Mashlight: a Lightweight Mashup Framework for Everyone

Matteo Albinola, Luciano Baresi, Matteo Carcano, and Sam Guinea
Deepse Group
Dipartimento di Elettronica e Informazione
Politecnico di Milano
{baresi | guinea}@elet.polimi.it

ABSTRACT

Recently, Web 2.0 has brought a storm on web application development. In particular, mashups have greatly enhanced user creativity across the web, allowing end-users to rapidly combine information from diverse sources, and integrate them into “new” goal-oriented applications. In the meantime, widgets (also built with Web 2.0 technology) have also gained a lot of momentum. Indeed, they have become ever more present in our everyday lives, even appearing as first class players in our operating systems (e.g., Mac OS X and Windows 7).

In this paper we present Mashlight: a lightweight framework for creating and executing mashups that combines these two worlds. Indeed, it provides users with a simple means to create “process-like” mashups using “widget-like” Web 2.0 applications. Even users that have no technical know-how can string together building blocks—taken from an extensible library—to define the application they need. The framework is implemented using common Web technology, meaning our mashups can be run from different kinds of devices, in a lightweight fashion without the overhead of a complex application server. The paper presents the main concepts behind Mashlight blocks and Mashlight processes, and demonstrate our prototype on a concrete example.

Categories and Subject Descriptors
D.1.7 [Software]: Programming Techniques—Visual Programming; H.4.m [Information Systems]: Miscellaneous

General Terms
Design

Keywords
Mashup, Process, Service, Lightweight, Web 2.0, Mashlight

1. INTRODUCTION

The integration of Web 2.0 technologies [5, 11] with the software as a service principle [12, 13] is imposing widgets as a new means to exploit and integrate information over the web. The idea is that disparate data are accessed remotely through service-based interfaces and can then be freely recombined by the user on his browser. These assemblies, like iGoogle, are often called mashups [10] and let users decide the information of interest and how to organize it on the browser with no particular programming skills and no significant effort.

Recently, this same idea has been applied to the composition of process activities instead of simply considering the aggregation of data. Some first proposals, like Popfly [8], provide means to let the user organize a real process on his browser. This is an evolution of data-oriented mashups with the aim of overcoming the limitations of current composition technologies for (Web) services. If we consider solutions like BPEL (Business Process Execution Language [4]), they foster the composition of Web services, but they also require substantial skills, impose the server-side deployment of designed compositions, and thus do not really support the flexibility and customization that are typical of user-centered processes. Oftentimes, these processes materialize very simple aggregations whose actual existence is limited to a single use by a particular user. There is thus no need for the definition of complex processes, which account for many different alternatives, and for their “permanent” deployment. The user would like to have a very simple and lightweight composition means to support the definition of easy and transient assemblies. Required skills must be very low, and the result must be usable immediately.

Given these requirements, the paper presents Mashlight, which is an attempt to borrow from data-centered mashups and the existing attempts to combine process activities, to extend them with presentation support, and to provide a simple and flexible framework for the definition of client-side compositions. Mashlight supports the seamless integration of many different activity types: from searching through the Yellow Pages to showing a location on a map. There is no need for heavy Web servers and container technologies, but the whole proposal is based on common Web technologies.

There are currently a number of approaches that can be used to create data, logic, and presentation mashups. Most of them, however, are destined to power users, i.e., users that do not have deep programming skills, but understand the technological difficulties of the problem being treated, and are comfortable with an intermediate level of abstraction. Mashlight’s goal, on the contrary, is to appeal to everyone, even to those who have no technical knowledge. Although approaches that are destined to these kind of users do exist, they lack the expressive power we need to create articulated mashups.

Mashlight reaches its goals by providing users with a simple means to create process mashups by using Web 2.0 application widgets. Even users that have no technical know-how
can string together building blocks—taken from an extensible library—to define the application they need. To a-priority limit the complexity, composition means are limited to sequences, branches, and loops to balance simplicity with the need for slightly articulated processes.

The paper provides an introduction to Mashlight. Besides setting the context of our proposal, it also presents its conceptual model and the first proof-of-concept prototype implementation. The idea is to provide the reader with a first glimpse on our proposal and its key differences with respect to its competitors. This is why its key characteristics are thoroughly demonstrated by means of an example: We exploit a well-known problem (a person who wants to organize a night out with friends) to let the reader focus on the prominent aspects of Mashlight.

The rest of the paper is organized as follows. Section 2 briefly surveys the state of the art and helps the reader set the context. Section 3 describes the conceptual model behind Mashlight, while Section 4 sketches how it is implemented. Section 5 materializes the main concepts on a simple example, and Section 6 concludes the paper.

2. STATE OF THE ART

Mashups commonly identify a Web-based application that dynamically composes information coming from different online sources. A classic example is given by HousingMaps.com, which allows users to visualize real estate information on a map. In this case data from Craigslist are combined with the single most popular mashup service, Google Maps. An impressive number of mashup development frameworks have been developed both in industry and in research, and it is a daunting task to present them all. To give a clear panorama of the industrial mashup tidal wave, we classify them along three orthogonal dimensions.

The first dimension is the nature of the mashups targeted by the framework. We can distinguish between data mashups, logic mashups, and presentation mashups. Data mashups are mainly interested in combining data coming from different sources. An example of a data mashup framework is Yahoo Pipes [3], which provides means to combine RSS feeds and JSON data. Logic mashups introduce ways for different services to collaborate with their logic towards the application's goal. A famous example of a logic mashup framework is Microsoft's Popfly [8]. Finally, presentation mashups do not combine neither data nor application logic, but simply present data or services taken from different sources using a common interface. A famous example is iGoogle, a personalized browser homepage where users can aggregate widgets (or Google Gadgets) such as news widget, a weather widget, or a calendar widget.

The second dimension is the type of users capable of using the framework to solve a specific problem by creating a mashup. We can distinguish between Developers, Power Users, and Users and Workers. Developers are people that have deep knowledge of Web technology, as well as programming skills, and could solve the problem at the lowest level of abstraction. Instead they are interested in using the mashup approach as a faster programming tool. Power users, on the other hand, do not have deep programming skills, but understand the technological problem, and are comfortable with an intermediate abstraction level. Finally, users and workers are people that only acknowledge the problem at the business level, and do not have any technical skills.

The third dimension regards "how" and "where" a mashup is run. We can distinguish between Client-side mashups and Server-side mashups. The former take advantage of client-side resources through the user's Web browser. In this case, the execution environment typically relies on lightweight Web technology such as JavaScript, XML, and JSON. The latter take advantage of server technology. In this case standard service oriented technology such as WSDL and SOAP can also be part of the picture. Indeed, the common definition of mashup is generic enough to include standard service compositions such as those performed using BPEL.

Figure 1 presents a map of the frameworks we have tested, in which they are categorized along the first two classification dimensions. As we can see from the map, there are some approaches that tackle data, logic, and presentation, such as Microsoft Popfly and Google Mashup Editor [1]. This means that with these approaches it is possible to define a wide variety of mashup applications. On the other hand, there are no approaches that target more than one specific group of designated users. For example, there are no approaches that are for both developers and power users, or for both power users and workers. Moreover, most of the approaches provide low and intermediate levels of abstraction, while fewer are tailored towards high-level users.

To conclude our presentation of the industrial state of the art we give a brief comparison of the major players. We will start with an approach dedicated to developers, and move towards higher levels of abstraction. Google Mashup Editor [1] is a developer-oriented Web 2.0 development environment that excels in integrating Google Services, such as Google Maps and Google Calendar. However, interaction with external information services is limited to RSS feeds, while external Web services are not available. Moreover, the design environment is Web-based and cannot be used off-line.

Yahoo Pipes [3] is tailored to power users that are accustomed to XML data such as RSS feeds. The framework does not require programming skills, but offers a set of simple action units that can be visually combined using "pipe" connections. Although composition is easy thanks to a Web 2.0 drag and drop environment, the set of action units is not extensible. This greatly limits the kinds of solutions that can be created using Yahoo Pipes. Moreover, the environment is Web-based and cannot be used off-line, meaning run-time execution is limited to Yahoo servers.

Microsoft Popfly [8] is a very powerful tool that can be used to create different kinds of mashups. The goal of the work was to provide an environment that could be used by everyone. Unfortunately, the result is an environment that requires technical know-how to create anything more complex than mere examples. It provides functional base components of various nature and the interaction with external information sources is well supported. It also allows the definition of new base components, allowing it to cope with ever new scenarios. The environment is a web application that cannot be used off-line, and the run-time environment is limited to Microsoft servers.

Finally, Intel Mash Maker [2, 7] is a simple tool that is tailored towards high-level users. It specializes in the creation of Web pages that combine information from diverse sources. The only information sources that can be used are those represented by web pages that are visited through a browser that has been extended using a special plugin. As
the user navigates, widgets with special data management capabilities can be added and linked to the pages being visualized. This makes the tool very easy to use but greatly limits the sources that can be integrated, since RSS feeds and Web services are ignored. Once again, the tool is Web-based meaning it cannot be used off-line. Moreover, every mashup is saved as a web page on an Intel server.

There are also many research-based approaches, and most of them concentrate on creating data-intensive mashups. PiggyBank [9] allows users to take advantage of Semantic Web content while they navigate the Web. It is implemented as a browser plugin that collects semantic content from RDF-enabled web sites. If a web site does not publish RDF data it uses screen scrapping techniques to automatically generate RDF data. The data it collects are sent to a shared Semantic Bank, i.e. a RDF repository, where they are collected and published, enabling collaborative scenarios. The approach allows users to browse the RDF repository and to tag interesting information, which are then saved to a local personal PiggyBank repository.

Dontcheva et al. [6] present three interaction techniques users can use to collect and organize Web content. The first technique consists of an application that allows users to define relationships between different Web sites. From that point on, every time a user extracts data, the system automatically extracts all the related data as well. The second technique consists of an application that allows user to merge data coming from different Web sites, and to organize them visually using “cards”. The third technique consists of an application that allows users to define search templates. It’s goal is to narrow the search space used by a search engine, to present a personalized view of the available Web content. The three techniques can also be used in unison: relations can feed into cards, while cards can be used to define search templates.

Tuchinda et al. [14] present Karma, an tool for creating mashups by example. They propose an integrated solution for data retrieval, source modeling, data cleaning, and data integration. Data retrieval uses a user-generated example to define what data should be extracted from a Web page, and to populate a data table. Source modeling asks the user to associate a name to each column in the table. Data cleaning looks at a user-generated example and tries to find out the cleaning algorithm it should apply to all the data in a given column. Finally, data integration compares the data in the table with data previously added to a repository and proposes join conditions that can be used to provide a data mashup.

Marmite [15] is a an easy-to-use mashup solution inspired by Apple Automator, a visual scripting tool for automating repetitive tasks under Mac OS X. Users can choose from a number of activities for extracting data from Web sites, and local and remote databases. The activities are then organized in simple sequential data flow. The system also provides run-time views of the data as they flow from one activity to the next. The activities can be of three kinds. Sources consist of Web data extractors, processors consist of data modifiers and aggregators, and sinks consist of data visualizers (e.g., a map). The authors also provide insight to what determines the usability of their approach: the fact that the system suggests compatible activities at design time, the fact that intermediate data is always shown, and the fact that they support incremental execution.

From our analysis, it is clear that a number of solutions exist, with very diverse features. However, it is also clear that there are some important requirements that have not been treated. Although it is possible to create mashups that integrate data, logic, and presentation, the approaches we have analyzed do not provide end-user tools for creating complex mashups that combine both data and process flows. On the contrary, either the process flow is extremely simple (e.g., Marmite), or power-user skills are required. A second aspect that emerges is that no approach offers a truly lightweight solution. The reason is that these approaches all create mashup applications that are deployed to servers for repeated use. None of these take into account the assumption that users might be interested in creating a disposable mashup application, i.e., one that solves the immediate

---

**Figure 1: A classification of industrial mashup frameworks.**

<table>
<thead>
<tr>
<th>Data Mashup</th>
<th>Logic Mashup</th>
<th>Presentation Mashup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Mash Maker</td>
<td>IBM Lotus Mashup</td>
<td>RIA Apps</td>
</tr>
<tr>
<td>Yahoo Pipes</td>
<td>Microsoft Popfly</td>
<td>Blogs, Wikis</td>
</tr>
<tr>
<td>Dapper</td>
<td>JackBe Presto</td>
<td>BEA Acqualogic Pages</td>
</tr>
<tr>
<td>Kapow</td>
<td>Serena Mashup Composer</td>
<td>HTML, Flash, Silverlight</td>
</tr>
<tr>
<td>SnapLogic</td>
<td>Google Mashup Editor</td>
<td>Silverlight Development</td>
</tr>
<tr>
<td>Mashery</td>
<td>DreamFace Mashup Kits</td>
<td>Developers</td>
</tr>
<tr>
<td><strong>Users</strong></td>
<td><strong>Power Users</strong></td>
<td><strong>Power Users</strong></td>
</tr>
</tbody>
</table>
problem at hand and is not needed anymore. This may be the case in which the approach is used as a fast prototyping tool. Our solution can be used both to create reusable and disposable mashups.

3. CONCEPTUAL MODEL

The requirements for our approach are that we want a mashup framework that can be used by anyone, from developers to workers, to create any kind of mashup, from data mashups to logic and presentation mashups. The framework must be simple enough for anyone to use, without loosing the flexibility required to satisfy the goals that even the most technical-savvy users may have. Moreover, the approach should be open to extremely diverse kinds of applications.

For example, the approach should support a worker that wants to define a mashup that can help organize a night out with his/her friends. In this case, the worker might want to (1) search and choose a restaurant near his/her current position, (2) view the restaurant on a map of the city, (3) search and choose a movie theater that is near the restaurant, (4) view the movie theater on a map, (5) see what movies are being shown in the theater and buy the tickets, and (6) produce a complete navigation route for the evening. The user’s goal in this case is to avoid loosing time going from one web site to another, searching and organizing the night in an unorganized fashion. This is clearly a high-level mashup. Most of the existing high-level approaches that exist in literature specialize in creating a composite view of information. Although we believe a single composite view, where all the information is shown, might represent a good solution at run-time, we also believe it to be not so intuitive during the design phase. It would be better to provide a tool in which the user can define a sequential set of steps (e.g., first search for a restaurant, then show me the restaurant on a map, etc.).

In another and completely different example, a developer might want to use a mashup environment to create a prototype of an application that interacts with external Web services. In this case the final application might have high quality of service requirements (e.g., transactionality, security, reliable messaging, etc.), and may be implemented with traditional technology such as WSDL, SOAP, and BPEL. However, a mashup might be just what he/she needs to investigate service behaviors and compatibility, before starting true development. The interested reader can find implementations of both examples at http://home.dei.polimi.it/guines/mashlight/.

In these examples ease of use is a key issue and development speed is critical. The worker will never create such a mashup if it takes too much time, while in development rapid prototyping and turn around times are of utmost importance. Another key requirement that emerges is that the framework should be lightweight. A final requirement emerges from our night-out example. The framework should allow the user to define the application on a desktop computer at work, and then migrate it over to his/her mobile device to take it with him/her on the night out. Summarizing, the framework must be:

- Flexible: it must support data, logic, and presentation mashups, in diverse scenarios.

- Usable: it must provide abstractions that are useful to developers, power users, and workers. The abstractions must allow the users to create mashups rapidly.

- Lightweight: its execution environment must be lightweight, and provide support for both desktop and mobile devices.

Mashlight is our proposal for a new and innovative mashup framework. With Mashlight, mashups are created through a “process-like” composition of logic units called “blocks”, taken from an extensible set.

With respect to standard process definition languages, such as BPEL, the level of abstraction proposed in Mashlight hides the classic constructs for sequencing activities, conditional branching, and looping. In our model a process simply defines “when” different functional blocks need to be activated, and using “which data”. This means that a Mashlight process can be modeled using an oriented graph, as shown in Figure 2. The nodes represent functional blocks, while the arcs can represent either data dependencies (dotted arcs), or sequential activation (normal arcs).

3.1 Functional Blocks

Mashlight blocks are functional and independent units. For example, a block could be a weather service that provides forecasts for a given location, a calculator that provide mathematical operations, or even a generic SOAP implementation for invoking remote Web services. As independent units, blocks can also provide their own graphical user interface.

Conceptually, Mashlight blocks are very similar to widgets, such as those that have become first-class players in our operating systems (e.g., Mac OS X and Windows 7). Indeed, in Mashlight, blocks are implemented and packaged using Web 2.0 technology, making it easy to port existing widgets (e.g., Google Gadgets) over to Mashlight.

The main difference with respect to widgets is that they need to collaborate exchanging data. This is why Mashlight blocks (see Figure 3) provide data input and output parameters (indicated as dark and light circles in our graphical notation). Depending on its needs, a Mashlight block can define any number of input or output parameters, and parameters can be mandatory or optional. Let us take for example a weather forecasting block. A reasonable input parameter would be the location for which we want a weather forecast. Regarding the output parameters, we could have
very diverse scenarios depending on implementation decisions. One implementation of the block could decide to provide the forecast as an output parameter, while another could simply show the forecast through its graphical interface and provide no output parameter at all.

Mashlight currently supports both simple parameters (i.e., simple XSD types such as string, integer, etc.) and complex parameters (i.e., parameters defined by complex XSD definitions). It also supports varargs parameters, i.e., parameters with multiple cardinality. The typical example is when we need to deal with routing and mapping. A navigational block based on Google Maps Navigator will take a starting location and a destination, and a list of intermediate locations to be shown on the map. Since it is impossible to know a-priori how many intermediate points we will have, the block is defined with an input parameter of multiple cardinality.

Besides data exchanges, Mashlight blocks also need to specify how they can be used within the sequencing of a process. This is done specifying inlinks and outlinks (indicated as dark and light diamonds in our graphical notation). By definition, Mashlight blocks only have one inlink. On the contrary, a block can define as many outlinks as it needs. However, only one of these outlinks will be chosen at run time, and it is the block’s responsibility to decide which one to activate. Things will become clearer once we introduce the notion of Mashlight process, in which parallel execution of blocks is not supported.

3.2 Processes

The Mashlight model distinguishes between Process Flows, Data Flows, and Execution Flows. Figure 4 shows an instructive example of the three different kinds of flows that exist in a Mashup process.

3.2.1 Process Flows

A process flow contains all the possible orders in which the composed Mashlight blocks can be executed. In a process flow an arc connects a block’s outlink to another block’s inlink, indicating that the termination of the first block will cause the execution of the second one. In Figure 4 (a), for example, the arc between A and B states that upon termination of A, B will be executed. In Mashlight, loops and self-loops are permitted. Indeed, in our example, we have a loop involving nodes A and D.

As previously stated, in Mashlight blocks can have more than one outlink. For example, D has an outlink that could be used to activate E, and another one that could be used to activate A. In Mashlight, parallel block execution is not allowed, meaning that the two blocks following D are actually in mutual exclusion. Indeed, when D terminates, it will communicate to the execution framework which of its outlinks needs to be activated.

In Mashlight, if multiple outlinks are connected to a single block’s inlink, we are indicating that the block can be activated in more than one way. This is the example of block E, which can be activated either from C or from D.

3.2.2 Data Flows

A data flow indicates the data dependencies that exist between the blocks in the process. Obviously, data dependencies have a profound impact on the how a process is executed, since a Mashlight block can only be executed when all its mandatory input parameters have received a value.

In Mashlight we support both simple and complex data, and the assignment of an input parameter can be made using either a constant value or a value taken from an output parameter. Figure 4 (b), for example, uses a constant value to set A’s input parameter, while output parameters are used in all the other assignments.

Obviously the definition of a mashup’s data flow must be given in conformity to the process flow. If we have two blocks that are in mutual exclusion, then it is going to be impossible for them to share data. For example, A’s two outlinks cause B and D to be in mutual exclusion. This means it does not make sense for B to expect any data from D, and vice versa. In this sense, Figure 4 (b) may appear to be erroneous. Indeed, in the process flow E can be reached both from C and D. However, in the data flow E only received data from D. This means that E can be activated after C only if E’s input parameter is defined as optional. If not, the process flow and the data flow would be inconsistent.

Mashlight allows data assignments with multiple destinations and sources. This means that the same data can be passed to more than one block. This is the case of block A which sends its output data to both B and D. Multiple sources and destinations means it is also possible to receive data on an input parameter from two different sources. This causes an assignment conflict at run time, which is solved, in our model, using priority values. In practice, we execute the assignment involving the source block that executed most recently.

3.2.3 Execution Flows

Finally, execution flows indicate the actual flows that can occur at run time, given certain process and data flows. Figure 4 (c), for example, illustrates two possible executions of the mashup process. The one one the left takes advantage of the fact that E’s input is defined as optional. The one on the right, on the other hand, occurs when A chooses to activate D instead of B. In this case D chooses to re-activate A. This is possible thanks to the loop in the process flow definition, and to the fact that A’s input parameter can always be set using the constant value. Notice that infinite execution flows can be produced in this example, due to the presence of the loop, which in theory can be iterated infinite times.
3.2.4 Advanced Features

Our model also provides an “undo” feature, which allows a process to be stopped, and re-started at the beginning of the most recently terminated block. This means that the undo feature looks at the execution flow to determine which block has to be re-executed.

The undo feature obviously requires data persistency, which can become an issue, especially when the process flow defines a loop. This is why the Mashlight model supports two kinds of data storage. “Complete” data storage consists in storing all the data ever produced in the execution flow. This allows us to support a complete undo of the process. “Finite” data storage, on the other hand, limits the amount of steps it is possible to undo. Which to use will depend on the device being used, i.e., desktop or mobile.

The undo feature can be problematic with blocks that have persistent effects. For example, if the user first uses a block to buy some tickets for a movie, and then uses a block to show a map of how to reach the movie theater. Should he/she decide to perform an undo, the standard semantics would eliminate the execution of the map block and go back to the beginning of the movie theater block. The user would have to buy the tickets again! The Mashlight model solves this problem by allowing blocks to identify themselves as not undo-able. This limits the amount of undos that can be performed in the process, and has benefits on the amount of data storage needed at run time, once the execution has passed the non-undo-able block. Indeed, any saved data from before that point can be discarded.

4. IMPLEMENTATION

The Mashlight framework (see Figure 5) is made up of four main components: the Block Builder, the Block Library, the Mashup Builder, and the Run-time Engine.

The Block Builder is a development environment for the definition of Mashlight Blocks. Each block must follow the Mashlight Block Specification (MBS) to be compatible with the Run-time Engine. The specification defines the internal structure of an archived block, and the interface the block must implement for a correct interaction with the engine.

Blocks are mini Web 2.0 applications, and are implemented using standard technology such as HTML/XHTML, XML, JavaScript, and CSS. Blocks follow the pre-defined internal structure shown in Figure 6 (a). The block has two main files: a main.html file and a block.manifest file. The main.html file is mainly responsible for the block’s presentation, while the manifest file contains all the information needed to execute it, such as the definition of its inlinks and outlinks, and of its input and output parameters. Notice that there is also a folder for any assets that may be required by the block’s internal code. The most important thing in a block’s definition is its manifest. It contains:

- **General information** regarding the block, such as a unique UUID identifier, a name, a version, a description, an icon, information regarding the author, and an indication whether the block is interactive or not.
Data type specifications, which are the XSD schemas for the data types used to define a block’s input and output parameters.

Outlink information, which consists of a unique name for each of the block’s outlinks.

Input and output parameter specifications, that name the parameters, link them to the data types specified above, and state whether they are mandatory or optional.

The Mashlight Block Specification also defines the interface a block must implement to interact with the Run-time engine. The interface contains functions initialize, abort, and undo. The first function is called by the engine to tell a block that it can initialize, while the other two are used to tell the block that it should terminate. The difference lies in the reasons for terminating the block. An abort signal that there has been a problem with the engine, while an undo signals that the user has performed an undo. In both case, the block should proceed to a graceful termination.

4.2 Mashup Builder

The Mashup Builder is the visual tool we provide for defining Mashup processes. Mashup processes need to follow the Mashup Process Specification (MPS), which defines a mashup’s internal structure (shown in Figure 6 (b)), and an XML specification language for process and data flows.

In a process archive there are two main files: the executable process definition (i.e., process.js, which is automatically generated from an XML definition of the process that mirrors the process’ graph structure, and the process manifest (process.manifest). There is also a folder for embedding blocks directly into the archive. This is not mandatory, since blocks can be loaded at run time directly by the engine. Finally there is a folder for any additional assets that might be needed.

The most important file in the archive is the process manifest. In contains:

General information regarding a unique UUID identifier, a name, a version, a description, an icon, and author information.

The flows are described using a list of variable elements and a list of node elements. Each variable refers to an input or output parameter that is going to be used in the process. Each node contains a list of link elements, and a list of assign elements. The former are used to define the process flow, while the latter are used to define the data flow. In the definition of the process flow, one of the nodes is always indicated as the process’ starting node.

4.3 Run-time Engine

The Run-time Engine is the component responsible for enacting the mashup. It is developed using Web 2.0 technology and can be run within any WebKit-based browser. Figure 7 illustrates the Run-time Engine’s layered structure. There are two main modules: the Core Engine and the CoreGUI.

The Core Engine is made up of four sub-modules. The Variable Container module contains all the APIs for managing variables, and for keeping track of the persistent storage needed for our model’s undo feature. The Manifest Container module contains all the APIs for managing the processes that have been deployed to the system. The Runtime module manages general information regarding the running process instance, and the Flow module contains the APIs used to execute the instance. The CoreGUI, on the other hand, is the module that is responsible for managing all presentation issues.

On top of these two core modules we have a Block API and a Process API. These allow the running block ad the running process to take advantage of the two Core modules.

For example, the block will use the Block API to access the CoreGUI module and manage key issues such as the positioning and resizing of its GUI, and to access to the...
CoreEngine for standard operation. Indeed, a block will use this API to ask the engine if all the input parameters it needs are available (isInputAssigned(inputName)), to get the inputs once they are available (getInput(inputName)), and to set the outputs once it has finished (setOutput(outputName)). The API also provides two functions for communicating it has finished operation. Upon normal completion end(outlinkName) is called, with a parameter that states which outlink has to be activated by the engine. When the block ends with an error, endWithError(error) is called.

The process, on the other hand, will use the Process API to access the CoreGUI module and manage its representation in the browser. Indeed, in the browser we always show a view of the overall process, to understand where we are in the execution, and what would happen should we perform an undo (see Figure 8).

When a mashup is launched by the user, the engine goes through a three step process. The first step is to load and initialize the process, the second is to prepare a block for launch, and the third is to execute a block. While the first step is only performed once, steps two and three are obviously repeated.

The first step can be articulated in four sub-steps. The first sub-step is to load manifest and the executable version of the process. The second sub-step is to read the process’ manifest to see what blocks are needed, and to load them from the Block Library. The third sub-step is to construct a graph representation of the process so that it can be shown using the CoreGUI module, while the last sub-step is to initialize the variables needed to deal with the input and output parameters of all the blocks. The required information is obviously read from the block manifests.

The second step consists in preparing the next block that needs to be run. This is an important step that is activated at the beginning of a process execution, and every time a block terminates correctly. If we are beginning the process, which block to launch can be read from the process’ manifest. If the process has already started, the next block to launch is chosen by the block that has just finished executing, by communicating which of its outlinks needs to be activated.

The third step consists in launching the chosen block. This step can be articulated in five sub-steps. The first sub-step is to assign the input parameters needed by the block. The second sub-step is to load the block’s graphical representation. The third sub-step consists in a call to the block’s initialize function. The fourth sub-step is the autonomous execution of the block’s internal logic, while the fifth and last sub-step is activated when the block has terminated. The engine reads the output parameters and assigns their values to the corresponding variables.

5. A NIGHT OUT

Now that enough details have been given regarding the Mashlight model and its implementation, we shall present a complete example (already described in Section 3). In this example, Mashlight is used to organize a night out in which the user wants to find a restaurant, and a movie (a screenshot of the example running can be seen in Figure 8).

The Mashlight process uses four different blocks: a Yellow Pages block to look for a restaurant and for a movie theater, a Google Maps block to show locations on a map, a general purpose movie theater block to see what movies are being shown and to buy tickets, and a Google Maps Navigator block for producing a navigational route for the entire evening.

Due to lack of space we cannot show the block manifests for all the blocks involved in the example, but only that of the Yellow Pages block (see Figure 9). As we can see, the
block defines a single outlink called `outlink1`, an input parameter called `searchQuery`, and an output parameter called `resultItem`. Both parameters are complex types, and their schemas are given under the `type` node. The searchQuery parameter is made up of four strings indicating what we are looking for (e.g., a restaurant), in what city, and near what street and number. When executed, the block shows a list of results, and allows the user to click on the one he/she wants. This feeds the resultItem output parameter, and causes the block to terminate. The resultItem parameter is made up of five strings and a coordinates set (defined using its own type specification). The information placed in the output parameter contains the name of the place that has been chosen, its address (i.e., street, city, postal code, and state), and its GPS coordinates (for use in the map block).

The process flow in this example is straightforward, since the blocks are simply placed in sequence. The data flow (shown in Figure 10, however, is slightly more complex. This is an example in which we have constants, complex data types, and data assignments that do not follow the normal process flow. Indeed, some blocks pass data to blocks that are much further down in the execution flow. In the figure the names of the input parameters are given in regular text, while the names of the output parameters (or of their parts in the case of complex types) are given in italics.

The data flow starts by assigning a set of constant values to the first Yellow Pages block. In this case we are looking for restaurants in Milano near Piazza Leonardo 1. When the block terminates, its output parameters feed the Google Maps block which shows where the selected restaurant is. When the Google Maps block terminates (i.e., is closed by the user), the engine launches a new instance of the Yellow Pages block and feeds it with a constant, indicating we are looking for a movie theater, and the city and address retrieved from the first Yellow Pages’ output parameter (i.e., the city and address of the chosen restaurant). The block executes and shows a list of the possible theaters. When the user chooses a theater, the block is terminated its output parameter is filled. Once again the engine starts a Google Maps block to show where the theater is on a map. The next step is to execute the Movie Tickets block. This block needs
a user account name, which is fed through a constant value, and the name of the theater. This is taken from the output data of the second Yellow Pages block. Finally, the engine executes the Google Map Navigator block. This block has three parameters: a from (indicating a starting point), a to (indicating the final destination), and an optional parameter called step, which is actually a list of intermediate locations with unknown cardinality. In this case, the starting point is “via Golgi 40, Milano”, and the destination is the address of the selected movie theater (data taken from the second Yellow Pages’ output parameter). In our example, there is one intermediate point, i.e., the selected restaurant. In this case we use an array like notation to indicate that the data arriving from the first Yellow Pages block should be placed in the first position of the list of intermediate locations.

Due to lack of space, we cannot show the entire process manifest, but only a part of it. Figure 11 shows a simplified version of our example called Search&Map. In this simplified version we have limited the process flow to a sequence of two blocks, and have also used simplified versions of our Mashlight blocks. In Search&Map first we perform a search using the Yellow Pages block, and then we show the result’s location on a map using the Google Map block. In the example, node n1 is of type i1, which refers to the Yellow Pages block through its UUID identifier, and is tagged as the starting node for our process. It has one outlink that is connected to node n2 of type 12, which refers to the Google Maps block. The Yellow Pages block has one input parameter called searchQuery, and one output parameter called resultItem, while the Google Maps block only has one input parameter called coordinates. For the input parameters the process defines variable n1_searchQuery and n2_coords, while for the output parameter it defines n1_resultItem. These are used to define the data flow. In the example, n1_searchQuery is filled using a constant value restaurant, while n2_coords is filled with the coordinates present in n1_resultItem. This is done using an XPath query that selects the coordinates from the complex schema of n1_resultItem.

### 6. CONCLUSIONS AND FUTURE WORK

In this paper we have presented Mashlight, an innovative approach to mashups that allow users to create “process-like” mashups using “widget-like” Web 2.0 applications. The approach presents a simple composition approach that limits complexity, without losing the flexibility needed to create articulated processes. Moreover, Mashlight is a lightweight approach that does not require any particular Web server or container technology, but can be run on any WebKit compliant Web browser.

Our future work will continue to evaluate the Mashlight model through the creation of domain-specific libraries of Mashlight blocks, and by conducting field tests with workers that do not possess technical knowledge. We are also going to continue to invest in the development of the Mashlight Builder, our visual composition environment for Mashlight processes. This is of paramount importance, since the ease of use of this environment will certainly define the usability of the entire approach. Finally, we are also interested in investigating the portability of our approach to mobile devices. A goal will be to allow processes to be ported at run time from one execution environment to another. A user will be able to begin the execution of his/her process at home, and continue on-the-road with his/her mobile device.

### 7. REFERENCES

[1] Google mashup editor

[2] Intel mash maker
http://mashmaker.intel.com/web/.


