ABSTRACT
In this paper we discuss a visual programming environment for the composition of services based on dataflow paradigm. This approach, strongly oriented towards web technologies, aims to offer the corporate application designer an integrated set of very high-level tools to design and test new services via functional composition of already available ones. The user experience is a mixture of (a) graphical actions and choices, for example the drag and drop of graphical widgets and the connection of input-output links, and of (b) interface coding, via the textual specification of the translation logic of data object types flowing through the input/output interfaces of the composed services. The new environment, named VisProSL (Visual Programming Service-Link), capitalizes on a standardized JSON syntax of exchanged data objects, is oriented to REST services, and makes use of a basic layer that provides the visual generation of interfaces towards location services.

Categories and Subject Descriptors
H.5.2 [Information interfaces and presentation]: User Interfaces; H.3.5 [Information storage and retrieval]: Online Information Services

General Terms
Visual programming

Keywords
Web Services, Service Composition, Orchestration, Visual Programming, HTML5, JavaScript

1. INTRODUCTION
In “the next 50 years” article, celebrating a half century of Communications of the ACM magazine, Rodney Brooks warrant us to expect new ways to understand computation and computational abstractions for our computing machinery: “The goal would be nonbrittle software modules that plug together and just work” [5], moving away from the traditional von Neumann abstraction, as demanded by the celebrated John Backus’ Turing Lecture [3], that inspired much work on functional languages.

Ray Kurzweil, in his visionary essay [9], estimated that “Within a few decades, machine intelligence will surpass human intelligence”. Whereas the goal of a future where “technology and human become one” by using semantics technologies looks quite faraway, other — apparently low-profile — approaches to enrich and automate the business processes have already given significant outcomes.

In particular, our applied research is attempting to reach efficient results in intelligent automation of narrower enterprise domains, and has already developed and deployed the VisProGW (Visual Programming by Graphical Widgets) module, a fast prototyping tool [4] for the development of applied Location-Based Services using the SOGEI’s corporate framework GEOPOI®.

The wide literature on composition and execution of web services as workflows that realize the functionality of a new service was analysed in [1] using multiple metrics, including composition effort, control, and ability to handle failures. In [6] the authors are motivated to approach the problem of semantic web services composition, and discuss the existing techniques for ‘Semantic Web Service’ and their limitations with respect to an appropriate set of required properties. The survey [10] of tools for automating service composition actually distinguishes between manual and automatic composition. The survey underlines that although the number of tools on 2010 was already impressive, the problem of automated composition was not solved. The very recent survey [13], after discussing a large number of systems, introduces a taxonomy of top-down and bottom-up designs, and distinguishes concrete from abstract workflows, finally identifying the emergence of new service models due to mobile devices.

In this paper we describe the design and the development of the new module VisProSL (Visual Programming Service
2. BACKGROUND

2.1 RESTful web services

A web service is a software system designed to support interoperability between different computers on the same network. It must provide a software interface that other systems can interact with. The interface is operated through messages included in an envelope. These messages are transported via the HTTP or HTTPS and formatted in some data representation language (either XML or JSON), using a lightweight protocol for exchange of messages between software components. Software applications, possibly written in different programming languages, and deployed on different hardware platforms, operate data exchange and execution of complex operations either on corporate networks or the Internet, via the interfaces they expose publicly and through the use of operations they make available. REST (Representational State Transfer) is a style of software architecture for distributed systems where service locations are modeled as URLs, and all resources are accessed stateless through a generic interface, i.e. by using basic HTTP primitives. “REST-based Web Services are increasingly being preferred for integration with backend enterprise services compared to SOAP due to their simplicity” [4].

2.2 Computations as data transformations

The abstract concept of function provides an important computational abstraction, since it encapsulates the type of computation and hides the details of the calculation to the user. This one only needs to know the mechanism of the function call, and not how the function works. Graphically, a function may be represented by a rectangle (box) and by its function name. We call $f(x)$ this graphical element. In abstract terms, it is a transformation (mapping) between the set of possible data (input) and the set of possible outcomes (output). In our visual programming approach, we only use two symbols, possibly instantiated in a set of specialized icons, to represent (at variable levels of detail) programs and data (including events fired by user interaction), respectively.

2.3 Composition of web services

A novel approach to corporate web-services was introduced in [4], and named VisProGW (Visual Programming Graphical Widgets). This tool establishes an initial library of interoperable software components, including location-based services and related interfaces, as template GUI components, and accumulates successful business processes as templates for new service classes.

The VisProGW environment supports higher-level functions, i.e. services (programs) that accept other services in input and/or that produce other services. A novel macro-service can be viewed as a stateless data-flow graph with pipelined data exchanges. The set of links determines a directed acyclic graph model of the computation. The processing is orchestrated by a modified DFS (Depth First Search) traversal algorithm, that requires that all the inputs necessary for the triggering of each single computation are available. The traversal ordering is established by visiting, in connection order, the first node with all input available.

3. PROJECT GOALS

3.1 Primary objectives

The main aim of the VisProSL (Visual Programming Service Link) project, described in this paper, is to go beyond location services, and develop a corporate-wide tool to support the client-based design and representation of REST-like web services, based on the dataflow paradigm [8], and including the description of their input/output parameters, though a JSON-based specialized language with highly expressive semantics. Therefore VisProSL is introducing a distributed interaction model among a network of REST services, whose integration is both accomplished and validated at client level. The output expected by a session of work with VisProSL is a new enterprise service, defined by the functional composition of pre-existing services, and reified as a bunch of HTML5 and JavaScript files.

In order to support the development of complex projects, the VisProSL environment provides the saving and restoring of its current status, via an up-to-date description of the project, loadable and executable in deferred with respect to its production, to allow for discontinued develop-
ment sessions. The VisProSL engine is designed to charge the minimal payload to corporate computational resources, also in order to minimise the security risks for the projects under development, that are often highly secretive. Vis-ProSL is therefore developed with languages and libraries able to take advantage of distributed resources, and is primarily executed on the web client supporting the user interface. A primary role in its development is assigned to advanced JavaScript technologies, most of which are listed in the following section. A further design requirement concerns the cross-browser and cross-platform (desktop and mobile) utilisation of the software tool via HTML5.

3.2 Further objectives

In order to improve the user experience in cross-browser and cross-platform web environments, we are experimenting with the new technologies made available by the HTML5 standard with respect to the building of user-interfaces via CSS3 Flexible Box Layout Module and the relative novel interaction capabilities, and therefore a responsive design of the user interface layout is taking a central role. The project is also experimenting with interactive documentation methods to minimise learning time when using the platform across the company. Finally, we are also testing the development of REST-like services and their input/output parameters, initially developed and tested though a JSON-based description language, via automatic translation towards a WADL-like description language. In this area we are making integration experiments of generic REST-like JSON-based services, e.g. for text documents, images, maps, etc. and legacy services (XML-based). In particular, such integration is required for the composition of services via the graphical user-interface.

4. WEB TOOLS

As recalled in the above section, VisProSL is being implemented making large use of client-side Web technologies — HTML5 + JavaScript + CSS3 — briefly synthesised in this section.

4.1 HTML5 technologies

The VisProSL development is making use of several novel technologies introduced to take advantage of the HTML5 stack. In the various VisProSL framework areas we are considering some alternatives, that we are carefully evaluating. In particular, we are currently using a bunch of HTML5 machinery, including:

1. **HTML5 File API** for the management of file-based project’s input/output;
2. **HTML5 Offline Application** for the use of the Vis-ProSL platform in the absence of connection;
3. **HTML5 Web Storage** for caching the resource data, that can be stored locally within the user’s browser;
4. **CSS3 Flexible Box Layout Module** for optimising the user interface layout.

4.2 JavaScript libraries

In this section we list the software resources we use in VisProSL. According to Wikipedia, “A library (from French ‘librairie’; Latin ‘liber’ = book) is an organized collection of resources made accessible to a defined community for reference or borrowing”.

1. **Cross-browser compatibility:**
   - **jQuery** — to simplify the client-side scripting of HTML, providing HTML document traversing, event handling, animating.

2. **Client-side MVC (Model View Controller) support:**
   - **Backbone.js** — to give structure to web applications by providing models with key-value binding and custom events, collections, and views.

3. **Manipulation of data structures (JSON Object and Array):**
   - **Lodash.js** — a utility library to deal with structured data, delivering performance, via custom build with only the needed features.

4. Handling of 2D vector graphics in web applications:
   - **Raphael.js** — small JavaScript library to simplify work with vector graphics on the web using SVG and VML, where every graphical object is also a DOM object, with possibly attached JavaScript event handlers.

5. Managing dependencies between modules (for details, the reader should refer to [11]):
   - **Brunch** — an assembler for HTML5 applications, agnostic to frameworks, libraries, programming, stylesheet and templating languages and backend technology.

6. Handling of AJAX request:
   - **SuperAgent** — a light-weight progressive ajax API crafted for flexibility, readability, and a low learning curve.

7. Feature Detection:
   - **Modernizer** — a JavaScript library that is able to detects HTML5 and CSS3 features in the user’s browser, to partly support legacy browsers.

5. SYSTEM ARCHITECTURE

VisProSL is configured as a Single-Page Web Application, using the Single-Page Interface (SPI) paradigm. Notice that in web applications without JavaScript, the state sequence is equivalent to the set of pages, whereas in a SPI application any change due to some interaction implies a new “state” of the single-page [14, 2]. Of course, the state of the application is fully owned by the client, and completely transparent to the servers of the REST transactions. According to the SPI model, the VisProSL architecture is strongly based on the Model-View-Controller pattern, declined into its client-side translation [12].
5.1 Proxy server

VisProSL is essentially a server-less web application. It runs completely on the client. But in order to allow cross-domain request, or in other words, to use third-party services, those services must be provided through mechanisms of resolution of the same origin policy \cite{same-origin-policy}. JSON-P \cite{json-p} or CORS \cite{cors}.

For third-party services that do not support such mechanisms it is necessary to provide a proxy server, in the same origin of VisProSL, which performs server-side requests and send back responses to the client.

5.2 Modeling a REST-like service

In VisProSL the main concept is the modeling of a REST-like service. Each such service is added to the VisProSL library through a description of it, expressed by means of a language of expressive power equivalent to WADL (Web Application Description Language), designed by W3C to provide a machine-processable description of a Web Application.

Each VisProSL service is modeled as a graphical box provided of inlets (input points) and outlets (output points). Inlet and Outlet points are connected by links that represent buses for data exchange. In particular, a data set is returned via an outlet by the service A as a JSON object and dispatched through a link towards an inlet of a service B. The visual model of a service is shown in Figure 2, where the service box is decorated with one inlet and two outlets, in order to distinguish the two alternative scenarios of either success or failure:

1. The output bus of JSON data objects is activated if the service request returns success (success outflow);
2. the alternate bus is activated when the service request returns failure (failure outflow);

![Figure 2: Modeling of a REST-like service.](image)

The overall execution flow of a macro-service (at enterprise level) is generally described by either a tree or a directed acyclic graph (DAG). More complex flow topologies are properly hidden from outside view and encapsulated into special services.

5.3 Linking, validation and integration of client-side services

The linking between two services is done via passing a JSON object. The integration of two services A and B is made by linking the success outlet of the service A with the inlet of the service B. The input/output description of each service guarantees that VisProSL may statically verify the compatibility of output data of service A with the input requested from service B. The validation consists in the comparison of JSON schemata of input and output objects on the two sides of a link. In order to abilitate the integration of non-compatible services, a special kind of transformer box is introduced, whose job is exactly the transformation of the input data structure, in order to match the data structure of the input requested by the service at the second end of the data link. The computational load required to operate these data transformations will be borne by the client.

![Figure 3: Transformer box.](image)

Data aggregation, where information is gathered and expressed in a summary form, may be modelled as a special multi-inlet/single-outlet transformer box, implementing a set of predefined “reduce” strategies.

![Figure 4: Multi-input transformer box](image)

5.4 Process communication

The linking of two boxes is always directed from an outlet towards an inlet, and not vice versa. In particular an inlet is connected to an outlet if interested to the flow of data passing through it. The action of connecting can be seen as a registration at the notification of an event:

1. the outlet is the event publisher;
2. the inlet is the event subscriber;
3. the release of a datum by the outlet is the triggering event;

when a funcbox releases an instance of its output, the outlet generates the event associated to the release of the datum, and notifies all the inlets interested to such datum.

5.4.1 Process registration and notification

Let us recall that within the JavaScript language the functions are first-class objects, and the scope is lexical, so that the functions have access to the context they are created within. This amazing language feature allow one to exploit the callback functions as connection apparatus.

In particular, the registration consists in passing a callback function from the inlet to the outlet. Conversely, the notification consists in the execution by the outlet of all the callback functions registered thereof. Also, the callback function employed in the registration maintains a full access to the context where it was created (i.e. to its closure) so transmitting its own knowledge to its publisher, that does not use a copy of it, but the function itself.
5.4.2 Triggering the funcbox computation

Each inlet has an associated callback function. The execution of the callback by the outlet provides the flow of the datum from the outlet to the corresponding inlet. Therefore, when a funcbox receives an input datum (via the inlet), verifies that all the other inputs are present and, if this predicate is true, applies to them the function associated to it (that we can call also box behavior). When the execution terminates, or when an item of the output stream has been generated, the funcbox will execute the callback functions registered on its outlet.

Figure 5: The action of connecting two Services via a link is a registration for receiving the notification of an event.

5.5 Input/Output DAG model

A VisProSL project can be defined as the description of a novel macro-service, graphically expressed as a directed acyclic graph with only one input (either a single JSON object or a single flow of JSON objects with the same structure) and only one output (as the above: either only one object, or a single flow of similar objects). A single source node (IN) allows to input the project data, and a single well node (OUT) is used to collect the project output. The simplest example of VisProSL project is shown in Figure 6.

Figure 6: Source and well nodes of a very simple VisProSL project.

5.6 Correctness inspection: testing mode

While developing a new VisProSL project, i.e. while designing a new macro-service by composition of pre-existing library services, it is of the utmost importance to be able to work in a test / debug mode, that would allow the designer to:

- verify the correct operation of the project;
- identify and remove any bugs in the execution flow;
- introduce test data to verify the correctness of service requests.

The insertion of the test data can be done in two ways:

- by reading a JSON document;
- by manually filling out a form that the system automatically generates from the description of the input required by the service connected to the source node.

Beyond the development and testing mode, useful debugging and/or understanding aids are provided even for normal operating modes. In particular, while in execution mode, the VisProSL application provides graphical tools for:

- access to input/output data of every box (services and transformations), in order to make correctness checks;
- in case of error, trace the flow of execution to have evidence of the point of execution failure.

5.7 User interface / User Experience

According to the design of the whole system, based on the Single-Page Interface (SPI) paradigm, the user interface is based on a user experience of single-window type.

5.7.1 UI structure

The user interface is structured as shown in Figure 7, and contains by default the following fields:

a) toolbar: Toolbar options;
b) workspace: Workspace available for project design;
c) library: List of available APIs for the project;
d) browser: List of the APIs used in the project:
e) inspector: Properties in the selected API function.

Figure 7: The default UI structure. It may be interactively restructured or partially hidden by the user, in case of need.

5.7.2 User experience

The user-system interaction occurs primarily through drag-and-drop. This type of interactive action is required for:

- insertion of new services — whose description is saved on a JavaScript file — located in the VisProSL Library;
- loading of pre-existing projects (saved in a configuration file);
• addition / removal of a service in the workspace (upgrading the project configuration file);
• addition / removal of a link between pair of services, i.e. configuring a service composition;
• arrangement of elements in the workspace (positioning and sizing of services);
• arrangement / sizing of sections of the layout.

6. EXECUTION MODEL

Once the dataflow DAG definition is finalised and the new service has successfully passed the testing stage, the project can be finally built and distributed. To allow for execution of built projects, a specialised execution Engine script must be included in the application page code.

6.1 Project building

Building a computer program is the process of collecting all the requirements and resources needed for execution. The output of the building phase of a VisProSL application is a JSON document organised as follow:

```json
{
  "services": [ ...
  ],
  "graph": [
  {
    "name": "service instance name",
    "type": "service type",
    "links": [
        "linked service instance name",
        ...
    ]
  }]
}
```

where:

• services is a list of descriptors of services utilised in the project being built;
• graph represents the project dataflow DAG as an array of node objects. Every node within the graph element represents a single service, and must specify the following information items:
  - name is the service instance name;
  - type is the type of the service (among those used in the project and listed in services);
  - links describes the service connections.

6.2 Service execution

To make the new service accessible, once it has been built, it can be distributed along with the Engine component. The Engine, which is a core component of the VisProSL architecture, consists of a specialized script to be included in the web pages in which the new service has to be made available. It implements an event driven execution model, which provides a solid communication mechanism while at the same time ensuring a complete decoupling of the Engine itself from the rest of the system (e.g. from the service editing and testing views user faces during new service composition). This design smartness allows the very same Engine to be employed in service editing as well as to be distributed along with the built project.

6.2.1 Execution Engine operation

The Engine executes a built project by instantiating an ExecNode for each node in the graph and by putting them in listening to each other’s particular events. An ExecNode is the fundamental execution entity, and represents a live version of a node in a built project. It can execute, listen for event firing, and fire events itself. More precisely, the responsibilities of an ExecNode are:

• generate the JavaScript Execution Function (or JEF) of the REST-like service wrapped by the node;
• trigger the node computation, once received the input;
• notify the end of computation, and passing forward the output;

To instantiate all the ExecNodes, the Engine needs a loop over all the nodes of the graph. Also, it needs a loop over all the node’s links to setup listeners and handlers. The execution time is then $O(n + e)$, where $n$ is the number of nodes and $e$ is the number of links in the project graph. The JavaScript core code of the Engine is demonstrated below:

```javascript
Engine.prototype.exec = function () {
  var execNodes = [];
  var links;
  this.graph.forEach(function (node) {
    var e = new ExecNode(node);
    execNodes.push(e);
  });
  var links = getAllLinks(execNodes);
  links.forEach(function (link) {
    var from = link.from;
    var to = link.to;
    from.on("complete", to.exec);
  });
};
```

6.2.2 Event lifecycle

As we stated before, the execution model is event driven. The event lifecycle implemented by each ExecNode is marked by the four events triggered, before, completed and failure.

Any entity in VisProSL that is interested in being informed about the trigger, start, conclusion or failure of a node execution, can listen to the respective event. It is what exactly happen, for example, in the test view: the UI listens to those events and displays the user with progress information about the execution of every node.

To understand in detail the event lifecycle, consider the outlet of a node $B$ connected to an inlet of a node $A$. The event lifecycle runs as follow:
1. node $B$ ends its computation and notifies all the interested nodes, and in particular the $A$ node — node $A$ fires the triggered event;

2. (a) node $A$ is multi-input, and it hasn’t already received a datum for each of its inlets — node $A$ rests in standby status;
   (b) node $A$ is neither multi-input, nor has already received a datum for each of its inlets — node $A$ fires the before event;

3. node $A$ executes;

4. (a) node $A$ correctly completes its computation and gets its results — node $A$ fires the completed event;
   (b) node $A$ does not successfully complete its computation because of kind of errors happened — node $A$ fires the failure event.

To start the execution of a built project, the Engine triggers all the nodes directly linked to a source (or input) node. Since every node has been registered to completed (or failure) event fired by its previous node, the correctness of execution is guaranteed.

### 6.2.3 Javascript Execution Function (JEF)

The main task of an ExecNode constructor is the definition of a function to wrap the underlying REST-like service, so ensuring the correct parametrization at runtime with the data expected from other services. We call this particular function Javascript Execution Function (or JEF). It could be considered as the “JavaScriptification” of a REST-like service, so it has to compose an HTTP request, handle the runtime input parametrization, perform the request, handle the HTTP response, manage the response data, as well as the success and error status-codes.

#### Runtime request parametrization.

The cornerstone in automatic generation of a JEF is the runtime parametrization, or rather how the function parameters have to be organized to automatically create and invoke such a function. Analyzing an HTTP request, it shows three parametrization points:

1. url templating
2. querystring
3. body

The first and the second parametrization points both accept data of the type key -> value. Conversely, any type of data can be sent within the packet body. We may think to combine homogeneous data joining key -> value parameters, and leave out the body parameters. In this way JEF comes to have the following signature:

```
function (parameters, body [, ...]) { ... }
```

where:

- **parameters** contains key -> value data. It is used to valorise the parametrization points 1. and 2. according to the service descriptor;

- **body** represents the data to be inserted in the request body.

The presence of additional parameters, besides parameters and body, points out the need to create a multipart request. In this case each function argument, starting from the second one, corresponds respectively to one of the body’s part.

Note that an AJAX request underlies each JEF. In other words, a JEF is intrinsically an asynchronous function, but no callback function appears in the argument list. The correct management of the application flow is ensured by the event lifecycle and by listener registrations operated by the Engine.

In addition, the last instruction of a JEF consists of the emission of one between the completed or failure events, to notify the end of the node computation, and to allow the output data to be passed forward to the next ExecNode.

### 7. CONCLUSION

This paper has discussed the design lines and the web technologies being used in the development of a novel tool for Visual Composition of REST-like services at enterprise level. This approach embraces a functional style of composition of web services, by using a stateless network of corporate services and (possibly location-based) web services, and supports a dataflow style, where JSON objects are properly transformed among the service links, suitably defined by the application designers.

A first module related to the visual programming of location-based services was already deployed through the SO-GEI company, that is a quite big state-owned company, and has produced the fast development of tens of location-based services among its state agency users. Our strong belief is that the increasing importance of client-side scripting and visual tools may provide real steps forward towards “nonbrittle software modules that plug together and just work”, especially for corporate workflows and mashups.

### 8. REFERENCES


