A Step-By-Step Debugging Technique To Facilitate Mashup Development and Maintenance

Waldemar Hummer, Philipp Leitner, and Schahram Dustdar
Distributed Systems Group
Vienna University of Technology
Vienna, Austria
<lastname>@infosys.tuwien.ac.at

ABSTRACT

With Web mashups, data from different Web documents and services are "mashed" together to create a new functionality. The mashup developer usually has a clear vision of the desired output, i.e., the resulting Web page to present to the end user. Complex mashups require multiple processing steps, and become hard to debug if the delivered result is not as expected. In this paper we propose an approach that supports step-by-step debugging for declarative development of data mashups. A dependency graph is constructed from the mashup definition, and developers are able to define breakpoints to inspect a snapshot of the running mashup execution. A Web 2.0 application provides direct visual feedback of the intermediate results in each processing step. On top of that, it is possible to specify expected and unexpected result elements. If the result does not comply with the specifications, the platform helps to identify the processing step which caused the error.

Categories and Subject Descriptors

H.3.5 [**Online Information Services**]: Web-based Services; H.3.3 [**Information Search and Retrieval**]: Retrieval models

General Terms

Mashup Debugging, Step-By-Step Mashup Development

Keywords

Mashup, Debugging, Web Data Aggregation

1. INTRODUCTION

In recent years, the emerging field of Web mashups [2] has attracted both industry and research. Mashups take advantage of the rapidly growing number of documents and services available across the Web, and combine heterogeneous data from different sources to create a new functionality. A number of mashup platforms have been developed by industry giants such as Yahoo[1], IBM[2], Google[3] or Intel[4].

An often cited, informal definition describes a mashup as *"an application that combines data, either through APIs or other sources, into a single integrated user experience"* [24]. In the same paragraph the paper states that the complexity of mashup programming is *"a barrier"* and prevents many users from actively developing mashups. This issue has been known to the community for some time and is now reflected in a variety of approaches that abstract from the technical complexity via Domain-Specific Languages (e.g., [15], [7]), or aim at making mashup development accessible to Web users with little or no programming skills (e.g., [23], [13]). Most of the proposed solutions have in common that they seek for a trade-off between a high level of abstraction (thereby reducing the programming skill requirements) and flexibility (increasing the range of mashups that can be implemented) [10]. In any case, arbitrarily querying, combining and transforming data remains a complex task, expressed in other words: *"The biggest problem of mashup is the data"* [14].

Some of the current and future challenges in the domain of mashup construction are mentioned in [12]. Part of these challenges are related to user experience (cataloguing, sharing and reusing) or cross-cutting concerns (security and identity, trust certificates). Here, we focus on the part that is concerned with technological challenges of developing and maintaining mashups. The paper names the challenge of data (non-)integrity, owing to the fact that mashups are defined on top of existing (mostly third-party) data sources. The mashup integrity and information quality needs to be ensured for all sources and for the mashup as a whole [5]. Also, if the underlying data or conditions change, it often occurs that the mashup cannot automatically adapt to the new environment, i.e., the entire mashup execution fails or the mashup delivers an unexpected result. Such situations require manual debugging by the mashup developer. Depending on the size and complexity of the mashup definition, the debugging process may be a tedious task if not appropriately supported by the development platform.

Although debugging is a well-established principle in software engineering, debugging is tedious in many mashup platforms (e.g., execute mashup, check output, refine mashup definition and execute again) and few existing works explic-

MASHUPS '10 Ayia Napa, Cyprus

[1] http://pipes.yahoo.com/pipes/
[2] http://ibm.com/software/info/mashup-center/
[3] http://code.google.com/gme/, discontinued in 2009
[4] http://mashmaker.intel.com/web/

itly address debugging of mashups. The survey carried out in [10] indicates that software engineering techniques are in general not well-supported in mashup platforms. For instance, only two out of six studied mashup makers provide some form of version control, and only three of the mashup makers support rudimentary debugging via console output messages. As an example that indicates the importance of the topic from a user perspective, a participant in a survey carried out among young students [17] critizised that the used mashup tool *"lacks support for interactive debugging"*.

In this paper we present a step-by-step debugging approach that facilitates development and maintenance of Web mashups. The presented concepts and implementation are based on WS-Aggregation [11], a platform for Web data aggregation which supports multi-step querying of heterogeneous data sources. Mashups are defined as a set of data source requests with data dependencies between them. The mashup definition is transformed into a graph representation comprising the individual process steps and the dependencies. Based on the mashup graph, developers can define breakpoints to pause the mashup execution at a specific point for inspection of the mashup state and the intermediate result. Furthermore, a developer may specify assertions about which elements should or should not occur in the mashup result, and the platform suggests at which point the mashup definition is likely flawed. A graphical debugging environment executes the mashup and indicates sources of errors (e.g., failed assertion, fault response from data Web service). To sum up the advantages of the approach, we demonstrate how mashup debugging is utilized for top-down mashup development, where the developer first specifies a template for the mashup output and then step by step defines the underlying data sources.

The remainder of this paper is structured as follows. Related work in the area of mashup debugging is discussed in Section 2. In Section 3 we present an illustrative scenario. Section 4 provides background information about WS-Aggregation and how it is used to realize the mashup scenario. The main part of the paper is Section 5, which highlights the capabilities and advantages of tool-aided mashup debugging. Our prototype implementation is described in Section 6. Section 7 concludes the paper with an outlook for future work.

2. RELATED WORK

In [1], mashup processes are depicted in a graph representation, which models both the control flow (sequential ordering) and the data flow between the blocks (functional, independent units). A concrete execution of the mashup process, which follows one of the possible process paths, is denoted as execution flow. The framework supports an "undo" feature, which allows to pause the execution to inspect the state of a running process. The execution path is used to resume the mashup process starting from the most recently processed block.

The authors of [3] present the Mashup Services System (MSS), a platform for mashup development and management. MSS relies on semantic descriptions of services and provides the Mashup Service Query Language (MSQL) that allows for automatic mashup generation. The presented matchmaking algorithm can detect incompatible messages or communication protocols. The notification about an incompatible configuration serves as the starting point for debugging the mashup query. A main difference to our work is that MSS requires semantic models for all participating services, which is not necessary in our approach. This is a major advantage, as our solution is directly applicable to real-world data services, most of which are not semantically annotated.

MoSaiC [19] presents a conceptual model for document services mashups. Its service taxonomy distinguishes the two main types of content source services and content transformation services. The data provided by a source service or the algorithm employed by a transformation service may change over time, influencing the proper behavior of the mashup. This aspect also plays a key role in mashup debugging. Mashups defined in MoSaiC have an interface to receive document service events, which contain notifications of changes made to a service. MoSaiC is related to our work, although it targets the slightly different aspect of runtime adaptation and employs an event-condition-action ruleset to dynamically react on changes in the mashup environment.

The authors of [4] conducted an experiment to study the debugging aspect of end-user mashup programming. Ten participants were asked to implement a given mashup using Microsoft Popfly[5]. The study procedure follows the *think-aloud* method, in which the participants pronounce their thoughts and impressions. Different debugging strategies were evaluated (e.g., testing, code inspection, dataflow reconstruction). One of the main observed problems is related to the dataflow strategy. Although analyzing the dataflow was positively correlated with success, users had difficulties applying it. Code inspection also appeared to be ineffective; the authors argue that this is because only a portion of the "code" (parameter settings) was visible at a time, and this may have led to cognitive overload for some users.

JOpera [18] is an integrated development environment for (data-centric) Web service orchestrations and Web mashups. The tool, which is based on the Eclipse platform, provides both a control flow view of the mashup process and a view of the data flow between the process steps. During execution the workflow engine provides basic monitoring and debugging information. Our approach goes beyond that aspect and allows for explicit debugging user input, e.g., in the form of breakpoints and assertion statements.

The Damia platform [20] is a lightweight enterprise data integration service. Its browser-based user interface supports graphical development and debugging of data mashups. Damia visualizes the mashup operators (e.g., filter, transform, merge) as drag and drop boxes. Whereas the mashup techniques greatly differ, a similarity to our approach is that users can select a single operator to see a preview of its effect on the result output.

Lixto [9] is a platform for scalable Web data extraction processes, also referred to as information pipelines. The pipeline steps are to 1) aquire and 2) integrate the required content from data sources, and to 3) transform and deliver the result to the end user. The logic-based language *Elog* is the basis for a visual specification framework to express data extraction processes. The Lixto suite allows visual debugging of the information pipeline. Lixto is different from our work since it focuses on analyzing the semantics of documents, e.g., providing functions to determine whether some text contains a date or currency, whereas we consider only the syntax of mashup data sources.

[5]http://www.popfly.com/, discontinued in 2009

3. SCENARIO

We base our contribution on an illustrative scenario of a US citizen who plans a tourist trip to Austria, Europe. Consider the example mashup depicted in Figure 1.

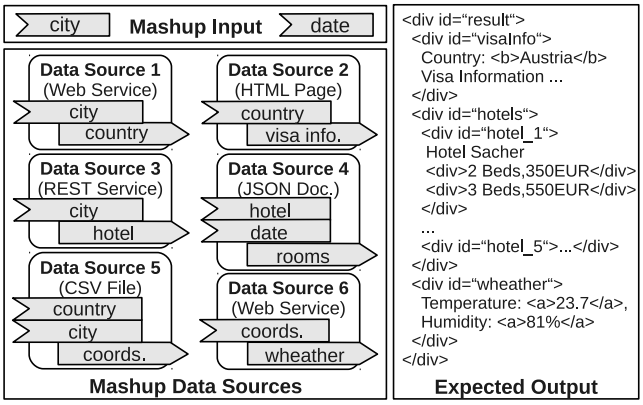

Figure 1: Tourist Info Mashup Scenario

As the input to the mashup the user provides the name of the target city (`city`) and the date on which the trip should take place (`date`). The mashup definition contains references to six different data sources (DS). These sources are a heterogeneous mix of Web services and documents. DSs 1 and 6 are Web services, DS 3 is a RESTful service, and DSs 2/4/5 deliver documents in the format of HTML, JSON (JavaScript Object Notation) and CSV (Comma-Separated Values), respectively. Each data source in the figure contains labels that signify the required input and produced output, e.g., Data Source 3 returns hotels (`hotel`) for a specified city name (`city`). The expected output of the mashup is an HTML document, which presents the tourist information (visa information, available rooms, wheather forecast) as a formatted website. As a constraint we define that the available rooms should be displayed for no more than 5 hotels, hence the hotel `div` elements in the expected output are named `hotel_1` to `hotel_5`.

The scenario mashup contains dependencies between the data sources. Data from two of the sources (DS 1 and 3) can be immediately requested using the mashup input `city`, but the remaining DSs depend on the output of some other DS. For instance, DS 2 can be queried as soon as the `country` is available, which gets returned from DS 1. In our implementation, which will be discussed in Section 4, the mashup developer specifies which element is required for a request and the WS-Aggregation platform takes care of resolving these data dependencies. We will discuss suitable debugging techniques which support the developer in achieving the expected result.

4. MASHUPS WITH WS-AGGREGATION

The scenario mashup is implemented using the WS-Aggregation platform, which we discuss in the following. WS-Aggregation provides a generic framework for distributed aggregation of Web services data. It incorporates the specialized query language WAQL (Web service Aggregation Query Language), which builds on XQuery [22] and provides additional features that are tailored to Web data aggregation. WAQL can be used to declaratively specify aggregation queries. WS-Aggregation is suited to function as a mashup platform, because in essence a mashup aggregates and combines data from different sources.

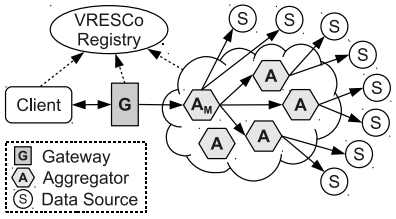

Figure 2: WS-Aggregation Architecture

The high-level architecture of WS-Aggregation is depicted in Figure 2. A number of aggregator nodes (`A`) are deployed to handle incoming requests and request data from data sources (`S`). Client requests are routed via a gateway (`G`), which selects a master aggregator ($\texttt{A}_\texttt{M}$). The framework transparently performs query distribution according to configurable strategies, i.e., the master aggregator delegates part of the request execution to partner aggregators. The system participants are stored in a *VRESCo* [16] service registry instance. The client discovers the gateway endpoint in the registry, the gateway queries for aggregator nodes and the aggregators dynamically bind to the data sources which match the user request.

WAQL distinguishes between three different query types: 1) *preparation* queries are used to transform data immediately after receiving them from a data source, 2) the single (optional) *intermediate* query is applied when data is passed between aggregator nodes, and 3) the single *final* query prepares the output for presentation to the user. As query distribution and inter-aggregator communication are out of the scope of this paper, we do not use intermediate queries here.

The realization of the scenario using WS-Aggregation is depicted in Table 1. The table lists for each data source (DS) 1) the numeric identifier, 2) the request expressed in WAQL, 3) an exemplary DS response, 4) the according WAQL preparation query, and 5) the prepared DS result which is the output of applying the preparation query to the example DS response. Note that only columns 3 and 4 are specified by the developer, whereas columns 3 and 5 (printed in italic font) contain runtime examples. The bottom of the table contains the finalization query (`F`), which gets applied to the concatenation of all prepared DS results and creates the final mashup output (compare Figure 1). The queries are based on XQuery, but the example contains some WAQL-specific constructs, which are briefly discussed in the following.

4.1 Non-XML Data Sources

The scenario mashup contains heterogeneous data sources, which use both different messaging protocols and data formats. WS-Aggregation builds on XML technologies such as XPath and XQuery, and seeks to integrate non-XML data sources to create a unified view on data. This is achieved using predefined conversion functions, two of which are used in the scenario mashup.

In the example, `jsonToXML` transforms the JSON document into an equivalent XML representation. The implementation follows the *BadgerFish* convention[6], which de-

[6]See http://www.bramstein.com/projects/xsltjson/

	WAQL DS Request	DS Response (Example)	WAQL Preparation Query	Prepared DS Result (Example)
1	`<getCountry><city>${city}</city></getCountry>`	*<country><name>Austria</name></country>*	`<country>{country/name/text()}</country>`	*<country>Austria</country>*
2	`/getVisaInfo?c=${//country}`	*.. ... ..*	`<visaInfo>{//div[@id='visas']/node()}</visaInfo>`	*<visaInfo>..</visaInfo>*
3	`<getHotels><city>${city}</city></getHotels>`	*..<hotel><name>Sacher</name><stars>5</stars>...</hotel>..*	`<hotelNames>{for $h in //hotel[position()<6] return <hotelName>{$h/name/text()}</hotelName>}</hotelNames>`	*<hotelNames><hotelName>Sacher</hotelName>...</hotelNames>*
4	`/getRooms?h=$(${//hotelName/text()})&d=${date}`	*{"hotel":{"name":{"$":"Sacher"},"rooms":{"room":[{"beds":{"$":"2"},"price":{"$":"350"}},{..},..]}}}*	`jsonToXML(/)`	*<hotel><name>Sacher</name><rooms><room><beds>2</beds><price>350</price></room>..</rooms></hotel>*
5	`/cityCoords?c=${//country}`	*Innsbruck,47N,11E Salzburg,47N,13E Wien,48N,16E ...*	`let $c=csvToXML(/)//row[col[1]='${city}']/col return <coords><lat>{$c[2]/text()}</lat><long>{$c[3]/text()}</long></coords>`	*<coords><lat>48N</lat><long>16E</long></coords>*
6	`<getWheather>${//coords}</getWheather>`	*<wheather temperature="23" date=".." humidity="81%"/>..*	-	*<wheather temperature="23" date=".." humidity="81%"/>..*

F
```
<div id="result">
  <div id="visaInfo">Country: <b>{//country/text()}</b> {//visaInfo/node()}</div>
  <div id="hotels">{ let $hotels:=//hotelName/text()for $h in $hotels return <div id="hotel_{index-of($hotels,$h)}">Hotel {$h}
     { for $r in //hotel[name/text()=$h]//room return <div>{$r/beds/text()} Beds, {$r/price/text()}EUR</div> } </div> }</div>
  <div id="wheather">Temperature: <a>{//wheather/@temperature}</a>,Humidity: <a>{//wheather/@humidity}</a></div>
</div>
```

Table 1: Scenario Implemented Using WS-Aggregation

fines a set of conversion rules to transform JSON strings into XML documents and vice versa. Essentially, the JSON notation defines name-value pairs, the names become the element names in XML and the values are either primitive or complex types. Element sequences are serialized as arrays in JSON (encoded in square brackets `'[..]'`), and text nodes are represented as a pair with name `'$'` and the text content as value. The argument of the `jsonToXML` function is the XPath root element selector (`'/'`). In order for this selector to evaluate properly in every case, each non-XML data source response is first wrapped in a temporary XML root element, before the preparation query gets applied.

Similarly, `csvToXML` creates an XML representation of the CSV file returned from DS 5. Again, the function argument is the XPath root element selector, which returns the CSV file wrapped in a temporary XML root element. Each line of the CSV document gets represented as a `row` element in the resulting XML markup and the contained values (separated by a comma or a similar delimiter) are wrapped in `col` elements.

4.2 Data Dependencies

The data dependency indicator (syntax: `${...}`) mandates that some data that matches the given XPath expression in the brackets has to be extracted from the result of another DS. The DS result denotes the result of applying the preparation query to the response of a DS. Note that the example uses the simplest version of data dependency, which specifies only *what* is required (in the form of an XPath) and not *where* (from which other data source) it should be extracted from. For instance, the request of DS 1 requires a `city` element, which is provided by the user input, and DS 2 requires a `country` element, which becomes available after retrieving the result of DS 1. In this simple case, the information which data source fulfils which data dependency is not available at design time, but can only be evaluated at runtime after having executed the mashup at least once. Apart from that, the data dependency provider can be specified explicitly, e.g., `$1{//country}` is used to receive the country from data source 1.

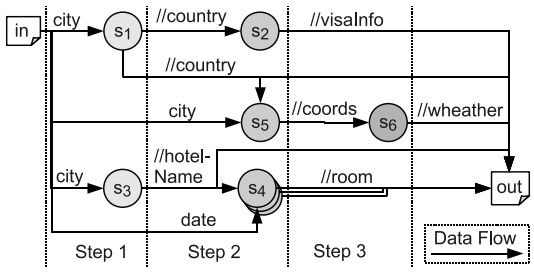

Figure 3: Dependencies and Processing Steps

A graph of the scenario's data dependencies is illustrated in Figure 3. The circles represent requests to data sources, and an arrow between circles signifies a data dependency, or the data flow, between two data source requests. Two requests (s_1 and s_3) can be immediately executed, since their dependencies are fulfilled by the mashup input (`in`). At this point, the newly received data is used to resolve the remaining dependencies. The dependency XPath expressions are matched against this new data and WS-Aggregation determines that the XPaths `//hotelName` and `//country` can now be satisfied. This enables the execution of requests s_2, s_4 and s_5. These requests are executed in a second step, and the remaining request s_6 in step 3. WS-Aggregation automatically constructs the dependency graph and performs the

data aggregation to create the final mashup output (`out`). An error message is issued in case some dependency cannot be fulfilled (see Section 5 below).

4.3 Generated Inputs

The node for data source s_4 is depicted as a stack of circles in Figure 3 because it uses *generated inputs*. Generated inputs are a convenience feature in WS-Aggregation to specify a request template along with value lists, whose elements are to be inserted into the template. Value lists are expressed with `$(..)` (note the round brackets as opposed to the curly brackets used for data dependencies), and the list contains an XQuery expression which, upon execution, returns the list items.

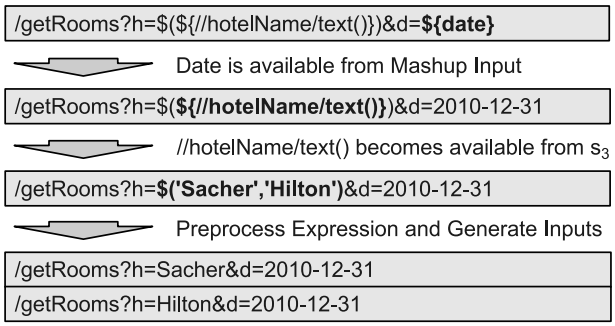

Figure 4: Data Dependency and Generated Inputs

Figure 4 illustrates an example of how the WAQL request for s_4 is processed at runtime. The date can be filled in from the mashup input. After retrieving the result from s_3, the data dependency `//hotelName/text()` can be satisfied (returning a list of hotel name strings) and the matching node is directly inserted into the source of the WAQL request. Since no more unresolved dependencies for s_4 exist, WS-Aggregation preprocesses the expression and generates the two final inputs. Both inputs are sent as requests to the data source, and the returned results are concatenated and appended to the output.

5. STEP-BY-STEP MASHUP DEBUGGING

Debugging is considered a core domain of software engineering and comprises a variety of methods and techniques. The comprehensive elaboration in [25] mentions different goals and purposes, three of which are of special interest for mashup debugging: *Asserting Expectations*, *Detecting Anomalies* and *Tracking Origins*. In the following we discuss how WS-Aggregation supports mashup developers in achieving the three goals, leading to the ultimate goal of *Fixing the Defect* [25]. Essentially, a defect in a mashup definition manifests itself in an unexpected output, i.e., at a certain stage in the mashup execution some data is mistakenly dropped, falsely added or improperly transformed.

5.1 Asserting Expectations

The mashup developer usually has a clear vision of the expected mashup output. In other words, the developer can express assertions concerning the mashup result. In WS-Aggregation, XPath and XQuery are used for that purpose. The specified XPath expressions define which XML nodes must and must not be present in the result document. If necessary, more complex assertion tests can be written with XQuery, which allows arbitrary selection and iteration over the nodes in the result document. All expressions must evaluate to `true` when applied to the mashup output.

#	Assertion Expression	DS	R/T	B/A
1	`count(//div[@id='hotels']/div)<=5`	E	T	A
2	`//div[@id='visaInfo']/b/text()`	E	T	A
3	`//div[@id='wheather'][a[1]/text()][a[2]/text()]`	E	T	A
4	`every $h in //hotelName/text() satisfies //hotel[name=$h]/rooms`	E	T	B
5	`string-length(//country/name)>0`	1	R	B
6	`every $r in //row satisfies count($r/col)>=3`	4	R	A

Table 2: Assertions for Scenario Mashup

Example assertions for the scenario mashup are printed in Table 2. The values in column `DS` indicate at which point in the mashup execution, i.e., after the request to which data source(s), the assertion should be applied; the special value `E` (end) means that the assertion is evaluated at the end of the mashup execution. Column `R/T` determines whether the assertion expression is applied to the response (`R`) of the data soure in `DS` or to the total result (`T`), i.e., the concatenation of all data source results collected so far. Note that the combination of `E` for `DS` and `R` for `R/T` cannot be used, because its semantics are undefined. The last column (`B/A`) specifies whether the assertion should be checked before (`B`) or after (`A`) applying the query. The respective query is either a preparation query, in case `R/T` equals `R` and `DS` references the numeric identifier of a data source, or the finalization query if `DS` equals `E`.

Assertion 1 expresses that the room availability in the HTML end result (`E`) after (`A`) applying the finalization query may be displayed for at most 5 hotels. Assertion 2 ensures that the country name is displayed in the visa information `div` element of the final document. The presence of the wheather data in the result is checked using assertion 3. Note that every assertion expression gets wrapped in an XPath `boolean(..)` function before evaluation and that the boolean value of a non-empty node sequence is `true`. Assertion 4 uses XQuery univeral quantification and ascertains that for each hotel name a `rooms` element needs to exist before (`B`) executing the finalization query. Note that this element may be empty (if no hotel rooms are available) but it is always present in the result from data source 4. Assertion 5 checks whether a non-empty country name is contained in the response (`R`) from DS 1 before (`B`) applying the preparation query. The quantification in assertion 6 mandates that each row (`row`) in the converted CSV document contains at least 3 columns (`col`).

5.2 Detecting Anomalies

We distinguish between different types of anomalies that can occur in a mashup environment, both at design time and at run time (see Table 3). The most obvious anomaly is when an involved data source or service itself indicates that an error occured. For instance, in the case of Web services, SOAP Faults are a common means to communicate application level faults to clients. Such faults are detected by the platform and displayed in red color in the graphical debugging environment (see Section 6). Another possible anomaly are WAQL query preprocessing errors. Take for instance the request for data source 3 in the scenario (see Figure 4). The nested language constructs (data dependencies, generated

inputs) are resolved in several processing steps. For easier debugging of erroneous queries, the platform allows inspection of the query before and after (pre-)processing.

Data dependencies are obviously also prone to errors and anomalies. For instance, a data dependency becomes unresolvable during execution if 1) the required data cannot be extracted from any data source result obtained so far, and 2) no further (independent) requests can be issued to receive new data. The required action is to add an (independent) data source, which can deliver the required data. The opposite anomaly is an ambiguous dependency, i.e., a situation in which a data dependency can be fulfilled by two or more data source results simultaneously. Solving this situation requires the developer to refactor the WAQL preparation queries and to transform the conflicting data source results in order to guarantee an unambiguous dependency resolution. Finally, a circular dependency (involving two or more data source requests) can arise if data dependencies target an explicit data source. In WAQL, the syntax `$s{x}` expresses that data matching the XPath `x` should be provided by the result of data source `s`, e.g., `$1{//country}` extracts the country from data source 1. These data dependencies with explicit provider information are analyzed and checked for circles statically (at design time).

Anomaly	Required Action
Fault Response from DS	Correct Request / Change Endpoint
Preprocessing Error	Fix Invalid WAQL Query
Unresolvable Dependency	Add Data Source
Ambiguous Dependency	Refactor WAQL Preparation Query
Circular Dependency	Revise Explicit Dependencies
Failed Assertion (Before Q.)	Correct Request / Change Endpoint
Failed Assertion (After Q.)	Correct WAQL Query

Table 3: Anomalies and Corrective Actions

Another category of anomalies are failed assertions. In the course of mashup debugging, WS-Aggregation executes the mashup and logs all individual requests, intermediate states and data flows. This allows to evaluate the defined assertion expressions against the traces of the mashup at runtime. If all assertions pass, i.e., all assertion expressions evaluate to `true`, the developer's expectations are met and the mashup is said to function correctly. However, the validity and significance of a successful test run highly depends on the number and quality of the specified assertions. In any case, assertions can help to ensure consistency of the mashup over time, since the data sources may unexpectedly change and critical changes can automatically be detected by evaluating the corresponding assertions.

5.3 Tracking Origins

Each node of the data dependency view depicted in Figure 3 is an abstract placeholder for a number of activities carried out by the WS-Aggregation platform. To track the origins of anomalies in a mashup, it is important to identify the potential points of failure in the system. On the one hand, the origins of anomalies may reside with a (third-party) target data source, e.g., the service is irresponsive or raises a fault. In this case the patform indicates that the faulty data source and the endpoint needs to be replaced.

Figure 5 illustrates the details of the mashup execution and indicates the potential points of failure. First, preprocessing of the WAQL request (`PP`) takes place, before the request is sent (`SR`) to the target data source. All matching assertions are then checked (`CA`), both before and after applying the preparation query (`PQ`) to the data source response. Finally, the newly obtained result is matched against all other requests in the mashup to update unresolved data dependencies (`UD`). In the case of s_4 the additional action of generating inputs (`GI`) is performed. The following actions (`SR,CA,PQ,UD`) are depicted as a stack of actions in Figure 5, because each action is executed for each of the generated inputs. The output is constructed by concatenating all individual data source results, and applying the finalization query to this document. Before and after this query, all assertions marked `E` (check at the end) and `T` (check for total result) are checked (`CA`).

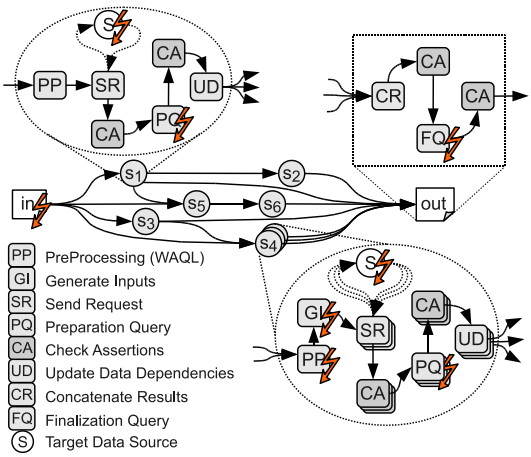

Figure 5: Potential Points of Failure

The lightning bolts in Figure 5 indicate potential points of failure, i.e., critical points in the mashup execution where mashup definition, user input and target data services may affect the mashup and lead to an unexpected result. It is therefore vital for the developer to gain an insight into how these critical steps affect the mashup behavior and output.

5.4 Debugging for Top-Down Development

The Web data aggregation concept employed by WS-Aggregation supports both bottom-up and top-down mashup development. With bottom-up, first the individual data source requests are constructed, then a preparation query for each of the responses, and as the last step the finalization query. This approach is straight-forward, but it requires the developer to keep a clear vision of the outcome during the successive refinement of the mashup. With top-down, the development process starts with defining the finalization query, which specifies the format of the result output and the required data to be extracted from the (yet to be defined) data sources. Using this approach has the advantage that the expected output can be defined up front. However, initially the data dependencies can of course not be satisfied, since no data sources have been defined. In the course of the mashup improvement, one data dependency after the other is resolved by adding new, matching data sources. This procedure is comparable to Test-Driven Development (TDD) [8], where a test case that deliberately fails is used to implement a new functionality such that the test passes. It is quite obvious that this methodology requires proper support by the mashup platform.

Imagine we follow a top-down approach for step-by-step

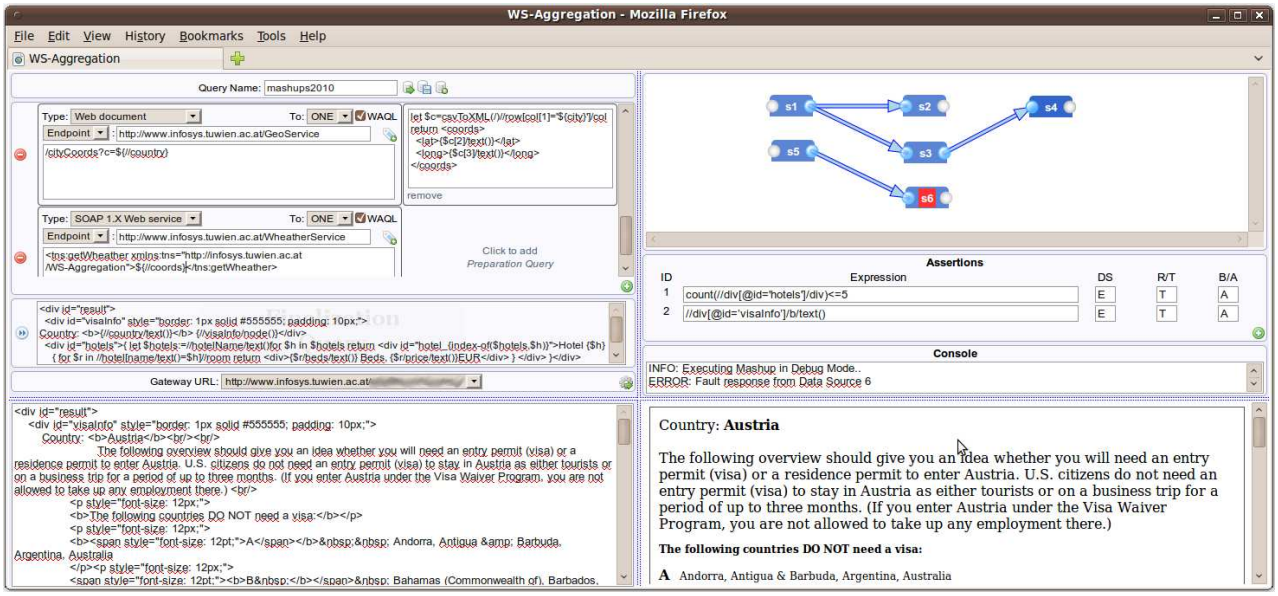

Figure 6: Graphical User Interface

development of the tourist information mashup scenario. The single development steps along with the data sources, assertions and unresolved dependencies in each step are summarized in Table 4. We first start with constructing the finalization query, which is printed in Table 1). We then define the assertions listed in Table 2. Of course, initially only the assertions 1 to 4 are known because they target the end of the mashup execution (`E`). The data source specific assertions 5 and 6 are added later when the according data source requests are defined.

#	Defined DSs and Requests	Assertions Defined	Assertions Violated	Unresolved Dependencies
1	-	1,2,3,4	1,2,3	-
2	3	1,2,3,4	2,3,4	-
3	2,3	1,2,3,4	2,3,4	`${//country}`
4	1,2,3	1,2,3,4,5	3,4	-
5	1,2,3,4	1,2,3,4,5,6	3	-
6	1,2,3,4,6	1,2,3,4,5,6	3	`${//coords}`
7	1,2,3,4,5,6	1,2,3,4,5,6	-	-

Table 4: Scenario Mashup Top-Down Development

When executing the mashup in debugging mode, the platform reports which assertions are violated and which data dependencies are cannot be resolved. Table 4 shows an example of how this debugging information is used to refine the mashup definition step by step. In each step, a new data source request is defined, the goal of which is to clear a violated assertion. Adding a data source request may temporarily introduce an unresolved dependency, but eventually all assertions are fulfilled and all dependencies are resolved.

6. IMPLEMENTATION

The backend of WS-Aggregation (gateway and aggregator nodes) is implemented in the Java programming language. Both the gateway and the aggregator nodes expose a WSDL interface and communication takes place using SOAP messaging. Metadata and endpoint information about services are stored in the VRESCo service registry. All aggregators and optionally the target data sources are contained in the registry. This allows for dynamic lookup and binding of aggregators and data services at runtime.

The frontend for development and debugging of mashups is implemented as a Web application. A screenshot of the graphical user interface (GUI) is depicted in Figure 6. The GUI is divided into four parts, which are displayed on a single HTML page: 1) the design view (top left) is used to construct the request inputs and WAQL queries, 2) the debug view (top right) displays an interactive data dependency graph, the assertions table and an output console, 3) the result view (bottom left) prints the XML source code of the mashup output, and 4) the preview (bottom right) shows the mashup output as an HTML document.

The Web GUI uses the JavaScript `XMLHttpRequest` object to communicate with the WS-Aggregation gateway, and the gateway routes the request to a master aggregator for distributed aggregation. In this architecture, the complete debugging information is collected by the backend, and the purpose of the Web application is solely to visualize the results and to react on user input. The reasons for using this architecture and not executing the mashup on the client side are manifold: first, WS-Aggregation constitutes an invokable data aggregation Web service, and a Web application is only one way to access it. Second, the platform implements configurable query distribution strategies, which are hard to implement with client-side execution. Besides, the query preprocessing and XQuery transformations are quite computation-intensive and not really suited for a browser environment. Our extensive performance evaluation carried out in `wsAggr` shows the good performance and scalability of WS-Aggregation.

7. CONCLUSION

In this paper we proposed an approach for step-by-step debugging of Web mashups. The realization is based on WS-Aggregation, a platform for distributed aggregation of het-

erogeneous data from Web services and documents. Mashups in WS-Aggregation are defined declaratively as a set of data source requests with data dependencies between them. The mashup execution is a multi-step process, and the developer specifies assertions about the mashup state in different intermediate steps. If an assertion fails at runtime, the platform helps to identify the reason of the error. The frontend prototype is implemented as a Web 2.0 application and provides direct visual feedback of the intermediate results in each processing step.

As part of our ongoing work, we are extending the debugging facilities of WS-Aggregation by analyzing advanced anomaly patterns, which may be useful for suggesting concrete corrections and improvements of the mashup under development. We further plan to take into account different roles and stakeholders of collaborative service mashup design and debugging, as discussed in [21]. Moreover, we intend to automatically generate assertions from the mashup definition, and we are moving to a more intuitive way of specifying data source requests and data dependencies, possibly with graphical XML editors and an XQuery mapper similar to [6].

8. ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement 215483 (S-Cube).

9. REFERENCES

[1] M. Albinola, L. Baresi, M. Carcano, and S. Guinea. Mashlight: a lightweight mashup framework for everyone. In *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web*, 2009.

[2] D. Benslimane, S. Dustdar, and A. Sheth. Services mashups: The new generation of web applications. *IEEE Internet Computing*, 12(5):13–15, 2008.

[3] A. Bouguettaya, S. Nepal, W. Sherchan, X. Zhou, J. Wu, S. Chen, D. Liu, L. Li, H. Wang, and X. Liu. End-to-end service support for mashups. *IEEE Transactions on Services Computing*, 3:250–263, 2010.

[4] J. Cao, K. Rector, T. H. Park, S. D. Fleming, M. Burnett, and S. Wiedenbeck. A Debugging Perspective on End-User Mashup Programming. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2010.

[5] C. Cappiello, F. Daniel, M. Matera, and C. Pautasso. Information quality in mashups. *IEEE Internet Computing*, 14:14–22, 2010.

[6] P. S. Corporation. Stylus studio xquery mapper. http://www.stylusstudio.com/xquery_mapper.html. Visited: 2010-09-23.

[7] F. Curbera, M. Duftler, R. Khalaf, and D. Lovell. Bite: Workflow composition for the web. In *International Conference on Service-Oriented Computing*, pages 94–106, Berlin, Heidelberg, 2007. Springer-Verlag.

[8] H. Erdogmus, M. Morisio, and M. Torchiano. On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering*, 31(3):226 – 237, March 2005.

[9] G. Gottlob et al. Lixto data extraction project: back and forth between theory and practice. In *Symposium on Principles of Database Systems*. ACM, 2004.

[10] L. Grammel and M.-A. Storey. An end user perspective on mashup makers. Technical Report DCS-324-IR, University of Victoria, 2008.

[11] W. Hummer, P. Leitner, and S. Dustdar. WS-Aggregation: Distributed Aggregation of Web Services Data. In *26th Symposium On Applied Computing (SAC), March 21-25, 2011*. ACM. To appear (Accepted on 2010-10-14).

[12] A. Koschmider, V. Torres, and V. Pelechano. Elucidating the mashup hype: Definitions, challenges, methodical guide and tools for mashups. In *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web*, 2009.

[13] J. Lin, J. Wong, J. Nichols, A. Cypher, and T. A. Lau. End-user programming of mashups with vegemite. In *International Conference on Intelligent User Interfaces*, pages 97–106, New York, USA, 2009. ACM.

[14] X. Liu, Y. Hui, W. Sun, and H. Liang. Towards Service Composition Based on Mashup. In *IEEE Congress on Services*, pages 332 –339, July 2007.

[15] E. M. Maximilien, H. Wilkinson, N. Desai, and S. Tai. A Domain-Specific Language for Web APIs and Services Mashups. In *Int. Conference on Service-Oriented Computing*, pages 13–26. Springer, 2007.

[16] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar. End-to-End Support for QoS-Aware Service Selection, Binding and Mediation in VRESCo. *IEEE Transactions on Services Computing*, 3(3):193–205, 2010.

[17] C. Pautasso and M. Frisoni. The mashup atelier. *ICSOC 2008 Workshop on Web APIs and Services Mashups*, pages 155–165, 2009.

[18] C. Pautasso, T. Heinis, and G. Alonso. Jopera: Autonomic service orchestration. *IEEE Data Engineering Bulletin*, 29(3):32–39, 2006.

[19] N. Schuster, C. Zirpins, M. Schwuchow, S. Battle, and S. Tai. The mosaic model and architecture for service-oriented enterprise document mashups. *Workshop on Web APIs and Services Mashups*, 2009.

[20] D. E. Simmen, M. Altinel, V. Markl, S. Padmanabhan, and A. Singh. Damia: data mashups for intranet applications. In *ACM SIGMOD International Conference on Management of Data*, pages 1171–1182, New York, USA, 2008. ACM.

[21] M. Vasko and S. Dustdar. Introducing collaborative Service Mashup design. In *International Workshop on Lightweight Integration on the Web*, 2009.

[22] W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, 2007.

[23] G. Wang, S. Yang, and Y. Han. Mashroom: end-user mashup programming using nested tables. In *18th International Conference on World Wide Web*, pages 861–870, New York, USA, 2009. ACM.

[24] N. Zang and M. B. Rosson. What's in a mashup? and why? studying the perceptions of web-active end users. In *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 31–38, Washington, DC, USA, 2008. IEEE Computer Society.

[25] A. Zeller. *Why Programs Fail - A Guide To Systematic Debugging*. Morgan Kaufmann, 2005.