UNIVERSITY OF TARTU

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Institute of Computer Science

Information Technology

Hans Mäesalu

Automated Rule-Based Selection and Instantiation of Layout Templates for Widget-Based Microsites

Master's thesis (30 ECTS)

Supervisor: Peep Küngas

Author: …..... “.....” May 2013

Supervisor: …..... “.....” May 2013

Approved for defence

Professor: …..... “.....” May 2013

Tartu 2013

Abstract

This thesis proposes a rule-based widget and layout template matchmaking solution for widget-based microsites. The solution takes as an input a set of widget descriptions and a set of layout templates with widget placeholders and returns a microsite, where the most suitable template has been instantiated with corresponding widgets. Matchmaking is based on applying a rule engine to metadata of widgets and placeholders about their content categories and dimensions,. Additional usability rules are used to further improve the results with respect to commonly accepted usability guidelines. Such a solution makes it possible to modularly enhance the usability results in the future simply by adding new usability rules and layout templates. Furthermore, the solution can be applied in mashup creation tools for layout selection.

The proposed solution has been implemented and is called Auto Microsite in this thesis. The system consists of a server-side and a client-side component. The server-side component matches widgets with layout template placeholders according to the given rules by using the OO jDREW RuleML engine. The client-side is responsible for presenting the mashup appropriately for the client device. The latter is based on OpenAjax Hub 2.0 framework, which enables secure sandboxing and communication of widgets in the generated microsite. Furthermore, OpenAjax Metadata 1.0 specification is used in this thesis to package the widgets such that they could be easily reused.

In order to evaluate the Auto Microsite system in practice two proof of concept (PoC) scenarios were implemented. The first scenario visualized “Hourly labour costs in Euros (European Union 1997-2008)” data using widgets for a map, a table and a summary. In the second scenario, also data was queried through a SOAP service and a Web site. In the scenario data was visualized using two table widgets and a map widget. The SOAP service and queries to the Web site were packaged as non-visual widgets to fit the framework. The POCs demonstrate that the Auto Microsite system is able to construct widget-based microsites. Furthermore, the framework is capable of constructing also more complex Web applications, with several pages and more content widgets, by adding new rules and templates.

Table of Contents

1 Introduction

Internet is the largest repository of human knowledge, but often this knowledge is scattered around different information systems and is difficult to use. To simplify integration of information from various sources to meet specific user requirements the concept of mashups has arisen.

Mashups are by their nature microsites that concentrate on solving a single user-oriented problem and combine data and functionality from different online sources. They are intended to add value by combining data in a meaningful way. Mashups are generally composed of widgets, which build up the user interface of a mashup and provide the necessary data.

A widget, sometimes refereed to as a gadget, is a small reusable application component, normally packaged and enriched with package-specific metadata. W3C categorizes widgets as regular desktop widgets, mobile widgets, and web widgets [1]. Desktop and mobile widgets are installed on a client device, although often still communicate with web services to receive additional information, like weather reports or news. Web widgets are deployed on web sites. They are built using web technologies, such as HTML, CSS, JavaScript and Flash. This thesis concentrates on web widgets.

Unfortunately with existing technologies combining widgets into mashups is still a time consuming and complicated task, often requiring programming knowledge. It becomes even worse when several different programming languages are involved. Modern web mashups are marked up using HTML, designed using CSS, may contain widgets written in JavaScript, Flash and Silverlight, and consume data in XML, JSON and CSV format. Namoun et al. [2] discovered that users are interested in mashing up different services because they see it as a way to increase productivity, but they fail to do so because average user without computer science background has poor understanding of technical details of web services and composing them. And when a mashup is intended to be used by several people, usability and accessibility will become crucial.

Usability is about effectiveness, efficiency and satisfaction with which users achieve their intended goals [3]. Accessibility is about ensuring equivalent access for everyone, including people with disabilities or devices with limited capabilities [4]. Unfortunately usability and accessibility are mostly considered relevant, just after users start complaining

about it or sales start to drop. However, a good Web site should be built with keeping good usability and accessibility in mind. These qualities also often suffer because mashup developers generally are not usability experts. Thus even though they start with good intentions in mind they simply lack the knowledge to consider all the aspects of usability and accessibility.

The aim of this thesis is to build a framework and a demonstrator that would automate the creation of visually simple Web pages [5], to which mashups are categorized. Simple web pages concentrate on one subject, have few links with easy to understand texts, have few and small images, use few and light colors and fit on screen without scrolling. Furthermore, visually simple web pages generally do not contain forms or advertisements.

First, the thesis will identify usability guidelines that are applicable to mashup layout composition. For example, many navigational guidelines are not applicable in case of visually simple mashups because such mashups only contain one or very few pages and fit on screen without the need for scrolling. On the other hand, guidelines that apply to content positioning are still valid because users consider mashup web sites as regular web sites and expect to find objects in familiar locations. Additional guidelines will be identified by studying Web sites that have been acknowledged for their good usability. Such Web sites are taken from the list of "15th Annual Webby Awards Nominees & Winners" [6].

Second, a set of machine-understandable formats for describing the identified guidelines and widget metadata will be compiled. The set of formats has to be flexible enough to be usable in ever-changing Internet technology landscape. At the same time the set of formats has to be easy to understand and based on existing standards, to minimize the learning curve. By using these formats a knowledge base of usability guidelines and a sample set of existing widgets will be constructed. This knowledge base will be extensible, to make it possible to add new rules in the future to further improve layout generation and new widgets for supporting wider set of mashup applications.

Third, the framework will be validated with two proof of concept scenarios, to prove its applicability in solving real world problems. For the first scenario, a mashup visualizing European average wage data over past years will be composed. The second, a more complex scenario, will integrate data from a Inforegister.ee SOAP service with a data widget and will semantically integrate syntactically different messages.

The rest of the thesis is organized as follows. Chapter 2 gives an overview of existing work related to mashups. Chapter 3 introduces technologies and standards used for the solution proposed in the thesis. Chapter 4 introduces the architecture of the framework application using a 4+1 architectural view model. Chapter 5 gives an overview of the implementation. Chapter 6 introduces two case studies that are intended to be solved by the framework described in this thesis. Finally, Chapter 7 concludes the work that has been done and Chapter 8 gives an overview of possible future work.

2 Related work

Before creating a new mashup construction tool it is important to study existing tools and techniques for mashups construction. Also, since usability is considered important for this thesis, literature on usability is reviews in this Chapter.

2.1 Web application and mashup usability

Mashups are generally seen as regular Web applications. There are many studies on Web application usability [7][8][9]. For example, Thung [7] proposes several navigational patterns, which were validated on the Web site of University Sains Malaysia School of Computer Sciences. More specifically, the study proposed set-based navigation, which means that content is distributed into sets of similar information, to be used together with search features, to make search more effective. Schmidt et al. [8] studied how changing design variables, such as font size or color, would affect usability and found that users may be willing to give up some usability for aesthetically pleasing Web site. Fox and Naidu [9] studied popular social networking sites and found that even though Facebook does not adhere to traditional usability guidelines, it had the most efficient user interface. Based on the studies, several books [10][11] about usability guidelines and patterns have been written. For example Vora [10] reviews patterns covering all aspects of web applications, from forms and navigation guidelines to accessibility issues. A book by Leavitt and Shneiderman [11] elaborates guidelines suggested by the U.S. Department of Health and Human Services.

In order to scientifically evaluate effectiveness of developed usability guidelines, eye tracking studies have been performed. For example, Dahal [12] reports a study of 25 USA university Web sites, which were tested on students. The study reports that people spend most of their time watching the main menu of a Web site, which is expected to be located at the top or on the left side of the screen, and the main contents of a Web site, which is expected to be located at the center of the screen. Russel [13] found that users first fixate their view in the top left and center areas of a Web sites, so this is where the most important pieces of contents should be located. It was also seen that it is possible to attract more attention to some location by coloring it differently from the rest of the Web site. For a usability study Goldberg and Kotval [14] constructed two interfaces, one of which was

poorly organized and another which was well organized. From eye tracking results it was seen that poorly organized interface may result in less efficient search behavior, which in return increases the time it takes to perform a specific task. However, sometimes eye tracking can give results that conflict with common usability guidelines. For example, it is commonly suggested to position main menu in the top or on the left hand side of a Web site [10][11], but Bailey et al. [15] found that users used rightaligned menus more efficiently, even more so on laptop computers. This was further studied by McCarthy et al. [16] who noticed that on the first page view users searched for the menu on the left hand side of the Web page. This made them slower to locate and use a menu on the right hand side of the Web page. But on consecutive page views users remembered where to look for the menu and the difference disappeared. This finding is also supported by Jacob's Law of Web user experience [16]:

"Users spend most of their time on other sites. Thus, anything that is a convention and used on the majority of other sites will be burned into the users' brains and you can only deviate from it on pain of major usability problems."

Anyway, the progress in identification of usability guidelines has resulted in so many suggestions that manual usability evaluation has become time consuming and error prone. In order to simplify usability evaluation automated usability evaluation tools have been developed. Dingli and Mifsud [3] introduce one such framework, USEFul. The framework uses a database of usability patterns, that have been collected from various usability guidelines, and evaluates Web sites with respect to them. The framework tool was validated with respect to manual usability evaluation results by usability professionals and it was revealed that even though it was not able to identify all the usability violations, that were manually identified by professionals, it was still able to identify usability violations that were not found manually. Therefore, it was concluded that for the best results automated and manual usability evaluation should be used together. The same conclusion was reached by Harty [17] who studied keyboard navigation on Web pages.

However, mashups often have characteristics that separate them from regular web sites. Cappiello et al. [18] proposed a quality model for mashups taking into account the component-based nature of mashups and other common mashup characteristics, for example that mashups are generally laid out on a single page. They identified that mashup quality depends on two major aspects: the components and the composition. The model of Cappiello et al. [18] consists of 3 dimensions: data quality, presentation quality and

composition quality. Data quality measures in which extent the data used in the mashup is accurate, complete, timely, consistent and available. Presentation quality measures whether the mashup is usable and accessible. Finally, composition quality shows whether the mashup introduces added value, whether suitable components are properly used, consistent and available. Figure 2.1 shows an overview of the mashup quality model.

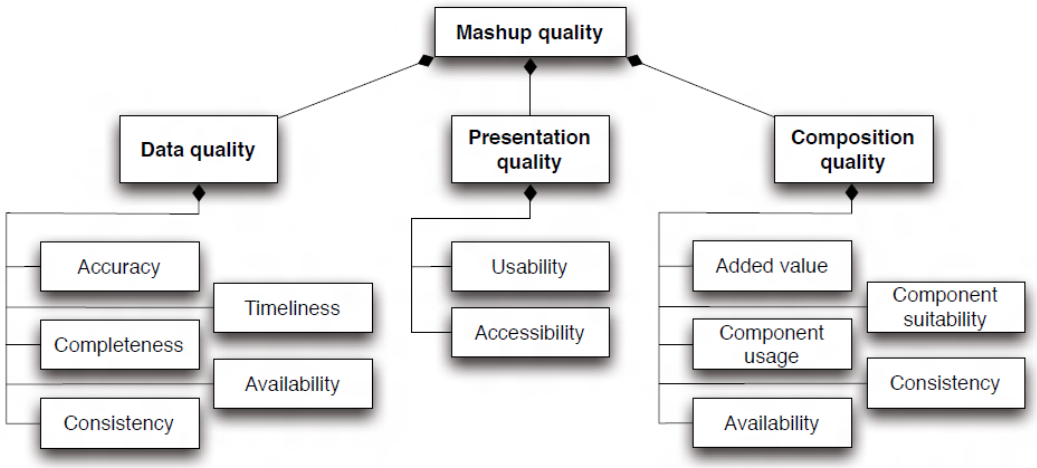


Figure 2.1: The mashup quality model by Cappiello et al. [18]

2.2 Widgets

W3C categorizes widgets as [1] desktop or mobile widgets and Web widgets. Desktop and mobile widgets are generally binary programs that are installed on the client device. In recent years usage of web technologies, like HTML5, in desktop widgets has become increasingly common. Web technologies are generally considered easier to master thus lowering the technological barrier. An instance of platforms, where widgets are written using HTML and JavaScript, is Tizen[1]. Another popular mobile platform for widgets is Android[2]. On Android widgets are used to control phone functionality, for example silence the phone, display recent news or whether, and access media, for example to display a picture or to play music.

Web widgets are written entirely by using web technologies, such as HTML, CSS and JavaScript. Sometimes Web widgets also include embedded content like Flash, Silverlight or Unity. However, unlike desktop widgets, web widgets are not installed on a client device - they are embedded into Web sites. This means that Web widgets run in restricted browser

1 https://www.tizen.org/
2 http://www.android.com/

sandboxes and, compared to desktop widgets, they do not have access to file-system and, furthermore, they cannot control or monitor other client resources. Web widgets often contain a server component, which provides data for the widget. One of the most popular widget platform is Facebook with over one billion users. Widgets on Facebook are used to personalize users' profiles, they can be used to share favorite music or travel locations. Additionally, Facebook provides its own public widgets, that can be used on other Web sites or in mashups.

2.3 Mashup tools

Volker Hoyer and Marco Fischer [19] have compiled an overview of existing mashup tools. The authors broadly classify mashup tools to catalog and editor tools. Catalog tools are collections of existing resources and they also mediate communication between different resources. Catalogs are further segmented into adapters, which deal with mediation of communication between resources, and repositories, which organize resources and widgets. Editor tools are used for combining resources into new applications. Editors are further distributed into transformation / aggregation tools, which deal with combining data from different sources, and presentation layer tools, which display data from different sources. Often real-world mashup tools combine both aspects.

A popular example of an adapter mashup tool is Yahoo!'s Dapper[3]. Dapper is a tool that allows users to extract information from Web sites into feeds that can then be used as data sources in mashups. Another adapter tool, Firecrow, is introduced by Maras et al. [20] with the aim of providing means to extract reusable user interface parts from existing Web sites that could then be used in mashups. It is implemented as a browser extension that works by recording interactions and then based on collected data extracts necessary CSS, HTML and JavaScript resources for the specific action.

Presentation and repository mashup tools are often combined. For example iGoogle[4] and Netvibes[5] are both tools that contain a repository of widgets and a customizable web portal. These allow users to create their own personalized portals by browsing the repository of widgets, such as weather information, clock and news, and adding them to their portals. Widgets in such environments generally cannot communicate with each other and have very limited customization options, which makes such tools very easy to use

3 http://open.dapper.net/
4 http://www.google.com/ig
5 http://www.netvibes.com

independently, with the expense of limitations in use of mashups. More complex presentation tool is OpenAjax hub based Scrapplet[6]. It has all the features of simple presentation tools, but because it is built on top of OpenAjax hub, widgets can communicate with each-other and widgets can be extracted from regular Web sites, which means it also has features of transformation and adapter tools.

One of the most traditional examples of a transformation and aggregation mashup tools is Yahoo! Pipes[7]. It is a web based service for combining and manipulating data from different feeds into mashups. Another more complex and enterprise oriented transformation and aggregation mashup tool is IBM DAMIA[8]. Similarily to Yahoo! Pipes, it is used to combine data from different feeds and it is bundled together with IBM Mashup Hub and QEDWiki to facilitate creation of user interfaces.

One of the most ambitious projects, which is currently under development, is OMELETTE [21]. This project aims to provide end-users with an environment that allows them to compose their own workspace according to their own specific needs. It is also meant to include a suggestions engine and automated composition engine, that suggest widgets based on defined patterns and user previous usage. The project aims to hide all the complexity into widgets with the aim of simplifying composition of mashups from widgets. OMELETTE combines existing technologies and standards to leverage mashup creation. More specifically, W3C Widgets family of specifications is used for describing widgets and Apache Rave is used as an application server. In order to simplify widget creation ServFace is extended for creating widgets from annotated SOAP services, and MyCoctail, for embedding RESTful services. This tool is supposed to join all four types of mashup tools into one single application framework.

2.4 Layout selection and construction

Layout modeling is a technique where a layout is modeled using some relatively easy to use visual tool that then generates the layout HTML code. Ceri et al. [22] introduce XML based Web modeling language WebML together with design tool suite ToriiSoft. The approach separates Web page modeling into 4 models: structural model, hypertext model, presentation model and personalization model. These models can be constructed by corresponding professionals and together give a complete view of a Web site. Tools, such

6 http://www.scrapplet.com/
7 http://pipes.yahoo.com/pipes/
8 http://link.ece.uci.edu/~yankaiw/damia/browser/html/home.htm

as ToriiSoft, can then generate HTML or even ASP pages with database backend based on these models.

Constraint based automated layout generation constructs a layout based on a set of constraints or rules. According to Lok and Feiner [23], there are two types of constraints: abstract and spatial. Abstract constraints describe a relationship between objects in a layout. For instance, an illustration references the text would be an abstract constraint. Spatial constraints describe a specific size or location of an object. For example, a constraint may set that a text area should appear below an illustration. Abstract constraints must be translated to spatial constraints before they can be used to generate a layout. Boring et al. [24] presented an architecture and implemented a prototype of constraint based layout manager. They proposed a system where constraints would be considered during page design-time for the basic layout construction and then on client computer the final dimensions and positions would be chosen using a client-side constraint solver.

Anyway, constraint based systems are a specific subclass of knowledge-based systems in general. Gonzales-Uriel and Roanes-Lozano [25] propose a knowledge-based system for layout selection for industrialized home building. They described a set of layout types for houses and a set of rooms, or components, that are placed into these layouts to form a house. Then they composed a knowledge base of criteria for house layout selection. These criteria covered climate-related, building-site-related and occupant-related issues. A set of concrete parameters are given to this system and based on the layouts and the knowledge base the approach provides the most appropriate layout for the house. For instance, in a cold climate a more compact house would be easier to keep warm. This approach is also relatively easy to extend by adding more rules to knowledge base. For example, it was proposed to include cost of materials and labor to minimize building costs. This thesis proposes a similar solution for Web mashup construction, where a set of layouts and rules are used to determine a layout for a specific set of widgets.

2.5 Guidelines

The following is a set of guidelines that have been identified as important for a layout of a mashup Web site. Rules are described textually and include references to sources that propose these rules. During the implementation these rules have been encoded either as RuleML, implemented in template files or programmed into the application.

Guideline 1 Layout has to be responsive [26]. This means that a Web site should

fill 100% width of the screen and should adjust to different screen resolutions. Guideline is proposed in [10] and [11]. Rule has been used in real world at CNN.com [27].

Guideline 2 Navigation menu has to be placed in the header or in the left hand side of the web site. In some situations both locations could be used simultaneously, for example a header menu for main sections and a left-hand side menu for subsections. The guideline is proposed in [10] and [28]. Rule has been used in real world at CNN.com [27], Skype.com [29] and Dropbox.com [30].

Guideline 3 Display a related illustration next to the main content, usually in the right-hand side. Such content is displayed at a higher position than the rest of the textual content. Guideline has been used in real world at Skype.com [29].

Guideline 4 Visualized data, for example a map or a graph, should be available as a table. Guideline has been used in real world at CNN.com [27].

Guideline 5 Visual feedback needs to be given in case page loading is performed. Without feedback users may get confused and think that the Web site is not working. Proposed in [10]. Used at CNN.com [27] and Dropbox.com [30].

Guideline 6 Important information should appear higher on a Web page. Users start reading from the top, this enables them to find important information in shortest time possible. Proposed in [10], [11] and [31].

Guideline 7 Use frames when some features of a Web page have to remain visible while scrolling others. Proposed in [11].

Guideline 8 Related information or functionality should be grouped together. This makes it easier to find necessary information. Proposed in [10] and [11].

Guideline 9 Web site should not be cluttered with information. Only display what is important for the user. Proposed in [11].

3 Background

In order to create an existing standards based application several existing standards and technologies were studied.

3.1 OpenAjax Metadata 1.0 Specification

OpenAjax Metadata 1.0 [32] specification was developed by IDE Working Group at OpenAjax Alliance to provide an industrial-strength metadata format for describing widgets. Although the format can be used to describe simple UI components, like buttons and text boxes, also more complicated mashup components or widgets, that contain complicated JavaScript logic and communicate with other widgets in the mashup, can be described with the format. Similarity it can be used to describe the APIs of JavaScript librarys, like jQuery[9] or Dojo[10]. OpenAjax Metadata is primarily targeted to IDE developers, who can use it to provide intelligent code assistants, and mashup assembly tools, where widget user interfaces and necessary resources can be described with the standard.

In this thesis OpenAjax Metadata is used to describe Web widgets. This leverages an elegant way to package the widgets and gives some extra information about the widget to the mashup creation framework. For example, OpenAjax Metadata allows defining the title of the mashup, suggested dimensions, required JavaScript files and other external files. Especially important for us is the ability to define topics, that the widget subscribes or publishes to, and data exchange formats. However, some extensions to the standard are needed for our usage, such as support for defining minimal and maximal widget dimensions, but most of the necessary information can be presented using the standard annotations.

3.1.1 Details of the standard

OpenAjax Metadata 1.0 widget specification file is a standard XML file that defines a widget and the resources it uses. A valid OpenAjax widget file name must end with "`oam.xml`".

9 http://jquery.com/
10 http://dojotoolkit.org/

Root element of each widget specification is `widget`. This element has attributes for defining a unique identifier of the widget, suggested dimensions, version, JavaScript class name for the widget and sandbox mode. The sandbox mode, which is enabled with the `sandbox` attribute, indicates that the environment has to completely isolate the widget from the rest of the widgets, apart from allowing communication through OpenAjax publish/subscribe APIs.

The `content` element provides widget's presentation in HTML format. It can be defined inline or refer to an external file. To load an external file, the `src` attribute is used, which defines the URL of the external file. The `content` element may contain any valid HTML, CSS and JavaScript code.

To load resources elements `javascript`, `require`, `library`, `preload` and `postload` are used. The `javascript` element is the simplest element for including JavaScript code that must be available at run-time. It can be inserted before content, after content or at the end of the mashup file by changing `location` attribute. The `require` element can, similarly to `javascript`, be used to include JavaScript code, but it can be also used to include CSS, images and other media. If `includeRef` attribute is set to `true` then this resource will be added into the HTML `head` element of the mashup page, otherwise it must be referenced within the widget content. For loading JavaScript libraries the `library` element is used. This element helps developer tools to identify widgets that share common libraries in which case these libraries are loaded only once. For the preceding, library name and version must be provided with `name` and `version` attributes. The `preload` and `postload` elements are library child elements that define JavaScript that is executed respectively before or after the library is loaded. All resources required by a widget, in order to operate properly, must be defined with these elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<widget spec="1.0" xmlns="http://openajax.org/metadata">
  <content><![CDATA[
    <div style="@@background@@;width:100%;height:100%;"></div>
  ]]></content>
  <property name="background" datatype="string"
defaultValue="#FFFFFF"/>
</widget>
```

Example 3.1: Sample usage of a property

The property element can be used to define properties for the widget. The properties can be modified either at design-time or at run-time to change behavior of the widget. Values of properties can be used inside widgets by using @@propertyname@@ variable substitution syntax or by defining getterPattern and setterPattern methods which can then be used in JavaScript code. Example 3.1 is a simple widget which background can be altered by changing the "background" property.

The topic element defines the topics that the widget subscribes to or publishes to using corresponding OpenAjax Hub primitives. Actual subscription or publication is performed in widget JavaScript code by using publish or subscribe APIs provided by OpenAjax Hub to the widget. Widgets can have several topic elements, but the name attribute of each topic element has to be unique. The topic element can have type attribute, which defines the type of the data structure that a published or subscribed message will contain. If type attribute is set to object then the topic element can contain property elements, which describe the structure of the object published by the widget or expected as input.

The category element can be used for categorization of widgets, while a widget may belong to multiple categories. For nested grouping double-colon sequence is suggested.

For localization, message bundle files are defined. These XML files have a root element messagebundle and any number of msg child elements that provide localization strings. Message bundle files are loaded into widget with locale element based on user locale and lang attribute of the locale element. Localized messages are inserted into elements by using locid attribute or localization variable substitution with ##localizationkey## syntax.

OpenAjax metadata 1.0 specification also provides compatibility elements for defining which browser or JavaScript library versions are required for specific features. Elements available and deprecated provide information about which widget or JavaScript library version specific feature is available for a browser set with userAgent.

Additionally several descriptive elements are specified in the standard. These elements are all optional, but they could be used to give additional information to the user or a widget catalog. For example the description element can be used to give a short description of the widget, the license element can be used to specify license terms, the

`icon` element can be used to specify widget icon for catalogs or developer tools.

For grouping of elements, OpenAjax Metadata 1.0 specification supports plural elements of many singular elements. For example `categories` for `category` elements, `topics` for `topic` elements and `requires` for `require` elements. Plural elements do not add any additional functionality, but they are used to make the widget file more readable by grouping element of the same type.

OpenAjax Metadata 1.0 specification includes JavaScript widget APIs for widget handling and communication through OpenAjax hub. To use these APIs widget must define JavaScript class by using `jsClass` attribute on the root element and class constructor function in either `javascript` or `require` element, either inline or in a separate JavaScript file. Widget loader then creates a widget object using class constructor and attaches OpenAjax APIs to this object. APIs enable several events, like `widget.onLoad`, which is fired when widget has finished loading all required resources, and `widget.onChange`, which is fired when some widget property has changed. There are also functions which can be used to get information about the widget or modify its behavior. For example the `widget.OpenAjax.getId()` function returns unique widget identifier which is assigned by the hub, and the `widget.OpenAjax.-requestSizeChange()` function tries to resize the widget. However, the latter may fail or change dimensions to not exactly the requested dimensions if mashup application does not allow it. Additionally, the `widget.OpenAjax.set-PropertyValue()` function changes widget property followed by firing `widget.onChange` event. Finally, OpenAjax hub instance is attached to `widget.OpenAjax.hub`, which implements OpenAjax hub `HubClient` interface, such that it can be used to subscribe or publish messages to some topic.

To ensure compatibility with future versions of the specification, it is not allowed to write extensions to OpenAjax Metadata specification within any OpenAjax XML namespaces. Extensions must have their own XML namespaces.

3.1.2 Example

Example 3.2 defines a Google Maps based widget. It loads jQuery library, Google JSAPI and local GoogleMaps.js file with widget logic. Additionally, it subscribes to topic "`AutoMicrosite.GoogleMaps`" for listening input messages, and describes the message semantically, as described in chapter 5.2. User interface of this widget can be seen in Figure 3.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<widget name="GoogleMapsWidget"
  id="AutoMicrosite/GoogleMapsWidget"
  spec="1.0" width="640" height="480"
  x:min-width="100" x:min-height="100"
  jsClass="AutoMicrosite.Widget.GoogleMaps"
  xmlns:x="http://deepweb.ut.ee/automicrosite/OpenAjaxMetadataExtension"
  xmlns="http://openajax.org/metadata">
  <library name="jQuery"
src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.1/" version="1.7.1">
    <require type="javascript" src="jquery.min.js"/>
  </library>
  <require type="javascript"
    src="https://www.google.com/jsapi" />
  <require type="javascript" src="GoogleMaps.js"></require>
  <content><![CDATA[<div id="__WID__map"
style="width:100%;height:100%;"></div>]]></content>
  <topic name="AutoMicrosite.GoogleMaps" type="object" subscribe="true">
    <property name="countryCode" datatype="string"
urlparam="https://www.inforegister.ee/onto/business/2013/r1/registrationC
ountryCode" />
    <property name="county" datatype="string"
urlparam="https://www.inforegister.ee/onto/business/2013/r1/countyName"/>
    <property name="city" datatype="string"
urlparam="http://schema.org/addressLocality"/>
    <property name="street" datatype="string"
urlparam="http://schema.org/streetAddress" />
    <property name="postalCode" datatype="string"
urlparam="http://schema.org/postalCode" />
  </topic>
  <categories>
    <category x:iri="http://schema.org/Map" />
  </categories>
</widget>
```

Example 3.2: Example of widget specification in OpenAjax Metadata 1.0

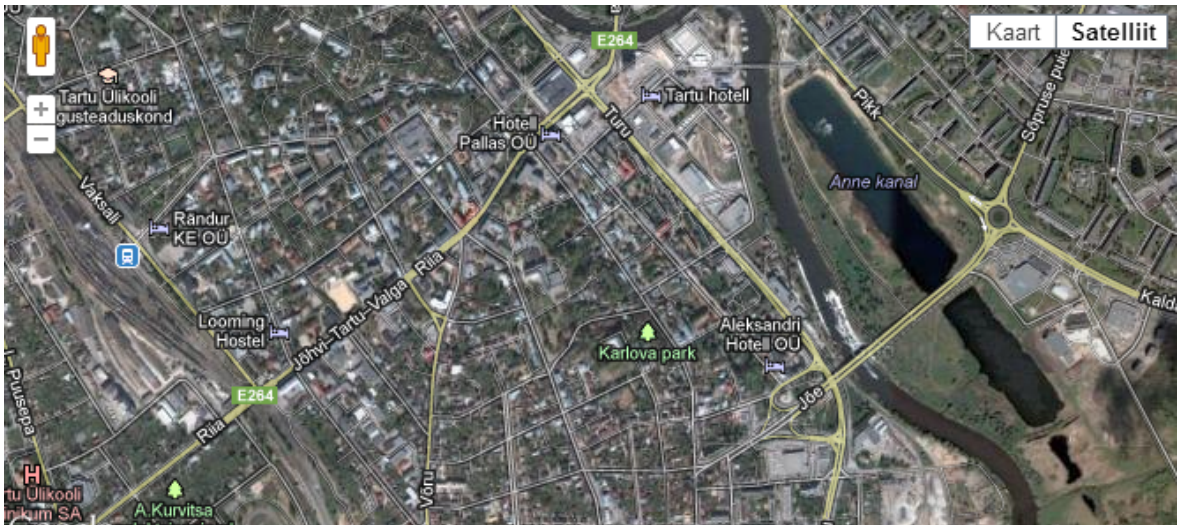


Figure 3.1: Example of a visual widget

3.1.3 Shortcomings

One of the greatest shortcomings of OpenAjax Metadata 1.0 specification is the lack of standardized or even suggested categories. This means that each implementer will have to come up with their own set of categories, which might lead to interoperability problems if certain application expects specific categories in order to work. In this master thesis also new set of categories had to be selected and extended.

In web development it is generally advised to use feature detection instead of browser or device detections. This is because browser versions change rapidly and also some features might be disabled by users in their browsers. In OpenAjax metadata specification only browser and version detection is possible, feature detection needs to be performed inside widget code.

3.2 RuleML 1.0

RuleML 1.0 [33] standard has been designed for the interchange of rules in an XML format that is uniform across various rule languages and platforms. It aims to cover most real world situations and is designed as an extensible family of languages instead of one single language. This makes easier to reuse a rule base designed for one application in another application.

RuleML consists of several subfamilies, languages and sublanguages. Figure 3.2 gives an overview of RuleML. RuleML languages can be broadly divided into deliberation and

reaction rule languages. Deliberation rule languages focus on derivation, they have Datalog RuleML as their core and add more features when necessary for the specific task. Reaction rule languages focus on actions that are performed in response to events and actionable situations. Reaction subfamily of RuleML addresses four types of reaction rules: production rules, event-condition-action rules, rule-based complex event processing, and knowledge representation reaction. Specific language constructs are structures as modules in the XML schema definitions. This facilitates maintainability and extensibility.

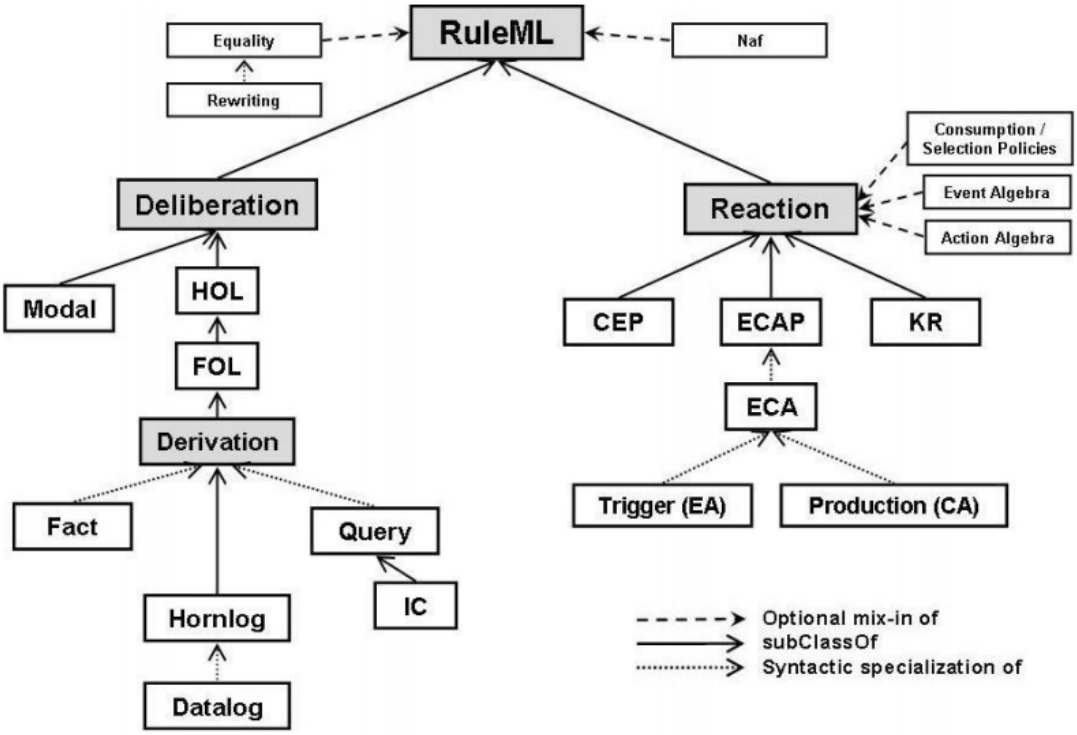


Figure 3.2: Taxonomy of RuleML rules from [34]

In this thesis Naf Datalog RuleML is used to describe rules and facts for the mashup construction application. These are evaluated by a rule engine and mashup is built from the results.

3.2.1 Details

The following is a description of Naf Datalog RuleML language syntax. Reaction RuleML languages were not used in this thesis so a description of syntax elements specific to these languages is not provided. Instead the readers are referred to the official RuleML 1.0 specification [33] for further details.

The root element of a RuleML document is `RuleML`. It defines RuleML namespace and schema location. It can have `Assert`, `Query` or `Retract` elements as its child elements. The element `Assert` implies that its content is asserted, i.e., knowledge is added. The element `Query` implies that element's content is queried from a ruleset. The element `Retract` means that element's content is retracted, i.e., knowledge is removed.

Facts are described using the `Atom` element. A fact is defined in terms of a relation constant `Rel`. It can contain any number of individual constant and data constant elements, respectively denoted with `Ind`, and `Data`, for defining relation arguments.

Implication rules are described using the `Implies` element. It has 2 child elements: `if`, that defines the premise or condition that must evaluate to true, and `then`, that defines the conclusion or consequent of the rule. These elements can either have an `Atom` or an `And` and several `Atom` elements as child elements of this `And`. The `And` element evaluates to true if all of its child elements evaluate to true or if it has no children. Similarly to facts, the `Atom` elements in implications can contain constants and data, but they can also contain logical variable elements denoted with `Var`.

In object-oriented RuleML, the `slot` elements can be used to create keyword-value pairs inside atoms. In this way the order of constants and variables in a relation is not important, constants and variables are matched based on keyword. Keyword is the first child of `slot` and its value is the second child. In order to provide even higher degree of flexibility, a `resl` element is supported, which allows matching of all slots that are not specifically defined.

3.2.2 Example

Example 3.4 is an example of a simple Datalog RuleML ruleset. It defines a fact that a widget "1" belongs to category "`http://schema.org/Map`" and an implication that if a widget belongs to category "`http://schema.org/Map`" then it shall be placed inside placeholder "2".

```
<RuleML xmlns="http://ruleml.org/spec">
  <Assert>
    <Atom>
      <Rel>Category</Rel>
      <slot><Ind>widget</Ind><Ind>1</Ind></slot>

<slot><Ind>category</Ind><Ind>http://schema.org/Map</Ind></slot>
    </Atom>
    <Implies>
      <if>
        <Atom>
          <Rel>Category</Rel>
          <slot><Ind>widget</Ind><Var>x</Var></slot>

<slot><Ind>category</Ind><Ind>http://schema.org/Map</Ind></slot>
        </Atom>
      </if>
      <then>
        <Atom>
          <Rel>Location</Rel>
          <slot><Ind>widget</Ind><Var>x</Var></slot>
          <slot><Ind>placeholder</Ind><Ind>2</Ind></slot>
        </Atom>
      </then>
    </Implies>
  <Assert>
</RuleML>
```

Example 3.3: A RuleML fact and an implication

3.2.3 Shortcomings

There is currently no complete open source RuleML evaluator engine available. RuleML covers very large range of languages which makes creating a complete engine very difficult and time consuming. For example OO jDREW[11] has implemented Naf Hornlog RuleML, which also contains Naf Datalog RuleML that is used in this thesis, but is completely missing reaction languages and also several deliberation languages, like First Order Logic. This means that rules written for one rule engine might not work with another engine without modifications, even though both engines evaluate RuleML.

11 http://www.jdrew.org/oojdrew/

3.3 Schema.org

Schema.org [35] is a joint operation by three major search engines to define a standardized set of schema that all major search engines would use and understand. It was jointly created by Google Inc., Yahoo Inc., and Microsoft Corporation to be used in their search engines. Schema.org vocabularies are also used by Yandex, the largest Russian search engine, and is open for everyone to use. It was created in the spirit of Sitemaps.org[12], a similar cooperation between search engine companies to create XML sitemap protocol that major search engines would recognize.

Schema.org provides an ontology for classifying content on web sites. This helps applications, like search engines, that are familiar with the schema, to understand what the information presented on the Web site means. For instance, the word "2012" might refer to a movie, a year, or just a number. Adding a Schema.org class "`Movie`" as an annotation to the corresponding Web site element would allow search engines to recognize that the text refers to the movie.

In the context of this thesis, Schema.org vocabulary is used to annotate widgets and based on these annotations layout rules are applied to widget descriptions. Because of the nature of widgets and web sites, only "CreativeWork" class of the Schema.org ontology and its subclasses are used for layout generation purposes. At the moment these annotations have to be added manually to the widgets, but in the future, once this standard gets recognized in wider scale, these tags could be automatically gathered from the source code of the widget.

3.3.1 Details

Schema.org provides a selection of commonly used content classes in a hierarchical fashion. All classes are children of class "`Thing`", the most generic class of an item. It has properties "`additionalType`", "`description`", "`image`", "`name`" and "`url`". It is extended with classes "`CreativeWork`", "`Event`", "`Intangible`", "`Medical-Entity`", "`Organization`", "`Person`", "`Place`" and "`Product`". All these classes add new more specific properties and are extended further by more child classes. Common data types are described in a separate "`DataType`" hierarchic, it includes "`Boolean`", "`Date`", "`DateTime`", "`Number`", "`Text`" and "`Time`".

12 http://www.sitemaps.org/

In situations where Schema.org does not provide necessary vocabulary it is allowed to use extensions of its elements in annotations. More specifically, users can extend existing class names in their annotations with a slash character, followed with an identifier of the introduced subclass. This allows existing applications to at least partially understand the class, even though the applications are not familiar with the extension. For example, after extending class "`Person`" with a subclass "`Engineer`" an annotation will be "`Person/Engineer`". If no suitable class exist in the schema to associate the extension with then it is also allowed to create new schemas. Extensions may be proposed for inclusion in Schema.org vocabulary, but this process is controlled by the companies that created this schema.

For a markup language for annotations, Schema.org creators have chosen Microdata. Microdata is HTML5 specification for embedding semantics into existing HTML documents. More detailed description of Microdata is given in Section 3.4.

3.3.2 Example

Example 3.4 is an example widget annotated with category "http://schema.org/Table". In this way we expose that the widget is a table widget and the developed layout selection can process it accordingly.

```
<?xml version="1.0" encoding="UTF-8"?>
<widget name="TableWidget" spec="1.0"
        xmlns="http://openajax.org/metadata"
xmlns:x="http://deepweb.ut.ee/automicrosite/OpenAjaxMetadataExtension
">
  <category x:iri="http://schema.org/Table" />
</widget>
```

Example 3.4: A categorized table widget

3.3.3 Shortcomings

Schema.org is not a truly open standard. Sponsors of Schema.org, Google, Yahoo and Microsoft, cooperate with W3C WebSchemas task force to get feedback from the community, but they keep control over the schema [36]. The vocabulary is closed to third-party contributions, only classes used by consortium members will be incorporated. This limitation may create interoperability problems since applications cannot be expected to be familiar with all extensions and some extensions might not be compatible with each other. Finally, Schema.org vocabulary is in its early steps and not yet very commonly used.

3.4 HTML Microdata

HTML Microdata [37] is an HTML5 specification for embedding semantics into HTML documents. It allows defining HTML elements as items and their descendants as properties of that item. Items can be given URIs to globally define their meaning.

In this project Microdata specification is only used for describing template placeholders for widgets and Microdata DOM API is used for finding the placeholders and replacing them with widget implementations. In future, when Microdata and Schema.org become more widespread, Microdata DOM API could be used to get widget annotations from the source code of widgets as well.

3.4.1 Details

Markup

An element is given an item scope with the `itemscope` attribute. It is a boolean attribute, adding it to an element without specific value evaluates to true which creates an item scope. To define the class of the item, `itemtype` attribute is used, which takes a URL as a value and defines in this way the class of the element globally so that all applications could understand the item in the same way. For example, the value of `itemtype` attribute could be a Schema.org class. If the item is given a global identifier, then it can be defined using `itemid` attribute. For example, for a book this value could be the ISBN of the book, and for a blog post it could be the URL of the blog post.

Microdata items can have properties, which are defined using `itemprop` attribute. If an item has several properties with the same name, then these are interpreted as a list of values. Generally the value of the item property is the text content of the element. However, if the item has an `itemscope` attribute, then the value is the item defined by that element. Furthermore, if the element is a `meta` element then its `content` attribute value is taken as the value of the property. For `audio`, `embed` and other media elements, the `src` attribute is the value of the property. For `a`, `area` and `link` elements, the `href` attribute value is the value of the propery. For `object` element the `data` attribute is the value of the property. For `date` element the `value` attribute is the value of the property. Finally, for the `time` element the `datetime` attribute is the value of the property.

DOM API

For easy access and manipulation of the data encoded in Microdata, a Microdata DOM API is defined in Microdata specification. All the attributes defined for HTML elements can be accessed using this API and a few extra calls have been defined:

- `document.getItems([types])` call returns a list of HTML elements that include all the items that include all the classes given as the attribute and are not part of any other item. This call is only available on the "document" element;
- `element.properties` returns all the `property` elements of an item. If element is not an item, i.e. it has no item scope defined on it, an empty list is returned;
- `element.itemValue` returns the value of the property and if an element is not a property then `InvalidAccessError` exception is thrown. It can also be used to set the value of the property.

3.4.2 Example

In Example 3.5 a template placeholder is annotated using Microdata. It defines that the `div` element is an item of class "`http://deepweb.ut.ee/TemplatePlaceholder`" and it has several "`category`" properties and "`min-width`", "`min-height`", "`max-width`" and "`max-height`" properties.

```
<div class="placeholder"
    itemscope
    itemtype="http://deepweb.ut.ee/TemplatePlaceholder"
    itemid="contentWidget">
  <span>Loading content...</span>
  <meta itemprop="category" content="http://schema.org/Map" />
  <meta itemprop="category"
content="http://schema.org/MediaObject" />
  <meta itemprop="category" content="http://schema.org/Table" />
  <meta itemprop="min-width" content="1" />
  <meta itemprop="min-height" content="1" />
  <meta itemprop="max-width" content="9999" />
  <meta itemprop="max-height" content="9999" />
 </div>
```

Example 3.5: Example of a template placeholder

3.4.3 Shortcomings

HTML Microdata standard is still a W3C working draft at the time of writing this thesis, which means it could still change. It also means that Microdata DOM API is not yet

implemented in any browsers, but it can be simulated quite easily with existing DOM tools.

3.5 Media queries

Media queries [38] is a W3C recommendation, which leverages media-dependent CSS rules. It is based on CSS 2 Media types [39], which allow defining different CSS rules for regular computer screen, printers, hand-held devices and couple of other screen types.

In the current project media queries combined with JavaScript code are used to adjust the layouts to as many screen resolutions as possible. Since most widgets cannot be re-sized indefinitely then the aim is to use column drop and off canvas patterns, as explained in [40]. Column drop pattern displays more columns with content next to each other on wider screens and on smaller screens while less important columns are moved below other columns. Off canvas pattern divides layout into sections such that on a larger screens more sections are shown at time and on a smaller screens some sections are hidden off the screen and can be shown by performing some action, for instance by clicking a button.

3.5.1 Details

Media queries can be used either inside the `media` attribute of HTML `link` element or by enclosing CSS rules inside within curly brackets and placing the media query before the brackets. In the first case the whole style sheet file is only loaded in case the device satisfies the query, otherwise only the enclosed CSS rules are applied in case the device satisfied the media query. Several queries can be separated using a comma, which expresses the logical “or”, or “and” keyword, which expresses the logical “and”.

In HTML 4 and CSS 2 media types, such as “`screen`”, “`tv`” and “`print`”, were defined and these are still present in HTML 5. These are used to define media specific CSS rules. For example “`screen`” means that the rules only apply when Web page is displayed on a computer screen, but “`print`” means that the rules are only applied when printing the Web page.

To define the dimensions of the display area, the `min-width`, `min-height`, `max-width` and `max-height` keywords are used. The units of the values are the same as in other parts of CSS, meaning that centimeters, pixels and em-s, which is the font size, are allowed. Additionally, the `device-width` and the `device-height` keywords are used to limit the actual screen size of the device. This may be different because for instance

Windows operating system based computers allow users to re-size the browser window size, which would change the display area, but the device dimensions would remain static. Similarily, Android based devices have address bar at the top of the screen and software buttons in the bottom of the screen, which reduce the usable dimensions.

To apply rules only in case the device is in portrait or landscape mode, the `orientation` keyword is used. The `aspect-ratio` and the `device-aspect-ratio` keywords are used to limit the rules to specific media with certain aspect ratio. For example, most new TV and computer screens are in 16/9 aspect-ratio, older screens were often in 4/3 aspect-ratio while the same 4/3 layout is used on both screens with a large blank space on the sides of 16/9 aspect-ratio screens.

Applying rules only to media with specific color output capabilities, `color`, `color-index` and `monochrome` keywords are used. The `color` keyword limits the number of bits per color component the device must be able to present. The `color-index` keyword limits the number of entries in the color lookup table of the device. The `monochrome` keyword is used to limit style sheet rules to black and white devices.

To further identify the device, the `resolution` keyword can be used to limit rules based on screen resolution. For example, many new smart-phones come with very high resolution screens, computer screens generally have less dots per unit and large TVs have even smaller resolutions.

3.5.2 Example

The HTML code in Example 3.6 loads special "`SmallStyle.css`" style sheet file in case the media type is `screen` and the screen is up to 600 pixels in width.

```
<link rel="stylesheet" media="screen and (max-width: 600px)"
href="SmallStyle.css" />
```

Example 3.6: Loading special CSS for small screens

```
@media all and (device-aspect-ratio: 4/3) {
    .wideScreen {
            display: hide;
    }
}
```

Example 3.7: A CSS rule that only applies to devices with 4/3 screens

The CSS code in Example 3.7 applies CSS rule that hides all elements with CSS class "wideScreen" from all devices that have screen aspect ratio of 4/3, for example old TVs and iPad.

3.5.3 Shortcomings

Media queries standard has just recently become a recommendation, so it is not yet implemented in many browsers. For instance, it is only implemented in Internet Explorer 9[13] and currently is not supported at all in Internet Explorer mobile[14]. Mozilla Firefox and Google Chrome have supported this specification since version 3.5 and 4.0, respectively[15]. However, it is supported in current major smart-phone operating systems, Android and iOS, and much of its functionality can be simulated using JavaScript.

3.6 Transformer widget

Transformer Widget [41] is a widget that when connected to OpenAjax Hub 2.0 hub routes and integrates messages between widgets with respect to the semantics of the messages. It is written using Google Web Toolkit and then compiled to JavaScript. In order to support message caching, to prevent situations where widget, that is not yet initiated, misses messages, it has is suggested to be used together with TIBCO PageBus[16] implementation of OpenAjax Hub 2.0.

Such middleware widget is needed because messages exchanged by different widgets created by different authors are often syntactically not compatible although they handle semantically similar data. Even more complicated are situations where one widget expects input message that is composed of output messages of several other widgets. Unless a mashup developer is able to modify all the widgets to be compatible, some sort of middle-

13 http://caniuse.com/css-mediaqueries
14 http://www.quirksmode.org/mobile/tableViewport.html#mediaqueries
15 http://caniuse.com/css-mediaqueries
16 http://developer.tibco.com/pagebus/default.jsp

ware is needed which does the translation and composition.

In order to perform the integration, mappings of all the messages exchanged must be defined. These mappings must be published to the Transformer Widget, there are 3 ways to do this. First option, is to publish the URL of the mappings XML file to "`ee.stacc.transformer.mapping.add.url`" topic. Second option, mappings may be placed in "`mappings.xml`" file that is in the file system folder with the Transformer Widget. These approaches are not always possible, because user needs access to the server where the Transformer Widget resides, so the third and preferred way is to publish the mappings data to "`ee.stacc.transformer.mapping.add.raw`" topic.

Mappings for a single message are defined using the `frame` element. All messages exchanged under one topic must follow the same structure, otherwise it would not be possible to map the message with annotations. Under the `frame` element `topic`, `format`, `schema`, `schema_data` and `mappings` elements can be used to define mappings:

- The `topic` element defines the name of the topic that the mapping describes. It has an optional parameter `outgoing_only` which, when set to "true", means that there are no widgets subscribed to that topic. This lets the Transformer Widget know that it does not have to compose such messages;
- The `format` element specifies the format of the data exchanged. Currently, the Transformer Widget only supports JSON and string data formats;
- The `schema` element defines the location of the JSON schema file which is used to generate a message for the topic. Since this was seen as inconvenient, Kirsimäe [42] added support for `schema_data` element, which allows adding the JSON schema data inline;
- The `mappings` element is the container for all the mappings. It may contain `mapping` elements and `repeating_element_group` elements;
- The `mapping` element contains mapping of a single data element. It has `global_ref` child element, which defines the reference to an OWL class. The `path` child element defines the location of the data element inside a message. The path is a slash (/) separated list of tokens which define the location of the element from root element. A `mapping` element may also contain a `default` element,

which defines a default value for the data element in case none is received;

- The `repeating_element_group` element maps a repeating element, such as an array. It must have a `path` attribute, which defines the location of the element in the message. Additionally, it may contain `repeating_element_group` elements and `mapping` elements.

Transformation widget maps messages based on the OWL classes specified for data elements. It is also capable of combining messages in order to create new messages. Once it has all the necessary data elements to form a message for a topic it is monitoring, it creates it using the JSON schema specified for the topic.

3.7 Proxy widget

Proxy widget [42] is an OpenAjax widget for surfacing SOAP services. It enables querying of SOAP services using OpenAjax Hub publish-subscribe APIs.

Proxy widget is necessary because consuming SOAP service from within browser with existing technologies is problematic. This is because making a cross-domain requests with JavaScript is limited due to browser same-origin policy [43], and generating and parsing of SOAP messages is difficult, because of lack of good XML processing tools.

A commonly used method to bypass browsers' same origin-policy is JSON-P [44]. It works by loading the third party content as a JavaScript file and passes the data to a callback function. This solves the problem of cross-domain requests, but it cannot be used to directly query SOAP services because the response message has to be inside a JSON-P callback function wrapper. It also introduces security concerns – since the message is evaluated as JavaScript then the publisher could run any code, potentially malicious, on client computer.

Another, more recent, approach to cross-domain domain requests is Cross-Origin Resource Sharing (CORS) [45]. It extends the existing domain-bound `XMLHttpRequest` with cross-domain request capabilities while keeping the communication secure. It would theoretically allow direct communication with the SOAP service, but since it requires that the server must send `Access-Control-Allow-Origin` HTTP header then it may still require server-side changes. This standard is also relatively new and only supported in modern browsers: Internet Explorer 8, Chrome 3 and FireFox 3.5 [46]. Lastly, CORS does

not solve the complexity of parsing a SOAP XML messages.

Third way, of achieving cross-domain messaging, is routing the messages through a less limited server-side proxy on the same domain. This means that no changes are necessary on the service side and it is also possible to transform the message to more easily understandable format by JavaScript, for example from SOAP XML to JSON. The only downside is that a server-side proxy needs to be set up, which may require programming knowledge and higher-level access to the server. The proxy widget handles all this complexity.

Proxy widget consists of client-side component, which includes the non-visual OpenAjax widget and utility functions, and server-side component, which generates mappings and proxies the request to service. The client-side widget and the server-side component must be hosted on the same domain. Client-side and server-side communicate using `XMLHttpRequest`, which is restricted with the same-origin policy. In order to generate the necessary mappings, the WSDL description of the service must be annotated using SAWSDL[17]. SAWSDL is an extension to WSDL that allows semantic descriptions within WSDL/XSD documents.

The client-side implementation consists of the widget code and utility functions for setting up the environment and widgets. When creating a new proxy widget instance, the URL of the WSDL for the service and the name of the operation to call are passed to the widget. The URL of the server-side component is taken from the URL of the widget, since they must reside on the same domain. Once the proxy widget has initiated, it constructs a URL to server-side mappings generator component and publishes it to transformer widget. Additionally, it generates URL to server-side SMD document generator component, that Dojo JSON-RPC component uses to create requests to proxy service.

The server-side component provides JSON-RPC proxy service to a SOAP service. In addition, it also generates mappings and JSON schema definition for the transformer widget, and a SMD document for Dojo JSON-RPC service wrapper.

17 http://www.w3.org/2002/ws/sawsdl/

4 Solution

The proposed solution to automated mashup layout selection is a rule-based matchmaker of widgets and layout templates. Rules are defined using RuleML and they are used to modify the matching behavior. Widget data is represented with OpenAjax Metadata 1.0 files. Layout templates are created using standard HTML, but special widget placeholders are placed inside the HTML code with Microdata markup. The mashup is built on top of OpenAjax Hub 2.0, which enables widget communication while securely separating widgets. This approach makes the application very flexible, since the result can be improved by adding new rules or layout templates. Also, the effort needed to make existing resources compatible with the application should be minimal because established standards and technologies are used.

In the following 4+1 architectural view model [47] is used to describe the architecture of the automated layout selection application. Process view describes the process of automated mashup creation. Development view describes the implementation of the components. Physical view gives an overview of the physical architecture. Logic view and scenarios have been omitted. They were deemed unnecessary because there is very limited direct user interaction with the application.

4.1 Process view

Mashup construction process is initiated when user submits a set of widgets with parameters to the Auto Microsite system.

4.1.1 Server-side component

Input is received and interpreted by the request handler component. The request handler first checks whether it can find a cached copy of the requested mashup. If if finds a cached copy then this is returned as result. If no cached copy is found, the mashup constructor component is initiated and the received widget references with parameters and configuration options are passed to it. Next, the mashup constructor component reads static rule files and sends them to the rule construction component which combines them and returns the combined ruleset. Next, the mashup constructor sends a list of widgets' metadata file URLs to rule construction component, which generates facts based on the

metadata. These facts are also combined with the previous ruleset. Next, template facts are generated by rule generator component and also combined with previous ruleset. Finally, the combined ruleset is sent to rule service.

Next, the mashup construction component starts querying the rule service. First, the mashup construction component queries the rule service client for a template that can accommodate all given widgets. Then, the mashup construction component separately queries the rule service for each widget for its widget metadata. Once done, mappings are generated for all semantically described widgets and added to the metadata.

Finally, the chosen template is prepared and widget metadata is appended to it. The resulting HTML code is returned to the request handler component which caches it and passes it back as response to the query.

4.1.2 Client-side component

Generated HTML code will be executed in client browser. First, OpenAjax Hub will be instantiated. Next, all the widgets will be instantiated and connected to the hub and placeholders. Data widgets will not be attached to a placeholder and instead will be added to the end of the document. When there are several widgets in a placeholder, the ones with higher priorities or lower work-flow order numbers are attached first. After loading of all the widgets is completed, the client-side component publishes widgets' mappings, when available, for the Transformer Widget. Next, menu widget, if available, is instantiated by populating it with data about widgets. Finally, the widget constructor will resize the visual widgets to appropriate dimensions. This resizing will also be done on each page resize.

This concludes the mashup construction process. Next, individual widgets will perform tasks according to their individual logic. An overview of the whole process of generating a mashup Web site is given in Figure 4.1 using BPMN diagram.

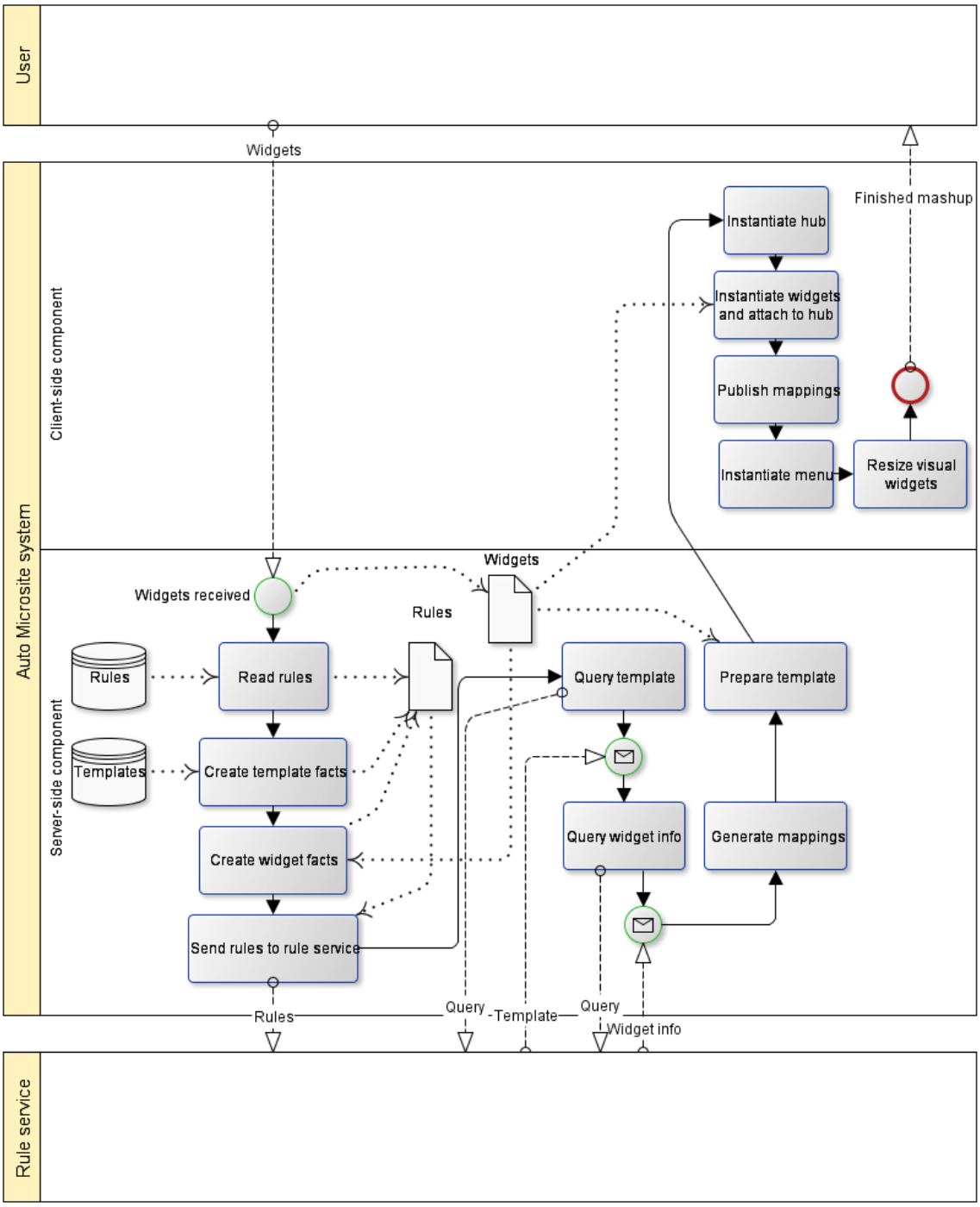


Figure 4.1: Process view BPMN diagram

4.2 Development view

On a high level, the Auto Microsite system can be divided into 3 main components: client-side component, server-side component and RuleML rule service. Figure 4.2 shows the component diagram of the application.

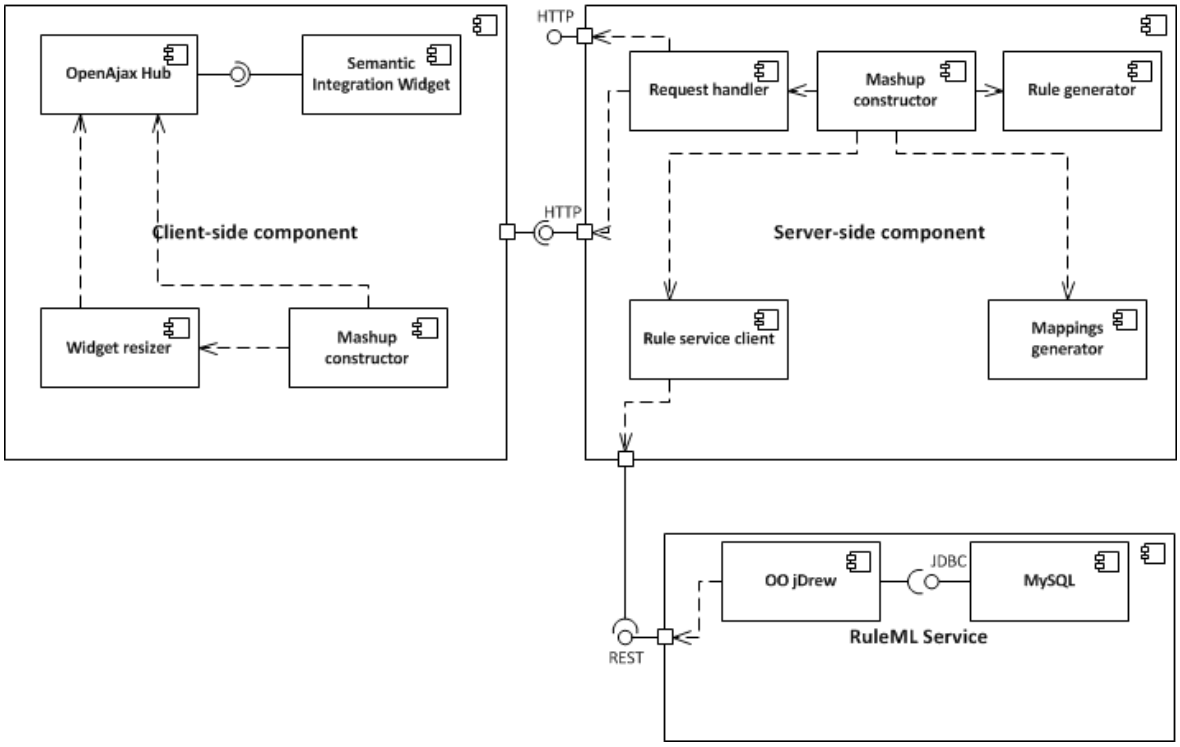


Figure 4.2: Component diagram of the layout construction application

4.2.1 Server-side component

Server-side component processes widgets that it receives as input and constructs HTML and JavaScript code for the client-side component.

The request handler component receives initial input from client in JSON format or as query string fields. The request handler component initiates the mashup constructor component, passing along widgets' data received as input. It also reads application configurations file. Because the evaluation of the rules can be resource intensive and generally just slow, then the request handler component also handles caching of the mashup.

The server-side mashup construction process is orchestrated by the mashup constructor component. It follows the process described in Section 4.1.1. It reads the static rule files, generates template and widget facts using the rule generator component and finally combines all the rules and facts into a single ruleset. Next, it initiates rule service client

component. Using the rule service client, a template is selected and then all widgets' info is queried one by one. Then, the mappings generation component is initiated, which generates mappings for all the widgets. Finally, the mashup constructor extends the chosen template file with necessary mashup JavaScript code and widget data such that the latter is appended to the end of the `head` element of the template HTML code.

Rule generator component generates rules and facts for the rule service. Rules and facts are generated from OpenAjax metadata files, described in Chapter 5.2, and layout template files annotated using Microdata, described in Chapter 5.3.

Mappings generator component generates mappings for Transformer Widget from OpenAjax Metadata files. In order to do this the topics have to be semantically described, as shown in Chapter 5.2.

Rule service client component mediates communication between the mashup constructor component and the rule service. It first sends the ruleset to rule service and later queries it for an appropriate layout template and widget information.

4.2.2 Client-side application component

The client-side application component handles the final construction of the mashup after necessary parameters have been set by the server-side application component. It is further divided into OpenAjax Hub, mashup constructor, Transformer Widget and Proxy Widget components.

The mashup constructor component orchestrates the life-cycle of the mashup from construction to event handling. It follows the process described in Chapter 4.1.2. It creates an instance of OpenAjax Hub that all the widgets are connected to.

OpenAjax Hub component is used as the backbone of the mashup. In order to support message caching, TIBCO PageBus 2.0 [48] implementation is used instead of the reference implementation of OpenAjax Hub 2.0. At the same time OpenAjax Hub basic functionality is used to handle secure messaging between widgets. OpenAjax Hub also includes the client-side JavaScript reference implementation of an OpenAjax Widget loader by OpenAjax Alliance. The widget loader parses OpenAjax Metadata 1.0 files and constructs OpenAjax widgets based on the metadata. Finally, the widget is connected to the hub and added to the placeholder.

Transformer Widget is a widget that is attached directly to OpenAjax Hub and it aims at assisting widget communication by semantically integrating syntactically different

messages with semantically similar content. This component is not required, when widget topics and messages are compatible with each other in which case they can exchange messages directly through OpenAjax Hub. Overview of semantic integration widget is given above, in Section 3.6.

Proxy Widget is a widget that enables consumption of SOAP services. It is an OpenAjax widget that also contains server side component for cross-domain requests. Overview of proxy widget is given above, in Section 3.7.

4.2.3 RuleML service

The RuleML service is a RESTful service for rule evaluation written in Java. It is based on OO jDREW RuleML engine and uses MySQL database for data storage. OO jDREW [49] (the Object Oriented Java Deductive Reasoning Engine for the Web) is object oriented extension to jDREW, a deductive reasoning engine for clausal first order logic. It introduces object oriented RuleML terms, `slots` and `rest`, to jDREW, allowing more flexibility, since all the constants do not have to be in the same order or even present in order to match facts. OO jDREW supports Naf Hornlog RuleML sublanguage of RuleML specification version 1.0. OO jDREW has support for some built in relations, such as “greater than” and “less than”, additional built in relations may be implemented in Java.

Rulesets are stored in a MySQL database to persist them over requests. This allows querying the ruleset without sending the whole ruleset with every request. New rules can also be appended to existing rulesets.

4.3 Physical view

The physical architecture of the application is designed to be flexible - all the high-level components may run on independent server nodes or on the same server node. Requirements for the environment are not very strict, any HTTP server or PHP version, newer than 5.3, may be used. Figure 4.3 is the deployment diagram of the application.

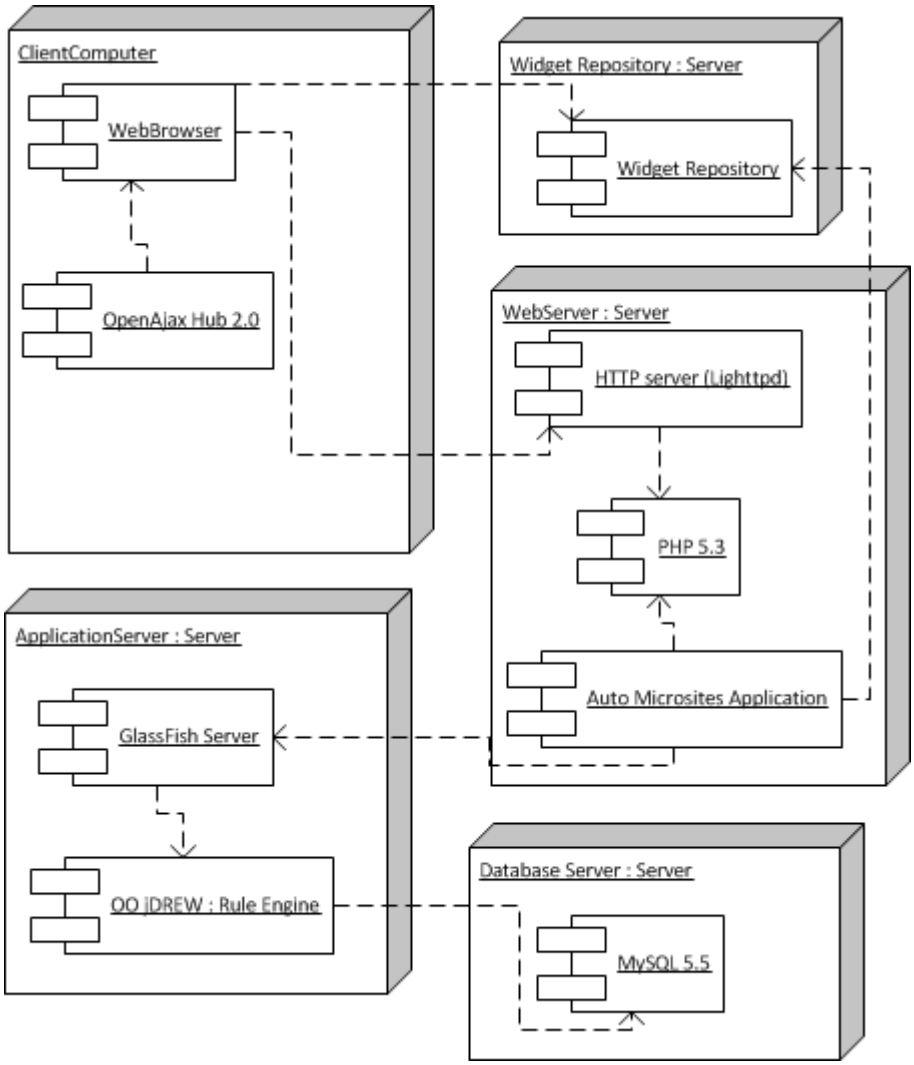


Figure 4.3:Physical view of the application

The server-side application component needs to be deployed on a node with HTTP server software, such as Apache HTTP server[18] or Lighttpd[19], and PHP 5.3[20] or newer. The client-side application component runs in a modern browser, such as Google Chrome, Mozilla Firefox or Microsoft Internet Explorer 9. Browser must have enabled JavaScript

18 http://httpd.apache.org/
19 http://www.lighttpd.net/
20 http://php.net/

and depending on specific widget, Flash or Silverlight plug-in might be necessary.

RuleML service needs a node with Java application server software. Application has been tested with GlassFish. In between requests, rules are stored in a database, for which MySQL database server is used.

5 Implementation

5.1 Categories ontology

In order to provide adequate matching behavior, a new ontology was defined at domain `http://deepweb.ut.ee/`[21] which also imported Schema.org onotology. This new ontology defines 6 new classes in addition to Schema.org ontology: "`BarChart`", "`Chart`", "`Dataset`", "`Form`", "`LineChart`" and "`PieChart`".

The "`Datasets`" class is an implementation of proposed "`Datasets`" [50] schema for Schema.org. Once the proposal has been added to Schema.org it could be used instead of the extension element. It is used to categorize content, i.e., non-visual, widgets.

The "`Chart`" class is for categorization widgets as charts widgets, it extends the "`http://schema.org/WebPageElement`" class. "`BarChart`", "`LineChart`" and "`PieChart`" are all different more specific classes of charts. The "`Form`" class is for categorization of a form widget, it extends the "`http://schema.org/WebPage-Element`" class.

5.2 Widgets

Widgets are defined using OpenAjax metadata 1.0 specification, described in Chapter 3.1 above. This allows storing of all the necessary information in one standard based metadata file, making the widgets portable. OpenAjax specification already has most of the necessary vocabulary, but some new extending attributes were added using a new namespace "`http://deepweb.ut.ee/automicrosite/OpenAjaxMetadata-Extension`".

For the widget element, the extension defines new `min-width`, `min-height`, `max-width` and `max-height` attributes. These define the minimum and maximum allowed widget dimensions in pixels. This way it is possible to avoid resizing to an extent that makes the widget unusable. The application will chose appropriate dimensions when none are defined.

21 http://deepweb.ut.ee/automicrosite/schema.org.owl

For the `category` element a new `iri` attribute is defined. This defines the internationalized resource identifier which has to belong to Schema.org class hierarchy or the Schema.org extension, described in Section 5.1.

For the `content` element a new `iri` attribute is defined. This is a URL to traditional OpenAjax widget HTML file that is implemented without metadata. The URL has to be absolute. This type of widget is always loaded inside an HTML `iframe` element and ignores the value of `sandbox` attribute for the widget.

```
<topic name="AutoMicrosite.BusinessRegister.QueryResponse"
type="object" publish="true">
  <example><![CDATA[
    {"name": "EVETERM OÜ",
      "code": 11375683}
  ]]></example>
  <property name="name" datatype="string"
urlparam="http://schema.org/Organization#legalName" />
  <property name="code" datatype="number"
urlparam="https://www.inforegister.ee/onto/business/2013/r1/registrat
ionCode" />
  <property name="registrationCountryCode" datatype="string"
urlparam="https://www.inforegister.ee/onto/business/2013/r1/registrat
ionCountryCode" />
</topic>
```

Example 5.1: A semantically annotated topic

In order to support semantic integration, all topics, that the widget communicates through, must be defined in the metadata file using `topic` element and structures of the messages must be defined using `property` elements. All `property` elements must have `name`, `datatype` and `urlparam` attribute values. The `name` attribute value is the name of the property in a JSON message that the widget consumes or publishes. The `urlparam` attribute value is used for an annotation and used when generating mappings for the Transformation Widget. The `datatype` attribute is the type of the property value, "`array`", "`boolean`", "`null`", "`number`", "`object`" and "`string`" are supported. Example 3.1 is an example of a topic that has been semantically described.

Widgets have to be "smart", meaning that they should contain most of the necessary logic and should be able to operate without dependencies to other widgets. The application will initialize them, set dimensions, provide them with initial data and format exchanged messages into acceptable format for all widgets, using the Transformation Widget, but widgets should contain necessary logic to store, process and display the data.

5.3 Layout templates

Layout templates are regular HTML files that describe the layout of a mashup. These files may also contain CSS and JavaScript or even external files, like CSS, JavaScript or image files, but these must be defined using absolute URLs. Widget placeholders are marked up using Microdata specification.

A new item class ("`http://deepweb.ut.ee/TemplatePlaceholder`") was defined in this thesis for layout template placeholders. The properties defined for the template placeholder type are `category`, `min-width`, `min-height`, `max-width`, `max-height` and `optional`. The `category` property defines the Schema.org categorization of widgets that may be used in the placeholder. If multiple `category` properties are defined then widget must match at least one of them. The `min-width`, `min-height`, `max-width` and `max-height` properties define the minimum and maximum dimensions of widgets that may be used in the placeholder. If no minimum or maximum dimensions are defined then a widget of any dimensions may be used in the placeholder. The boolean property `optional` defines whether the placeholder has to be filled or may be left empty in the generated mashup. The default value is "`false`", which means that the placeholder must be filled with a widget.

Similarly to widget, templates have to be "smart". The client-side application will manage widgets, but a template will have to be implemented with respect to usability guidelines described in Chapter 2.5.

5.4 Rules

Rules are defined using RuleML 1.0 specification OO Naf Datalog sublanguage. Rules are either statically stored in `ruleml` files or generated dynamically based on widget and template files. Relations in rules are defined using URIs such that they are globally unique. For example, relation named "`http://openajax.org/metadata#category`" defines that widget belongs to some category.

Statical rules are applied to all mashups in the same way. For the application, statical rules are distributed to three types: generalization rules, priority rules and other rules. Generalization rules allow Schema.org element children to inherit rules from parent elements. For example, rules associated with Schema.org class "`MediaObject`" also apply to its subclasses "`AudioObject`", "`ImageObject`", "`MusicVideoObject`"

and "VideoObject". Priority rules, however, allow ranking of widgets by their importance such that more important ones are positioned closer to the header in a Web page or on the first pages of a more complex multiple-page Web application. For example, if priority rule states that Schema.org "MediaObject" widget has priority "10", while "Table" widget has only "1", then "MediaObject" must be positioned higher in the layout. Other rules manipulate the widget-template matchmaking process. For instance, there is a rule that says that when a widget categorized as Schema.org category "Table" is placed inside a placeholder together with some visualization wiget and there exists a menu widget, then the table widget must be placed on a separate page.

Dynamically generated rules and facts are generated based on widget OpenAjax Metadata and Microdata layout template files. The widget facts generated from OpenAjax metadata define their dimensions and categories. The layout template facts, generated from template files, define allowed dimensions and categories of placeholder, plus whether a placeholder is optional and may contain more than one widget. Additionally, implications are generated for checking whether there are widgets for all required template placeholders and that all widgets have compatible placeholders.

5.5 Usage of rules

The process of matching templates with widgets is guided by rules. Since all the rules are available in RuleML format in the appendix, this section explains only the most important rules in the matching process using first order logic.

$$template(t) \wedge placeholder(p, t) \wedge widget(w) \wedge \neg isDataWidget(w) \wedge category(w, c) \wedge templateCategory(t, c) \wedge badDimensions(w, t, p) \rightarrow widgetPlace(w)$$

Rule 1: When there is a template t *with placeholder* p *and a widget* w *that is not a data widget and widget* w *and placeholder* t *share a category* c *and there are no dimensions conflicts then widget* w *matches placeholder* t*.*

The logic behind the rules aims at providing the best match between a set of given widgets and available layout templates. Matchmaking is done by matching the categories of templates' placeholders to widgets' categories. Additionally, rule engine considers widget dimensions and layout template placeholder dimensions, in order to avoid stretching the user interface of a widget or a layout. This is expressed with Rule 1.

Rule engine will evaluate rules to try and find a placeholder for each widget and then it will check that there is a widget for each required placeholder. It is also possible to match several widgets into a single placeholder. This is expressed with Rule 2.

$$templateFilled(t) \wedge \neg widgetMissPlace(w) \rightarrow templateMatch(t)$$

Rule 2: When template `t` has widgets for all non-optional placeholders and no widget `w` is without a placeholder then template is matched

Generally we assume that a template with more placeholders is more specific. This is because a layout that contains a specific placeholder for each widget is likely to be more specific for a case than a layout template with one or two placeholders that are able to fit widgets of any class. Therefore, whenever several layout templates match the widgets set, the one with the most placeholders is chosen, i.e., the most specific template with respect to a given selection of widgets. This is expressed with Rule 3.

$$template(t) \wedge templateMatch(t) \wedge \neg templateNotHighestPriority(t) \rightarrow templateQuery(t)$$

Rule 3: When a template `t` matches widgets and there is no template with higher priority then template is returned as query response

Data widgets are not suppose to be placed in placeholder, so it is important to identify data widgets. The Rule 4 identifies widget as a data widget.

$$widget(w) \wedge category(W, \text{Datasets}) \rightarrow isDataWidget(w)$$

Rule 4: When a widget `w` belongs to category "Datasets" it is a data widget.

In order to use a menu widget it must be recognized first. Widget is regognized as a menu widget with the Rule 5.

$$widget(w) \wedge category(W, \text{AutoMenu}) \rightarrow isMenuWidget(w)$$

Rule 5: When a widget `w` belongs to category "AutoMenu" it is a menu widget.

5.6 Server-side component

Server-side automated microsite generation application is written in PHP 5.3 programming language. Most of the server-side application components are implementations based on interfaces or abstract classes, with the exception of mashup constructor component, and constructed using factory pattern. This enables loose coupling of components, meaning that the implementation of one component can be altered without

affecting the rest of the application, as long as the interfaces stays intact.

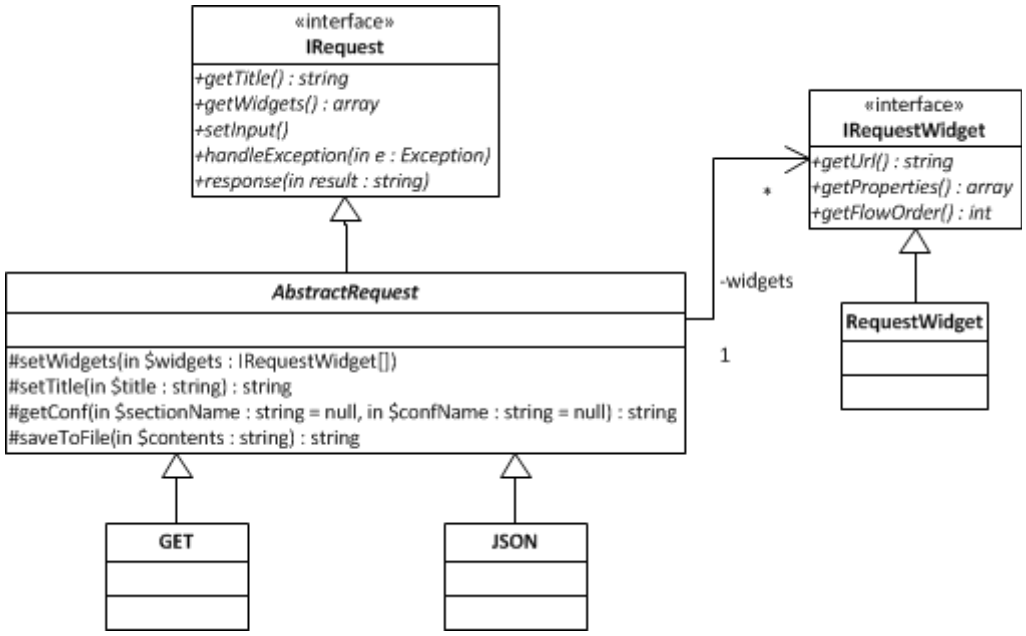


Figure 5.1: Request handling component

The request handling component implements the `IRequest` interface, that can be seen in Figure 5.1. The `setInput` method parses user input and prepares in such a way that it can be accessed using methods `getTitle` and `getWidgets`. Widget objects, accessible using the `getWidgets` method, must implement the `IRequestWidget` interface. The `handleException` method is called whenever an uncaught exception is received from the mashup constructor. The latter must respond appropriately, for example with an HTML error page. The `response` method is called with finished mashup HTML code as input when the mashup constructor has finished constructing the Web site. It must respond to client request, either by outputting the data in some format or by saving the data and providing the target URL. An abstract class `AbstractRequest` has been implemented based on the interface. The class provides common functionality for different request handling implementations. It loads the configuration file "`conf.ini`" and makes the contents accessible using `getConf` method. It also provides `getCache` and `saveCache` methods for cashing of the request.

Cache handling is implemented based on hash values constructed from URLs of all the widget OpenAjax metadata documents and widget properties in mashups together with particular mashup names. When an existing cache entry with a matching hash is found, that

is also no older than the rule files, it is used to return previously generated mashup instantly instead of running the server-side mashup generation process from scratch.

```
http://deepweb.ut.ee/automicrosite/?title=My+Mashup&widget
%5B1%5D=http%3A%2F%2Fdeepweb.ut.ee%2Fautomicrosite%2FWidgets%2FTable
%2FTable.oam.xml&property%5B1%5D%5BbackgroundColor%5D=
%23FFFFFF&property%5B1%5D%5BforegroundColor%5D=%23000000
```

Example 5.2: Example HTTP GET request

Currently, two request handling classes have been implemented: `GET` and `JSON`. The `GET` request handling class receives the data from HTTP GET request query string fields, and in case of the `JSON` request handling class the request will be encoded in JSON format and sent as a HTTP POST request body. In both cases the following attribute-value pairs are used. The `title` field sets the title of the mashup. The `widget` field is used to send the widget metadata URLs. In the case of HTTP GET request it is an array of widgets where the index is also used as work-flow order number, so the widget that is intended to be used first should have the lowest index. The `property` field is used to set property values. It is also an array where the index must correspond to widget index in `widget` field. Example 5.2 is an example of GET request. In the case of HTTP POST request, `widget` field is an array of objects with widget info. Widget info object contains `url`, `properties` and `flowOrder` fields. The `url` field contains the URL of the widget metadata file. The properties field contains an object of properties for the widget. The `flowOrder` field contains the work-flow step order number of the widget, this is optional. Example 5.3 is an example JSON request input.

```
http://deepweb.ut.ee/automicrosite/json.php
{"title":"My Mashup","widget":
[{"url":"http:\/\/deepweb.ut.ee\/automicrosite\/Widgets\/Table\/Table
.oam.xml","properties":
{"backgroundColor":"#FFFFFF","foregroundColor":"#000000"},"flowOrder"
:1}]}
```

Example 5.3: Script JSON input

The server-side mashup construction process is orchestrated by the mashup constructor component. It follows the process described in Section 4.1.1. The component receives widget data from the request handling component. The mashup constructor component

reads static RuleML files, defined in the configurations file under `[rules]` section with configurations `generalization`, `priority` and `other`, and sends them to the rule construction component which combines them into a ruleset. Next, it sends all the widget metadata URLs to the rule generator component, for fact generation, and combines them with the rest of the ruleset. Next, it reads all the layout template files from templates directory, defined with configuration `templates_dir`. Template files are sent to the rule generation component, for fact generation, and are combined with the ruleset. Next, rule service client object is created by the mashup constructor component, passing the created ruleset and queries, defined with `template_query` and `widget_info_query` configurations, to it. First, a template is selected through the rule service client component. Next, all widgets' info is queried one by one. Then, the mappings generation component is initiated, which generates mappings for all the widgets. Once the server-side processing is finished, mashup constructor returns created mashup to request handler component.

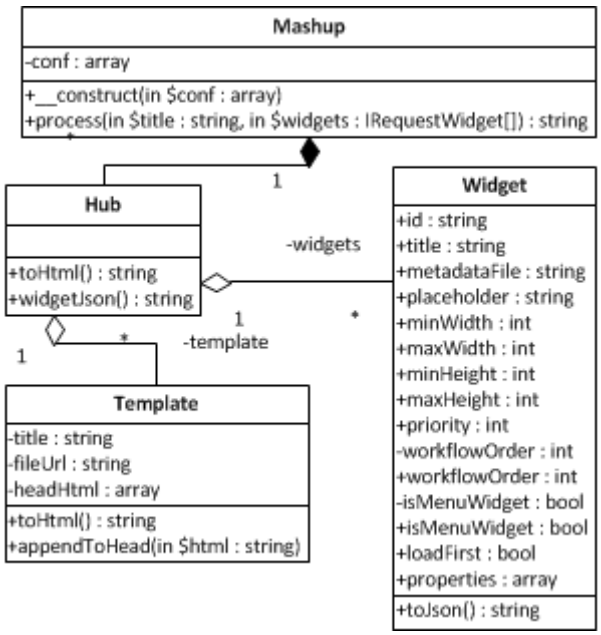


Figure 5.2: Mashup constructor component

Widget data is combined into objects created from the `Widget` class, shown in Figure 5.2, which is then serialized as JSON and returned to the client-side component. The `id` field is internally used unique widget identifier. The `title` field is loaded from the metadata file, when available, it is used in a menu widget, when necessary. The

`metadataFile` field holds a URL to widget's metadata file. The `placeholder` attribute contains identifier of the template placeholder that the widget belongs to. The `minWidth`, `maxWidth`, `minHeight` and `maxHeight` fields describe the maximal and minimal dimensions of the widget. The `priority` field describes the priority of the widget, higher priority means that it is placed higher in the mashup. The `workflowOrder` field describes the widget execution order, lower value means that the widget is placed closer to the header of the mashup. The `isDataWidget` field is "`true`" when the widget is data widget only, i.e., it has no user interface. The `isMenuWidget` field is "`true`" when the widget is menu widget usable by the application for widget pagination. The `loadFirst` field is "`true`" when the widget has to be loaded before other widgets, e.g., Transformer Widget has to be listening to mappings of other widgets so it has to be loaded first. The `properties` field contains properties that are given to the widget when it is loaded. It is an associative array, where the key is the name of the property and the value is the value of the property. The `separatePage` field is "`true`" when widget has to be placed on a separate page in a multiple-widget placeholder, e.g., a table widget when there is a visualization widget in the same placeholder.

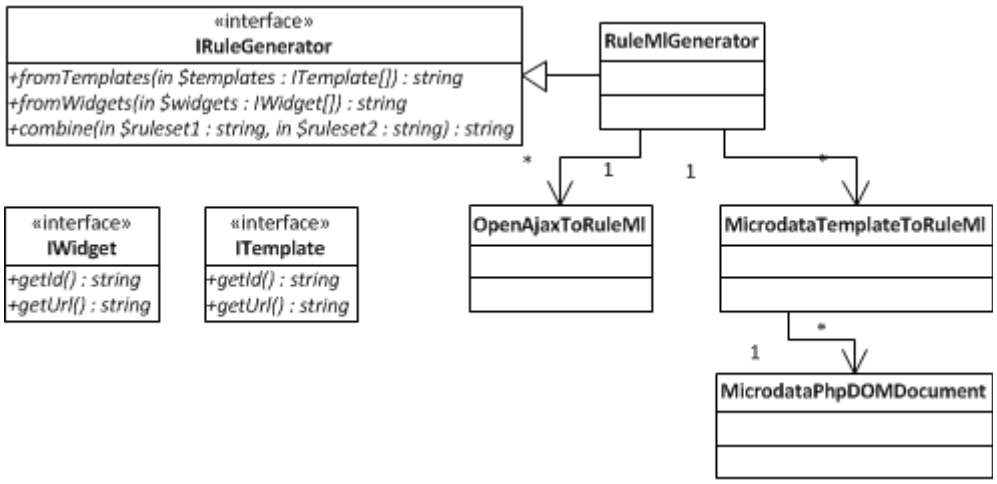


Figure 5.3: Rule generator component

Rule generator component implements the `IRuleGenerator` interface given in Figure 5.3. The `fromTemplates` and `fromWidgets` methods generate rules from template and widget files. The `fromTemplates` method takes an array of objects based on `ITemplate` interface as input and returns generated ruleset as a string. The `fromWidgets` methods takes an array of objects based on `IWidget` interface and

returns generated ruleset as a string. The `combine` method is used to combine rulesets. The concrete implementation for this thesis generates rules for widgets from OpenAjax Metadata 1.0 files, XSLT transformation is used to perform this task. XSLT transformation rules are stored in file `Rules/OpenAjaxToRuleML.xsl`. Rules for templates are generated from Microdata template files using DOM API. PHP 5.3 DOM extension does not yet implement Microdata DOM API, so an implementation of the API by Lin Clark[22] is used for parsing the template files.

Mappings generator component realizes `IMappingsGenerator` interface shown in Figure 5.4. The `getMappings` method takes a URL of a OpenAjax metadata widget file and returns the resulting mappings as a string. The algorithm implemented for mappings

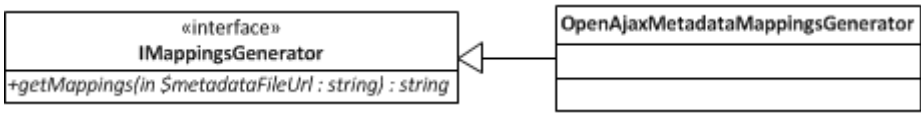


Figure 5.4: Mappings generator component

generator finds all the `topic` elements inside a metadata file and then by recursively going through all the `property` child elements constructs JSON schema and XML mappings necessary for the Transformer Widget. The `datatype` attribute is used as the type of the element in JSON schema, the `name` attribute is used as the property name in JSON schema and also for constructing the `global_ref` element value for the `mapping` element. In case an element with `datatype` value "`array`" is met, a `repeating_element_group` element is created in the mappings.

22 https://github.com/linclark/MicrodataPHP

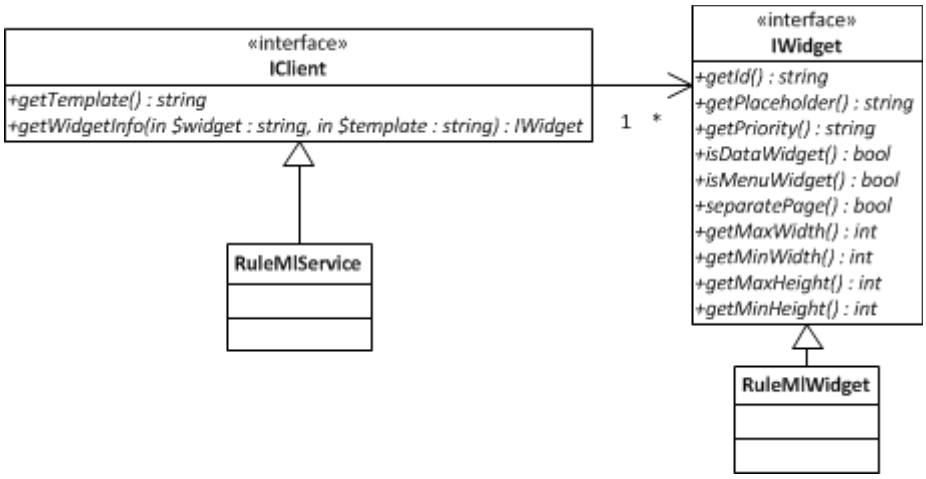


Figure 5.5: Rule service client component

Rule service client implements `IClient` interface, shown in Figure 5.5. In the concrete implementation, a client is implemented for the RESTful RuleML service described Chapter 4.2.3. PHP Client URL Library (cURL) is used for communication. URL, ruleset, template query and widget information query are passed as string parameters to the constructor of the client class. The method `getTemplate` queries rule service for a template that satisfies the ruleset, URL of the template is returned. The method `getWidgetInfo` takes widget identifier and template URL as an input and constructs the query based on these values. This method returns an implementation of the `IWidget` interface.

5.7 Client-side component

The client-side application component is written in JavaScript programming language using Dojo 1.8 library [51].

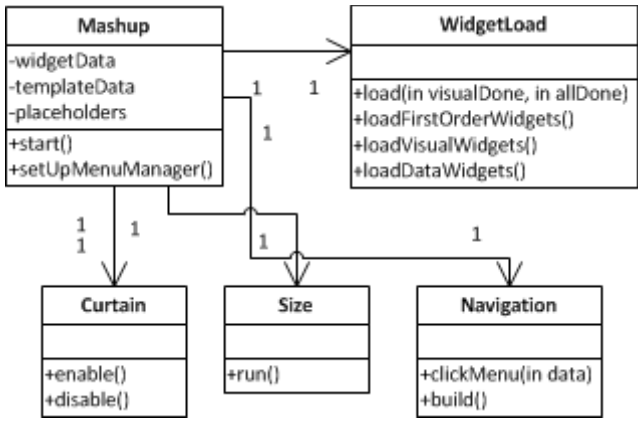


Figure 5.6: Client-side mashup constructor component

The mashup constructor component orchestrates the life-cycle of the mashup from construction to event handling, as described in Chapter 4.1.2. The process is programmed in `start` method. The `setUpMenuManager` method registers a listener to "`AutoMicrosite.MenuClick`" topic and forwards every message received in said topic to `Navigation` object `clickMenu` method. Figure 5.6 is the class diagram of client-side mashup constructor component.

The `WidgetLoad` class loads all the widget metadata files, starting with widgets that have been defined for first loading, and passes them to OpenAjax Hub component which constructs widgets. It then loads visual widgets and finally data widgets. When there are several widgets in a placeholder, the ones with higher `priority` value and lower `workflowOrder` are attached attached first. When a widget has mappings data available, this class will extend `onLoad` event handler of the widget to publish the data to "`ee.stacc.transformer.mapping.add.raw`" topic for Transformer Widget.

The `Curtain` class activates a black see-through overlay with loading message. It is used while the widgets are being loaded. It blocks user interactions with the mashup in order to prevent errors and gives visual feedback about the status of the mashup. It is required by Guideline 5.

The `Size` class handles the sizing of widgets. It is first executed when all visual widgets have finished loading, later it is invoked every time a browser window size changes or a navigation button is pressed. The algorithm for resizing the widgets goes through all the placeholders one by one. For each placeholder, it finds the dimensions of the placeholder and all the widgets inside it. Then it goes through all the placeholder

widgets in the order of their priority. Widgets are placed next to each other as long as they fit and then a new line is started. When no minimal or maximal dimensions are available, the application will use dimensions that fit best with the layout. When all the widgets cannot be fit in the width or height of the placeholder with their minimal dimensions, a scrollbar is used. Resizing is required by Guideline 1.

The `Navigation` class handles navigation between widgets. When a widget has the `separatePage` option value set to "`true`" it will be made invisible by default and will be shown when the mashup constructor receives a click event from a menu widget. It populates menu widget by setting its `buttons` property value. Example 5.4 represents an example of a button property value. It is an array that contains menu button objects. Each button object contains `label` property, which defines the visible label of the widget, and `href` property, which contains the identifier of the widget. The value of `href` property is published to "`AutoMicrosite.MenuClick`" topic when the button is clicked. `Navigation` class also subscribes to that topic and switch widget visibility whenever it receives a message.

```
[
  {label: "My Mashup", href: {widget: null, placeholder:
"contentWidget"}},
  {label: "Table", href: {widget: null, placeholder:
"contentWidget"}}
]
```

Example 5.4: Menu widget input

OpenAjax metadata loader reference implementation was thesis extended to add support for the `iri` attribute on the `content` element. This kind of widget is simply attached with IFrame container without rest of the OpenAjax metadata widget headers.

5.8 Deployment

The deployment of Auto Microsite system consists of two parts: the deployment of the Auto Microsite application itself and the deployment of RuleML service.

5.8.1 Auto Microsite application

The server-side Auto Microsite system component requires HTTP server with PHP 5.3 or later, as described in Chapter 4.3. In order to deploy the application, the application files

have to be uploaded to a HTTP server and "log" directory has to be configured to be writable by PHP user. Auto Microsite system is configured using conf.ini file. Configurations file is distributed into general, rules and rule service sections.

General configurations section ("[general]") holds configurations for cache, templates, rule generator component and mappings generator configuration. To enable caching the cache configuration has to be set to "1". Directory for cache entries is set with cache_dir configuration. Cache directory has to be accessible and writable by the PHP user, so read and write permissions may have to be granted to all users. The template_dir configuration sets the directory from within the application will read template files. The rule_generator configuration sets the rule generator implementation which is used for rule and fact generation and combination. Only "RuleML" has been implemented for this thesis. The mappings_generator configuration sets the mappings generator implementation which is used for Transformer Widget mappings generator. Only "OpenAjaxMetadata" has been implemented for this thesis.

Rules configuration section ("[rules]") holds configurations for rule files locations. The generalization configuration sets the location of generalization rules file. The priority configuration sets the location of priority rules file. The other configuration sets the location of other rules file. The template_query configuration sets the location of template query file. The widget_info_query configuration sets the location of widget information query file.

Rule service configuration section ("[rule_service]") holds configurations for the rule service. The type configuration sets the rule client implementation that is used for querying the rule service. Only "OOjDREW" has been implemented for this thesis. The url configuration sets the location of the rule service.

5.8.2 RuleML service

In order to deploy RuleML service the RuleMlApp2 project needs to be built into a "`RuleMlApp2.war`" file. This file can be deployed on a Java application server, such as Glassfish.

A MySQL database needs to be set up for RuleML service. The "`RuleMlApp2/db.sql`" file needs to be imported to the database in order to create necessary knowledgebase table. RuleML service connects to database using JDBC Resource named "`jdbc/MySQL`".

6 Proof of Concept

In order to validate the solution two proof of concept scenarios were constructed. First one visualizes "Hourly labour costs in Euros (European Union 1997-2008)" [52] data as a map and a table, while the second proof of concept visualizes debt information from Inforegister.ee database as a table. Additionally, Schema.org ontology extension defined for semantical notations of the service is described here.

6.1.1 Schema.org extension

For semantical annotation of the widgets in the following scenarios a new `https://www.inforegister.ee` ontology[23] was defined, which also imports Schema.org ontology. It defines new "`AccessKey`", "`DebtSum`", "`NumericRange`", "`Organization`" and "`PostalAddress`" classes with properties.

6.2 Proof of Concept 1

The aim of this proof of concept is the creation of a mashup for visualizing "Hourly labour costs in Euros (European Union 1997-2008)" data [52]. This mashup will load the data from a text file and will visualize it using a map and a table. The table is considered a secondary backup visualization, in case the map is difficult to understand, so it is hidden to a separate page. Mashup also shows a summary of data selected on the map.

6.2.1 Components

Widgets

Five widgets are required for this mashup. Widgets are described using OpenAjax metadata 1.0 specification:

- Data widget loads the data from the service and publishes it to other widgets for further consumption. The data is loaded from data.txt file using AJAX. The widget has category "`http://deepweb.ut.ee/Datasets`".
- Map widget is the primary data visualization widget in this case. It displays data about all countries and only about one year at time. Year, that is displayed, can be selected by clicking on it in the menu that is above map. It receives data from

23 http://deepweb.ut.ee/automicrosite/business.owl

"AutoMicrosite.LabourCost.Data" topic and publishes summary of data for the selected year to "AutoMicrosite.LabourCost.Summary" topic. Map widget has category "http://schema.org/Map". Allowed minimum dimensions for this widget are 100 pixels in width and 50 pixels in height, no maximum dimensions have been defined.

- Table widget is used as a secondary data visualization method. It displays data across all years and countries at the same time. It receives data from "AutoMicrosite.LabourCost.Data" topic. Table has category "http://schema.org/Table". Allowed minimum dimensions for this widget are 100 pixels in width and 100 pixels in height, no maximum dimensions have been defined.
- Summary widget displays a short summary of the data it receives from the map widget. It receives data from "AutoMicrosite.LabourCost.Summary" topic. Summary has category "http://schema.org/WPFooter". Allowed minimum dimensions for this widget are 100 pixels in width and 25 pixels in height, no maximum dimensions have been defined.
- Menu widget allows switching between visual widgets. It reads the buttons to display from a buttons property and publishes click events to AutoMicrosite.MenuClick topic. Menu widget has categories "http://schema.org/SiteNavigationElement" and "http://-deepweb.ut.ee/AutoMenu", the latter one is used by the application to recognize menus the application is able to use for navigation widgets. Menu widget has minimum width 200 pixels and minimum height 25 pixels.

Templates

For the given scenario a simple template with 3 placeholders is required. Template consists of a header, a content and a footer area. According to Guideline 2, menu is placed in the header area. According to Guideline 6 content is placed right below the header and the footer is placed below the content area. Header allows categories "http://schema.org/SiteNavigationElement" and "http://schema.-org/WPHeader". Content placeholder allows categories "http://schema.org/-Map", "http://schema.org/MediaObject" and "http://schema.org/-

Table". Footer placeholder allows category "http://schema.org/WPFooter" and it is optional.

6.2.2 Mashup construction

Input

All the widgets are combined into a JSON object, as described in Chapter 5.6. The resulting object is sent as POST request body to JSON API of the application. Example 6.1 shows an example input for creating such a mashup.

```
{"title":"My Mashup","widget":[{"url":"http:\/\/localhost\/Automated-
generation-of-
microsites\/AutoMicrosite\/widgets\/Data\/Data.oam.xml"},
{"url":"http:\/\/localhost\/Automated-generation-of-
microsites\/AutoMicrosite\/widgets\/Map\/Map.oam.xml"},
{"url":"http:\/\/localhost\/Automated-generation-of-
microsites\/AutoMicrosite\/widgets\/Menu\/Menu.oam.xml"},
{"url":"http:\/\/localhost\/Automated-generation-of-
microsites\/AutoMicrosite\/widgets\/Summary\/Summary.oam.xml"},
{"url":"http:\/\/localhost\/Automated-generation-of-
microsites\/AutoMicrosite\/widgets\/Table\/Table.oam.xml"}]}
```

Example 6.1: Mashup construction input

Server-side component

Server-side procedure proceeds as described in Chapter 4.1.1. Menu widget is placed into header placeholder, because they both contain "http://schema.org/Site-NavigationElement" category. Summary widget is placed into the footer placeholder because it matches "http://schema.org/WPFooter" category. Map and table widgets are both placed into the content placeholder because they match categories "http://schema.org/Map" and "http://schema.org/Table", respectively. Table widget is placed on a separate page because there is menu widget available and it is placed inside the same placeholder with visualization widget.

Client-side component

After the server-side process has finished, the client-side process will proceed as described in Chapter 4.1.2. Once all the widgets have finished loading, the data widget will publish the data to visual widgets. Map widget will further publish the summary of the data to Summary widget. A screenshot of the resulting mashup is depicted in Figure 6.1.

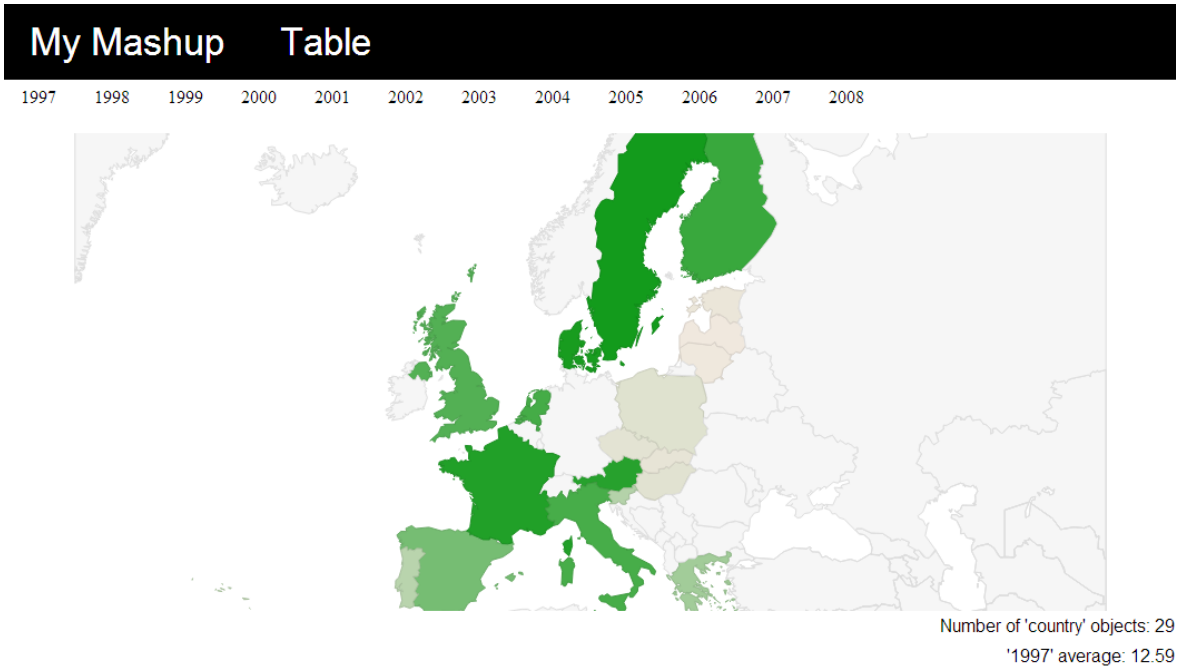


Figure 6.1: Screenshot of proof of concept 1 microsite

6.3 Proof of Concept 2

A second, more complex, scenario was used to validate that the application works in cooperation with Transformer Widget, introduced in Chapter 3.6, and automated OpenAjax hub SOAP Proxy Widget generator, introduced in Chapter 3.7. The mashup will query Estonian business registry with a business name from where it will receive registration code as response. It will then use this registration code to query Inforegister.ee SOAP service "`getOrganizationDetails`" operation, to get information about the business, and "`getDeptSummary`" operation, to get dept related information about the business. It will then display business information in one table, dept information in another table and business address will be visually displayed on a Google Map next to rest of business information.

6.3.1 Components

Widgets

Seven different widgets are required for this mashup and two instances of Proxy Widget are created. Widgets are described using OpenAjax metadata 1.0 specification:

- Google Maps widget is used for visually displaying business address. It is based on Google Maps API, address will be marked using a red pin. It listens to topic "`AutoMicrosite.GoogleMaps`" for an object that contains an address. Map widget has a category "`http://schema.org/Map`". It has `min-width` and `min-height` values of 100 pixels.
- Organization information table widget is used for displaying business information in a table form. It shows name of the organization, registration code, establishment year, address and the field of business. It listens to topic "`AutoMicrosite.-Table.OrganizationData`" for organization information. It has categories "`http://schema.org/Table`". and "`http://schema.org/About-Page`". It has `min-width` and `min-height` values of 100 pixels.
- Organization debt information table widget is used for displaying debt information about the business. It display the summarized debt information of the company, including tax debt and debt listen in Inforegister.ee database by third parties. It

listens to "AutoMicrosite.Table.OrganizationData.Debt" topic for the information. Debt information table has "http://schema.org/Table" category. It has min-width and min-height values of 100 pixels.

- Business registry query widget is a non-visual widget that takes business name as input and publishes business registry code of that business. It queries Estonian business registry through a server side proxy script to obtain this information. It receives name to query from a property name and publishes the result of the query to topic "AutoMicrosite.BusinessRegister.QueryResponse". Business registry query widget has category "http://schema.org/Data-set".
- Key widget publishes SOAP access key when necessary. It subscribes to "AutoMicrosite.BusinessRegister.QueryResponse" to be notified when the key is required. Access key is read from the property key. Key widget has category "http://deepweb.ut.ee/Datasets".
- Transformer widget integrates structurally different data by using semantic information. This way visual widgets do not have to subscribe to exactly the same topics that data widgets publish to and the structure of the messages exchanged does not need to be exactly the same. Transformer widget has category "http://deepweb.ut.ee/Transformer". Longer description of this widget is given in Chapter 3.6. All the other widgets have been semantically described for this scenario.
- Proxy widget is a non-visual widget for surfacing SOAP services. Two instances of this widget are necessary, one for "getOrganizationDetails" operation and another for "getDebtSummary" operation. It takes three parameters as input: wsdl, the URL of the WSDL file for the service, operation, the name of the operation that is run when the proxy widget is called, and proxy, the URL of the proxy service. Proxy widget has category "http://schema.org/Dataset". Longer description of this widget is given in Chapter 3.7.

Templates

Template necessary for this scenario contains 3 template placeholders. In the top of the Web page there are two content placeholders. One on the left side for textual content and one on the right side for illustration. This corresponds to Guideline 3. These content placeholders both take 50 percentage width and 50 percentage height, to satisfy Guideline 1. The placeholder on the left side allows widget with category "`http://-schema.org/Table`" or "`http://schema.org/AboutPage`". The one on the right side allows widget with category "`http://schema.org/MediaObject`" or "`http://schema.org/Map`". Below these two placeholders there is a content placeholder that takes 100 percentage width and 50 percentage height, to satisfy Guideline 1. This placeholder has category "`http://schema.org/Table`".

6.3.2 Mashup construction

Input

Mashup creation process is started by submitting widgets described above to Auto Microsite system. All the widgets are combined into a JSON object, as described in Chapter 5. For the business registry query widget the `name` parameter is sent with the name of the business to query, e.g. "EVETERM OÜ". Proxy Widget is sent twice, for both of the instances `wsdl` parameter is set with the WSDL URL and `proxy` parameter is set with the URL of the proxy service. In addition, the `operation` parameter is set. For the first instance "`getOrganizationDetails`" is used and for the second instance "`getDebtSummary`" is used. Example 6.2 shows an example input for creating such a mashup.

```
{"title":"My Mashup","widget":
[{"url":"http:\/\/deepweb.ut.ee\/automicrosite\/Widgets\/GoogleMaps\/
GoogleMaps.oam.xml"},
{"url":"http:\/\/deepweb.ut.ee\/automicrosite\/Widgets\/OrganizationD
eptInfo\/OrganizationDeptInfo.oam.xml"},
{"url":"http:\/\/deepweb.ut.ee\/automicrosite\/Widgets\/OrganizationI
nfo\/OrganizationInfo.oam.xml"},
{"url":"http:\/\/deepweb.ut.ee\/automicrosite\/Widgets\/ProxyWidget\/
ProxyWidget.oam.xml","properties":
{"wsdl":"http:\/\/deepweb.ut.ee\/automicrosite\/wsdl\/krdxInterfaceSe
rvice-liisi-1-enhanced-
again.wsdl","operation":"getOrganizationDetails","proxy":"http:\/\/de
epweb.ut.ee\/proxywidget\/"}},
{"url":"http:\/\/deepweb.ut.ee\/automicrosite\/Widgets\/ProxyWidget\/
ProxyWidget.oam.xml","properties":
{"wsdl":"http:\/\/deepweb.ut.ee\/automicrosite\/wsdl\/krdxInterfaceSe
rvice-liisi-1-enhanced-
again.wsdl","operation":"getDebtSummary","proxy":"http:\/\/deepweb.ut
.ee\/proxywidget\/"}},
{"url":"http:\/\/deepweb.ut.ee\/automicrosite\/Widgets\/TransformerWi
dget\/TransformerWidget.oam.xml"},
{"url":"http:\/\/deepweb.ut.ee\/automicrosite\/BusinessRegister\/Busi
nessRegisterQuery.oam.xml","properties":{"name":"EVETERM O\u00dc"}},
{"url":"http:\/\/deepweb.ut.ee\/automicrosite\/Widgets\/Key\/Key.oam.
xml","properties":{"key":"API_KEY_HERE"}}]}
```

Example 6.2: Mashup construction input

Server-side component

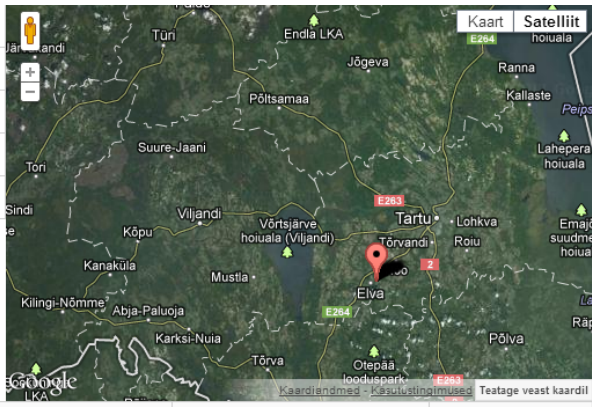
The server-side component works as described in Chapter 4.1.1. The template described above is chosen because matching visual widgets are found for all mandatory placeholders in that template.

Business registry query widget, Transformer Widget and Proxy Widget are identified as data widgets, since they have only "`http://schema.org/Dataset`" or "`http://deepweb.ut.ee/Transformer`" category. This means that these widgets do not get a placeholder, priority or dimensions. For the right side content placeholder, Google Maps widget is found to be a match. Both have category "`http://schema.org/Map`" and there are no size restrictions in that placeholder. For the left side content placeholder organization information widget is found to be a match. Both have categories "`http://schema.org/Table`" and "`http://schema.org/AboutPage`", and there are no size restrictions in that placeholder. For the bottom content placeholder organization debt information widget is found to be a match. Both have category "`http://schema.org/Table`" and there are no size restrictions in that placeholder.

Client-side component

The client-side component attaches all the widgets to the mashup, following the process described in Chapter 4.1.2. Once all the widgets have been loaded, business registry query widget will make a request to business registry with the property `name`. It then publishes the response with registration code to "`AutoMicrosite.Business-Register.QueryResponse`" topic where Transformer Widget routes it to SOAP Proxy Widget topics. Proxy Widget responses are again routed to organization information and organization debt information widgets by the semantic integration widget. A screenshot of the resulting mashup is depicted in Figure 6.2.

Legal name	BEST CANDY OÜ
Registration code	12345678
Establishment year	2007
Address	Kesk 32, 61507 Elva, EE
Description of field	Selling candy



Tax debt as reported by Estonian Tax and Customs Board	Currency code of the tax debt sum	Date and time of the last update of tax debt information	The sum on active claims against the organization	Currency code of the debt sum
1000000	EUR	2013-05-18	100000000	EUR

Figure 6.2: Screenshot of proof of concept 2 microsite

7 Conclusions

This thesis describes a solution for the problem of automated layout selection for a specific class of Web sites, namely microsites, which are visually simple Web sites according to Harper et al. [5] and consist of one or a couple of content pages and fit on page without scrolling. More specifically, in this thesis, Auto Microsite system was implemented, which enables automated layout selection and packaging of microsites made of widgets.

The literature review in this thesis revealed that usability studies generally concentrate on regular Web pages, but mashups have some distinctive characteristics, which means that not all the existing guidelines are applicable in the case of mashup Web sites. Also, several existing mashup tools were compared. Most of the tools were found either too simplistic to solve real-world problems or too complex for an average computer user.

To encounter shortcomings identified from the literature review rule-based matchmaking of widgets with layout templates was proposed as solution. For this, widgets, with categories and dimensions, and layout templates, with acceptable categories and dimensions, are defined. The layout templates and widgets must also satisfy usability guidelines, in order for the result to satisfy usability guidelines. Widgets and layout templates are then matched and additional rules are applied to modify the result.

An overview of used technologies and standards was given. OpenAjax Metadata 1.0 specification was used to describe the widgets. Mashups themselves are constructed on top of OpenAjax Hub 2.0, which enables secure widget separation and communication. RuleML 1.0 rule markup language was used to write matching rules. Schema.org ontology was used to categorize widgets and as the ontology for semantic integration of messages. Microdata was used to mark up widget placeholders on layout templates. Semantic integration widget by Rainer Villido was proposed to be used for semantic integration of messages. Proxy widget by Karli Kirsimäe was proposed to be used for communication with SOAP services.

The Auto Microsite system was validated on two proof of concept scenarios. The first one was simple visualization of EU wages data from one source. The visualization included a Google Chart Tools based map widget, a table widget, a summary widget and a

menu widget, that would enable switching between the map and the table widgets. The second scenario, combined data from Estonian business registry database with Inforegister.ee debt information SOAP service. The semantic integration widget was used for easing widget communication and the proxy widget was used for querying SOAP service. The visualization consisted of two tables, for displaying general organization and debt information, and a Google Maps based widget, for visualizing address of the organization.

The proof of concept scenarios gave satisfactory results, several ways of further improving the application were recognized. These are given under future work chapter.

8 Future work

The application developed for this thesis allows automatic construction of visually simple Web sites, i.e., Web sites that concentrate on one topic, fit on page without scrolling and have no input forms. In real-world situations more complex mashups might be necessary, especially in the case of enterprise mashups. In order to support more complex Web pages, for example with forms and several pages, the layout selection solution should be developed further. Existing infrastructure should be able to handle more complex Web sites, but more rules and layout templates are required to be developed for results of better usability.

The simplest way to improve the resulting mashup would be writing new rules and layout templates for more cases. For the two proof of concept scenarios only a few layout templates were created and the rules were also scenario-specific. These rules and templates allow generation of mashups relatively similar to proof of concept scenarios, but when there are more or different widgets then there are no compatible layout templates to map them to.

Another way to improve the layout selection would be to use more detailed ontology than Schema.org. This would allow describing the nature of widgets more specifically, which in return would allow more specific rules. For instance, for the proof of concept scenarios two different map widgets were developed: one based on Google Maps API and the other based on Google Chart Tools API. Both of these widgets were described using the Schema.org class Map, which means that for the application they are the same, but in fact they are used in completely different scenarios. Google Maps API based widget is used for illustrating addresses or coordinates of places, Google Chart Tools API based widget is used for displaying summarized data. In some situations they may have to be positioned differently.

Additionally, more data could be used in the decision process. For example, the topics are already annotated in order to generate semantic integration mappings, the same data could be used in the widget-template matchmaking process to group together similar widgets.

Also the RuleML service component needs further development before it can be used in a production environment. It is currently a very basic RESTful service with no

authentication or resource usage monitoring, anyone with service endpoint URL can run any ruleset on it. This is a problem because certain rulesets can run for a very long time or even crash the server.

9 Abstract (in Estonian)

Automaatne reeglitel põhinev veebilehe struktuurimallide valimine ja rakendamine

Magistritöö (30 EAP)

Hans Mäesalu

Resümee

Veebi avatud arhitektuuron loonud soodsa pinnase veebisolevate andmete kasutamiseks nii keerulisemates kui lihtsamates veebirakendustes. Andmete kogumise ja visualiseerimise lihtsustamiseks lihtsates veebirakendustes on loodud hulganisti tööriistu, mille seas on ka mashup'ide loomise tööriistad. Olemasolevate tööriistadega kõrge kasutatavusega mashup veebilehe loomine võib aga paraku olla keerukas, kuna nõuab erinevate tehnoloogiate ning programmeerimiskeelte tundmist, rääkimata kasutatavuse juhtnööridega kursisolemist. Kuigi osad mashup'ide platvormid, a'la OpenAjax Hub, lihtsustavad olemasolevate komponentide kombineerimist, on lahendamata probleemiks siiani nende rakenduste kasutatavus.

Käesolev magistritöö kirjeldab reeglipõhist lahendust andmete visualiseerimise vidinate jaoks sobiva veebilehe malli automaatseks valimiseks vastavalt enimlevinud veebilehtede kasutatavuse juhtnööridele. Selleks laetakse vidinate ning struktuurimallide kirjeldused koos kasutatavuse juhtnööridest saadud reeglitega reeglimootorisse ning kasutatakse reeglimootorit ekspertsüsteemina, mis soovitab sobivamaid malle vastavalt etteantud vidinate komplektile. Lahenduse reeglipõhine ülesehitus võimaldab uute vidinate ning mallide lisandumisel või juhtnööride muutumisel operatiivselt reageerida nendele muutustele reeglibaasi täiendamise kaudu.

Väljapakutud lahendus realiseeriti käesoleva töö raames Auto Microsite rakendusena, mis koosneb serveri- ning kliendipoolsest osast. Serveri poolel toimub reeglite abil vidinate komplekti visualiseerimiseks sobiva malli valimine kasutades OO jDREW RuleML reeglimootorit ning rakenduse paketeerimiseks koodi genereerimine. Kliendi poolel kasutatakse OpenAjax Hub raamistikkuvidinate turvaliseks eraldamiseks ning omavahel suhtlemapanemisel. Samuti on kliendi poolel lahendatud genereeritud veebilehe vastavusse

viimine brauseri võimalustega.

Katsetamaks Auto Microsite rakendust praktikas loodi seda kasutades realisatsioonid kahele lihtsale stsenaariumile. Esimesel juhul viusaliseeriti Euroopa 1997-2008 tööjõukulude (Hourly labour costs in Euros (European Union 1997-2008) ing. k.) andmeid kaardi, tabeli, kokkuvõtte ja menüü vidinatega. Teisel juhul kasutati lisaks andmete visualiseerimise vidinatele ka väliseid andmeallikaid, mis olid realiseeritud mittevisuaalsete vidinatena. Saadud andmed visualiseeriti kahe tabeli ning ühe kaardi vidinaga. Näidisveebilehtede loomise tulemusena järeldub, et rakendus sobib lihtsate veebilehtede loomiseks. Lisaks on võimalik lahendust täiendada keerukamate veebirakenduste automaatseks loomiseks läbi vastavate mallide ning reeglite lisamise.

10 Bibliography

Bibliography

[1] M. Caceres. Widgets 1.0: The Widget Landscape. 2008, http://www.w3.org/TR/2008/WD-widgets-land-20080414/. Cited: 15.05.2013

[2] A. Namoun, T. Nestler, A. D. Angeli. Conceptual and Usability Issues in the Composable Web of Software Services. *ICWE'10 Proceedings of the 10th international conference on Current trends in web engineering*, pages 396-407, 2010

[3] A. Dingli, J. Mifsud. USEFul: A Framework to Mainstream Web Site Usability Through Automated Evaluation. *International Journal of Human Computer Interaction*, pages 10-30, 2011

[4] Web Accessibility and Usability Working Together. http://www.w3.org/WAI/intro/usable. Cited: 10.01.2013

[5] S. Harper, E. Michailidou, R. Stevens. Toward a Definition of Visual Complexity as an Implicit Measure of Cognitive Load. *ACM Transactions on Applied Perception*, Volume 6, Number 10, pages 1-18, 2009

[6] 15th Annual Webby Awards Nominees & Winners. http://www.webbyawards.com/webbys/current.php?season=15. Cited: 06.02.2012

[7] P. L. Thung. Improving a Web Application Using Design Patterns: A Case Study. *Information Technology (ITSim)*, Volume 1, pages 1-6, 2010

[8] K. E. Schmidt, Y. Liu, S. Sridharan. Webpage aesthetics, performance, and usability: Design variables and their effects. *Ergonomics*, Volume 52, Number 6, pages 631-643, 2009

[9] D. Fox, S. Naidu. Usability Evaluation of Three Social Networking Sites. 2009, http://usabilitynews.org/usability-evaluation-of-three-social-networking-sites/. Cited: 20.05.2013

[10] Pawan Vora. *Web Application Design Patterns*. Morgan Kaufmann Publishers, 2009

[11] M. O. Leavitt, B. Shneiderman. *The Research-Based Web Design & Usability Guidelines*. U.S. Government Printing Office, 2006

[12] S. Dahal. Eyes don't lie: understanding users' first impressions on website design using eye tracking. MSc thesis, , Missouri University of Science and Technology, Missouri, 2010

[13] M. Russell. Using Eye-Tracking Data to Understand First Impressions of a Website. *Usability News*, Volume 7, Number 1, 2005

[14] J. H. Goldberg, X. P. Kotval. Computer interface evaluation using eye movements: methods and constructs. *International Journal of Industrial Ergonomics*, Volume 24, Number 6, pages 631–645, 1998

[15] R.W. Bailey, S. Koyani, J. Nall. Usability testing of several health information Web sites. *National Cancer Institute Technical Report*, 2000

[16] J. D. McCarthy , M. A. Sasse , J. Riegelsberger. Could I have the Menu Please? An Eye Tracking Study of Design Conventions. *In Proceedings of HCI2003*, pages 401-414, 2003

[17] J. Harty. Finding usability bugs with automated tests. *Communication of the ACM*, Volume 54, Number 2, pages 44-49, 2011

[18] C. Cappiello, F. Daniel, M. Matera. A Quality Model for Mashup Components. *ICWE '9 Proceedings of the 9th International Conference on Web Engineering*, pages 236-250, 2009

[19] V. Hoyer, M. Fischer. Market Overview of Enterprise Mashup Tools. *ICSOC '08 Proceedings of the 6th International Conference on Service-Oriented Computing*, pages 708-721, 2008

[20] J. Maras, M. Štula, J. Carlson. Extracting Client-side Web User Interface Controls. *ICWE'10 Proceedings of the 10th international conference on Web engineering*, pages 502-505, 2010

[21] O. Chudnovskyy, T. Nestler, M. Gaedke, F. Daniel, J. I. Fernández-Villamor, V. I. Chepegin, J. A. Fornas, S. Wilson, C. Kögler, H. Chang. End-user-oriented telco mashups: the OMELETTE approach.. *WWW '12 Companion Proceedings of the 21st international conference companion on World Wide Web*, pages 235-238, 2012

[22] S. Ceri, P. Fraternali, A. Bongio. Web Modeling Language (WebML): a modeling language for designing Web sites. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, Volume 33, Number 1-6, pages 137-157, 2000

[23] S. Lok , S. Feiner. A Survey of Automated Layout Techniques for Information Presentations. *Proceedings of the 2001 SmartGraphics Symposium (SG2001)*, 2001

[24] A. Borning, R. K.-H. Lin, K. Marriott. Constraint-based document layout for the Web. *Multimedia Systems*, Volume 8, Number 3, pages 177-189, 2000

[25] A. Gonzales-Uriel, E. Roanes-Lozano. A knowledge-based system for house layout selection. *Mathematics and Computers in Simulation*, Volume 66, Number 1, pages 43-54, 2004

[26] K. Knight. Responsive Web Design: What It Is and How To Use It. 2011, `http://coding.smashingmagazine.com/2011/01/12/guidelines-for-responsive-web-design/`. Cited: 20.05.2013

[27] Home and Away: Iraq and Afghanistan War Casualties. `http://edition.cnn.com/SPECIALS/war.casualties/`. Cited: 10.07.2012

[28] Yahoo! Design Pattern Library. `http://developer.yahoo.com/ypatterns/`. Cited: 10.07.2012

[29] Skype.com. `http://www.skype.com/intl/et/home`. Cited: 10.07.2012

[30] Dropbox.com. `https://www.dropbox.com/home`. Cited: 10.07.2012

[31] K. Perzel, D. Kane. Usability Patterns for Applications on the World Wide Web. *Pattern Languages of Program Design 1999 Proceedings*, 1999

[32] OpenAjax Metadata 1.0 Specification. `http://www.openajax.org/member/wiki/OpenAjax_Metadata_Specification`. Cited: 20.05.2013

[33] H. Boley, T. Athan, A. Paschke, S. Tabet, B. Grosof, N. Bassiliades, G. Governatori, F. Olken, D. Hirtle. Schema Specification of Deliberation RuleML Version 1.0. 2012, `http://ruleml.org/1.0/`. Cited: 20.05.2013

[34] H. Boley, A. Paschke, O. Shaq. RuleML 1.0: The Overarching Specification of Web

Rules. 2012, http://cs.unb.ca/~boley/papers/RuleML-Overarching.pdf. Cited: 03.02.2012

[35] Schema.org. http://www.schema.org. Cited: 20.05.2013

[36] D. Brickley. Web Schemas TF and Schema.org. 2011, http://www.w3.org/2001/sw/interest/schema.org-collab.html. Cited: 15.05.2013

[37] I. Hickson. HTML Microdata. , http://www.w3.org/TR/2012/WD-microdata-20120329/. Cited: 20.05.2013

[38] F. Rivoal, H. W. Lie, T. Çelik, D. Glazman, A. Kesteren. Media Queries. 2012, http://www.w3.org/TR/2012/REC-css3-mediaqueries-20120619/. Cited: 20.05.2013

[39] Media types. http://www.w3.org/TR/CSS21/media.html. Cited: 20.05.2013

[40] L. Wroblewski. Multi-Device Layout Patterns. 2012, http://www.lukew.com/ff/entry.asp?1514. Cited: 20.05.2013

[41] R. Villido. Semantic Integration Platform for Web Widget Communication. MSc thesis, Institute of Computer Science, University of Tartu, Tartu, 2010

[42] K. Kirsimäe. Automated OpenAjax Hub Widget Generation for Deep Web Surfacing. MSc thesis, Institute of Computer Science, University of Tartu, Tartu, 2011

[43] M. Zalewski. Browser Security Handbook, part 2. 2009, https://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy. Cited: 4.03.2013

[44] K. Simpson. Defining Safer JSON-P. 2010, http://json-p.org/. Cited: 4.03.2013

[45] A. Kesteren. Cross-Origin Resource Sharing. 2013, http://www.w3.org/TR/cors/. Cited: 4.03.2013

[46] M. Hossain, Using CORS. *HTML5 Rocks*. Cited: 12.03.2013

[47] P. Kruchten. Architectural Blueprints—The "4+1" View Model of Software

Architecture. *IEEE Software*, Volume 12, Number 6, pages 42-50, 1995

[48] TIBCO PageBus. `http://developer.tibco.com/pagebus/default.jsp`. Cited: 20.05.2013

[49] M. Ball, H. Boley, D. Hirtle, J. Mei, B. Spencer. Implementing RuleML Using Schemas, Translators, and Bidirectional Interpreters. 2005, `http://ruleml.org/w3c-ws-rules/implementing-ruleml-w3c-ws.html`. Cited: 15.02.2013

[50] WebSchemas/Datasets. `http://www.w3.org/wiki/WebSchemas/Datasets`. Cited: 26.12.2012

[51] The Dojo Toolkit. `http://dojotoolkit.org/`. Cited: 05.04.2012

[52] Hourly labour costs in Euros (European Union 1997-2008). `http://www-958.ibm.com/software/data/cognos/manyeyes/datasets/hourly-labour-costs-in-euros-europ/versions/1`. Cited: 26.12.2012

11 Appendix

11.1 Source code

The source code of the implementation is available at GitHub repository: https://github.com/hansm/Automated-generation-of-microsites

11.2 RuleML rules

RuleML rules written for the application are available through GitHub repository:

- https://github.com/hansm/Automated-generation-of-microsites/blob/master/AutoMicrosite/Rules/General.ruleml
- https://github.com/hansm/Automated-generation-of-microsites/blob/master/AutoMicrosite/Rules/Generalization.ruleml
- https://github.com/hansm/Automated-generation-of-microsites/blob/master/AutoMicrosite/Rules/Priority.ruleml

Non-exclusive licence to reproduce thesis and make thesis public

I,
Hans Mäesalu
(date of birth: 07.02.1988),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,
Automated Rule-Based Selection and Instantiation of Layout Templates for Widget-Based Microsites
supervised by Peep Küngas,

2. I am aware of the fact that the author retains these rights.
3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 21.05.2013