups on unmodified browsers. In *WWW '08*, pages 535–544, 2008.

[24] M. Lenzerini. Data integration: a theoretical perspective. In *PODS '02*, pages 233–246, 2002.

[25] B. Marinoiu, S. Abiteboul, and P. Bourhis. Distributed monitoring of peer-to-peer systems. In *ICDE*, 2008.

[26] N. Milanovic and M. Malek. Current solutions for web service composition. *IEEE Internet Computing*, 8(6):51–59, 2004.

[27] A. Nigam and N. Caswell. Business artifacts: An approach to operational specification. *IBM Syst. J.*, 42(3):428–445, 2003.

[28] Openajax. http://www.openajax.org/.

[29] Opensam. http://opensam.org/.

[30] Programmableweb. http://www.programmable.com/.

[31] D. Quass and J. Widom. On-line warehouse view maintenance. In *SIGMOD '97*, pages 393–404, 1997.

[32] J. Rao, P. Küngas, and M. Matskin. Logic-based web services composition: From service description to process

model. In *Proceedings of the IEEE International Conference on Web Services*, page 446, 2004.

[33] A. Riabov, E. Bouillet, M. Feblowitz, Z. Liu, and A. Ranganathan. Wishful search: interactive composition of data mashups. In J. Huai, R. Chen, H. W. Hon, Y. Liu, W. Y. Ma, A. Tomkins, X. Zhang, J. Huai, R. Chen, H. W. Hon, Y. Liu, W. Y. Ma, A. Tomkins, and X. Zhang, editors, *WWW'08*, pages 775–784. ACM, 2008.

[34] M. Smith. Portals: toward an application framework for interoperability. *Commun. ACM*, 47(10):93–97, 2004.

[35] B. Srivastava and J. Koehler. Web service composition - current solutions and open problems. In *ICAPS*, 2003.

[36] J. Tatemura, A. Sawires, O. Po, S. Chen, K. S. Candan, D. Agrawal, and M. Goveas. Mashup feeds: continuous queries over web services. In *SIGMOD '07*, pages 1128–1130, 2007.

[37] J. Ullman. *Principles of Database and Knowledge Base Systems*. Computer Science Press, 1989.

[38] W3c. http://www.w3.org/.

[39] J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996.

[40] J. Wong and J. I. Hong. Making mashups with marmite: towards end-user programming for the web. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 1435–1444, New York, NY, USA, 2007. ACM Press.

[41] G. Yang, M. Kifer, and C. Zhao. Flora-2: A rule-based knowledge representation and inference infrastructure for the semantic web. In *CoopIS/DOA/ODBASE*, pages 671–688, 2003.