A Product-Line Architecture for Web Service-based Visual Composition of Web Applications

Marcel Karam[1] Sergiu Dascalu[2] Haidar Safa[1] Rami Santina[1] Zeina Koteich[1]
[1]*American University in Beirut, Lebanon* [2]*University of Nevada, Reno, USA*
marcel.karam@aub.edu.lb *dascalus@cse.unr.edu*

Abstract

A web service-based web application (WSbWA) is a collection of web services or reusable proven software parts that can be discovered and invoked using standard Internet protocols. The use of these web services in the development process of WSbWAs can help overcome many problems of software use, deployment and evolution. Although the cost-effective software engineering of WSbWAs is potentially a very rewarding area, not much work has been done to accomplish short time to market conditions by viewing and dealing with WSbWAs as software products that can be derived from a common infrastructure and assets with a captured specific abstraction in the domain. Both Product Line Engineering (PLE) and Agile Methods (AMs), albeit with different philosophies, are software engineering approaches that can significantly shorten the time to market and increase the quality of products. Using the PLE approach we built, at the domain engineering level, a WSbWA-specific lightweight product line architecture and combined it, at the application engineering level, with an Agile Method that uses a domain-specific visual language with direct manipulation and extraction capabilities of web services to perform customization and calibration of a product or WSBWA for a specific customer. To assess the effectiveness of our approach we designed and implemented a tool that we used to investigate the return on investment of the activities related to PLE and AMs. Details of our proposed approach, the related tool developed, and the experimental study performed are presented in this article together with a discussion of planned directions of future work.

1. Introduction

A web service (WS) is a reusable, extensible, platform and language independent component that is used over web protocols [1, 3, 4]. Stock quotes, weather updates, and currency converters – to name only a few – are examples of web services (WSs). Interface and technical details that are related to used protocols, web services' supported operations, and the packaging and exchanging of data, are described using the Web Services Description Language (WSDL) [1]. A WS, once published by a person or an organization, can be looked up from a services registry. Published and running WSs operations are bound to ports and run on hosts. Creating a composite WS requires the integration of existing published WSs that are structured in a workflow-like fashion, using WS composition languages such as the Business Process Execution Language for Web Services (BPEL4WS) [4] which is an XML-based language used to define a set of service partnerships and structured invocation schemes. For example, a complete traveling WS could come from the structured collaboration of the following services: travel agent, airline, hotel, car rental, entertainment, and billing. A newly composed WS can be described using WSDL, registered, and invoked as a new WS.

The task of engineering a web application, entirely or partially, from a set of WSs is considered crucial for an agile web application development process that, combined with the use of an appropriate reference architecture, can help overcome many problems of software use, deployment, and evolution. In this direction, the work in [3] relates product line models with WSs development. In the same direction, the work in [5] proposes an approach to integrate heterogeneous services developed under a service product lines using XML technologies, whereas the work in [6] presents a lightweight product line model with specific variability information to support the composition of WSs. With regard to web applications in general, very little work has been carried out in viewing such applications as software product-line. For example, in [7], the authors advocate the use of a lightweight product line as a suitable agile approach [8] for building web applications and improving the evolution of these systems. In [9] a specialized architecture, OOHDM-Java2 (a model-view-controller architectural extension) was introduced to develop web applications; however, since commonalities are very general in web applications, the system family

defined by OOHDM-Java2 is representative to a certain extent for the whole class of possible web applications. The work in [10] describes a product-line architecture for web applications that are obtained as the result of the composition of reusable components that carry suitable variability determination mechanisms which are assembled directly into the components, and thus allow for products to be instantiated and managed with a higher degree of flexibility by means of a domain-specific language [11].

A related research direction in service-based web applications emphasizes the creation of effective tools to describe and co-ordinate the composition of WSs. For example, the work in [12] led to a generic visual flow language for coordinating software components with a development tool tailored for WS composition where dataflow, execution sequence, and fault handling can be specified with a simple visual syntax. In [13], a domain-specific visual language was developed to support modeling complex interactions between WS components. A common drawback in these visual languages and environments is that they do not provide a reference and flexible infrastructure with reusable assets; rather, they operate at the visual code level, thus providing some agility measures for the orchestration of composed WSs.

In a separate but related direction of work, that of desktop applications, Ito and Tanaka [14] describe a tool that "wraps" parts of existing web applications that can be considered as WSs, and plug them onto a desktop application using the Intelligent Pad Architecture (IPA). The work in [15] extends the IPA to allow users to clip elements from existing applications and form cells on a spreadsheet, connect these cells using formulas, and clone the cells to provide the mechanism of handling multiple parallel requests.

Despite the fact that the use of WSs in web applications is a key factor in an agile web application development process, we could not find reports on a product line infrastructure that is combined with an integrated development visual environment to develop, with agility, these special kinds of applications that depend to a great extent on WSs. The work we present in this article is rooted in the body of some of the aforementioned related work. In particular, we used the lightweight product line model proposed in [6] and extended it to support a domain-specific visual language and environment at the application engineering level thus allowing us to perform agile calibration and customization of WSbWAs. At the domain engineering level, our approach includes the identification of commonalities and variability of

WSs in WSbWAs domain as well as the construction of a Model View Workflow (MVWf)-based framework that is instantiated to identify a specific product or WSbWA. At the application level, our approach supports agile methods, in particular by relying on a domain-specific visual language and environment with innovative extraction capabilities of WSs directly from web sites that are "imported" into our visual environment. This speeds up the development process by facilitating the composition and customization (or calibration) of a product or WSbWA for a specific customer.

There are considerable advantages to our combinatorial approach, and these advantages can be divided into three groups, as described next. The first group of advantages is related, at the domain engineering level, to the use of a flexible architecture modeled as an interrelated component hierarchy consisting of Page, Workflow, WSs (single and composite), Model, and View components and based on two high-level design grounds, the Model-View-Controller (MVC) design pattern [16, 17] and the workflow paradigm [18, 19]. At the application level, web engineers must extend these components to instantiate a WSbWA. In this way, reuse of components (pages, workflow, and WSs) is ensured and our product line domain engineering phase remains more or less the same as that of its conventional counterpart [6]. The second group is related to the product line application engineering level. In essence, the constructed architectural framework at the domain engineering level is augmented with a powerful graphical interface environment that allows a web engineer to visually construct WSbWAs and manage their dependencies as workflows in web pages. For example, given a specific business logic and user requirements, a web engineer can use the provided visual environment and domain specific visual syntax to perform agile creation, calibration, and variant bindings of WSs (with built-in variability handling mechanisms) and their workflows in a webpage. WSs are represented in the visual environment as visual WebPads constructs. More on these *WebPads* can be found in Section 2. Our approach, which allows to create and bind variant points of a WS in a workflow of a webpage in a WSbWA, relies on the ability of the web engineer to: (1) import a web page into the visual environment; (2) visually configure the *WebPad*'s variant information: input/output, exceptional handling, predetermined alias configuration in case the WS fails, and WS discovery for dynamically discovering and invoking a WS; and (3) visually

assemble in a workflow-like these *WebPads* with their views and controls. The third group of benefits is related to the overall "intuitive" nature of the approach for developing WSbWAs, which provides the compelling benefits of reducing the time, effort, and overall complexity involved in re-engineering these services. Furthermore, wrapping WSs as components with variant specifications in web applications also reduces the maintenance and upgrading involved in having such service integrated in one's own web application, since it is provided as an "off-the-shelf" service.

This article describes our approach that takes advantage of the factors mentioned above and combines a lightweight product line engineering method with agile techniques that are provided at the application level using a visual tool and environment we call *VisualWebC* (short for *Visual Web Composition*) to support the visual composition of WSs. The remainder of this article is organized as follows: Section 2 describes our WSbWA-specific Product Line Architecture (PLA), its architectural component hierarchy, and the *WebPad* component and its variants; Section 3 describes our prototype (*VisualWebC*) and its visual authoring support for WSs; Section 4 presents a case study that was conducted to evaluate our approach; Section 5 presents a discussion regarding our overall approach; and Section 6 rounds up the article with pointers to future work and several concluding remarks.

2. A WSbWA-Specific Product Line Architecture

One defining characteristic of Product Line Engineering (PLE) is represented by a key dual software process, consisting of *domain engineering*, which establishes and realizes the commonality and the variability of the product line and a supporting reference architecture construction, and *application engineering,* which draws on the strengths of the product line and builds software applications that reuse common domain artifacts while also ensuring that required product variability is satisfied [20, 21, 22]. In this section we explain our design approach to a lightweight PLE, in particular the architectural component hierarchy of the WSbWA-specific PLA as well as the *WebPad*'s architecture and the issues related to the implementation of its variants.

2.1 MVWf Architectural Component Hierarchy

Calling different software pieces or WSs at run-time to accomplish the functionality specified in the requirements of a business process adds great flexibility in the development and maintenance of WSbWAs, and shortens their time to market. Since these WSbWAs are expected to be used in similar environments to fulfill similar tasks, it is appropriate to design them as members of a product family [6]. One way to achieve this is to create a "web services-oriented" reference architecture that is based on an architectural paradigm with adequate flexibility to support the creation of business processes as workflows containing a set of simple and/or composed WSs, thus facilitating workflow and WSs development and reuse. This architecture should help developers maintain the state of each business process or workflow to create a system that is easier to maintain and augment. Finally, the architecture should also benefit from mature testing methodologies that can be applied to each workflow separately, resulting in higher quality and re-usable workflows and their WSs.

As previously mentioned, to facilitate the development, maintenance, and evolution of WSbWAs we modified, as depicted in Figure 1, the lightweight PLE model that was originally proposed in [6] to support the construction of a reference WSbWA-PLA that is based on two major design concepts, the MVC pattern and the workflow paradigm. Many commercial and research frameworks such as Ruby on Rails [23] and Struts [24] support a combination flavor of these paradigms. The MVC development approach is based, in general, on the MVC design pattern. In this pattern, the Model component encapsulates data manipulation, extraction, and storage procedures. The View component handles the data and user interface rendering. The Controller component implements the flow of control in the application by decoding the events generated by the View and invoking the appropriate Model operations. In general, MVC-based frameworks build on the idea of having one global controller for all pages in an application. With the workflow paradigm, the development approach represents a business process as a workflow that maintains its state and uses the current state along with the received event to direct the flow of control within the application.

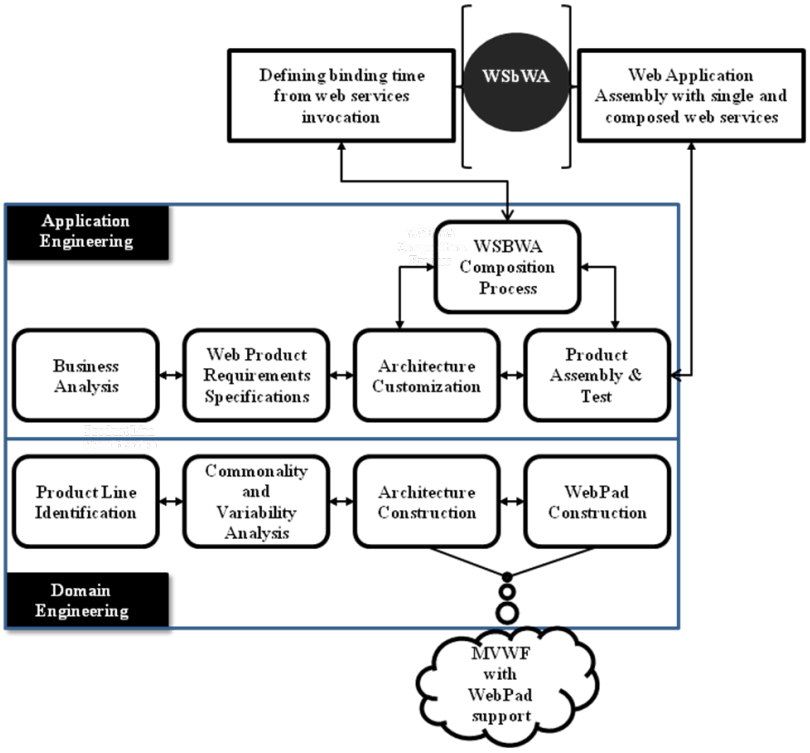

Figure 1 – The lightweight Product Line model adapted from [6] and extended to support our approach.

The workflow and MVC paradigms can be combined to create a Model-View-Workflow (MVWf) paradigm [19] in which one controller exists for each procedural component (traditionally referred to as a set of related pages). In the MVWf framework, the workflow's graph consists of a set of action (Model) and result (View) nodes. The action nodes access and modify the Model in the MVC pattern whereas the result nodes are the Views in the MVC pattern that are rendered by the browser. In this context, the workflow engine embeds the Controller of the procedural component that now behaves as a separate mini-application. For reuse purposes and to keep workflows confined in one webpage, our PLA takes advantage of the MVWf paradigm to further restrict a business process to a single page thus allowing several business processes or workflows to exist within that page. As such, a web page becomes simply the rendering engine of the workflows, containing, among other things, a set of related simple or composed WSs that reside within the page. As shown in Figure 2, the PLA of *VisualWebC* is mainly based on a five-component hierarchy or layer: Page, Workflow, Model (Action), View, and Behavior&Layout. For the sake of simplicity, in Figure 2 we refer to each component in this hierarchy as:

p, *wf*, *m*, *v*, and *b&l,* respectively. Developing a web page with a set of workflow processes in a WSbWA involves the instantiation and interaction of all components in the hierarchy depicted in Figure 2. More formally, a web page $\boldsymbol{\Psi}$ in a WSbWA developed using a framework implementing our *VisualWebC's* MVWf paradigm is denoted $\boldsymbol{\Phi}_{(WSbWA)} = WF = \{wf_1, wf_2, ..., wf_n\}$, where each $wf_i \in WF$ is a workflow process that is used to develop $\boldsymbol{\Psi}$ and is associated with two sets of components: Views $V = \{v_1, v_2, ..., v_n\}$ and Models $M = \{m_1, m_2, ..., m_n\}$. The Behavior&Layout component delimits the specifications and behavioral constraints of all fields present within a View component. As such, a View component becomes a collection of fields (stored in forms), the behavioral definitions of which being stored within the appropriate Behavior&Layout component layers.

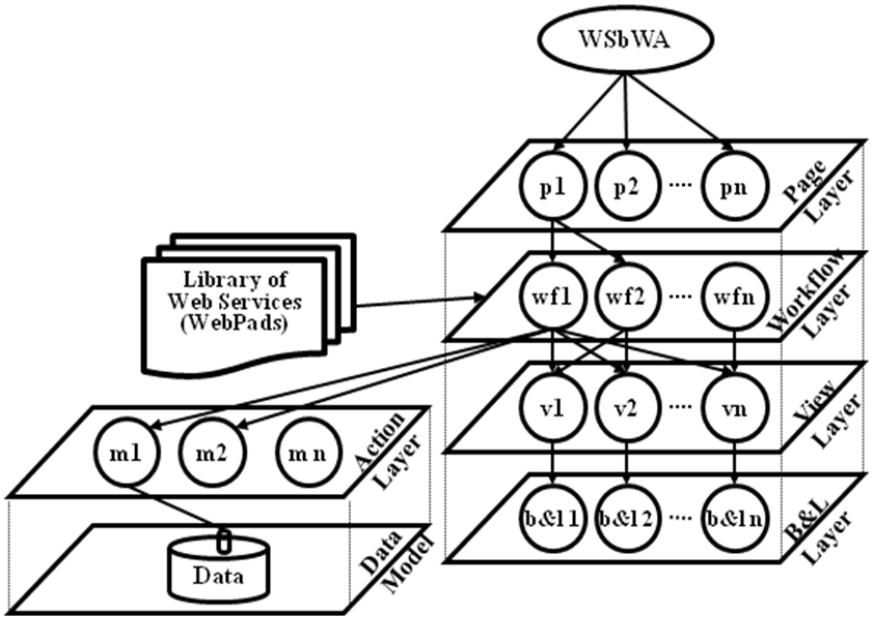

Figure 2 — Abstract component hierarchy of *VisualWebC*.

As depicted in Figure 2, a WSbWA can be viewed as an association of pages, their workflows, and their WS components (the latter as a library of stored *WebPads* in workflows). In particular, WSbWA is a product consisting of a number of pages, each of which containing a number of workflows that collect contents dynamically provided by the selected WS components. The WS components in a workflow are designed to capture both the commonalities and variabilities of a class of behavior in an application domain. A WS component is a bound component as soon as some of its variant methods are selected in

the product. As previously mentioned, WS components are represented in our visual environment as *WebPads*. Variant points for our *WebPads*, as we shall see in Section 2.2, can be visually resolved, and include: WS selection, WS input and output data, WS aliases, exception handling, and quality factors. Unlike in typical WS composition, in our approach the variability determination and management capabilities can be visually accomplished during the product instantiation phase using the visual environment and the direct extraction capabilities of *WebPads* as WSs from imported web pages. This is one of the major contributions of our approach. We next explain the design and implementation of a *WebPad*.

2.2 The WebPad Component

To provide a way to visually configure and assemble readily available and tested WSs into a WSbWA, we modified the desktop-based Intelligent Pad Architecture presented in [14] to create a generic component, called *WebPad,* with variants that the web engineer needs to bind during the configuration of a *WebPad* in a workflow representing a business process. Possible values for these variants are:

- WS selection — the URL address of the webpage from which a WS is chosen;
- WS input and output data — the XPaths in the DOM representation of the webpage containing the input and output of the WS;
- Exception handling — the URL to display or page/view to load when exceptions are encountered as a result of an unknown problem;
- Quality factors — a preference WS that has been statistically proven in terms of speed and reliability;
- WS aliases — all the above information is required to be bound to replace a failed WS at run-time, thus making a WSbWA more resilient to future changes and failure. In essence, to deal with the possibility of having failed remote services, the web developer is asked to specify one or more aliases for a remote service in a *WebPad*. The alias is a website (URL and input-output object and

their XPaths) that provides a similar functionality as the original remote component. The developer can specify as many aliases as needed in an effort to reduce the number of possible "error" occurrences.

WebPads are designed to interact with each other only through input and output ports. Data coming into a *WebPad* port can be directly or indirectly sourced from a user's input or another *WebPad*. Much of our implementation of the *WebPad* architecture relies on the Document Object Model (DOM) [25], which is an application programming interface (API) for HTML and XML documents that structures the content of a document as a tree. DOM-based techniques include identification of objects within HTML documents and the browser's API ability to reference the components, manipulate their content, and trigger their associated methods. To assign the XPath of an input object or an output object of a WS from an imported webpage to a *WebPad*, the authoring environment of *VisualWebC* first extracts the DOM representation of the imported webpage then uses it to create a transparent and interactive layer on top of the imported webpage to allow a web engineer to select input and output objects, extract their XPath, and assign those XPaths to the *WebPad*. The assignment of the XPaths to the input and output object can be done by lassoing the input or output object and associating it to an input or output *WebPad* port, respectively.

The run-time environment of a *WebPad* is implemented as a Dynamic Link Library (DLL) that encapsulates its information in a workflow and ensures that a WS is triggered and results are generated. The run-time environment takes the input that triggers a specific workflow, along with the URL address of a certain website and returns the output as specified by the user. To process the input of the user when WS is triggered in a workflow, the DLL creates a thread *t* to: (1) retrieve the URL of the webpage containing the WS and opens the webpage contents; (2) access the DOM tree representation of the website and check if the XPath specified for an input object *X* (in the *WebPad*) is isomorphic to that of the webpage; and (3) generate an assignment statement such as: *Set X.text = local_textBox.text* (where *local_textBox* is a textbox in which the user enters data); append the DOM tree with new value of *X*; and generate a 'submit' action to submit the form on the webpage. In case the page fails to open successfully,

then *t* forks a process to repeat, using the bound alias variants, the three steps described above. In case an alias fails or is not specified, an error page is displayed. To retrieve the output of a called WS and display the result in the WSbWA, a thread *t* is created to: (1) retrieve the XPath of the output, or the regular expression of the XPath, and wait until the page containing the WS is loaded or until the output specified by the XPath is ready; (2) retrieve the data of the objects specified by the XPath, and generate assignment statements such as *Set local_outputBox.text = retrieved text*; and (3) close the connection to the website and update the local page/view. In case an error occurs while loading the page, an error page is displayed.

This design solution ensures that the code responsible for connecting to remote web applications will be reused for each application, which in turn reduces the overhead of regenerating this code for every *WebPad*. While this design solution ensures accurate invocation of various WSs, it does however create an overhead since every *WebPad* in a workflow that requires that its input comes from a WS has to wait. This delay is due to the time needed to process each web request at the remote server. Additional *WebPad* variants such as automatic discovery of a WS can be added to our variant implementation since it provides a way to improve the chances of the survival of a composed WS (in case of a failure). The variant points in this case should be associated with the WS static lookup and dynamic discovery using WSDL and services registry, respectively. WS publishing and dynamic discovery techniques are very promising XML-based technologies and we expect that they will neither be complicated to integrate into our PLA nor difficult to provide visual representations for their variant binding process at the visual environment level. This is in fact part of our ongoing enhancement of *VisualWebC's* WS discovery.

At the page and workflow levels of the architecture presented in Figure 2 other component variants can also be defined. These variants at the workflow level can be considered as the orchestration model of the overall workflow, while at the page level variants can be defined in terms of any "variable" that a page can use and process.

2.2.1 WebPad Variants and Their Implementation Issues

We describe next implementation issues pertaining to two *WebPad* variants: *input and output* and *aliases* types which are used to handle different kinds of inputs and outputs to cover a large portion of the possible components that need to be taken from a WS. We next explain each of them:

Types of Inputs and Outputs — we have devised a method for extracting the output of a remote functionality (service) as a list of results. A good example to illustrate this feature is a search engine where the user needs to specify all the results as output in order to display all of them on his or her webpage. Thus, in order to reduce the overhead, we provided the user with the facility to retrieve the list of results by specifying the output of the component as of type "list" and giving the user the option to wrap only the first two results and specify as "sub-variant" the number of results that need to be wrapped for the output display. Our implementation internally applies a string-matching algorithm to retrieve the difference in the HTML paths of the results, uses this difference to compute the HTML path of the remaining results in the list, and addresses the remote elements with the computed HTML paths. Currently, our prototype handles several types of inputs and outputs. An input/output can be almost any element of a website, including a text field, text area, label, button, and so forth. However, unsupported elements can be very easily integrated, as most inherit from an abstract GUI class.

Aliases — the generation of an alias is partially automatic to assist the developer and eliminate the overhead of linking, creating, and re-initializing the alias as the original. As we have noticed in our experimentation with the framework, this approach minimizes the errors that might occur during the re-initialization process.

3. Visual Authoring Support for Web Services

According to the Agile Manifesto [26] it is of highest priority to satisfy customers with early and continuous delivery of valuable software. Key concepts in agile methods to quickly develop running software and reduce time to market are short, time-boxed iterations, short feedback cycles within the development team and with the customer, continuous integration, and automated regression tests. Our

visual environment in *VisualWebC* implements a visual code-centric approach that provides time-boxed iterations with short feedback cycles that allows a web engineer to quickly assemble a WSbWA and observe some feedback.

To better understand our visual environment and consequently facilitate the discussion of our innovative method to visual composition of WSs, we next introduce some of the visual syntax and constructs and describe the main interface areas of *VisualWebC*. Then, we describe using an example the visual composition of WSs.

3.1 Visual Syntax and User Interface

VisualWebC has several visual key constructs and features that help the user rapidly develop a WSbWA. The visual constructs shown in Figure 3 are some of the building blocks currently available in the environment. They represent, from left to right: the *WebPad,* which is used to construct simple WSs; the *MultiTaskWebPad,* which is used to represent a service that requires more then one phase to provide the input to a WS; the *ComposedWebService WebPad,* which is used to wrap many other *WebPads* and create complex WSs; a *conditional operator,* which is used to define conditional dependencies in a workflow between various visual constructs; an *output GUI element table* (other GUI elements are also available); and a *connector*, which is used to connect visual constructs and define their dependency relationships. Circles on the left and right sides of these visual constructs represent the data input and output ports, respectively.

Figure 3 — A Subset of the visual constructs in *VisualWebC*.

The example we use to illustrate the main interface areas and capabilities of *VisualWebC* involves a set of requirements that are given to a web engineer by an investor who wants to monitor from his or her web application stock quotes of some US companies in Japanese yen. For the sake of brevity, we will simply refer to this example hereafter as the "running example". Based on these requirements, the web

engineer, after importing and browsing a couple of financial sites, realizes that: (1) *Quote.com* provides real-time stock-price browsing service of US companies in US dollars; and (2) *Yahoo.finance.com* provides a service that converts US dollars into Japanese yen based on the current exchange rate.

As depicted in Figure 4, *VisualWebC* has four major interface areas that have been annotated with circled letters *A*, *B*, *C*, and *D*. The interface area *A* is a tabbed web browser that allows the web engineer to import and navigate many web sites looking for a particular WS that satisfies part of the requirements. For example, in Figure 4 area *A* shows two tabbed web sites: one for *new.quote.com* and the other for *yahoo.financial.com*. Once a website's webpage is imported into a tab and a WS of interest is found the webpage's DOM is extracted and used to build a transparent layer that is superimposed on the imported webpage. This transparent layer allows the developer to lasso a point of interest (input or output of WS) and assign it to the input or output of a *WebPad*, respectively.

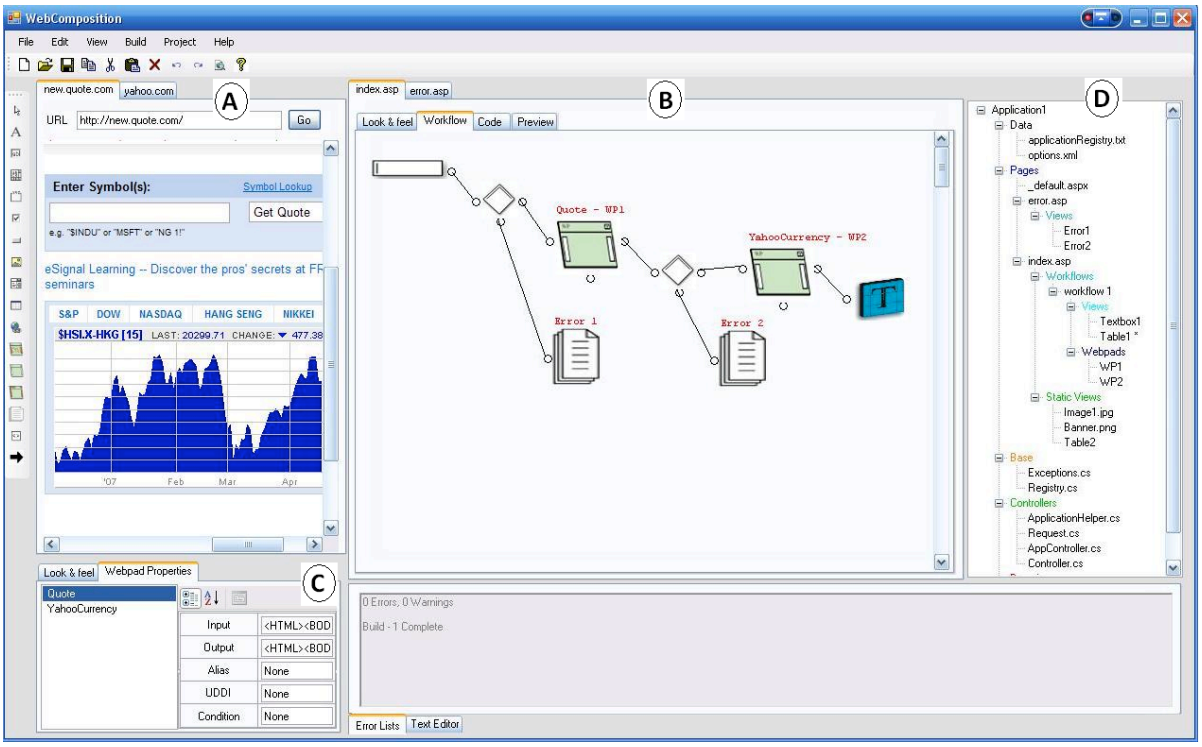

Figure 4 — A Sample project showing the composed currency exchange web service.

The interface area *B* is a double-layered tab structure that contains, for each page file in the *Pages* tree structure of interface area *D*, four associated tabs: Look&feel, Workflows, Code, and Preview. The Look&feel tab is where the user can visually arrange on a canvas the GUI components of a web page and its WSs elements. The Workflow tab is where the developer visually composes single or complex WSs. For example, the composition process of a simple WS involves the creation of a workflow that comprises a set of interconnected *WebPads*. To create a *WebPad*, the user right clicks in the Workflow tab and chooses to add a new *WebPad* or simply drag and drop it (from the toolbar) into the Workflow tab. Once a *WebPad* is created the user can then proceed to visually configure its variants. Variant points such as the input and/or output can be first determined either from interface area *A* by choosing their appropriate paths from the DOM representation of the designated webpage or from the output of another *WebPad*. For example, in our running example the developer locates the *quote.com* website, chooses the input and then assigns it to the *WebPad* labeled "Quote–WP1". The assignment is performed by connecting the input port of Quote-WP1 to a GUI input element *Textbox1* that now represents the path *p* in the DOM of *quote.com*. The developer then connects the output of Quote-WP1 to the input port of another *WebPad*, labeled "YahooCurrency–WP2". To satisfy the client's requirements, the developer then locates the *yahoo.com* website, chooses the output, and then assigns it to the output GUI element *Table1*. Both the input and output (*Textbox1* and *Table1*) will then be automatically generated in the Look&feel tab of the interface area *B*. The conditional constructs between the various components in Figure 4 are designed to satisfy certain constrains in the requirements. For example, the data between the input box and WP1 is checked to see if it satisfies a certain format condition that is required as the input to the next visual construct. The Code tab provides a conventional text editor for the webpage. The Preview tab is where the user can view the result of the composition and experiment with it.

The interface area *C* is a tow-tabbed structure that contains the GUI elements in the look & feel tab and *WebPad* properties in the WebPad Properties tab. The GUI Elements, as the name indicates, gives the user access to GUI elements such as *textboxes*, *tables*, *labels*, and *buttons* that can be dragged and dropped into the Look&feel tab of interface area *B*. The WebPad Properties tab allows the user to

instantiate a *WebPad*, define some of its variant points, and drag and drop it into the Workflows tab of interface area *B*.

Different types of dependencies can exist between the *WebPads* that we are wrapping from different web applications. Some *WebPads* might depend on an input from a local component while some other might depend on an output from another *WebPad*. Let us consider in Figure 4 *Quote-WP1* of *www.quote.com*, and, *YahooCurrency–WP2* of *finance.yahoo.com/currency*. *Quote–WP1* depends on an input from a GUI component (*Textbox1*) and *YahooCurrency-WP2* depends on the output of *Quote-WP1*. Such dependencies are usually hard to track in a non visual compositing approach. However, our *VisualWebC*'s representation of the links between components in a workflow gives the user the ability to visualize the dependencies and ensure at run time that if an error occurred (e.g., within *YahooCurrency-WP2*) its dependencies can be easily tracked and fixed. This solution also leads to several other benefits, including: (1) visualizing the flow of information between different elements in different websites; (2) identifying problems in case of data flow errors or dependency failures; (3) tracking the effect of changes in the format of data; and (4) providing the user with a way to easily add new functionalities, given that input/output dependencies and their types are available.

Generated pages and views are rendered according to a very simple design strategy that displays their elements in an ordered manner. To enhance and enrich our experimental studies, we provided the users with a way of importing an existing page (HTML or ASP), loading it into our system, and using it when generating the web application. When a page containing design elements is imported our environment uses the page's DOM representation to first identify its different elements (*textboxes*, *buttons*, etc.) and then redefine their spatial relationships with existing elements. Imported pages do not affect the model of the website that is generated; they may change only the display of the elements.

3.2 Agility and Visual Composition of Workflows and Web Services

Our approach in *VisualWebC* leverages the modularity of our WSbWA-specific PLA and its underlying MVWf paradigm to provide a visual environment and syntax for creating, manipulating and

reusing – with notable agility – pages, their workflows, and the workflows' associated WSs to instantiate products or WSbWAs. From this perspective, the framework can be seen as a black box-like architecture that allows web engineers to use agile techniques in the visual creation and calibration of WSbWAs. In the context of our running example, we describe next the interactions between the visual constructs at the visual coding level as well as the WSbWA-specific PLA components. The example is deliberately kept simple to facilitate the overall discussion and illustrate agility in visual composition, component reuse, and collaborative aspects of our WSbWA-based PLA.

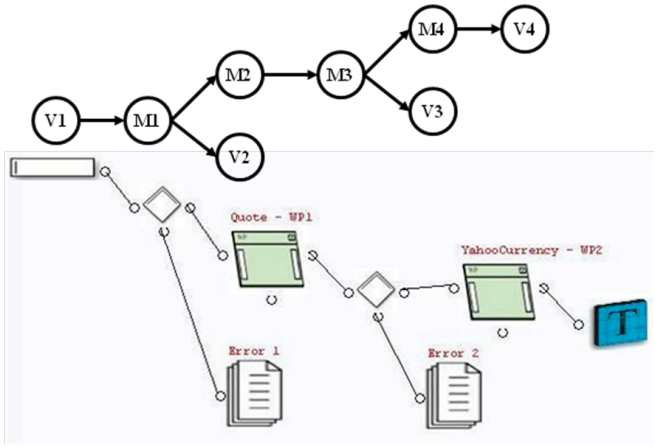

Figure 5 — The visual syntax and workflow graph of the currency exchange example.

In our running example, the web engineer implements the requirements with only one business process or workflow within one page, *index.asp*, which is a container comprising one workflow component that runs independently within the page. In the Workflows tab, the web engineer right clicks and instantiates a workflow component or *WF1* then assembles its elements and binds their variants. The visual syntax satisfying the requirements of our running example as well as the syntax's conceptual workflow graph are depicted in Figure 5. In the workflow graph, the node labeled *V1* represents the state of the "textbox" before a new event is generated by an "enter" action that creates a transition from node *V1* to node *M1* where some conditional action is performed on the data coming from *V1* or the output port of the "textbox". The conditional action results in two possible transitions, true and false, one to *M2* or the input port of *Quote-WP1*, and the other to node *V2*. A conditional action is also performed on the data

coming out of node *M2* or the output port of *Quote-WP1* which results in two possible transitions, true and false, one to node *M3* or the input port of *YahooCurrency-WP2*, and the other to node *V3*. Finally, the data from *M3* or the output port of *YahooCurrency-WP2* is passed to the *Table T*. Since each workflow maintains its own state, workflow reuse is possible and the state of a page becomes the collective state of its workflows. A workflow appears to a user as one of its View nodes where the workflow is pending. The page then appears to the user as a collection of pending states – the nodes where the page's workflows are pending. The events that can be triggered in a View node are those attached to the GUI element within this node. The events attached to each GUI element are defined within the Layout&Behavior component of the GUI element. In our running example, after entering the data into the text box, the user can trigger a "pass data" event (by pressing the Enter key) and move to the first conditional action. The same principle applies to all node and link definitions within the workflow.

The View component groups the GUI elements used within a business process so that a browser can render them based on the specifications in their GUI Layout&Behavior components. In our running example, the web engineer instantiates a View component called *V1*. The grouping of GUI elements manifests itself as a set of GUI elements belonging to the same HTML form in a page. A View component (text box or table in our running example) consists of the visible entities that the user interacts with as part of a business process. The Model component embodies an activity that manipulates the web application's data. It implements functional procedures over the application's data model or performs operations in the context of the application. To perform its function, the model component can access the data model content and the GUI elements presented to the user through the web browser. The Layout&Behavior component defines both the layout (markup) and behavioral specifications of the field. The markup specifications define how the data in a GUI element is rendered in the web page. The behavioral specifications define how a GUI component behaves in the web page. They also define the events triggered during a user's interaction with the GUI component.

To specify possible events that might be used for transitioning between Views and between Models at the visual syntax level, the developer is given a straightforward selection facility to choose the element

that triggers the transition (e.g., “on-text-changed”, “on-failure-of-remote-data”, “on-button-click”, etc.). These events are then identified as XML tags in the GUI layout and Behavior XML files.

4. Case Study and Lessons Learned

To obtain meaningful information about the effectiveness of our approach in: (1) providing a reasonable amount of agility in the visual composition and management of WSs compared with the more traditional XML-based orchestration languages for composing and managing WSs; and (2) reusing WSs and workflows to satisfy new requirements, we conducted a short study with two scenarios. The first scenario required web engineers to deliver five products, each of which containing a certain number of workflows and their appropriate web services. The second scenario required these web engineers to handle requirements changes in of these products. We next describe the design setup, results, and lessons learned.

4.1 Experimental Design and Setup

In setting up the design of our case study, we asked a group of graduate students who are not familiar with the ongoing study to build, using the .NET Framework©, seven web applications and place them on different web servers in our lab. The Service Provider (SP) number, name, and the description of the services it offers, as well as the WS number are shown in the header of Table 1.

The SP names are used for illustrative purposes and are not part of any online application. The SP number and WS number in the first and, respectively, third column of the table are used to facilitate further referencing.

The graduate students were then asked to assemble the components in column 2 of Table 2 to create a framework to describe, register, and publish the WSs shown in column 2 of Table 1. The framework’s full components and their technological specifications are shown in Table 2. We next assembled two groups of five teams, each of which containing one graduate student and one senior undergraduate student. The graduate students were chosen after completing a term project in a graduate course that focused on the creation and management of substantial MVWf-based web applications. The senior

undergraduate students were a mix of intermediate and advanced programmers. As part of our training sessions, we trained the first group (the VGroup) on how to use *VisualWebC*, and provided a thorough understanding of its MVWf architectural model and visual syntax and environment. We then trained the second group (the TGroup) on only the MVWf architectural model of *VisualWebC*, leaving out its visual syntax and environment. We also provided training for the TGroup on the use of the WS composition language BPEL4WS [4]. Each team of pair programmers across the two groups can be considered advanced developers of web environments and proficient in related scripting languages, especially in HTML, XML, ASP, and PHP.

Table 1 – The WSs offered by the service provider's web sites.

SP#	Service Provider (SP) name	WS number	WSs Description
SP_1	Department of Motor Vehicle	(WS_1)	Get Driving Abstract
		(WS_2)	Validate Address Info
SP_2	Credit Bureau	(WS_3)	Get Credit Report
		(WS_4)	Validate SSN
SP_3	Security Services	(WS_5)	Get Security Clearance
SP_4	Immigration Services	(WS_6)	Get Citizenship Status
		(WS_7)	Get Work Permit Status
		(WS_8)	Validate SSN
SP_5	Car Rental	(WS_9)	Book a Car
		(WS_{10})	Return a Car
SP_6	Hotel	(WS_{11})	Book a Room
		(WS_{12})	Check In
		(WS_{13})	Check Out
SP_7	Weather	(WS_{14})	Get 24 hour-update
		(WS_{15})	Get 5-day-update

Table 2 – The WS Framework for describing, registering and publishing WSs.

Average Time/requirements change	Average successful requirements change
Java SDK and Runtime environment	Sun Java 2 SDK 1.4.0_02
Application Server	Sun Java Enterprise edition (J2EE) 1.3.1
Transport Facility that Processes SOAP Messages	Apache Tomcat 4.1.31
XML Parser	Apache Axis 1.2 RC 1
Build Tool	Apache Ant 1.6.5 – Apache Xerces-J 2.5.0
Framework for Accessing UDDI within Java	UDDI4J 2.0.4
Database Engine	MySQL 4
UDDI Registry	jUDDI 0.9.4

Scenario 1 — Agility and the Visual Syntax. In this scenario, each team in both the VGroup and the TGroup were asked to deliver five products. Each product consisted of a set of workflows, each of which containing a combination of the WSs depicted in Table 1. The number of workflows and their number of web services for each product are shown in Table 3. In total, there were 17 workflows and 54 WSs.

Table 3 – The number of workflows and their web services in each product.

Product #	Number of workflows	Number of web services in each product
Product 1	3	12
Product 2	5	14
Product 3	2	7
Product 4	3	11
Product 5	4	10

One product example is a web application for a head hunter (or Product 1 in Table 3) that requires accessing certain records from various web sites to determine a potential employee's credentials. The requirements for this product example asked that the product consists of three simple web pages, each of which providing the headhunter with a different business process. Table 4 summarizes the requirements for the headhunter product example (Product 1) by detailing the business processes (workflows and their required WSs) for each of the three web pages involved.

Table 4 – The requirements for the headhunter product example in Scenario 1.

Page1/Workflow 1	Page2/Workflow 2	Page3/Workflow 3
WS_3,$WS_4(SP_2)$, WS_5, and WS_7,	$WS_4(SP_2)$, WS_6, and WS_7	WS_1,WS_3, $WS_4(SP_2)$, WS_6, and WS_7

Every team in the VGroup was asked to deliver the five products using *VisualWebC*; whereas, every team in the TGroup was asked to use the underlying MVWf architectural framework of *VisualWebC* (but not its visual environment) to deliver the products using the traditional code-based approach.

For each team in each group, we recorded for each product: (1) the time it took a team to finish a workflow successfully; and (2) the number of successful workflows. We then calculated for each team in each group the average time and successful workflows across the total number of products. Table 5 and Table 6 show the average time and the average success for each workflow across all products for both the VGroup and TGroup.

As shown in Tables 5 and 6, it was noted that all five teams in the VGroup had a remarkable time advantage over the teams in the TGroup. Although the workflows in this scenario were straightforward, the agility that was exhibited in the creation and calibration of the workflows and their *WebPads* by the VGroup was noteworthy.

Table 5 – Average time and average successful workflows across the delivered products for each team in the VGroup.

VGroup	**Average time/ workflow across all products** [min]	**Average success/ workflow across all products** [%]
T1	4.4	89%
T2	6.2	92%
T3	5.8	89%
T4	5.4	96%
T5	4.5	87%

Table 6 – Average time and average successful workflows across the delivered products for each team in the TGroup.

TGroup	**Average time/ workflow across all products** [min]	**Average success/ workflow across all products** [%]
T1	10.8	95%
T2	11.5	82%
T3	8.8	86%
T4	12.4	92%
T5	14.5	81%

Scenario 2 — Reuse and Agility. In this scenario, both groups were asked to implement requirements changes in the original products. These requirement changes mainly asked for change in the WS source, reuse of composed web services, and workflows. For example, a sample requirement change in the

headhunter product example stated that all three workflows should be placed in one page, and the workflow [WS_4 (SP_4), WS_6, and WS_7] be used as a complex composed WS such that the value of WS_4 be obtained from SP_2 instead of SP_4. Table 7 shows the new requirements in its first column. In the second column it shows a complex composed web service ($CCWS_1$), which can be formed by creating a WS from the second workflow in the first column of Table 7.

Table 7 – Changes in new requirements of the headhunter product example in Scenario 2.

Workflows/Page1	**Complex Composed WS**	**Resulting Workflows**
WS_1, WS_3,$WS_4(SP_2)$, WS_5, and WS_7,		WS_1, WS_3,$WS_8(SP_4)$, and WS_7,
$WS_4(SP_4)$, WS_6, and WS_7	$CCWS_1$	$WS_8(SP_4)$, WS_6, and WS_7
WS_1,WS_3, $WS_4(SP_2)$, WS_6, and WS_7		WS_1,WS_3, and CCWS1

For each team in each group, we recorded for each product: (1) the time it took a team to successfully implement a requirement change; and (2) the number of successful requirement changes. We then calculated for each team in each group, the average time and average successful requirement changes across the total number of products. The collected data is shown in Table 8 for the VGroup and in Table 9 for the TGroup.

Based on Tables 8 and 9, it was noted that all five teams in the VGroup had a remarkable time advantage and agility, especially in the visual composition of these workflows and in properly satisfying the new requirements.

It was also noted that, when asked to make changes to satisfy the new requirements, the TGroup had more difficulty implementing (at code level) a solution for the new requirements; whereas, it was remarkably easier for the VGroup to provide such solution, since they could quickly "adjust" the design of the solution to meet the new requirements (by using simple drag and drop movements within the environment's visual interface). The creation of the workflow was particularly difficult to implement for the TGroup. The VGroup however found this to be a simple task of cut, copy and paste, and showed

remarkable agility in the calibration of *WebPad* variants and in the customization of Scenario 2 workflows.

Table 8 – Average time and average successful requirements changes across all products in the VGroup.

VGroup	**Average time/ requirements change** [min]	**Average successful requirements change** [%]
T1	6.4	100%
T2	7.2	95%
T3	5.8	96%
T4	8.4	96%
T5	6.5	100%

Table 9 – Average time and average successful requirements changes across all products in the TGroup.

TGroup	**Average time/ requirements change** [min]	**Average successful requirements change** [%]
T1	15.8	85%
T2	14.5	72%
T3	11.8	66%
T4	16.4	82%
T5	19.5	73%

4. 1 Threat to Validity

We discussed a case study conducted to develop a better understanding of our combined approach, extract a number of empirical observations, and obtain several preliminary results that could indicate the degree of agility in our proposed approach. Our informal analysis, based on a limited set of original requirements and changes in those requirements, showed some promising return on investment. To address a limited set of agile questions, we addresses the agility indexes listed in the second column of Table 10. As columns 3 and 4 of this table indicate, there are many unknowns that are still yet to be determined. Therefore, no definite conclusion should be drawn from our case study. As such, we believe that to better understand the strengths and weaknesses of our approach a larger, more comprehensive set of formal empirical studies needs to be conducted over time. In fact, this is precisely one of our main directions of future work.

Table 10 – Some agility indexes, their description, and their values in both groups.

Agility Index	Description	VGroup	TGroup
Flexibility	Does the approach accommodate expected or unexpected changes?	Yes	Yes
Speed	Does the approach produce results quickly?	Yes	No
Leanness	Does the approach follow simple and quality instruments for production?	Yes	No
Learning	Does the approach apply updated prior knowledge and experience to learn?	N/A	N/A
Responsiveness	Does the approach exhibit sensitiveness?	N/A	N/A
Responding to change over following a plan	Does the approach value responding to change over following a plan?	Yes	Yes
Keeping the process agile	Does the approach help in keeping the process agile?	Yes	No
Keeping the process cost effective	Does the approach help in keeping the process cost effective?	Yes	No

5. Discussion

While operating essentially as agile (quick, flexible, adaptable, user-oriented) instruments in the realm of engineering web applications, which in itself represents an interesting area of exploration for PLE [20, 21, 27, 28] and AMs [26, 29, 30], the *WebPads*-based approach introduced in this article and its supporting tool, *VisualWebC*, offer excellent examples of support for *software reusability*. After all, the WSs incorporated in new applications are remarkable illustrations of reusing software components, be they weather forecast modules, stock evolution monitors, shopping carts, online unit converters, or keyword-based searching mechanisms. In terms of *domain engineering* and *application engineering*, one can envision applying *WebPads* and tools similar to *VisualWebC* to building web applications using a more rigorous and systematic process directions of the PLE methodology, combined with (intrinsic to

WebPads) elements of development agility. Our approach being organized around easily modifiable, reconfigurable and reusable pages, workflows, and *WebPads* (WSs), can effectively support both the common artifacts of a web development "domain" (e.g., the on-line shopping "domain") and the particular aspects of a specific application in that "domain" (e.g., on-line shopping of automotive parts, such as tires or wheels).

Finally, *documentation* and, in general, *process* and *product organization*, all important to PLE, could significantly benefit from using easy-to-built and efficient web-based solutions for collaborative software product development made possible by the proposed *WebPads*-based approach (e.g., web-based error-tracking tools, jointly updated product data dictionary, and online tools for project progress reporting, assessment, and prediction). In summary, numerous interesting directions of research and development are offered by the prospect of placing *WebPads*-based WSbWAs and tools (the latter, very agile in nature) under the joint umbrella of PLE and AMs, thus answering in practice the need for rightly balancing agility and discipline [31].

6. Future Work and Conclusions

Our approach presents several significant advantages. First, it allows the creation of web applications with complex functionality with relatively little effort and time. Importantly, it allows novice programmers to build websites easily and reduces the cost and time of creating such websites. Second, the use of trusted, well tested, permanently updated, and secure services [32] allows the newly generated web applications to inherit these features and gain the trust of their users, who do not have to worry about the security of the remote components they rely on. Third, testing a new web application is reduced to testing the inter-services relationships that exist between different wrapped web applications rather than testing the intra-services relationships that characterize each contributing application (because the latter are already tested by their providers). Fourth, if one of the remote web applications updates its database or enhances its mechanisms of providing the outputs, these changes will be directly reflected in the new web application, since it only depends on the paths of the inputs and outputs. Consequently, there is no need to

worry about updating the web application, especially because the connection to the remote services is established dynamically.

Regarding future work, a problem that we plan to tackle next is to implement a proxy class between the website created and the websites referenced in order to minimize the effects of changes in their data formats. Specifically, when a referenced website changes the format of its inputs and/or outputs an interface is needed between the two applications to accommodate the changes. Also part of future work, we are interested in developing a more rigorous model using client/server agents that will track changes made on the remote components and refresh the local components, thus keeping the information always up-to-date. We also plan to integrate a component to facilitate: (1) publishing our composed services; and (2) dynamically discovering aliases via automatic WS discovery.

Finally, as discussed in the previous section, placing the naturally agile (fast, flexible, service-oriented, reconfigurable, easily manageable, user-friendly) *WebPads*-based web development approaches and tools in the scope of the PLE methodology offers many promising directions of future exploration and expansion. Larger examples of web application developments using the *WebPads* concept, the *VisualWebC* tool, and the methodological directions provided by PLE and AMs is also part of our future work.

Acknowledgments

We would like to thank the students at AUB for their collaboration and patience in the experimentation sessions. Our thanks go also to the ACM SIGCHI for allowing us to modify templates they developed.

References

[1] W3C. Web Services Description Language (WSDL) 1.1, 2001, http://www.w3.org/tr/wsdl

[2] Thakkar, S. Knoblock, A.C., Ambite, J.L., and C. Shahabi, "Dynamically Composing Web Services from On-line Sources", *Proceedings of the AAAI Workshop on Intelligent Service Integration*, Edmonton, Alberta, Canada, 2002, pp. 1-7.

[3] van Zyl, J., "Application Assembly Using Web Services", *Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC'02),* IEEE Press, 2002, pp. 493-500.

[4] IBM, Specification: Business Process Execution Language for Web Services Version 1.1, accessed May 10, 2007, at http://www.ibm.com/developerworks/library/ws-bel/

[5] Sillitti, A., Vernazza, T., and G. Succi. "Service Bases Product Lines," *Proceedings of the 3rd International Workshop on Software Product Lines: Economics, Architectures and Implications*, ICSE-2002, Orlando, USA, 2002.

[6] Capilla, R. and N. Yasemin Topaloglu. "Product Lines for Supporting the Composition and Evolution of Service Oriented Applications." *Proceedings of the Eighth International Workshop on Principles of Software Evolution,* 2005, pp. 53-56.

[7] Capilla, R. and J. C. Dueñas, *Evolution and Maintenance of Web Sites: A Product Line Model.* Chapter "Managing Corporate Information Systems Evolution and Maintenance," Idea Group Publishing, 2005, pp. 255-271.

[8] Svetinovic, D. "Architecture-Level Requirements Specification," *Proceedings of the 2nd International Software Requirements to Architecture Workshop (STRAW'03),* Portland, Oregon, USA, 2003, pp. 14-19.

[9] Jacyntho, M.D., Schwabe, D. and G. Rossi. A software architecture for structuring complex web applications. *Journal of Web Engineering* **1**(1):37-60, October 2002.

[10] Balzerani, L., Di Ruscio, D., Pierantonio, A. and G. De Angelis. "A Product Line Architecture for Web Applications." *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC'2005)*, ACM Press, New York, NY, USA, 2005, pp. 1689-1693.

[11] Balzerani. L., Problemi di Generazione e Configurazione dei Sistemi a Componenti. Tesi di Laurea in Informatica, Università degli Studi di L'Aquila, 2005.

[12] Pautasso, C. and G. Alonso "Visual Composition of Web Services," *Proceedings of IEEE HCC'03*, Auckland, New Zealand, 2003, pp. 92-99.

[13] Liu, N., Grundy, J. and J. Hosking. "A Visual Language and Environment for Composing Web Services," *Proceedings of the 2005 ACM/IEEE International Conference on Automated Software Engineering*, Long Beach, CA, 2005, pp. 321-324.

[14] Ito, K. and Y. Tanaka. "Visual Wrapping and Functional Linkage of Web Applications," *Proceedings of the Workshop on Emerging Applications for Wireless and Mobile Access*, Budapest, Hungary, 2003.

[15] K. Ito, and Y. Tanaka. "A Visual Environment for Dynamic Web Application Composition," *Proceedings of the 14th ACM Conference on Hypertext and Hypermedia*, 2003, pp. 184–193.

[16] Java Sun website, Design Patterns: Model-View-Controller, accessed May 12, 2007 at java.sun.com/blueprints/patterns/MVC.html.

[17] Leff, A. and J.T. Rayfield. "Web-Application Development Using the Model-View-Controller Design Pattern," *Proceedings of the 5th IEEE Enterprise Distributed Object Computing Conference*, 2001, pp. 118-124.

[18] Ginsburg. M. A Comparison of Web Visualization Frameworks for eBusiness Applications Using the Example of an Online Shop. Master Thesis. Accessed on May 12, 2007 at www.sts.tu-harburg.de/pw-and-m-theses/2003/

[19] Karam, M., Keirouz, W. and R. Hage. "An Abstract Model for Testing MVC and Workflow Based Web Applications," *Proceedings of the IEEE Advanced International Conference on*

Telecommunications and International Conference on Internet and Web Applications and Services (AICT-ICIW'06), 2006, pp 206/1-7.

[20] Van Gurp, J., Bosch, J. and M. Svahnberg. "On the Notion of Variability in Software Product Lines," *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, 2001, p. 45.

[21] Pohl, K., Böckle, G. and F. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.

[22] Carbon, R., Lindvall, M., Muthig, D., and P. Costa, "Integrating Product Line Engineering and Agile Methods: Flexible Design Up-front vs. Incremental Design," Proceedings of the 1st International Workshop on Agile Product Line Engineering Baltimore, MD, USA, 2006. Available online as of May 15, 2007 at http://www.lsi.upc.edu/events/aple/CarbonEtAl.pdf.

[23] Ruby on Rails, "Web Development that Doesn't Hurt," accessed May 12, 2007 at http://www.rubyonrails.org

[24] The Apache Software Foundation web site, Struts, accessed May 12, 2007 at http://struts.apache.org/

[25] W3C Document Object Model (DOM), accessed May 12, 2007 at www.w3.org/DOM

[26] The Agile Manifesto, accessed May 12, 2007 at http://www.agilemanifesto.org

[27] P. Clements, L. M. Northrop, et al. A Framework for Software Product Line Practice, version 4.2. Technical Report. SEI Carnegie Mellon University, Pittsburgh, 2004.

[28] van Zyl, J. An Approach to Assemble Software Products Using a Product Line Approach. The International Workshop on Product Line Engineering (PLESS'03), Technical Report 139.0/E at Fraunhofer ISE, Erfurt, Germany, 2003, pp. 43-49.

[29] Agile Software Development – Wikipedia, accessed May 15, 2007 at http://en.wikipedia.org /wiki/Agile_software_development

[30] Fowler, M. "The New Methodology," December 2005, accessed May 15, 2007 at http://www.martinfowler.com/articles/newMethodology.html

[31] Boehm, B. and R. Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley, 2004.

[32] Turner, M., Budgen, D. and P. Brereton, Turning software into a service. *IEEE Computer* **36** (10): 38-44, October 2003.