Journal of Visual Languages and Computing 25 (2014) 414–432

Contents lists available at ScienceDirect

Journal of Visual Languages and Computing

journal homepage: www.elsevier.com/locate/jvlc

End-User Development of Mashups with *NaturalMash* ☆

Saeed Aghaee *, Cesare Pautasso

Faculty of Informatics, University of Lugano (USI), Switzerland

A R T I C L E I N F O

Article history:
Received 17 July 2013
Received in revised form
13 November 2013
Accepted 31 December 2013
Available online 20 January 2014

Keywords:
Mashups
End-User Development
Mashup tools
WYSIWYG
Natural language programming
Programming by Demonstration

A B S T R A C T

Context: The emergence of the long-tail in the market of software applications is shifting the role of end-users from mere consumers to becoming developers of applications addressing their unique, personal, and transient needs. On the Web, a popular form of such applications is called mashup, built out of the lightweight composition of Web APIs (reusable software components delivered as a service through the Web). To enable end-users to build mashups, there is a key problem that must be overcome: End-users lack programming knowledge as well as the interest to learn how to master the complex set of Web technologies required to develop mashups. End-User Development (EUD) is an emerging research field dealing with this type of problems. Its main goal is to design tools and techniques facilitating the development of software applications by non-programmers.

Objective: The paper describes the design and evaluation of *NaturalMash*, an innovative EUD tool for mashups (a mashup tool). *NaturalMash* aims at enabling non-professional users without any knowledge of programming languages and skills to create feature-rich, interactive, and useful mashups.

Methods: The design of *NaturalMash* adopts a formative evaluation approach, and has completed three design and evaluation iterations. The formative evaluations utilize usability testing, think aloud protocol, questionnaires, observation, and unstructured interviews. Additionally, we compare the expressive power of naturalmash with the state-of-the-art mashup tools.

Results: The results from the formative evaluations helped us identify important usability problems. From an assessment point of view, the results were promising and sggested that the proposed tool has a short and gentle learning curve in a way that even non-programmers are able to rapidly build useful mashups. Also, the comparative evaluation results showed that *NaturalMash* offers a competitive level of expressive power compared with existing mashup tools targeting non-programmers.

Conclusion: As the evaluation results indicate, *NaturalMash* provides a high level of expressive power while it is still highly usable by non-programmers. These suggest that we have successfully achieved the objective of the proposed tool, distinguishing it from existing mashup tools that are either too limited or highly specialized for non-professional users.

1. Introduction

With the proliferation of Web APIs (i.e., reusable software components published on the Web), the Web [1] has become a highly programmable platform. A lightweight form of Web applications that is widely developed and used on this platform is called mashup. Mashups are

☆ This paper has been recommended for acceptance by Shi Kho Chang.
* Corresponding author.
E-mail addresses: saeed.aghaee@usi.ch (S. Aghaee), c.pautasso@ieee.org (C. Pautasso).

1045-926X/$ - see front matter © 2014 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.jvlc.2013.12.004

usually built by users themselves by composing different Web APIs in an ad hoc fashion [2]. As a result, they provide users with the opportunity of rapidly satisfying their situational needs in various domains of application [3,4], ranging from daily utilities of Web users to specialized domains, such as e-learning [5], bioinformatics [6], health care [7], emergency management [8] and enterprise integration [3].

In spite of the growing demand for mashups, their development barriers (e.g., knowing how to code in Web scripting languages like PHP and JavaScript, understanding Web API protocols such as HTTP) can hinder their proliferation. This is due to the fact that the dominant type of mashup users in various application domains are those with little or no knowledge in programming and related technologies. In order to cope with this challenge, therefore, these non-professional users need to be empowered to create mashups. End-User Development (EUD) [9,10] is a research area that is committed to address this type of problems. Research and development in EUD for mashups have resulted in the emergence of dedicated *mashup tools* [11] that provide end-users with an intuitive composition language and environment for on-the-fly and code-free development of mashups.

In this paper, we present in detail the design and the evaluation of an innovative mashup tool called *NaturalMash*. *NaturalMash* provides adequate expressive power to create non-trivial, feature-rich, and interactive mashups out of the composition of Web APIs provided through different technologies (ranging from REST and SOAP services to JavaScript and HTML5 widgets). *NaturalMash* is designed to be usable by non-professional users by ensuring that it is easy to understand and easy to learn with a gently sloped learning curve (thanks to a highly interactive, live programming environment, featuring immediate feedback and autocompletion). Many mashup tools with the same level of expressive power (e.g., IBM Mashup Center (http://www.ibm.com/software/info/mashup-center), and JackBe Presto (http://www.jackbe.com/) are, however, designed in a way that is too specialized for non-professional users. On the other hand, mashup tools explicitly targeting non-professional users, such as `IFTTT` (https://ifttt.com) and ServFace Builder [12], do not provide adequate expressive power to freely compose any type of Web APIs.

This paper also contributes a novel, hybrid end-user programming technique [13] based on natural language programming [14], live programming, WYSIWYG [15] (What You See Is What You Get), and Programming by Demonstration [16] (PbD). *NaturalMash* is one of the first live mashup tools [17] that combines natural language processing techniques [18] with model-driven Web engineering [19] in order to provide immediate feedback to the users and show them the resulting mashup as they are typing up its recipe. *NaturalMash* was first introduced in [20]. This paper includes additional material describing our user-centric design approach with the complete history of its formative evaluations, an extensive comparison with related approaches, as well as additional usage examples to demonstrate its expressive power and information on the internal architecture of *NaturalMash*.

A formative user-centered design approach enabled us to collect early feedback on the system by two groups of users differing in their computer science knowledge: programmers and non-programmers. This approach helped us better focus the design and avoid gaps between the user expectations and the delivered system. As of yet we have completed three iterations of design and evaluation. Initial findings from the evaluations indicate that users with little or no programming experience can become productive and successfully build useful mashups, confirming the validity of some of the design decisions behind *NaturalMash*.

The rest of the paper is organized as follows. Section 2 presents the goals, requirements, and rationale behind the design of *NaturalMash*. We explain our approach to use natural language programming for mashup development in Section 3. Sections 4 and 5 thoroughly describe, respectively, the graphical user interface environment and the architecture of *NaturalMash*. Section 6 reports on the formative evaluation (second iteration) of the system and discusses the impact of users′ feedback in terms of usability assessment and suggested areas to improve. In Section 7 we compare *NaturalMash* against the state-of-the-art mashup tools in terms of their expressive power and the chosen end-user programming techniques. We provide a comprehensive discussion – summarizing the lessons learned in form of design guidelines – of the evaluation and comparison results in Section 8. We draw the conclusions in Section 9.

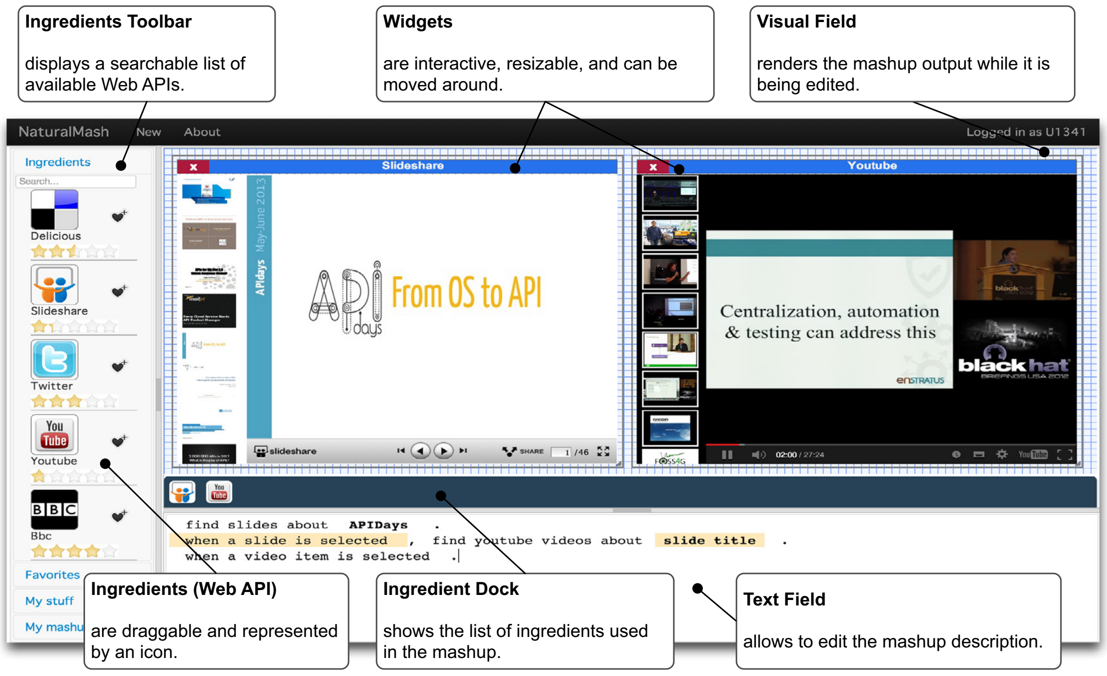
2. Design goals, requirements, and decisions

One of the main challenges in designing mashup tools consists of balancing the trade-off between the tool expressive power and the assumed user skills [21]. In addressing this challenge, the design of *NaturalMash* seeks to empower non-professional users (i.e., those who do not know programming) to rapidly create useful and feature-rich mashups with minimal prior learning. To achieve this, we tailored our design to meet three requirements:

(i) a high degree of usability by non-professional users (**R1**),
(ii) a competitive level of expressive power (**R2**), and
(iii) usefulness in a sense of being able to create useful mashups (**R3**).

In order to fulfill the above requirements, we made the following design decisions.

- *Familiar metaphor* (*D1*): Many users, specially non-professional users, might not be familiar with the technical terminology related to mashups such as service composition and Web APIs [22]. In order to bridge this gap, we designed *NaturalMash* based on the familiar metaphor of *cooking*, according to which Web APIs are referred to as “ingredients”, and the mashup source code (composition) is the “recipe” to mix these ingredients.
- *Simple graphical user interface* (*D2*): A simple yet powerful graphical user interface can potentially reduce the learning barriers and make the tool more intuitive. To



Ingredients Toolbar
displays a searchable list of available Web APIs.

Widgets
are interactive, resizable, and can be moved around.

Visual Field
renders the mashup output while it is being edited.

```
find slides about APIDays .
when a slide is selected , find youtube videos about slide title .
when a video item is selected .
```

Ingredients (Web API)
are draggable and represented by an icon.

Ingredient Dock
shows the list of ingredients used in the mashup.

Text Field
allows to edit the mashup description.

Fig. 1. *NaturalMash* environment: users type the recipe of the mashup in the text field and immediately see the output in the visual field. The output contains interactive widgets that can be resized and relocated. The ingredients toolbar helps with API discovery, while the dock gives a summary of the APIs used in the current mashup. Web APIs are abstracted away from the technologies they use and are represented as icon.

this end, we designed *NaturalMash* (Fig. 1) to have a Single Page Application (SAP) interface composed of merely four main components that together control all the functions of the system: (i) *text field* providing advanced support for typing in the recipe of a mashup integration logic, (ii) *visual field* implementing the WYSIWYG (What You See Is What You Get) interface for both the design and preview of the user interface of the mashup being created, (iii) *ingredient dock* graphically representing the APIs used by the mashup, and (iv) *ingredients toolbar* containing all the mashups created by the users and a searchable list of Web APIs.

- *Live programming based on WYSIWYG* (*D3*): *NaturalMash* incorporates the live programming paradigm [23,24], in which the edit/compile/run development life-cycle is fully automated by the system. As a result, users can more easily bridge the gulf of evaluation (the degree of difficulty of assessing and understanding the state of the system [25]). This in turn leads towards an improved learning experience [26].
- *Natural language programming* (*D4*): Natural language programming as an end-user programming technique can potentially provide a good level of expressive power. Also, natural languages (e.g., English) are readily understandable by their speakers. In the design of *NaturalMash*, natural language programming is enabled through a *controlled natural language* (CNL) — a subset of a natural language (e.g., English) restricted in terms of vocabulary and grammar. The reason for using a CNL is to ensure the accuracy of the system compiler. From the expressive power point of view, the *NaturalMash* CNL empowers users to describe relatively complex process orchestration and data integration logic as well as the composition of widgets (all at a very abstract level).
- *WYSIWYG* (*D5*): The type of live programming [17] utilized in the system employs a WYSIWYG interface (the visual field) providing not only the recent version of the mashup being edited but also direct manipulation capabilities (e.g., moving and resizing the widgets) to design the mashup user interface. Moreover, the visual field also facilitates natural language programming through visual demonstration and interactions with widgets (e.g., clicking a map widget adds the corresponding natural language description to the text field, being, for instance, "when the map is clicked"). In other words, the visual field uses PbD to provide a direct way to partially manipulate the application logic of the mashup being created. The combination of WYSIWYG and natural language programming makes the user interface much more intuitive as it can support both direct manipulation (visual field) and descriptive representation (text field) of the mashup being created.

3. *NaturalMash* controlled natural language

The *NaturalMash* CNL is an abstract, executable language for modeling the presentation integration, process integration, and data integration layers of mashups. Before describing its syntax and semantics we introduce the language with a few examples.

Listing 1 is the recipe (executable text written in the CNL) of a mashup that searches Slideshare (it is a Website for sharing and finding presentations and documents, http://www.slideshare.net/developers) for a topic or event (in this example "`APIDays`"), and then uses the title of each resulting slide to accurately search for its corresponding presentation video in YouTube (https://developers.google.com/youtube/).

```
Find slides about APIDays. When an item is selected,
find YouTube videos about slide title.
```

Listing 1: A presentation search mashup

Listing 2 is an example recipe of a map-based mashup. The mashup includes a user interface composed of a Google Maps widget (https://developers.google.com/maps/) and an HTML table widget. The content of the table displays a stream that aggregates content from the BBC News (http://www.bbc.co.uk/news/10628494) and CNN News feeds (http://rss.cnn.com/rss/edition.rss). When a news item in the table is selected, the Yahoo! Placemaker service (http://developer.yahoo.com/geo/placemaker/) extracts geographical data (e.g., longitude and latitude) from the text. The geographical data is used to place a marker representing the news item on the map.

```
Combine BBC Top News with CNN Top News.
When an item is selected, extract the location of the item,
and place a marker on the map.
```

Listing 2: A location-based news feed mashup

Listing 3 builds a mashup combining Twitter (https://dev.twitter.com/), the YouTube player, a HTML table, and a regular expression component, extracting values from an input using a set of predefined patterns (e.g., "`YouTube video link`", "`Flickr image link`", etc.). It displays tweets about a certain keyword in a table, and allows users to play them if they contain a link to a YouTube video.

```
Find tweets about Lugano.
When an item is selected, extract youtube video from it, and
play video.
```

Listing 3: A player for videos linked from tweets

The above recipe examples all conform to the CNL grammar and vocabulary and can be automatically executed by *NaturalMash*. The CNL imposes specific grammatical constraints that, for instance, only certain types of sentences can be constructed. The rest of this section describes in detail the CNL grammar as well as the way Web APIs are specified in *NaturalMash*.

3.1. Abstract mashup components

The underlying implementation of the CNL accommodates an abstract component model that

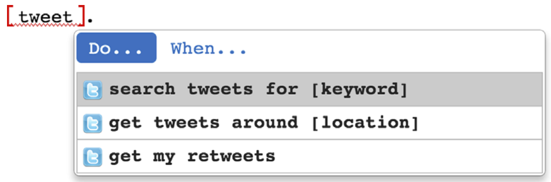
(i) gives a unified technology-neutral description of Web APIs, and
(ii) models them in an abstract textual form.

The *NaturalMash* component model classifies and describes various aspects of Web APIs. Different Web APIs are broadly categorized as three types: *Data Source*, that delivers a snapshot or a stream of data from remote sources on the Web (e.g., BBC News), *Service*, that provides remotely accessible business logic (e.g., Yahoo! Placemaker), and *Widget*, that is a stand-alone Web application with a self-contained (reusable) user interface (e.g., Google Maps widget).

The component model also distinguishes two types of functionality provided by Web APIs, namely, *Task* – a passive atomic operation, and *Event* – an active source of control. More in detail, an Event describes a condition that (if satisfied) may produce a message. In the Listing 1 example, selecting an item (slide) is an Event of the table widget. A Task, on the other hand, takes some input data, does some processing, and then produces data. Finding

```
(*MASHUP RECIPE*)
nlmd = paragraph  (EMPTY_LINES  paragraph)* (EMPTY_LINES?  EOF);

(*PARAGRAPH*)
paragraph = sentence+;

(*SENTENCE*)
sentence = imperative  |  causal;

(*IMPERATIVE SENTENCE*)
imperative = command  (command_delimiter  command)*  DOT;
command = (*created from a Task label*)
command_delimiter = COMMA AND?;

(*CAUSAL SENTENCE*)
causal = WHEN case COMMA imperative;
case = (*created from an Event label*)
```

Fig. 2. The grammar of *NaturalMash* CNL represented in Extended Backus–Naur Form (EBNF).



search tweets for [keyword]
get tweets around [location]
get my retweets

Fig. 3. Typing "tweet" results in the autocomplete list showing the labels associated with the Twitter API.

slides, given a keyword, exemplifies a Task behavior of the Slideshare API.

The data consumed and produced by Tasks and Events of Web APIs is modeled as, respectively, one or more input and output *parameters*. Each parameter has a meaningful and unique (only within the API scope) name; its syntactic and semantic types are defined but not shown to the end-users.

To give a natural language representation of APIs, each of its Tasks and Events is annotated with a specific natural language description called a *label*. For instance, "find slides about [keyword]" is a label describing the slide-searching Task of the Slideshare API. The input parameter name keyword is enclosed within square brackets, creating a placeholder for the object of the verb used in the label. In a mashup recipe, the actual object of the sentence may be a parameter name referring to the output of previous tasks, a constant value, or an anaphora — in linguistic, an anaphora (e.g., "that") is defined as an expression linking two elements in a document — pointing to a specific part of the recipe text (e.g., in Listing 3 the clause "play it" contains the anaphora "it" pointing to the output parameter of the event produced by the table Widget). As an example of an Event label, "an item is selected" is associated with the selection event of the table widget. Note that an Event does not receive input parameters, and thus its label does not need to contain any placeholder.

3.2. CNL grammar

Fig. 2 illustrates the grammar of the *NaturalMash* CNL. The top-level structure of a mashup recipe text is decomposable into paragraphs, which are, in turn, a collection of sentences. To describe how to compose together Web APIs, the CNL imposes specific grammatical constraints, which limit the types of sentences that may be constructed. We distinguish

- *Imperative sentences*: They are composed of multiple imperative mood clauses. Each clause is built from a Task label by replacing the placeholders of the label with objects. For example, given the label "find songs titled [keyword]" the corresponding clause can be "find songs titled mashup", where the object is replaced with a constant value "mashup".
- *Causal sentences*: They are written in causal form in which the time conjunction "when" introduces a passive clause (a label associated with an Event) followed by a set of imperative mood clauses (like an imperative sentence). The passive clause describes the cause of the event; the imperative mood clauses represent the effect to be realized when the cause of the event happens. Consider the causal sentence in the example (Listing 1). "When an item is selected, search youtube videos about title." In this sentence, when the Event described as "an item is selected" happens, the Task following it "search youtube videos about title" is executed.

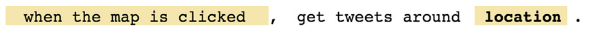
The CNL is compiled to an executable language with both *imperative* and *event-driven* semantics. Its imperative aspect is that the natural sequential order of the sentences in English (i.e., from left to right) defines the control flow of a mashup from one sentence, clause, or phrase to another.

The event-driven execution semantics of the *NaturalMash* CNL is modeled by causal sentences. Causal statements are activated (but not immediately executed) when the main control flow reaches them. An activated causal sentence is ready to later receive control whenever its Event happens, after which the control is immediately passed to its imperative part. In doing so, the imperative part initiates a parallel and independent control flow, which can be repeatedly executed for every occurrence of the Event.

4. *NaturalMash* composition environment

The *NaturalMash* environment is designed to provide an innovative selection of features that are meant to enhance the user experience and the ability of users to build sophisticated mashups. The design of the environment has been evolved over two years, as a result of a formative user-centered process. In this section, we consider the current version of the environment as this paper is being written. We postpone the details of the evolution and evaluation of the environment to Section 6. In the following, we first describe each feature individually and later show in a usage scenario how they are used in conjunction to build a mashup.

- *Inline search*: To enhance component discovery, the *NaturalMash* environment provides an *inline search* feature (Fig. 3) in the text field that allows users to
  - (i) directly type in the text editor the (approximate) name of the ingredient (Web API) they are looking for, which results in the user getting a list of labels associated with the ingredient matching or approximating the given keyword, or
  - (ii) type what the ingredient is supposed to do (in case they do not know or cannot guess the exact name of the ingredient), by doing which the input text will be matched against all the labels associated with all the ingredients in the library.



when the map is clicked , get tweets around location .

Fig. 4. The source of the object (the "map click" label) as well as the object itself "location" get highlighted as soon as the cursor is placed in the object text.

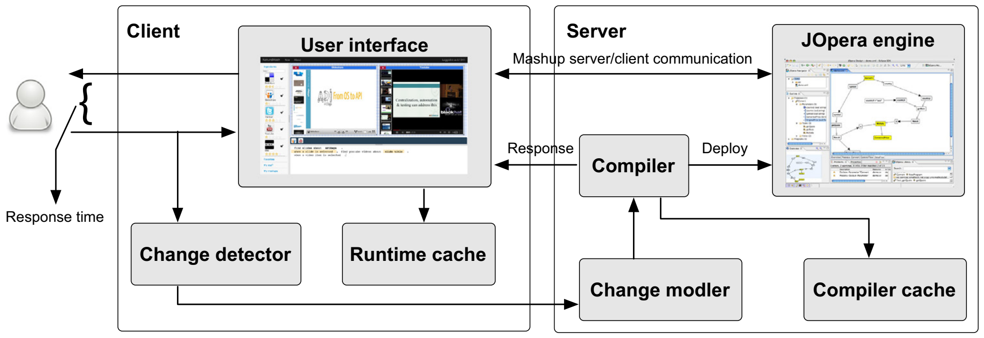


Fig. 5. The high performance architecture of *NaturalMash* aims at minimizing the response time to support live programming.

In the latter case, the mechanism of searching labels is based on

(i) exact match,
(ii) word synonym (e.g., "`search`" and "`find`"), or
(ii) word semantics (e.g., "`location`" and "`map`").

- *Autocompletion*: The *NaturalMash* CNL is based on a restricted grammar meaning that not all possible input combinations are acceptable. Consequently, the CNL acts as a learning barrier as the users need to master the grammar and syntax of the language. The autocomplete feature (Fig. 3) is utilized to lower the CNL learning barrier. Based on what users type in the text field, a list automatically appears and shows suggestions for Task/Event labels (to support component discovery and reuse), and data flow (i.e., referencing suitable objects within Task labels).
- *Semi-structured text editor*: To support the users' learning experience, the text field provides a semi-structured text editor ensuring that user input will not cause syntax errors, while still allowing a high degree of freestyle editing. To be specific, the text field:
  (i) restricts input characters to avoid accidental syntax errors (for instance the new line characters are disabled while typing an object in a placeholder),
  (ii) automatically inserts the separators "`,`", "`and`", and "`and,`" if the cursor is positioned before and after clauses (manual insertion of the separators is also possible), and
  (iii) streamlines selecting and moving text objects (clauses) via, respectively, double-click and drag-and-drop.
- *Data flow highlighting*: In the text field, objects indicating flow of data are displayed in boldface (Fig. 4). Moving the cursor on the text representing an object results in the highlighting of the text describing its source Task or Event. This way, users can discover the source of an object not only when browsing the data flow suggestions in the autocomplete list but also after a data flow suggestion has been entered in the text.
- *Error highlighting*: If there is an ambiguity in the input text (e.g., the input does not match any Task or Event label), the compiler produces an error that is reported to the user as the text is being entered (Fig. 3). Similar to many "spell-checking text editors", the error is shown using a red wavy line under the text that produced it. An autocomplete list containing possible suggestions to disambiguate the label is shown whenever the user moves the cursor (or click) on the highlighted erroneous text, and thus offering the opportunity to the user to quickly correct the mistake.
- *Drag-and-drop*: The ingredients toolbar gives a visual overview of the available ingredients. From there, users can drag-and-drop an ingredient into the text field, visual field, or ingredient dock. If the ingredient is a widget, it will be displayed in the visual field. Also, the autocomplete list will appear containing the corresponding Task/Event labels.
- *Programming by Demonstration*: Interacting with widgets in the visual field results in appending the corresponding Event label to the text field. For instance, clicking on Google Maps widgets results in showing the text "`when the map is clicked`" in the text field. To grab the attention of users the text corresponding to the event is highlighted both after it has been added and when it will be executed.
- *Synchronized multi-perspective modeling*: The three main interaction components of the environment (ingredient dock, text field, and visual field) are all kept synchronized during every user interaction:
  (i) editing text in the field updates the visual field and the ingredient dock;
  (ii) selecting a widget from the visual field or a ingredient from the ingredient dock results in highlighting its corresponding text in the text field and vice versa (moving the caret through a portion of the text highlights its associated widgets and ingredient icons);
  (iii) deleting an icon from the dock or a widget from the visual field results in the removal of its corresponding text (and vice versa).

4.1. Usage scenario

The following illustrates a common and complete usage scenario of *NaturalMash*, whereby a user builds the mashup example described in Listing 1.

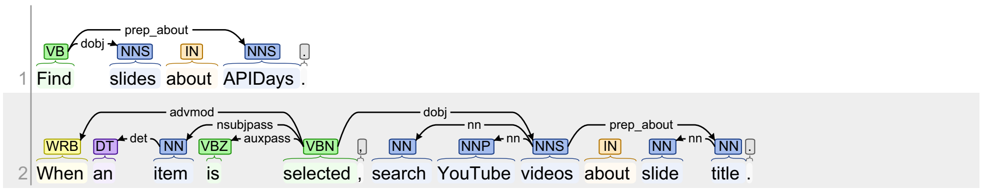

prep_about
dobj
VB NNS IN NNS .
1 Find slides about APIDays .
advmod
nsubjpass
auxpass
det
dobj
nn
nn
prep_about
nn
WRB DT NN VBZ VBN , NN NNP NNS IN NN NN .
2 When an item is selected , search YouTube videos about slide title .

Fig. 6. The visualized output of Step 1 (linguistic information) for the Listing 1 example using the Stanford CoreNLP online tool (http://nlp.stanford.edu:8080/corenlp/). The input is split into two sentences. Part-of-speech tags (VB: base form verb, NNS: noun plural, IN: proposition, DT: determiner, WRB: wh-adverb, NN: noun, VBZ: present verb, VBN: past participle verb) are associated with each word in the input text. Grammatical dependencies (det: determiner, advmod:adverbial modifier, nsubjpass: passive nominal subject, auxpass:passive auxiliary, dobj: direct object, nn: noun compound modifier, prep_about: prepositional modifier) are shown using arrows.

The first step is to discover the right ingredients for finding slides. This step can be facilitated by the inline search feature which enables the users to type what the ingredient he is looking for is supposed to do. For instance, the user can start by typing "`search slides`", which results in the text field providing an autocomplete list of labels that contain the input words or synonyms for the words. Once the autocomplete list is displayed, the user can select a proper suggestion (in this case, "`find slides about [keyword]`") by either pressing the Enter key or pointing with the mouse and clicking.

After selecting a label from the autocomplete list,

(i) the label is inserted into the text field,
(ii) ambiguity is resolved, in case there are one or more similar labels,
(iii) the mashup is rebuilt and executed,
(iv) another autocomplete list containing data flow suggestions for the label is displayed.

For the input parameter "`[keyword]`", the user may type a constant string like "`APIDays`" resulting in a mashup that uses Slideshare API to search for slides and document matching the input constant, and automatically shows the results in the Slideshare widget.

The output mashup is interactive and supports PbD in a way that, for instance, clicking an item in the Slideshare widget results in not only showing the item in the embedded frame of the widget, but also appending the corresponding Event label (i.e., "`when an item is selected`") to the text field as well as setting the focus in a way that makes it easier for the user to add some Task labels to complete the causal sentence. For example, the user may type "`video`" in the text field or, alternatively, search for the YouTube API in the ingredients toolbar and then drag-and-drop the ingredient to the text field, both of which result in displaying an autocomplete list containing the YouTube API labels. Immediately after selecting the suggestion, another autocomplete list containing data flow suggestions is shown to the user. The user can select the output parameter "`slide title`" from this list, or type an anaphora pointing to the item such as "`it`", both referencing the click event label of the Slideshare widget. The data flow highlighting feature helps users to figure out the source of an object. Leaving the object placeholder empty results in the compiler error that is shown by a red wavy line under the placeholder. Clicking on the wavy red lines displays the autocomplete list associated with the placeholder.

While typing the mashup recipe, the user may modify the mashup user interface layout in the visual field. The final mashup can then be deployed in production, with a single click. Even after a mashup has been published, it can still be modified and redeployed at any time.

5. Architecture

NaturalMash is designed as a live mashup tool, which completely automates the repetitive task of compiling, deploying, and running mashup recipes. Considering that mashups are compositions of remote and distributed Web APIs, it is rather technically challenging to comply with the requirements of liveness, as the changes made by the user to the mashup design must be reflected in the result of the mashup execution with minimal delay.

We present the client/server architecture of *NaturalMash* (Fig. 5), which aims at achieving performance improvements that result in decreased response time elapsing between the instants the user manipulates a mashup being created to the instant the resulting mashup is compiled, executed, and displayed to the user. The client-side runs in a Web browser and presents the user interface of the mashup tool and of the resulting mashup (in the visual field). The server-side handles the compilation of the recipes into executable representations and supports their runtime, which involves the interactions with external Web services and Web data sources.

To realize which APIs are affected during the user interactions intended for live development, consider the following scenario. The life-cycle of each interaction begins with the user making a single modification to the model of the mashup being developed on the client. This modification should target either the visual field (modifying the user interface) or the textual field (editing the recipe). Each modification may result in the server (re) compiling, (re)deploying, (re)executing, and (re)rendering the target mashup. It is essential that the whole interaction has a fast turnaround time to enhance the overall quality of the user's experience and reduce the user's anxiety about the effect of their actions on the mashup they are developing.

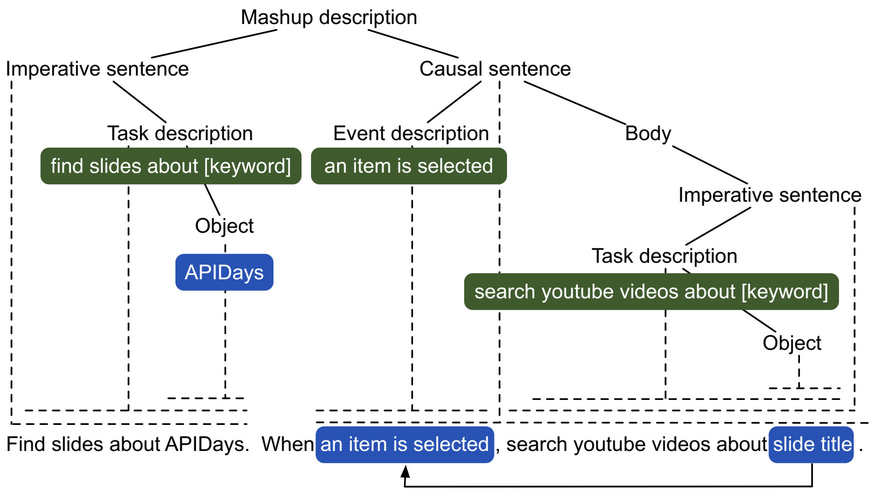


Fig. 7. The annotated syntax tree corresponding to the mashup label of Listing 1.

5.1. Incremental change detector of mashup models

Mashup compilation and deployment can be time-consuming tasks. Therefore, initially, and as soon as the user makes a modification to the mashup, the `change detector` component (client-side) identifies whether or not the modification requires issuing a (re)compilation request. The mechanism behind this component is based on classifying possible modifications as follows:

- Front-end modifications are applied to the mashup user interface (e.g., reorganizing widgets within the mashup user interface layout, and adding or removing widgets), and therefore, can be handled on the client-side without a need for recompiling and redeploying the whole mashup. These modifications are temporarily stored and previewed, and once the user's session is finished or idle (to avoid losing the modifications in the case of disconnection), the modifications are sent to the server for persistent storage.
- Incomplete modifications require further modifications to take a visible effect. These may leave the mashup recipe in a temporarily incorrect state as the user needs to complete them before they can be executed. For instance, a new Task is being added, but since no data flow source has been bound to its input it is not yet possible to display its results in a suitable Widget. These modifications may trigger the display of the autocompletion menu or the highlighting of errors so only a partial compilation is required.
- Logic modifications change the back-end of the mashup, and thus require to recompile, redeploy and re-execute the mashup on the server-side.

Once the change detector decides a recompilation is required, a request is sent to the server (over WebSockets). To facilitate and speed up the compilation process, the architecture includes the `change modeler` component that supports the notion of incremental compilation [27]. More in detail, a source model and a target model are required to generate a change model. The source and target models are, respectively, the high-level (abstract) and low-level (code) models of the mashup. The change model conforms to a metamodel that defines possible changes that may occur on the mashup (i.e., on both the source and target models) and contains information to link these changes from the source model to the target model. Since in many cases mashups are grown incrementally by adding one API at a time, it is possible to extend the low-level code without having to regenerate it from scratch.

5.2. Compilation and deployment

The resulting change model is passed to the `compiler` component, which is responsible to

(i) generate executable code corresponding to the change model,
(ii) merge it with the existing code, and finally
(iii) redeploy the mashup.

This component should allow to terminate an unfinished compilation process to avoid continuous compilation requests in short-intervals that put a heavy load on the server.

The compiler component in *NaturalMash* provides a pipeline that transforms mashup recipes into executable models of Web service compositions that are executed by the JOpera engine [28]. In the following we briefly describe the main steps of the *NaturalMash* compilation process.

The process relies on a representation that initially contains the input recipe text, but later is augmented with a list of components used by the mashup, the specifications of the layout of the mashup user interface (e.g., the size and position of the widgets), and an abstract syntax tree containing data flow information (i.e., source and destination of objects) and a mapping between the text

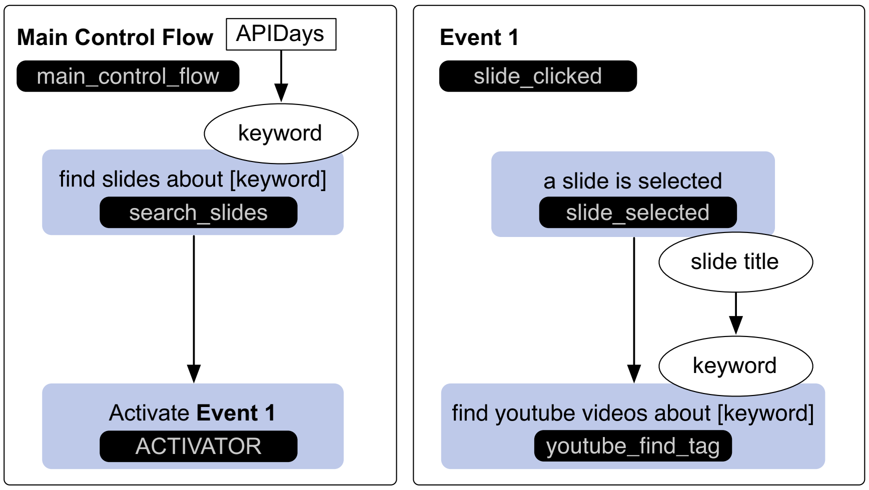


Fig. 8. The intermediate model generated for Listing 1. It contains two control flow graphs: `Main Control Flow` that corresponds to the imperative sentence "`find slides about APIDays`", and `Event 1` that is associated with the causal sentence "`when a slide is selected, find youtube videos about slide title`". The passing of input data flows to output data flows is represented by ovals.

chunks (i.e., clauses and phrases) and Task and Event labels. The mashup representation is recycled with each round of compilation and is continuously updated as the mashup is being developed.

- *Step*1. *Natural Language Parsing*: The input recipe text is parsed and its linguistic information is extracted (Fig. 6). This step is implemented using the Stanford CoreNLP library (http://nlp.stanford.edu/software/corenlp.shtml), which enables users to tokenize the input text and split it into sentences, parse the text and assign a part-of-speech tag (verb, noun, etc.) to each word, process grammatical dependencies, and create the anaphora resolution graph.
- *Step*2. *Constrained Natural Language Parsing*: An abstract syntax tree based on the *NaturalMash* CNL grammar is produced. In this step, we use a formal lexer and a parser (implemented using ANTLR, http://www.antlr.org/) to extract and identify sentence types as well as to extract their chunks (i.e., imperative or passive clauses).
- *Step*3. *API Binding*: The output of Steps 1 and 2 is consumed to build a mapping between the text chunks extracted in Step 2 and their corresponding Event/Task label. To attain this mapping, we first gather all the labels associated with the Tasks and Events of the APIs registered within the *NaturalMash* library. These are matched against the text chunks by ignoring the parameter placeholders. The result is a mapping between each text chunk and the corresponding Task or Event. In this step, ambiguity may occur when more than one Task/Event label match the same text chunk. Assuming that multiple APIs sharing the same label are equivalent, the ambiguity can be resolved automatically based on well known QoS-driven dynamic binding techniques [29]. Manual intervention through the tool's autocompletion feature is required only if there is an aliasing problem.
- *Step 4. Data Flow Resolution and Suggestion*: The mapping generated from Step 3 is used to extract objects references and complete the syntax tree (Fig. 7). To do so, the placeholders found within the Task labels representing input data are bound to the output data

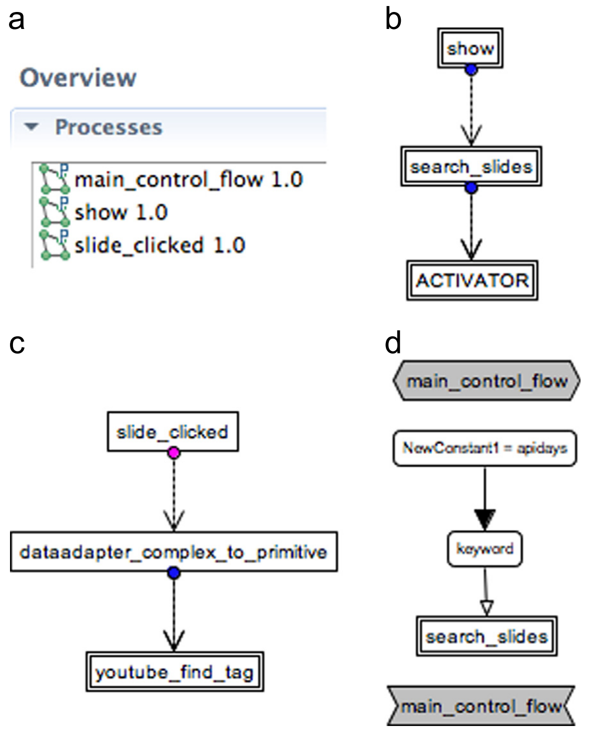


Fig. 9. The generated JOpera visual composition code [28] for Listing 1: (a) list of processes created for the mashup: `main_control_flow` implements the control flow for the imperative sentence, `slide_selected` implements the control flow associated with the causal sentence (slide select event), and `show` is the process responsible for creating the user interface of the mashup, (b) control flow implementing `Main Control Flow` in the intermediate model (Fig. 8), (c) control flow triggered whenever a slide is selected (it corresponds to `Event 1` in the intermediate model), and (d) data flow associated with `main_control_flow` ("`apidays`" is a constant passed to the `search_slides` input parameter).

referenced from the actual text. Using the results of the linguistic analysis (Step 1) also the anaphoric objects are resolved. The data flow operations (e.g., matching and conversion) are delegated to the *NaturalMash* semantic framework for data integration. The framework is based on a schema for input and output parameters of APIs. The schema contains metadata such as data type (primitive or complex), MIME type (e.g., application/xml and application/json), and ontology-based semantic annotations. Explaining the semantic data integration framework in detail is out of the scope of this paper. In the case of ambiguity (having more than one suggestion), we use the autocomplete feature to let the user specify the correct data flow references.

- *Step*5. *Intermediate Model* The disambiguated syntax tree is consumed to generate an intermediate model (Fig. 8) that includes control flow and data flow graphs representing the algorithmic structure of the mashup. This structure includes a "`main`" control flow graph containing all of the imperative sentences found in the current mashup recipe (and their constituent executable chunks like clauses and phrases built from Event and Task labels) as well as a set of "`event-triggered`" control flows associated with the causal sentences of the recipe. Causal sentences get activated in the "`main`" control flow. The "`main`" control flow begins when the mashup starts executing, whereas "`event`" control flows are executed every time their corresponding causal sentence occurs. The nodes of these graphs store a mapping between the executable and data elements of the recipe (technology-neutral) and the target executable model of JOpera (technology-specific).
- *Step*6. *Emitter*: The intermediate model is transformed into the target composition code (Fig. 9), which is directly executable by a JOpera mashup engine, which further transforms it internally to Java bytecode for efficient execution.The mashup execution is controlled by *NaturalMash* through a REST API, which also allows users to retrieve and display its results. By replacing the emitter it is possible to target other mashup runtime platforms.

The performance of the incremental compilation process is boosted through the use of the `cache` storing all the previous target models and their corresponding generated code. In doing so, the compiler component can save considerable time and computation resources by first looking up the target model in the cache and reusing its corresponding generated code. Also, the cache is populated through crowdsourcing to contain the target models generated not only by the same user but also by other concurrent users of the tools. This way, the chances of the compiler finding the appropriate target model are higher.

5.3. Runtime

After the mashup is compiled and deployed, a response is immediately sent back to the client. Since the response is asynchronous, there are three ways for the client to receive it:

(i) using HTTP short polling (i.e., polling the server by sending continuous requests in short-intervals),
(ii) using HTTP long polling (i.e., keeping the connection open for a long time-interval), and
(iii) using Web Sockets.

Among these solutions Web Sockets is much faster and more reliable [30], and is therefore, used by *NaturalMash* as the main communication mechanism. Moreover, this technology is nowadays mature and fully supported by the widely used Web browsers (e.g., Google Chrome, Firefox, Internet Explorer, etc.).

Immediately after the client is notified about the end of the deployment, it proceeds to execute the mashup with the given user input data. With regard to the response time, the most important phase in mashup runtime lifecycle is the first phase, in which the mashup is initiated in the visual field. To optimize this phase we propose two mechanisms based on parallelization and caching.

On the client-side, each Widget is initialized in parallel by using WebWorkers to call the associated JavaScript code. The server is responsible for interacting with external Web services and fetch third-party data sources due to the Same-Origin-Policy (SOP) sandboxing limitations imposed by browsers. JOpera provides support for a shared cache of Web service invocation results that can – seen as a form of pre-fetching – reduce the execution time of popular mashups. Caching is also a crucial feature in live mashup tools minimizing excessive calls to the Web services and data sources of a mashup, which most probably have some sort of call rate limit that may hinder the continuous re-execution of mashups invoking them.

6. Formative evaluation

As it was mentioned earlier in this paper, *NaturalMash* evolved over the past two years following a formative user-centered design approach [31], which proposes an iterative and incremental process for design and development of software systems. In the process, each iteration cycle consists of design, implementation, and formative evaluation. The evaluation is conducted at the end of each iteration to inform the next iteration and ensure that users were kept central in the design so as to avoid as much as possible mismatches between users′ expectations versus system behavior.

So far, we have completed three iteration cycles. In this section, we present the results from the formative evaluations we conducted at the end of each iteration, and show how the evaluations have driven the design of *NaturalMash* (Table 1). In general, the results suggest the success of our decisions in meeting the design requirements and goals (Section 2).

6.1. First iteration

The first iteration involved the design, development, and evaluation of the initial prototype of the system. For the evaluation, we conducted an expert review with 10 mashup experts. We individually interviewed the experts

Table 1
The evolution of *NaturalMash* during the formative user-centered design process in terms of added/removed features. `V0`, `V1`, and `V2` correspond to the versions of the tool during, respectively, the first, second, and third iterations.

Features	V0	V1	V2	**Change rationale**
Autocompletion	×	×	×	In addition to the label text, suggestions are represented with the corresponding ingredient icon
Error highlighting	–	×	×	Give immediate feedback about errors to user
Semi-structured editor	–	–	×	Prevent syntax errors
Ingredient stack	×	×	–	Replaced with Ingredient dock
Ingredient dock	–	–	×	Make the current APIs more visible to users while interacting with the visual and text fields
Side-bar	–	–	×	Let users tag favorite ingredients (APIs), and retrieve their mashups
API search box	×	–	–	Users did not realize they could search for APIs
Inline search	–	×	×	Replaced the search box to let users search for APIs from within the text field
Ingredients toolbar	–	–	×	Search *and* browse existing APIs
Drag and drop	–	–	×	Let users directly use a API selected from the ingredients toolbar
Programming by Demonstration	–	–	×	Allow users to (visually) demonstrate what they want
Auto-visualization	–	–	×	Visualizing output data using a suitable widget, thus reducing the complexity of the mashup description with natural language

in order to shed light on existing usability problems, and asked them to define how serious these were. Each expert was asked to interact with the system (after a short tutorial) for as long as they needed to provide feedback. The experts were researchers and practitioners in mashups that we met in mashup-related workshops such as the 5*th International Workshop on Lightweight Integration on the Web* (*ComposableWeb*) and the 6*th International Workshop on Web APIs and Service Mashups* (*Mashups*). The goal of the review was to identify common usability errors before doing a user study.

6.2. Second iteration

The valuable feedback from the expert review informed the re-design of the tool. Specifically, the feedback suggested to (1) remove the search box, that was initially designed as part of the component stack to help with component discovery, and replace it with the inline search functionality (i.e., component discovery within the text field); (2) reorganize the user interface layout (e.g., moving the stack from the left to the right); (3) efficiently organize and add icons to the autocomplete menu; (4) provide visual cues to distinguish labels of ingredients already in use out of all returned suggestions within the autocompletion menu; and (5) change short-cut keys for properly working with the autocomplete menu; and (6) improve the environment color scheme and fonts.

At the end of the first iteration, we conducted a user study with the main goal of identifying early major usability problems.

6.2.1. Users

We repeated the study with two groups of users with similar profiles: 5 high school students, and 6 first year students attending the USI Bachelor of Informatics. The recruited high school students had volunteered to attend a one-week computer science promotion program at our University. We performed the usability testing at the very beginning of the program in order to avoid students to be influenced by any programming activity.

The first-year students, who just started their studies at our University, even if inexperienced had an interest in Web technologies and this motivated them to volunteer to participate in the user study. In terms of programming knowledge, the high school students were all non-programmers. Likewise, among the group of bachelor's students, there were 2 programmers and 4 non-programmers. Participants in both the groups had neither heard of mashups nor ever created a Web application.

6.2.2. Method

Following Nielsen's discount usability (http://www.useit.com/alertbox/discount-usability.html) we consider the size of our sample sufficient to provide meaningful feedback at this early stage of development. In the beginning of the usability testing, we gave a short tutorial about the tool and guided them through completion of a warm-up task that, overall, took around 10 min. We then asked the participants to perform five tasks (developing five different mashups) with increasing complexity (in terms of the number of APIs to be mashed up):

Warm-up task:	Get upcoming events in a place specified using Google Maps (two APIs).
Task 1:	Play YouTube videos selected from Delicious public bookmarks (two APIs).
Task 2:	Show Flickr images about the twitter trending topic (two APIs).
Task 3:	Aggregate BBC news, CNN news, and Delicious feeds (three APIs).
Task 4:	Display tweets and events around a selected map location (three APIs).
Task 5:	Create a mashup on your own (open task).

The final open task was designed to assess the ability of the participants to independently come up with a mashup idea, and then transform it to a concrete implementation.

Since the tasks themselves were described in English, we attempted to minimize the similarity of the Task labels with their corresponding solutions in the *NaturalMash* CNL. After the usability testing, we asked the participants to fill out a questionnaire to assess their satisfaction with the tool as well as to gather their opinions on the overall usability of the tool (e.g., their opinion on how helpful or unhelpful the environment features were in the context of the given tasks).

6.2.3. Results

Overall, the users performed correctly 52 out of 55 tasks. On average, participants took less than 5 min to correctly complete each task.

82% of the participants believed that the autocomplete and inline search features helped them quickly and easily discover appropriate Task or Event labels. As indicated by 80% of them and according to our observation, the live preview in the visual field was really engaging and had a positive impact on the completion of the tasks. However, correcting mistakes in the tool was difficult for 55% of them.

6.2.4. Lessons learned

In this iteration, the results confirm that we are on the target to meet the design requirements. The satisfaction results as well as the low turnaround time and high rate of task completion and correctness suggest that we have fulfilled the requirements **R1** (usability). In the case of the open task, the completion and correctness rate also show that we have met the requirements **R2** (useful expressiveness) and **R3** (usefulness).

The evaluation helped to identify early critical usability problems. The freestyle editor in the text field makes it easy for users to make syntax errors, which in turn causes anxiety and stress. Another usability problem, revealed in the open task, concerns the lack of an overview of the available ingredients. More importantly, most users expected that the results of their commands would have been displayed immediately. Instead, in order to display the results of a Task on the visual field (e.g., "`find tweets around location`"), users had to learn how to manually find the right widget and explicitly add its corresponding Task to the mashup recipe (e.g., "`show the result in the table`").

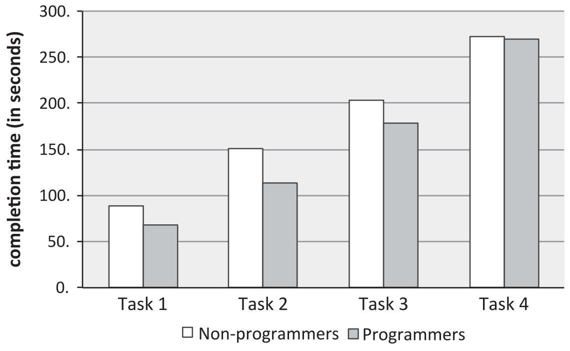


Fig. 10. The completion time grows with the complexity of the task at hand. Programmers have a slightly shorter completion time than non-programmers.

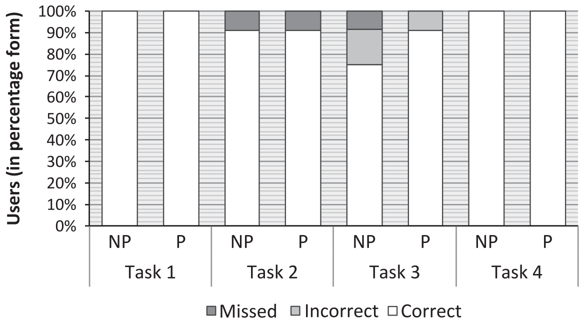


Fig. 11. The majority of tasks were accomplished successfully (correctly). In terms of accuracy, however, there is no major difference between programmers (P) and non-programmers (NP).

6.3. Third iteration

In the third iteration, we attempted to address the usability problems identified in the previous iteration by implementing a set of feature additions to *NaturalMash*:

- The semi-structured editor, making it unlikely to make syntax errors, while allowing freestyle editing.
- The ingredients stack that gives a searchable overview of existing ingredients and the corresponding drag and drop support. We extended the component library by adding 15 more popular ingredients to enable innovation in the open task (it used to contain 7 ingredients in the first iteration).
- The automatic visualization of the results in a widget (e.g., table, map, or chart) without users having to explicitly mention the widget.

We also moved the component dock from the right side (where the stack currently is) to the top of the text field to be more visible while users interact with the text field or the visual field. After observing some users dragging the icons of some ingredients over the widgets in the visual field we decided to add support for PbD in the second version.

To conclude this iteration, we conducted a formative evaluation on a larger group of participants. The evaluation aimed at not only to test the design hypotheses and identify usability problems, but also to assess the success of the new features in addressing the usability problems identified in the previous iteration.

6.3.1. Users

We recruited a total of 22 participants, mostly from young university staff and students volunteers both at the University of Lugano and at the University of Trento. In terms of programming skills, they were equally divided into programmers and non-programmers.

6.3.2. Method

The participants were given four tasks of growing complexity (in terms of the number of APIs to be mashed up), after receiving a short tutorial (5 min) in the form of a warm-up task (with the complexity of two APIs):

Warm-up task:	Get upcoming events in a place specified using Google Maps (two APIs).
Task 1:	Search Flickr images with location from Google Maps (two APIs).

Task 2: Show upcoming events in a selected location on the map. Get information about each event from Google (three APIs).
Task 3: Find slides about "Web APIs". For each slide found, show relevant videos, tweets, and images (four APIs).
Task 4: Create a mashup on your own (open task).

During the study we recorded the user sessions (video, audio, and screen) and asked the users to think aloud about their activities. The recordings were complemented by an informal interview as well as an exit questionnaire (containing Likert scale questions) asking them about their overall reaction and satisfaction with the tool. The aim of the extended post-study interviews was to have a deeper, but informal discussion with each participant with the opportunity to reflect on what was not captured by the questionnaires and to further discuss the rationale behind some answers. For instance, we asked "do you have any personal mashup/use-case of the tool?" and "why do you feel comfortable with the tool?".

6.3.3. *Results*

In terms of accuracy and efficiency (Figs. 10 and 11), the majority of participants completed all the tasks correctly and in a very short time (around 3 min on average), with a slightly better performance on the programmers′ side. Out of the total of 88 tasks, 11 programmers produced 86 correct tasks, while the 11 non-programmers achieved 84 correct tasks. Moreover, by the end of each user study session, the majority of participants felt confident about their mastery of the tool and reported on a high level of satisfaction in the exit survey (72% felt satisfied with the tool, 89% were interested in continuing using it, and 87% wanted to suggest it to their friends).

Our recorded observations, together with the feedback from participants through both the exit questionnaire and the informal discussion, reported positively on the individual features of the *NaturalMash* environment (Section 4). More in detail, the following percentage of participants reported that the features were helpful or very helpful for the completion of the given tasks: autocomplete (92%), inline search (91%), live execution (86%), ingredients toolbar (82%), and PbD (73%). The ingredients toolbar was more frequently used for component discovery and selection than the inline search with the text field (in average, 84% of component discovery and selection tasks were done using the ingredients toolbar). Instead, users employed the inline search feature when they were looking for a specific operation that could be perfectly described verbally.

Overall, the participants were engaged with the tool (77% felt that the tools were stimulating or very stimulating). In the open task, all created distinct, useful, and non-trivial mashups. One example was a mashup that finds an audio album in eBay and plays it in YouTube by first searching and finding the exact name of the album using the Last.fm API. Another example was a mashup that shows news, Flickr images, and tweets all related to a selected location on the map, and then allows users to share the results on Facebook. Indeed, some of the mashups created in the open task were actually meant to address a real pressing need of the participants. For instance, one of the participants created a mashup to automate the analysis of online presence within the tourism domain. The mashup searches tweets for a specific tourism-related keyword, and then for each tweet found, it searches for the Facebook profile given the name of the author of the tweets.

6.3.4. *Lessons learned*

Similar to the second iteration, the results collected in this iteration appear to support our design decisions towards meeting the requirements. The high rate of task completion time and correctness for the predefined tasks, the satisfaction results, and the post-study interviews suggest that we have fulfilled the requirement **R1** (usability). The results from the open tasks reassure that we have successfully met the requirements **R2** (useful expressiveness) and **R3** (usefulness).

In terms of usability problems, we observed that some users – especially non-programmers who lack algorithmic thinking abilities – would benefit from receiving suggestions not only for individual Event/Task labels but also for hints on how to compose them together in the right order.

Another major usability problem is concerned the way PbD is applied in the visual field, i.e., interacting with widgets results in the corresponding Event label being added to the text field. However, many participants confused capturing the general behavior of a widget (i.e., the event) with the recording of the concrete action on the widget they just triggered (e.g., a specific location they have clicked). For example, clicking on the map would add the map-click Event label (`when the map is clicked`) to the text. This is meant to be completed by appending Task labels (e.g., "`show upcoming events around location`") that form the body of the causal sentence. Indeed, we observed – in the same map example – the users correctly generating the Event labels using PbD and completing it with an imperative clause, expected to immediately see the results from the location they had originally selected (demonstrated) to create the causal sentence, as opposed to having to click again on the map to obtain the results. In other words, they did not realize that they had created a parametric mashup that shows events for any location on the Map. A similar problem occurred with other widgets supporting PbD, such as the table.

7. Related work

In recent years a number of mashup tools have been designed in both academia and industry. In this section, we give an overview and comparison of the state-of-the-art mashup tools in terms of the level of expressive power they offer as well as the end-user programming techniques they utilize. We also review the related works in natural language programming to put our approach in context.

Table 2
Comparison of the expressive power of state-of-the-art mashup tools.

Mashup tools	Data integration			Process integration				Presentation	
	Aggregate	**Filter**	**Extract**	**Sequence**	**Condition**	**Loop**	**Event**	**Layout design**	**Wiring**
NaturalMash	×	×	–	×	–	–	×	×	×
Mashroom [32]	×	×	×	×	–	–	–	×	–
Husky	×	×	–	×	–	–	–	–	–
Karma [33]	×	×	×	×	–	–	–	–	–
MashMaker [34]	×	×	×	–	–	–	–	×	×
Vegemite [35]	–	–	×	×	–	–	–	–	–
Yahoo! Pipes[a]	×	×	×	×	–	×	–	–	–
IBM Mashup Center[b]	×	×	–	×	–	×	–	×	×
JOpera [28]	–	–	–	×	×	×	×	–	–
JackBe Presto[c]	×	×	–	×	–	×	–	×	×
Marmite [36]	×	×	×	×	–	–	–	–	–
MashArt [37]	×	×	–	×	–	–	–	×	×
ResEval Mash [38]	×	×	–	–	–	–	–	–	–
MyCocktail[d]	×	×	–	–	–	×		×	×
MashableLogic[e]	×	×	–	×	–	–	–	×	×
Swashup [39]	×	×	–	×	×	×	×	–	–
WMSL [40]	×	×	–	×	×	×	×	–	–
ServFace [12]	–	–	–	–	–	–	–	×	×
DashMash [41]	×	×	–	–	–	–	–	×	×
Omelette [42]	–	–	–	–	–	–	–	×	×
CRUISE [43]	–	–	–	–	–	–	–	×	×
RoofTop [44]	–	–	–	–	–	–	–	×	×
FeedRinse	×	×	–	–	–	–	–	–	–
d.mix [45]	×	×	×	×	×	×	–	–	–
Open Mashups [46][f]	–	–	–	×	–	–	–	–	–
IFTTT[f]	–	–	–	×	×	–	×	–	–

[a] http://pipes.yahoo.com/
[b] http://www.ibm.com/software/info/mashup-center
[c] http://www.jackbe.com/
[d] http://www.ict-romulus.eu/web/mycocktail
[e] http://www.mashablelogic.com/
[f] https://ifttt.com/

7.1. *Mashup composition expressive power*

The expressive power of a mashup tool plays an important role to determine the success of the tool. Limiting the expressive power of a tool may restrict the diversity of the useful mashups it can create. Being able to only create the so-called "toy" mashups has been one of the main criticisms against existing mashup tools. This assertion can be backed by the fact that almost none of the mashups registered in the Programmable-Web (http://www.programmableweb.com/) — a Website listing thousands of mashups and popular Web APIs — have been created by end-users using a specific mashup tool.

However, increasing the expressive power may come at the price of sacrificing the usability of the tool. To this end, one solution is to design mashup tools in a domain-specific and semi-closed manner to merely offer "enough" expressive power suiting the needs of users in a specific domain of application [47]. Despite the issues concerning the closeness of domain-specific mashup tools (e.g., the assumptions about an application domain may change over time and thus it is frequently required to first identify the new assumptions and then redesign and customize the tool accordingly), a higher expressive power even in this case serves as an added bonus (or even a necessity) to the users. This is because higher expressive power fosters much more innovation and engagement, and avoids limiting the exploratory boundaries of users that are learning how to create diverse and rich mashups.

In this section, we show that *NaturalMash* offers a competitive level of expressive power compared with the state-of-the-art mashup tools, while it is still usable by absolute non-professional users (Section 6).

7.1.1. *Method*

We collected 26 tools (including *NaturalMash*) based on their availability in terms of being able to read about or use them to extract required information about them. Also, we made sure that there are enough mashup tools from both the industry and the academia.

We define the maximum expressive power of a mashup tool as the ability of composing mashups at all the layers of presentation, process integration, and data integration [48]. More specifically, data integration is accomplished by aggregating, filtering, and extracting (i.e., Web scraping [49]) different sources of data on the Web. Process integration involves application logic creation constructs such as conditions, loops, and sequences as well as catching events triggered by services or data sources (e.g., receiving updates in a stream of data). At the presentation layer, various

Table 3
A classification of the state-of-the-art mashup tool based on end-user programming techniques. The big **×** and smaller × represent, respectively, the main and secondary techniques.

Mashup tools	Spreadsheets	PbD	Visual P.	DSL	WYSIWYG	Form-based	Example modification	NLP
NaturalMash		×			×			**×**
Mashroom	**×**	×			×			
Husky	**×**							
Karma	**×**	**×**						
MashMaker	×	**×**			×			
Vegemite	×	**×**						×
Yahoo! Pipes			**×**			×		
IBM Mashup Center			**×**		×	×		
JOpera			**×**			×		
JackBe Presto			**×**	×	×	×		
Marmite	×		**×**			×		
mashArt			**×**		×	×		
ResEval Mash			**×**					
MyCocktail			**×**		×	×		
MashableLogic			**×**		×	×		
Swashup				**×**				
WMSL				**×**				
ServFace Builder			×		**×**	×		
DashMash					**×**			
Omelette					**×**	×		
CRUISE					**×**	×		
RoofTop					**×**	×		
FeedRinse						**×**		
d.mix				×			**×**	
Open Mashups								**×**
IFTTT								**×**

widgets can be superficially rearranged (i.e., layout design) or wired with each other (i.e., one end of a wire represents an event fired by a widget, and the other end is attached to a functionality offered by another widget).

7.1.2. Results and discussion

As it can be seen in Table 2, industrial mashup tools generally offer much more expressive power than academic tools. Compared with *NaturalMash*, only a few industrial mashup tools such as IBM Mashup Center and JackBe Presto as well as scripting languages like Swashup, WMSL, and d.mix offer as a competitive level of expressive power as *NaturalMash*. These scripting languages are not usually approachable by absolute non-programmers due to the learning barriers. The industrial mashup tools either are explicitly claimed to be usable only by expert users, or are provided with limited evidence of usability by non-professional users.

In terms of process integration, *NaturalMash* lacks the support for expressing loops and conditions, for which we have not yet noticed a clear reason to support (e.g., iterating over data can be automated without users having to explicitly use loops to do so). However, supporting data extraction in *NaturalMash* seems essential and can be done through incorporating a third-party visual Web scraper like Dapper (http://open.dapper.net/). Following the user-centered approach, we plan to continue fine-tuning the trade-off between the tool expressive power and the assumed end-user skills. The benefits of the added complexity would have to be evaluated with an additional user study.

7.2. End-user programming technique

Generally, existing mashup tools can be classified according to the end-user programming techniques they utilize as follows (Table 3):

- *Spreadsheets*: The advantage of using spreadsheets for creating mashups lies in their ease-of-use, intuitiveness, and expressive power to represent and manage complex data [50]. Mashroom [32] adapts the idea of spreadsheets and adds the nested tables to support complex data formats such as XML and JSON. Husky (http://www.husky.fer.hr/) is also another spreadsheet-based tool aiming at streamlining service composition. The main shortcoming of such tools is the lack of support for designing the mashup UI.
- *Programming by Demonstration* (*PbD*): PbD enables users to teach a system to do a task by demonstrating how the task is done [16]. Karma [33] allows users to extract, filter, and aggregate content on the Web through demonstration [49]. Intel MashMaker by [34] utilizes PbD to extract, store, manage, and integrate data from the Websites being browsed by the user. Vegemite [35] is another browser-based tool like MashMaker which adds scripting capabilities (based on CoScripter by [51]). The use of scripting allows users to augment and operate the extracted data. The focus of these tools is more on data extraction and visualization, and does not provide support for service composition and orchestration.
- *Visual programming*: Programming languages can also be expressed by visual symbols and graphical notations [52]. Visual programming is widely used by existing

mashup tools in the form of wiring diagrams, in which users drag-and-drop mashup components (visualized as boxes) and connect them to form a mashup. Examples are Yahoo! Pipes (http://pipes.yahoo.com/pipes/), IBM Mashup Center (http://www.ibm.com/software/info/mashup-center), JOpera [53], JackBe Presto (http://www.jackbe.com/enterprise-mashup/), Marmite [36], mashArt [37], ResEval Mash [38], MyCocktail (http://www.ict-romulus.eu/web/mycocktail), and MashableLogic (http://www.mashablelogic.com/). One of the strengths of visual languages is their ability to support more than one view at the same time [54], e.g. showing both the design time and run time environments in the same screen. However, a recent study conducted by [22] has suggested that the wiring paradigm that is widely used by mashup tools can be difficult to understand by non-programmers.

- *Textual domain-specific language* (*DSL*): DSLs are small languages targeted for solving certain problems in a specific domain [55]. DSLs can also be used as an EUP technique for reducing programming efforts [56]. Swashup [39] is a DSL for mashups, based on Ruby-on-Rails. It simplifies invocation, integration and aggregation of Web APIs and data sources. As an example of a DSL having its own paradigm and syntax, the Web Mashup Scripting Language (WMSL) is a scripting language for mashup development and automatic semantics and ontology creation [40]. Although these DSLs help us to reduce programming efforts, they still cannot be used by non-programmers due to the difficulty of learning their syntax and vocabulary [16].
- *What-You-See-Is-What-You-Get*: WYSIWYG enables users to create and modify a mashup on a graphical user interface which is similar to the one that will appear when the mashup runs. ServFace Builder [12], DashMash [41], Omelette [42], CRUISE [43], and RoofTop [44] exemplify WYSIWYG mashup tools. They provide a set of connectable configurable boxes whose current visual positions in the design time are the same as during the runtime. Since users always see the resulting mashup, the whole development process is streamlined. Another potential benefit is the increase of the tool directness. Users place visual objects exactly in the places where they are meant to be. However, the application logic of a mashup such as data filtering and conversion happens in the backend where is not visible in the graphical user interface, and therefore, is not directly accessible for modification using a pure WYSIWYG tool.
- *Form-based*: In form-based interaction, users are asked to fill out a form to create a new or change the behavior of an existing object. *FeedRinse* (http://feedrinse.com/) provides a form-based mechanism to create data mashups by filtering and aggregating Web feeds. Filling out online forms has nowadays become an ordinary task for end-users on the Web. This can be interpreted as a proof for "naturalness" of form-based tools [57]. Form-based tools cannot handle complex composition patterns [58] such as the ones used for mashups.
- *Programming by example modification*: Another powerful technique to remove the burden of programming is to let end-users modify and change the behavior of existing examples, instead of programming from scratch [59]. *d.mix* [45] allows users to sample elements of a Website, and then generates the corresponding source code producing the selected elements. These source codes are stored in a repository, where they can be discovered and edited by others. Provided that adequate mashup examples are available, in most cases the modification of a mashup example or the customization of a predefined mashup template requires a small effort. Still, searching for appropriate examples as a suitable starting point for the work is a challenging task for non-programmers. With the ever increasing number of Web APIs, providing adequate mashup examples derived from all possible combinations of these APIs is not feasible.
- *Natural language programming*: *Natural Mashup* [46] incorporates natural language programming for composing mashups on mobile devices. In the area of personal information management, Belaunde and Hassen [46] presented *Automate* that uses a simplified CNL for context-sensitive personal automation. Another similar system is IFTTT (https://ifttt.com) which, even though it is based on natural language, restricts the user's input using a structured visual editor. Also, IFTTT only allows users to create mashups based on a specific control-flow pattern (if this then that) using a predefined list of components. Even though natural language is considered as a natural way for humans to command computers [60–62] it cannot be efficiently used for user interface integration and design which are integral part of mashup development [63].

The novelty of *NaturalMash* distinguishing it from the above-mentioned tools lies in its novel hybrid end-user programming technique, being an effective combination of natural language programming (back-end development), WYSIWYG, and PbD (front-end design). While natural language programming is intuitive and can give a high level of expressive power for developing the mashup back-end, it cannot be effectively used for the front-end design. On the other hand, WYSIWYG and PbD provide intuitive direct manipulation facilities for the front-end design, while they cannot provide adequate expressive power for developing the back-end. The reason for selecting and combining these techniques lies in the fact that they augment one another's strengths and compensate for one another's weaknesses. As it was seen (Sections 7.1 and 6), this has resulted in *NaturalMash* offering a high level of expressive power, while it is still usable by absolute non-professional users. To the best of our knowledge, *NaturalMash* is the only mashup tool that has developed such unique capabilities.

7.3. Natural language programming

Natural language programming has a long history and has been recently successfully used in different application domains. For instance, the system presented in [64] allows users to use natural language to express formal rules defined in the RoboCup coach language. System English (http://www.system-english.com/) is also another example that enables users to refer to MATLAB function calls using regular sentences. In the area of personal information management, Van Kleek [65] presented *Automate* that uses a simplified CNL for context-sensitive personal automation. CoScripter [66] is a natural language based script for automating Web browsing activities. The idea of sloppy programming utilized by CoScripter is closely related to the natural language programming style of *NaturalMash*. More recently, Smart et al. [67] proposed a controlled natural language interface to simplify the development of semantic media wikis.

However, there are a number of critical viewpoints on the use of natural language programming as a replacement for existing formal programming languages [68,69]. Many are concerned with the so-called general-purpose nature of natural programming languages. For example, METAFOR [70] could transfer a natural-language description of a program to the skeleton of the program rendered in one of the supported general-purpose programming languages including Python, Lisp and Java. In this case, the issues of ambiguity and imprecision discussed in the literature become relevant.

To support our choice in favor of a natural language approach, we argue that the *NaturalMash* CNL is not a general-purpose language but a specific high level language tuned specifically for the domain of mashup composition. Additionally, *NaturalMash* is the first to support natural language programming in a live fashion, fostered by means of direct manipulation and immediate preview of the mashup user interface through a WYSIWYG interface.

8. Discussion

From a technical perspective, one of the main tasks of mashup tools is to hide the heterogeneity and complexity of Web technologies behind an easy-to-understand abstraction. From a user modeling perspective, the challenge lies in the broad diversity of user skills that need to be targeted and in the large number of domains in which mashups can be applied to. The evaluations presented in this paper (iteration formative evaluations in Section 6 and comparative expressive power evaluation in Section 7.1) showed that the design of *NaturalMash* has found a sweet spot within this trade-off. In the following we enumerate a set of important lessons learned from our experience with the design and evaluation of *NaturalMash*. We believe that these lessons can help addressing the mentioned trade-offs towards the design of efficient and effective mashup tools letting non-professional users create sophisticated mashups.

- *Design at meta-level*: Mashups can be built and used in different domains of applications. These domains range from daily utilities of Web users (i.e., consumer market) to narrowly specialized domains and enterprise environments. It is important to identify the application domain in which users are willing to and have shown a clear need to develop mashups [47]. This is a well known problem in EUD, as the importance of task and domain specificity was already pointed out by Nardi [13] in the context of end-user programming. From the point of view of the tool designer, a closed approach which narrowly targets a single application domain may present some limitations. Application domains usually change over time. This may result in changes of the initial requirements and assumptions based on which the mashup tool was designed. Also, a mashup tool targeting a specific domain may not perfectly fit into, or be easily transformed into, a tool targeting another domain. Therefore, we advocate a meta-design approach [71], whereby a generic mashup meta-tool is designed and from it domain-specific mashup tools can be derived by its users over time. The meta-design elements in *NaturalMash* include

  (i) selection of the available ingredients (components),
  (ii) the look and naming of ingredients, and most importantly
  (iii) the language style used to describe them.

  In the latter case, the labels associated with the ingredients used in the text field can be changed by users to tailor the "language" of the tool to their domain.
- *Support different levels of expressiveness*: An effective mashup tool should provide enough expressive power to allow the creation of sophisticated mashups. On the other hand, the usability of a system may be affected by the degree of expressiveness it offers. To avoid this issue, Mørch [72] proposes three levels of user tailoring including customization, integration, and extension. In the case of mashups, all these three levels are relevant and thus should be supported by a mashup tool. Customization means modifying an existing mashup through parameterization or user interface manipulation. Integration (discussed in Section 7.1) is the process of creating new mashups and should be allowed at all the levels of data, business logic, and presentation tiers. Extension allows extending the functionality of the mashup tool by developing new ingredients.

  In *NaturalMash*, customization is enabled through the visual field. Integration is mainly supported by natural language programming. The plan is also to enable extension for professional users to create and add ingredients to the tool library.
- *Build an online community*: Online communities are of importance in boosting the ability of users to learn how to use the tool through creating, sharing, and reusing mashups, knowledge, and experience [13]. Crowdsourcing can also be applied in an online community to persuade professional users to enrich the ingredients library for non-professional users [73].

  We still plan to investigate the mentioned impacts of online communities in the context of mashup EUD. More importantly, we are interested in in-the-wild testing of our meta-design using an online community.

9. Conclusion

In this paper we presented *NaturalMash*, a "natural" tool for end-user mashup development. *NaturalMash* is based on a novel hybrid composition technique combining a controlled natural language tuned for mashup development with an interactive WYSIWYG and drag-and-drop interface allowing PbD and live execution preview and modification of the resulting mashup user interface. The design of *NaturalMash* has adopted an incremental, user-driven approach in which iterative formative evaluations inform the next steps to be taken to improve the usability of the tool. The results of our formative evaluations helped us to identify several usability problems and gather ideas on how to address these problems. Also, the results provided positive feedback about the tool design, demonstrated its usability by non-professional users as well as its high level of expressive power, compared with existing mashup tools, to let users create useful and non-trivial mashups.

For the next iteration, in the near future we plan to

(i) enable autocompletion of mashup compositions based on semantic and syntactic matching to assist non-programmers with limited algorithmic thinking capabilities,
(ii) boost the use of Programming by Demonstration to generate output in addition to behavior,
(iii) enhance the live execution feature by saving and restoring the current state of all user interface widgets across compilation cycles, and
(iv) propose and adopt a set of design guidelines for enhancing the usability of the widgets.

We also plan to explore the following open research directions. First, our intention is to target possible specific domains of application for mashups, so that we can find end users that may lack programming skills but have a significant expertise in a given domain. This way, the next user study can take advantage of the user's *motivation* and *domain-relevant* skills, as these play an important role in EUD [71]. To this end, we are considering to port the tool to use non-English languages. We are also interested in applying end-user software engineering [74] in the realm of mashups. For instance, end-user debugging [75], testing, and versioning [76] are all very important open challenges in end-user mashup development.

Acknowledgements

We are grateful for the support and the expertise of Monica Landoni and Antonella De Angeli with the preparations and the analysis of the formative evaluation results. This work is partially supported by the Swiss National Science Foundation with the SOSOA project (SINERGIA grant nr. CRSI22 127386).

References

[1] T. O'Reilly, What is Web 2.0: design patterns and business models for the next generation of software, Commun. Strateg. 20 (2007) 17.
[2] D. Benslimane, S. Dustdar, A. Sheth, Services mashups: the new generation of web applications, IEEE Internet Comput. 12 (2008) 13–15.
[3] A. Jhingran, Enterprise information mashups: integrating information, simply, in: Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB Endowment, pp. 3–4.
[4] C. Anderson, The Long tail: why the future of business is selling less of more, Hyperion, 2008.
[5] M. Eisenstadt, Does elearning have to be so awful? (time to mashup or shutup), in: Proceedings of the 7th IEEE International Conference on Advanced Learning Technologies (ICALT), IEEE, pp. 6–10.
[6] C. Goble, R. Stevens, et al., State of the nation in data integration for bioinformatics, J. Biomed. Inform. 41 (2008) 687–693.
[7] M.N. Kamel Boulos, S. Wheeler, The emerging web 2.0 social software: an enabling suite of sociable technologies in health and health care education1, Health Inf. Libr. J. 24 (2007) 2–23.
[8] A. Bellucci, A. Malizia, P. Diaz, I. Aedo, Framing the design space for novel crisis-related mashups: the estorys example, in: Proceedings of the 7th International ISCRAM Conference, 2010.
[9] H. Lieberman, F. Paternò, M. Klann, V. Wulf, End-user development: an emerging paradigm, in: End User Development, Springer, 2006, pp. 1–8.
[10] F. Casati, How end-user development will save composition technologies from their continuing failures, in: End-User Development, Springer, 2011, pp. 4–6.
[11] S. Aghaee, M. Nowak, C. Pautasso, Reusable decision space for mashup tool design, in: Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, ACM, pp. 211–220.
[12] T. Nestler, M. Feldmann, G. Hübsch, A. Preußner, U. Jugel, The servface builder—a WYSIWYG approach for building service-based applications, in: Web Engineering, Springer, 2010, pp. 498–501.
[13] B.A. Nardi, A Small Matter of Programming: Perspectives on End User Computing, MIT Press, 1993.
[14] L.A. Miller, Natural language programming: styles, strategies, and contrasts, IBM Syst. J. 20 (1981) 184–215.
[15] J. Rode, M.B. Rosson, Programming at runtime: requirements and paradigms for nonprogrammer web application development, in: Proceedings of the IEEE Symposium on Human-Centric Computing Languages and Environments, 2003, pp. 23–30.
[16] A. Cypher, D.C. Halbert, Watch What I Do: Programming by Demonstration, MIT Press, 1993.
[17] S. Aghaee, C. Pautasso, Live mashup tools: challenges and opportunities, in: Proceedings of the 1st International Workshop on Live Programming (LIVE) 2013.
[18] R. Mihalcea, H. Liu, H. Lieberman, NLP (natural language processing) for NLP (natural language programming), in: Computational Linguistics and Intelligent Text Processing, Springer, 2006, pp. 319–330.
[19] S. Casteleyn, F. Daniel, P. Dolog, M. Matera, Engineering Web Applications, Springer Publishing Company Incorporated, 2009.
[20] S. Aghaee, C. Pautasso, A. De Angeli, Natural end-user development of mashups, in: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2013.
[21] S. Aghaee, C. Pautasso, End-user programming for web mashups: open research challenges, in: Proceedings of the 11th International Conference on Current Trends in Web Engineering, 2012, pp. 347–351.
[22] A. Namoun, T. Nestler, A. De Angeli, Service composition for non-programmers: prospects, problems, and design recommendations, in: Proceedings of the 8th IEEE European Conference on Web Services (ECOWS), IEEE, 2010, pp. 123–130.
[23] N. Collins, A. McLean, J. Rohrhuber, A. Ward, Live coding in laptop performance, Org. Sound 8 (2003) 321–330.
[24] S.L. Tanimoto, VIVA: a visual language for image processing, J. Vis. Lang. Comput. 1 (1990) 127–139.
[25] D.A. Norman, S.W. Draper, User Centered System Design: New Perspectives on Human–Computer Interaction, L. Erlbaum Associates Inc., 1986.
[26] A. Repenning, A. Ioannidou, What makes end-user development tick? 13 design guidelines, in: End User Development, Springer, 2006, pp. 51–85.
[27] G. Bergmann, I. Ráth, G. Varró, D. Varró, Change-driven model transformations, Softw. Syst. Model. 11 (2012) 431–461.
[28] C. Pautasso, G. Alonso, The JOpera visual composition language, J. Vis. Lang. Comput. 16 (2005) 119–152.
[29] A. Strunk, QoS-aware service composition: a survey, in: Proceedings of the 8th IEEE European Conference on Web Services (ECOWS), IEEE, pp. 67–74.
[30] P. Lubbers, B. Albers, Harnessing the power of HTML5 web sockets to create scalable real-time applications presentation, Web2.0 Expo SF, 2010.
[31] K. Vredenburg, J.-Y. Mao, P.W. Smith, T. Carey, A survey of user-centered design practice, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 2002, pp. 471–478.
[32] G. Wang, S. Yang, Y. Han, Mashroom: end-user mashup programming using nested tables, in: Proceedings of the 18th International Conference on World Wide Web, ACM, 2009, pp. 861–870.

[33] R. Tuchinda, C.A. Knoblock, P. Szekely, Building mashups by demonstration, ACM Trans. Web (TWEB) 5 (2011) 16.
[34] R. Ennals, E. Brewer, M. Garofalakis, M. Shadle, P. Gandhi, Intel mash maker: join the web, ACM SIGMOD Record 36 (2007) 27–33.
[35] J. Lin, J. Wong, J. Nichols, A. Cypher, T.A. Lau, End-user programming of mashups with vegemite, in: Proceedings of the 14th International Conference on Intelligent User Interfaces, ACM, 2009, pp. 97–106.
[36] J. Wong, J.I. Hong, Making mashups with marmite: towards end-user programming for the web, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ACM, 2007, pp. 1435–1444.
[37] F. Daniel, F. Casati, B. Benatallah, M.-C. Shan, Hosted universal composition: models, languages and infrastructure in mashart, in: Conceptual Modeling-ER 2009, Springer, 2009, pp. 428–443.
[38] M. Imran, F. Kling, S. Soi, F. Daniel, F. Casati, M. Marchese, Reseval mash: a mashup tool for advanced research evaluation, in: Proceedings of the 21st International Conference Companion on World Wide Web, ACM, 2012, pp. 361–364.
[39] E.M. Maximilien, H. Wilkinson, N. Desai, S. Tai, A domain-specific language for web apiAPI and services mashups, in: Proceedings of the International Conference on Service-Oriented Computing (ICSOC 2007), Springer, 2007, pp. 13–26.
[40] M. Sabbouh, J. Higginson, S. Semy, D. Gagne, Web mashup scripting language, in: Proceedings of the 16th International Conference on World Wide Web, ACM, pp. 1305–1306.
[41] C. Cappiello, M. Matera, M. Picozzi, G. Sprega, D. Barbagallo, C. Francalanci, Dashmash: a mashup environment for end user development, in: Web Engineering, Springer, 2011, pp. 152–166.
[42] O. Chudnovskyy, T. Nestler, M. Gaedke, F. Daniel, J.I. Fernández-Villamor, V. Chepegin, J.A. Fornas, S. Wilson, C. Kögler, H. Chang, End-user-oriented telco mashups: the omelette approach, in: Proceedings of the 21st International Conference Companion on World Wide Web, ACM, 2012, pp. 235–238.
[43] S. Pietschmann, M. Voigt, A. Rümpel, K. Meißner, Cruise: composition of rich user interface services, in: Web Engineering, Springer, 2009, pp. 473–476.
[44] V. Hoyer, F. Gilles, T. Janner, K. Stanoevska-Slabeva, SAP research rooftop marketplace: putting a face on service-oriented architectures, in: Proceedings of the 2009 Congress on Services—I, IEEE, pp. 107–114.
[45] B. Hartmann, L. Wu, K. Collins, S.R. Klemmer, Programming by a sample: rapidly creating web applications with d.mix, in: Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology, ACM, 2007, pp. 241–250.
[46] M. Belaunde, S.B. Hassen, Service mashups using natural language and context awareness: a pragmatic architectural design, in: Proceedings of the 15th IEEE International Enterprise Distributed Object Computing Conference Workshops (EDOCW), IEEE, 2011, pp. 404–411.
[47] F. Casati, F. Daniel, A.D. Angeli, M. Imran, S. Soi, C.R. Wilkinson, M. Marchese, Developing mashup tools for end-users: on the importance of the application domain, Int. J. Next Gener. Comput. Perpetual Innov. 3 (2012).
[48] J.J. Hanson, Mashups: Strategies for the Modern Enterprise, Addison-Wesley Professional, 2009.
[49] M. Schrenk, Webbots, spiders, and screen scrapers: a guide to developing Internet agents with PHP/CURL, No Starch Press, 2012.
[50] D.D. Hoang, H.-y. Paik, B. Benatallah, An analysis of spreadsheet-based services mashup, in: Proceedings of the 21st Australasian Conference on Database Technologies, vol. 104, Australian Computer Society, Inc., 2010, pp. 141–150.
[51] G. Little, T.A. Lau, A. Cypher, J. Lin, E.M. Haber, E. Kandogan, Koala: capture, share, automate, personalize business processes on the web, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ACM, 2007, pp. 943–946.
[52] N.C. Shu, Visual Programming, Van Nostrand Reinhold Co., 1988.
[53] C. Pautasso, Composing restful services with JOpera, in: Software Composition, Springer, 2009, pp. 142–159.
[54] B.A. Myers, Taxonomies of visual programming and program visualization, J. Vis. Lang. Comput. 1 (1990) 97–123.
[55] A. Van Deursen, P. Klint, J. Visser, Domain-specific languages: an annotated bibliography, ACM Sigplan Not. 35 (2000) 26–36.
[56] H. Prähofer, D. Hurnaus, H. Mössenböck, Building end-user programming systems based on a domain-specific language, in: Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling (DSM), 2006, p. 33.
[57] J.C. Thomas, J.D. Gould, A psychological study of query by example, in: Proceedings of the National Computer Conference and Exposition, ACM, May 19–22, 1975, pp. 439–445.
[58] R. Jeffries, J. Rosenberg, Comparing a form-based and a language-based user interface for instructing a mail program, ACM SIGCHI Bull. 17 (1986) 261–266.
[59] G.M. Olson, S. Sheppard, E. Soloway, Empirical Studies of Programmers: Second Workshop, Ablex, 1987.
[60] C. Green, et al., A summary of the psi program synthesis system, in: Proceedings of the 5th International Conference on Artificial Intelligence, vol. 1, 1977, pp. 380–381.
[61] G.E. Heidorn, Automatic programming through natural language dialogue: a survey, IBM J. Res. Dev. 20 (1976) 302–313.
[62] E. Kaufmann, A. Bernstein, How useful are natural language interfaces to the semantic web for casual end-users? in: The Semantic Web, Springer, 2007, pp. 281–294.
[63] J. Yu, B. Benatallah, R. Saint-Paul, F. Casati, F. Daniel, M. Matera, A framework for rapid integration of presentation components, in: Proceedings of the 16th International Conference on World Wide Web, ACM, pp. 923–932.
[64] R.J. Kate, Y.W. Wong, R.J. Mooney, Learning to transform natural to formal languages, in: Proceedings of the National Conference on Artificial Intelligence, vol. 20, Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, p. 1062.
[65] M. Van Kleek, B. Moore, D.R. Karger, P. André, et al., Atomate it! end-user context-sensitive automation using heterogeneous information sources on the web, in: Proceedings of the 19th International Conference on World Wide Web, ACM, 2010, pp. 951–960.
[66] G. Leshed, E.M. Haber, T. Matthews, T. Lau, CoScripter: automating & sharing how-to knowledge in the enterprise, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ACM, 2008, pp. 1719–1728.
[67] P.R. Smart, J. Bao, D. Braines, N.R. Shadbolt, Development of a controlled natural language interface for semantic mediawiki, in: Controlled Natural Language, Springer, 2010, pp. 206–225.
[68] E.W. Dijkstra, On the foolishness of “natural language programming”, in: Program Construction, Springer, 1979, pp. 51–53.
[69] S.R. Petrick, On natural language based computer systems, IBM J. Res. Dev. 20 (1976) 314–325.
[70] H. Liu, H. Lieberman, Metafor: Visualizing stories as code, in: Proceedings of the 10th International Conference on Intelligent User Interfaces, ACM, 2005, pp. 305–307.
[71] G. Fischer, E. Giaccardi, Y. Ye, A.G. Sutcliffe, N. Mehandjiev, Meta-design: a manifesto for end-user development, Commun. ACM 47 (2004) 33–37.
[72] A. Mørch, Three levels of end-user tailoring: customization, integration, and extension, Comput. Des. Context 20 (1997) 51–76.
[73] M. Nebeling, S. Leone, M.C. Norrie, Crowdsourced web engineering and design, in: Web Engineering, Springer, 2012, pp. 31–45.
[74] A.J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, et al., The state of the art in end-user software engineering, ACM Comput. Surv. (CSUR) 43 (2011) 21.
[75] J. Cao, K. Rector, T.H. Park, S.D. Fleming, M. Burnett, S. Wiedenbeck, A debugging perspective on end-user mashup programming, in: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), IEEE, 2010, pp. 149–156.
[76] S.K. Kuttal, A. Sarma, G. Rothermel, History repeats itself more easily when you log it: versioning for mashups, in: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), IEEE, 2011, pp. 69–72.