Mashup Model and Verification using Mashup Processing Network

Ehtesham Zahoor, Olivier Perrin, Claude Godart

Mashup Model and Verification using Mashup Processing Network

Ehtesham Zahoor, Olivier Perrin, and Claude Godart

LORIA, INRIA Nancy Grand Est Campus Scientifique
BP 239 54506 Vandoeuvre-lès-Nancy Cedex, France
{zahooreh,operrin,godart}@loria.fr

Abstract. Mashups are defined to be lightweight Web applications aggregating data from different Web services, built using ad-hoc composition and being not concerned with long term stability and robustness. In this paper we present a pattern based approach, called Mashup Processing Network (MPN). The idea is based on Event Processing Network and is supposed to facilitate the creation, modeling and the verification of mashups. MPN provides a view of how different actors interact for the mashup development namely the producer, consumer, mashup processing agent and the communication channels. It also supports modeling transformations and validations of data and offers validation of both functional and non-functional requirements, such as reliable messaging and security, that are key issues within the enterprise context. We have enriched the model with a set of processing operations and categorize them into data composition, transformation and validation categories. These processing operations can be seen as a set of patterns for facilitating the mashup development process. MPN also paves a way for realizing Mashup Oriented Architecture where mashups along with services are used as building blocks for application development.

Key words: Mashups, Service Composition, SOA

1 Introduction

Mashups are defined to be lightweight Web applications aggregating data from different sources such as Web services. In the literature [1, 2, 9, 4], it has been suggested that mashup principles can be the new wave for composing Web services. In this new agile programming paradigm, component services can be assembled with very little or no programming effort, without requiring heavy orchestration techniques such as WSBPEL [7]. Moreover, freely available mashup creator tools ease the process of creating mashups by integrating content from more than one Web services and mashups can be created and published in minutes.

The mashups principles, initially targeted to end-users, can also be used within an enterprise context, called enterprise mashups, to facilitate the process of service composition within an enterprise. Enterprise mashups lack the formalizations and concepts needed to properly describe, model and validate Mashups.

In this paper, we present an approach for facilitating the creation, modeling and the validation of enterprise mashups, introducing the Mashup Processing Network (MPN) approach. The idea is based on Event Processing Network [6, 8]. Mashup Processing Network provides a view of how different actors interact for the mashup development namely the producer, mashup processing agent, consumer and the communication channels. It also illustrates the role of mashup application, which acts both as mashup processing agent and as a data flow consumer.

The Mashup Processing Network approach supports modeling transformations and validations of data, and offers validation of both functional and non-functional requirements, such as reliable messaging, security, and fault-tolerance, that are key issues within the enterprise context [10]. We have also enriched the model with a set of processing operations and categorize them into data composition, transformation and validation categories. These processing operations can be seen as a set of patterns for facilitating the mashup development process. MPN also paves a way for realizing Mashup Oriented Architecture (MOA) where mashups along with services are used as building blocks for application development.

The rest of this paper is organized as follows. Section 2 discusses background and related work, then we will present a sample health-care scenario in Section 3 as the basis for discussing processing operations. We introduce mashup processing network in Section 4. Mashup processing operations can be classified into three categories: composition (Section 5), transformation (Section 6) and validation of data (Section 7). Section 8 models the sample scenario using the mashup model presented earlier, while we discuss implementation details in Section 9. Finally, Section 10 concludes.

2 Related work and motivation

Classically, a Web service is defined as a self contained and modular unit of application logic which permits communication and data transfer between heterogeneous systems in a distributed environment such as Internet. Using Web services as the basic building blocks for application development results in a totally decentralized architecture called Service Oriented Architecture (SOA). In SOA, individual services may need to be composed to form composite services and WSBPEL [7] is the most commonly used method for services composition. WSBPEL, though very powerful and widely used, lacks the primitives to easily handle data validations and transformations. This makes it difficult to build mashups using WSBPEL because of the data inherent nature of mashups, in contrast to the control flow oriented nature of WSBPEL.

Mashups in an Enterprise context aim at enabling the users to dynamically compose and interconnect their own operational environments and processes in a very flexible fashion. An Enterprise mashup architecture will facilitate flexible, useful, and effective user interaction and management with all kind of resources (SOAP or REST-based Web services, Atom or RSS data sources, or other

mashups), and in this sense, such an architecture can be seen as a real collaboration tool. In contrast to the developer centric approach of traditional service composition, data driven mashup programming is a new agile application development paradigm in which knowledge workers, who do not have previous coding skills but do have extensive domain expertise, visually assemble and combine off-the-shelf components or services with both development and runtime rendering capabilities. Then, traditional service composition is an interface level composition and relies on composing operations while mashups are an application level service composition and focus on composing the data from Web services.

There are various freely and commercially available mashup tools for creating mashups using service composition without requiring technical expertise. These include Microsoft Popfly[1], Yahoo Pipes[2], Google Mashup editor[3] and IBM DAMIA[4]. Using these efficient frameworks, it becomes possible to rapidly develop applications and to remove dependence on IT staff. Focus is on simplicity and on creating the mashup application with minimal expertise, effort and time. However, in an Enterprise context, validation is very important, and the mashup tools introduced above lack the advanced customizations to handle validations/synchronization modes and some transformations. Google's Mashup Editor or IBM DAMIA are relatively more powerful and customizable as they are targeted to users with technical expertise, but still lacks handling of advance features such as synchronization modes and validation. Further, their usage within an enterprise context is limited due to the fact that they lack the primitives to manage validations and transformations which can be of critical importance, or primitives to handle security. In this paper, we propose an extension of Enterprise Integration Patterns [5] to take into account the classical problems of data integration, but also the problem of validation of Enterprise mashups.

As mashups are dedicated to data level composition, they introduce the classical data related challenges. Thus, the proposed MPN model will take into account following challenges:

- data heterogeneity and integration: data from different services can be of different format (XML, JSON, data localization, date or currency formats).
- service heterogeneity: as similar to data heterogeneity, services being used can be either RESTful, SOAP based, or using some other protocols.
- streaming data: data provided by different services can be streaming and thus mashup application should be able to handle streaming data.
- data quality: data quality is of high importance in enterprise context. Data returned from services should be valid and well-formed and should be according to constraints imposed by the service consumer.
- data security: some data can be confidential any breach in data security can be critical in the enterprise context. Data exchanged between service providers and consumers should be secure.

[1] http://www.popfly.com/
[2] http://pipes.yahoo.com/pipes/
[3] http://editor.googlemashups.com/editor
[4] http://services.alphaworks.ibm.com/damia/

– data reliability: data stream from service provider should be reliable and in case of non availability, alternatives reliable sources should be provided.

3 Patients Checkout Handling Mashup Example

In order to illustrate enterprise mashups, we consider the example of a Health Care System (HCS) implementing SOA, with services for various redundant operations and systems which use these services. Let us consider a mashup being set up for handling patients checkout within SOA based HCS (Figure 1).

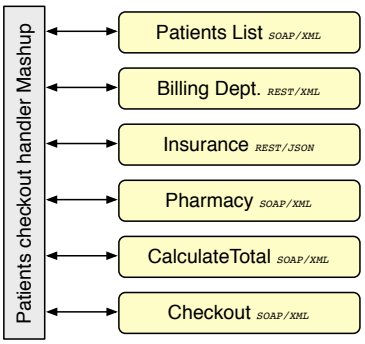

Fig. 1. Patients checkout handling mashup

For the example mashup, there exists a streaming Web service which will provide the social security number for patients checking-out on a particular day. Our mashup application will first fetch this streaming data from patients list Web service (*patientsList*) and for each patient, it will collect outstanding dues from *pharmacy* and billing department (*billingDept*) Web services and will also request the amount to reimbursed by the insurance company Web service (*insuranceCompany*). Our example mashup will then forward this information to *calculateTotal* Web service for calculating the amount to be paid by patient. Finally, it will checkout the patient.

SOA based implementation in HCS can be of great advantage as different systems can be built on top of these services without replicating the functionality. Moreover, community-based collaborations can be fostered, thanks to the introduction of a share, reuse and assembly culture of collaboration. However, it remains important to validate the application obtained by composition, in order to guarantee data reliability, data security and data quality, but also the consistency of the entire mashup.

Before going further, lets us discuss how mashup can ease service composition within the proposed scenario. Implementing the above scenario using traditional service composition techniques such as WSBPEL, will require the end user (domain expert but without technical expertise) to learn the language, and have

some knowledge about Web service concepts such as their interfaces and APIs. On the other hand, a mashup tool will allow end-user to create mashups on the fly without very little or no programming skills. The popularity of mashups thus stems from their ease of usage and flexibility in contrast to the traditional service composition techniques. However, mashup application, when used within an enterprise context such as the example above, requires modeling and validation which may or may not be left to the end user.

4 Modeling Mashups

4.1 Introduction

In our mashup model, a *data flow* – response message from a Web service – is obtained by sending a message to a service (a universal resource identifier, URI). Different types of data flows can be obtained based on different synchronization modes, that can be:

- **synchronous** - synchronous data flow mode requires mashup to suspend its execution after requesting the data from the service providers until it receives the data. Thus, mashup has a data existence dependency on requested data. Synchronous data flow is normally used when the response is of high importance and response time is known to be short.
- **asynchronous** - asynchronous data flow mode requires mashup to continue its execution after requesting the data from the service providers. Thus it has no dependency on requested data and it can later "pull" the data from provider or alternatively data is "pushed" to mashup by service providers, when it is ready. Asynchronous data flow is normally used when the requests take a long time to produce response.
- **streaming data** - mashups can process streaming data. Streaming data transfer can be either service driven or mashup driven [3]. We assume services to be continuously supplying data and mashup can "pull" data from the Web services whenever mashup is ready to process that data. In this case mashup can never be overloaded with data. On the other hand, services can continuously "push" the data to mashup and thus it should have capacity for the storage and the processing of data.

4.2 MPN: Mashup Processing Network

In this section, we will introduce the concept of Mashup Processing Network (MPN) based on Event Processing Network approach as presented in [8]. Event Processing Network is based on Event Driven Architecture (EDA) [11] which is defined to be a pattern promoting the production, detection, consumption and reaction to events. Event producers asynchronously broadcast events as they occur to be later consumed by some receiving system resulting in a totally decoupled architecture. EDA also supports Complex Event Processing (CEP)

[6], which enables event driven applications to react not only to a single event but a complex composition of events happening in different times and contexts.

Our MPN model consists of four components: Dataflow Producer (DP), Dataflow Consumer (DC), Mashup Processing Agent (MPA) and the communication channels to send requests to and receive responses from Web services (see Figure 2).

In MPN, Dataflow Producers, or just producers, are the Web services producing data to be consumed. Communication protocols are used to invoke the Web services and to receive responses. The data flows are the response messages from the Web services. Mashup application acts as both the Mashup Processing Agent (MPA) and as a Consumer by considering mashups as both service composition and as service providers (Mashup-As-Service); they can aggregate data from multiple sources and provide aggregated data to other mashups while acting as a service. Formally, an MPN is a graph $G = (Vertices, Edges)$ where:

$Vertices = DP \cup CC \cup MPA \cup DC$ and $Edges = \{(u, v)|$
$(u \in (DP \vee MPA) \rightarrow v \in CC) \wedge (u \in CC \rightarrow v \in (DC \vee MPA))\}$

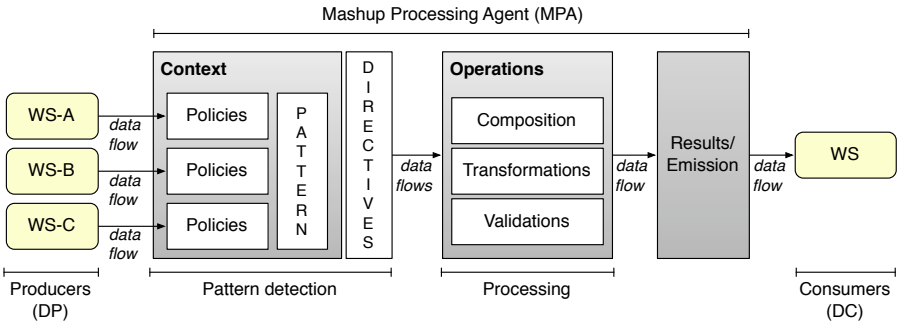

Fig. 2. Mashup processing network: components and stages

Mashup Processing Agent (MPA) is the core component of the model, and is divided into following three stages (see Figure 2):

- *Pattern*[5] *detection* stage is responsible for selecting data flows matching a particular pattern.
- *Processing* stage is responsible for applying processing functions to patterns detected in pattern detection phase and thus resulting in derived data flows.
- *Results/Emission* stage, which is responsible for either emission of derived data flows or storing them as results.

The **Pattern detection** stage is further divided into following components:

[5] Patterns in a MPN are the relations between data flows, not to be confused with design patterns in general.

- *Context*, which specifies relevance of participating data flows and can be either *temporal*, *spatial* or *semantics* based. Temporal context may restrict for example, to consider only the data flows received in some specific amount of time. Spatial context may restrict to consider only data flows received from Web services in specific geographical locations while semantics context can be used to express relevance between participating data flows through a mutual object or entity. If the context for pattern detection is not specified (*None* value), every data flow will be considered as a candidate for pattern detection.
- *Policies* include decisions to either use *first*, *last* or *each* of dataflow in stream for pattern detection. As an example, we can specify policy to use the *last* data flow received from a streaming Web service to ensure that we use the most recent data flow. Policies can also apply constraints to only include the data flows satisfying a predicate on their attributes, for example to use only secure and reliable data flows. Further, policies can be used for specifying expiry time for data flows.
- *Patterns* specify the relationship among data flows complying policies and that are within specified context. The MPN model, as similar to EPN model, neither restricts operators used for pattern detection nor the semantics given to them. Some examples of operators include the operator *any*, meaning that any data flow within context and complying policies results in pattern detection and the operator $all(df_1, df_2...df_n)$ requiring that all data flows $(df_1, df_2...df_n)$ need to exist for pattern detection.
- *Directives* specify the directives for reporting pattern detection to processing stage. For instance, the directive *immediately* specifies to report immediately to the processing stage as the pattern is detected. Other directives include to report the pattern detection at the end of the detection interval or to report pattern detection at specifiable periods.

The **Processing** component is responsible for processing the data flows contributing to pattern detection and it is the core of Mashup application. Processing operations can be of *data composition*, *data transformation* or *data validation*. We will describe these three categories in sections 5, 6, and 7.

The **Results/Emission** component is responsible for either emission of derived data flows or storing them as results. It also supports data validations in the form of post conditions before emitting the data flows.

5 Data composition operations

Mashup is the composition of data flows extracted from Web services. There exist many ways to compose these flows and the *data composition operations* are further divided into *routing* and *aggregate* sub-categories.

5.1 Routing

The *data routing* operations characterizes the manner in which data flows (and their multiple parts) can influence the operation of other aspects of the mashup,

particularly the control flow perspective (each part can be processed in a different way). We identify the following operations.

- **Sequence** - the *sequence* operation is the most basic operation for routing information. Given a service, and a message sent to this service, a data flow is obtained and this data flow is used to call a new service.
- **Content based routing** - mashups can route the input data to different services based on their content. If the data flow content matches a given criteria, it is routed to the output of the mashup, otherwise, it is discarded.
- **Data routing** - similar to the content based routing operation, the data routing operation splits the incoming data to different services. However, in contrast to above mentioned operation, this splitting does not have to be content based. For an example consider that the mashup application decides to split different incoming request to different Web services as a form of load-balancing to avoid over-loading a single service.

For the content based routing and data routing operations listed above, actual split can take one of three forms: **AND-Split**: same input data is routed to **all** services, **OR-Split**: same input data is routed to **at-least one** of the services, and **XOR-Split**: same input data is routed to **exactly one** of the services. We ensure that every *split* operation must be later followed by an *aggregator* operation, which we describe below.

5.2 Aggregator

The *Aggregator* is an operation that receives a stream of data flows and identifies data flows that are correlated. Once a complete set of data flows has been received, the Aggregator collects information from each correlated data flow and publishes a single, aggregated data flow to the output channel for further processing. In order to decide that a set is complete, we introduce the following aggregation schemes: **all** – all the data flows should be considered in order to consider the set complete, **exactly-one** – one the data flows should be considered in order to consider the set complete, **at-least-one** – the first data flow that is received is considered, and **subset** - only a subset of data flows is merged.

We earlier mentioned that any *split* operation must be later followed by an *aggregate* operation, however the converse is not true. We can aggregate different data flows which may not be obtained as an result of a split operation.

Let us review the scenario presented in section 3 and focus on how different data composition operations fit into the example scenario. For the checkout handler mashup (see Figure 3), response from **all** of *billingDept*, *pharmacy* and *insuranceCompany* Web services will be sent to *calculateTotal* service, and a **subset** of responses from source services will be used to checkout the patient. Further, data from *patientsList* service is **routed** to *billingDept*, pharmacy and *insuranceCompany* Web services using **AND-split**. As we discussed earlier, split operation is followed by an **aggregation** operation, and here we later aggregate

these data flows using **aggregate-all** operation. Similarly **content based routing** on data from the source service can be used to decide to which insurance company the claim request should be sent.

6 Data transformation operations

Mashups receive data from different Web services that expose their data in different formats (JSON, XML,...). Thus the mashup may need to *translate* input data to some common format. Mashups may also need to *transform* the input messages even if they have the same binding, for example to normalize or filter the input messages. In order to manage this transformation, we have introduced data transformation operations:

Translator Mashups provide an abstraction layer for dealing with heterogeneous data from different sources. The translator function converts data from one format to some other common format decided by the mashup application.

Wrapper To wrap data inside an envelope that is compliant with the infrastructure. The *wrapper* operation can be used to include encryption facilities or QoS properties.

Data enricher Mashup may enrich the input data to append new data to the data flow. Examples of this kind of operation may include expanding acronyms, including metadata or other similar transformations.

Filter This operation is the dual of data enricher operation. The *filter* operation can be used to remove unwanted data elements from a data flow leaving only necessary items. The *filter* operation can be useful to simplify the structure of the XML document, thanks to a SAX parser for instance.

Normalizer Mashup may require to normalize the input data from different sources to handle localized information such as currency rates, date formats and so on. We can associate a *normalizer* with a *translator* so that the resulting data flows match a common format.

Resequencer The *resequencer* can receive a stream of data flows that may not arrive in order. The role of *resequencer* is twofold; to re-order the data flows and secondly, to re-order message parts within a data flow (for instance, re-ordering the nodes in the XML tree of the response message).

For the sample scenario presented earlier, responses from different services are in different formats and service calls are based on different protocols (see Figure 3). **Translator** is thus needed for data transformation to some common format. Response from different services need to be **filtered** to include only the relevant information, before forwarding data to other services. Finally, data from different services may require to be **normalized** before use: the **normalizer** may be needed to convert currency formats before sending data to *calculateTotal* Web service if the patient is covered by some foreign insurance company.

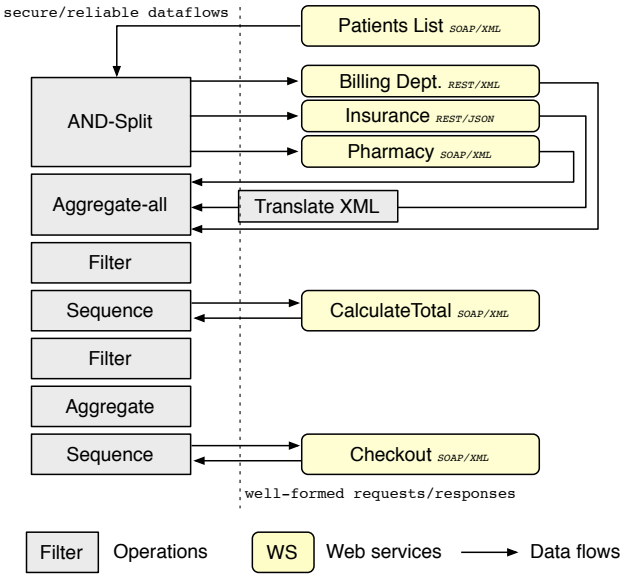

Fig. 3. Mashup operations for the motivating example.

7 Data Validation

Validation for mashups is of high importance. This validation may include checking data accuracy, security checks or some other data level properties. In our model, data validation for mashups can occur at three different levels and consequently it is handled at three different levels in MPN: **pre-conditions** which are constraints imposed by mashup before pattern detection (they can be specified as constraints on data flows in the *policies* for pattern detection), **validations** which are the constraints imposed by the mashup after pattern detection, possibly after applying processing operations such as transformations (they can be specified as validation functions in *processing* phase after pattern detection), and finally **post-conditions** which are the constraints on data being sent by the mashup application and they can be specified as constraints before emitting data flows. In our model, we consider the following validation operations.

Existence Mashups can put constraints on existence of data and it acts as a **pre-condition** for the mashup. Mashups have constraints on existence from either one specific data source or from a set of data sources. Existence constraint is handled by specifying the appropriate policies description in mashup processing network model presented earlier.

Format Constraints on format of input data may also be needed. Although the *translate* processing operation provides abstraction layer for different data formats, it will allow transformation for some specific (already known) data formats. This constraint thus acts as a **pre-condition** for the mashup and is

handled by specifying the appropriate policies description in mashup processing network model presented earlier. As an example, consider now that our mashup application sets constraints input to be only XML based, as it may not be able to handle JSON based data.

Value Mashups can also impose constraints on data values. This constraint may require translation processing operation and thus can be validated after data transformation; it is thus considered as a **validation** constraint.

Quality Mashups can have some restrictions on data quality and in terms of a HCS, it is of critical importance for data to be of high quality. As an example, mashup application may impose constraint to receive only well formed and valid XML by demanding DTD or XML Schema file in response for data validation.

Streaming Data Constraints Different type of data constraints can be imposed by mashup applications for streaming data. These constraints partially depend on the approach used by mashup to handle the streaming data.

On one hand, data availability can be **mashup driven** and it can "pull" the data from streaming data source, whenever data is needed and mashup is ready to process the data. Different types of constraints can be imposed when using this approach, these include **Max idle time** constraint, which require timely reception of data and specifies the maximum amount of time mashup application should wait for data before concluding that data stream has ended.

On the other hand, data availability can be **source driven** and data from streaming data source can be continuously "pushed" to mashup. The constraints using this approach include **Update rate** constraint which specifies the rate at which data should be pushed to mashup by data source. It is important as it can help avoiding data overloading/over-writing at mashup.

Some other constraints are independent of "mashup driven" or "source driven" approaches discussed above. These include **Data Size** constraint, which is the maximum amount of data mashup can process at a time, and data should be pushed or pulled from data source in chunks based on data size specified by mashup. These constraints may also include **Freshness** constraint, which specify the time interval during which data is considered "fresh" or valid.

Finally, these constraints can be imposed on both requests to and responses from Web services, so they can either be in the form of **pre** or **post** condition.

Data Security Constraints can also specify that data from the service providers should be on secure channels or should use some security standards. In case of HCS it is of critical importance as health records are confidential, similar is the case when, for example, using credit cards to make payments. Data Security constraints can be imposed on both requests to and responses from Web services, so they can either be in the form of **pre** or **post** conditions.

Data Reliability Mashups can also impose reliability constraints on data flow that can be in the form of **pre** or **post** conditions.

For the patients checkout handler mashup, application has **data existence pre-condition** on data from *patientsList* Web service (see Figure 3). Let us assume that mashup application can only handle REST/ SOAP based XML and JSON data thus it has **data format pre-condition** on data from all Web services. Then, *patientsList* Web service is streaming and thus different temporal pre-conditions can be imposed as well. Mashup application **pulls** data for each patient from *patientsList* Web service thus it has **data size pre-condition** on input data. Similarly **freshness pre-condition** can be added to specify that the data is valid for a given amount of time. For each contributing data flow, we can also have **pre/post conditions** to only use secure/reliable data flows, and requiring well-formed requests/responses as a form of **data quality** constraint.

8 Mashup Validation

In this section, we will attempt to model and validate patients checkout handler mashup using Mashup Processing Network (MPN) as presented earlier. We will use the processing operations identified in Figure 3 and will model them using the template below.

A1 - Handle response message from patientsList Web service
Detection:
-Context: *None* - every data flow is considered for pattern detection
-Policies: *secure/reliable* data flows - *last* df_patientsList as PL
-Pattern: *Any*
-Directives: *Immediate*
Processing:
Operation : *Validate*
-If *invalid* patient record then *Reject and terminate* processing
Operation : *Split*
-df_billingDept, df_insuranceCompany, df_Pharmacy
Emission/Results
Post Conditions - use *secure/reliable* channels
-*Emit* PL.social_security_no to billingDept service
-*Emit* PL.social_security_no to insuranceCompany service
-*Emit* PL.social_security_no to Pharmacy service

Mashup application first fetches the patient record from the *patientsList* streaming Web service. We have modeled the role of mashup application (acting as MPA) in agent A1. In this template, we first specify the name of agent, and brief description of its role. Below we discuss different phases of the pattern detection:

– first, we specify the *context*, which in this case is *None* meaning that every data flow arriving to mashup application will be considered as a candidate. However, the *context* can be either temporal, spatial or semantic based as discussed earlier. As an example of *temporal* context, we can specify to consider all the data flows arriving in a specific time interval. In the example above, we

can specify the *maximum idle time* streaming data constraint by specifying the time interval in which data flow should be received.
- in the *policies* part of pattern detection we have specified predicates on data flows within specified context. In the example above we have specified preconditions to use secure/reliable data flows, we can also specify the expiry time for data flows as an example of *freshness* streaming data constraint. Further, we have specified to use the *last* data flow we have received from *patientsList* Web service; this will ensure that we use the most recent data flow as the *patientsList* Web service is a streaming Web service.
- then, for the pattern part we specify the relation between data flows that are within specified context and which comply the specified policies. MPN model neither restricts operators used for pattern detection nor the semantics given to them, some examples of operators include *Any* meaning that any data flow within context and complying policies results in pattern detection, it can also be of the form $all(df_1, df_2...df_n)$ requiring that all data flows $(df_1, df_2...df_n)$ need to exist for pattern detection. In the example above, we have specified pattern to be *any*, thus the last data flow we received from *patentsList* Web service (context) and is reliable and secure (policy) will mark pattern detection.
- finally, we specify the directive to be *immediate*, meaning that as soon as we will detect the pattern, we will move on to processing phase.

In the processing phase, we specify the processing operations on data flows detected in the pattern detection phase. In the example above, we first perform a validation operation to check if the message we have received is well-formed and valid and in case of invalid record, processing by this agent will be terminated. Then, we perform a split operation and specify the targets to which the data flow should be split. Finally, in the processing phase we will emit the social_security_no field from *patientsList* service to the targets identified in the split operation. We will then aggregate these splitted data flows later in agent run A2 and we will use the same template to model the MPA functionality. Further, we have modeled the *calculateTotal* Web service invocation in agent run A3 and finally patient checkout is modeled in agent run A4 (see Figure 4).

Now we will briefly discuss how can we validate a mashup using the mashup model presented earlier. We define a mashup to be valid, if constraints for every data flow in mashup are satisfied and dependence between operations and/or there usage conditions are respected. From the above mashup model for checkout handler mashup presented earlier, we can say that a mashup is valid if all the constraints (pre/post conditions and validations) are satisfied i.e secure/reliable channels are used for sending requests and to receive responses. To illustrate the dependence between operations and its application to mashup validation, we take the case of *routing/splitting* and *aggregation* operations. We believe that every *routing/splitting* of data flow should be followed by aggregation somewhere later in mashup flow. For illustrating the usage conditions for an operation, we cannot perform an aggregation of data flows of different format; for every such instance we first need to translate the data flow(s) and then aggregate. Similar

A2 - Process response from billingDept, insuranceCompany, and Pharmacy Web services
Detection:
-Context: *None* - every data flow is considered for pattern detection -Policies: *secure/reliable* data flows, *first* df_billingDept as BD, *first* df_insuranceCompany as IC, *first* df_Pharmacy as PH -Pattern: *all*(BD,IC,PH) -Directives: *Immediate*
Processsing
Operation : *Validate* - If *invalid* BD, IC, PH then *Reject and terminate* processing Operation: *Translate* df_insuranceCompany' := translate(df_insuranceCompany, XML) Operation: *Aggregate* df_calculateTotal := df_billingDept.gross_total ∪ df_Pharmacy .gross_total ∪ df_insuranceCompany'.amount_rembursed
Emission/Results
Post Conditions - use *secure/reliable* channels -Emit df_calculateTotal to calculateTotal Web service

A3 - Process response from calculateTotal Web service
Detection:
-Context: *None* - every data flow is considered for pattern detection -Policies: *secure/reliable* data flows, *first* df_calculateTotal as CT -Pattern: *any* -Directives: *Immediate*
Processsing
Operation : *Validate* - If *invalid* CT then *Reject and terminate* processing Operation: *Aggregate* df_patientCheckout := df_calculateTotal.gross_total ∪ df_patientsList .name ∪ df_patientsList.id_number ∪ df_patientsList.social_security_no ∪ current_date
Emission/Results
Post Conditions - use *secure/reliable* channels -Emit df_patientCheckout to patientCheckout Web service

A4 - Process response from patientCheckout Web service
Detection:
-Context: *None* - every data flow is considered for pattern detection -Policies: *secure/reliable* data flows, *first* df_patientCheckout as PC -Pattern: *any* -Directives: *Immediate*
Processsing
Pattern : Validate - If *invalid* PC then *Reject and terminate* processing
Emission/Results
Post Conditions - use *secure/reliable* channels -Store df_patientCheckout Result

Fig. 4. Patients checkout handler mashup - Model

is the case with possible data flow ordering and filtering to match target Web service input, before performing sequence operation.

9 Implementation

To illustrate how the proposed set of processing operations can be used as patterns for mashup development, we have implemented a Java based server side mashup application for patients checkout scenario presented in section 3 (see Figure 5). We have programmed Web services which return sample data for patients from different systems including billing department, insurance company, pharmacy and others. These Web services are intentionally programmed to support different data formats (XML, JSON) and access protocols (SOAP, REST).

To simulate a streaming based service, we have programmed a SOAP based *patientsList* Web service which provides data for one patient at a time. This web service is pull based and our mashup application will pull the next patient record after checking out current patient. So in the first step, mashup application fetches the patient record from *patientsList* Web service and it then splits this information(using AND-split) to REST/XML based *billingDepartment*, REST/JSON based *insuranceCompany* and SOAP based Pharmacy Web services. The data returned from these services will include billing amount for *billingDepartment* and *Pharmacy* Web services and amount reimbursed information from *InsuranceCompany* Web service. This information will then be aggregated (using aggregate-all) and sent to SOAP based *calculateTotal* Web service, which will return the amount to be paid by the patient. Finally, the information from before mentioned Web services will be merged and will be sent to *patientCheckout* service to checkout the patient. User can then move on to processing next patient record from the *patientsList* Web service.

The patients checkout handler mashup discussed above uses the sample data from Web service written specifically for this purpose. In order to test our implementation on publicly available Web services, we have programmed a search-Mashup which can search user specified query from various Web services, including del.icio.us Web service, Yahoo search services, YouTube Web service and others. The services we have chosen for the mashup application are heterogenous as they support different data formats (XML/JSON) and support different API formats (SOAP/RESTful). In addition some services are being called asynchronously. Users are given option to select Web services and constraint, transform, filter the data returned. Space limitations restrict us to discuss these options in detail.

10 Conclusion

In this paper,we have proposed an approach for building, modeling and validating enterprise mashups. We introduce the concept of Mashup Processing Network (MPN) which illustrates how different actors interact for the development of mashup. The MPN model consists of four components: Dataflow Producer (DP), Dataflow Consumer (DC), Mashup Processing Agent (MPA) and the communication channels to send requests to and receive responses from Web services.

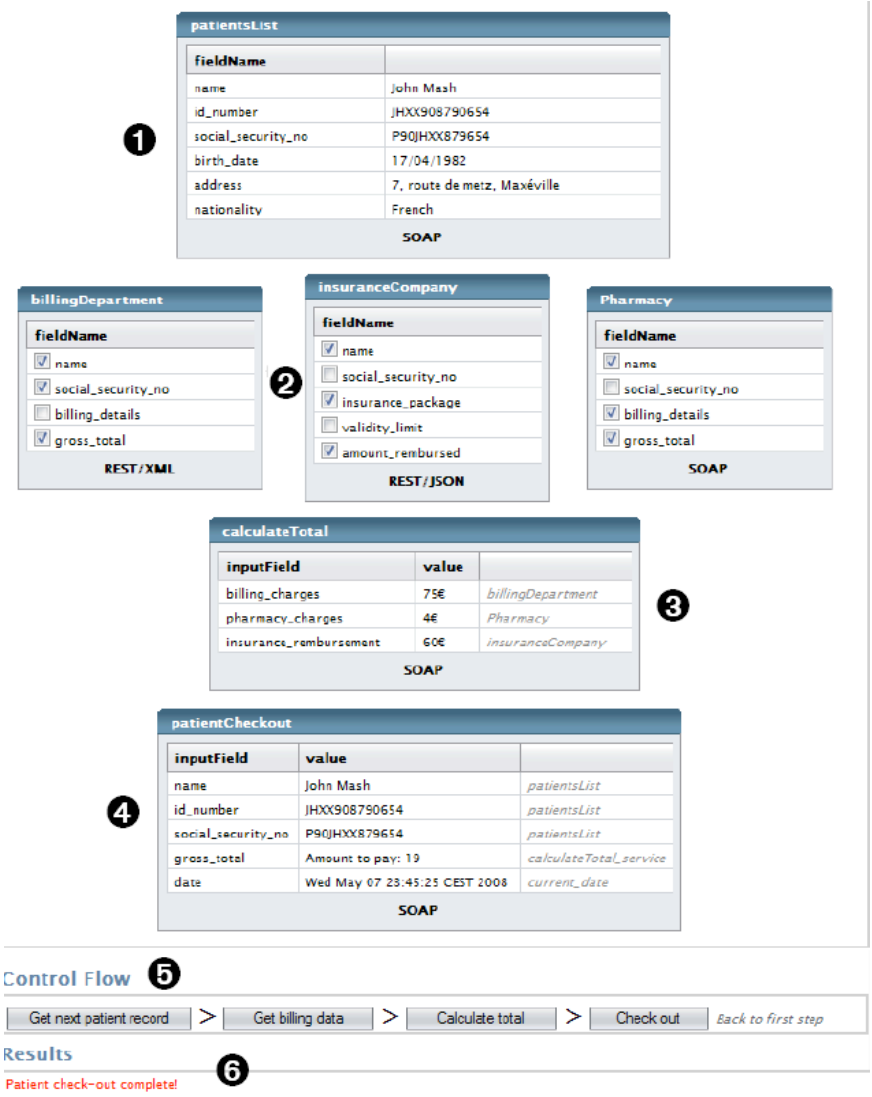

Fig. 5. Patients checkout handler mashup - Implementation

We further enriched this model with a set of processing operations that can be used to process data flows. These operations are divided into three categories: composition operations (routing and aggregation operations), transformation operations (operations to update data flows such as translator, wrapper, filter, data enricher. . .), and validation operations.

Data validation operations are divided into pre/post-conditions and validations. They are necessary in order to express data level properties and they are

handled at three different stages in the MPN: before using data, after processing data, and before emitting data.

We have also presented a sample scenario for the SOA based Health Care System to illustrate these concepts, and we have modeled patient checkout handler mashup using the MPN model introduced in the paper. We have also implemented the sample scenario to discuss how we can use the proposed processing operations as patterns for mashup development and have also tested our implementation on freely "real life" Web services to create a search mashup application, which allows content to be searched using various search Web service including Yahoo search Web service, YouTube Web service and others.

In our future work, we will focus on finding the sufficient conditions to consider a mashup application to be Valid and Well-formed: we will also work on providing a management framework for enterprise mashups and we will work on the security aspects and the runtime management of enterprise mashups.

References

1. X. Liu, W. Sun, Y. Hui, and H. Liang, *Towards Service Composition Based on Mashup*, In: Proc. of the SCW 2007 International Workshop at ICWS'07, Salt Lake City, July 2007.
2. B. Bioernstad, and C. Pautasso, *Let it flow: Building Mashups with Data Processing Pipelines*, In: Proc. of Mashups'07 International Workshop on Web APIs and Services Mashups at ICSOC'07, Vienna, September 2007.
3. B. Bioernstad, C. Pautasso, and G. Alonso, *Control the Flow: How to Safely Compose Streaming Services into Business Processes*, In Proc. of the 2006 IEEE International Conference on Services Computing (SCC 2006), Chicago, September 2006.
4. C. Pautasso, and G. Alonso, *Parallel Computing Patterns for Grid Workflows*, In: Proc. of the HPDC2006 Workshop on Workflows in Support of Large-Scale Science (WORKS06), Paris, France, June 2006.
5. G. Hohpe and B. Woolf, *Enterprise Integration Patterns : Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, Boston, 2004.
6. D. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*, Addison-Wesley Professional, Boston, 2002.
7. OASIS, *Web Services Business Process Execution Language (WSBPEL) 2.0*, 2006.
8. G. Sharon, O. Etzion, *Event-processing network model and implementation*. IBM Systems Journal, 2008.
9. D. Merrill. Mashups, *The new breed of Web app - An introduction to mashups*. http://www-128.ibm.com/developerworks/xml/library/x-mashups.html
10. D. Hinchcliffe, *The quest for enterprise mashup tools*. http://blogs.zdnet.com/Hinchcliffe/?p=59
11. G Hohpe, *Programming Without a Call Stack - Event-driven Architectures*. http://www.eaipatterns.com/docs/EDA.pdf