TOWARDS FAST METAMODEL EVOLUTION IN LIQUIDML

ESTEBAN ROBLES LUNA[1,2], GUSTAVO ROSSI[1, 3], JOSE MATIAS RIVERO[1, 3]
FRANCISCO J. DOMINGUEZ-MAYO [4], JULIAN A. GARCIA-GARCIA[4], MARIA J. ESCALONA[4]

[1] LIFIA, Facultad de Informática, UNLP, La Plata, Argentina

{esteban.robles, mrivero, gustavo}@lifia.info.unlp.edu.ar

[2] Also at CIC [3] Also at Conicet

[4] Web Engineering and Early Testing Group, University of Seville, Seville, Spain
julian.garcia@iwt2.org; {fjdominguez, mjescalona}@us.es

The software industry is applying Model-driven development approaches due to a core set of benefits, such as raising the level of abstraction and reducing coding errors. However, their underlying modeling languages tend to be quite static, making their evolution hard, specifically when the corresponding metamodel does not support primitives and/or functionalities required in specific business domains. This paper presents an extension to the LiquidML language to support fast metamodel evolution by allowing experts to abstract new language concepts from primitives while supporting automatic tool evolution and zero application downtime. To probe our claims, we evaluate the evolutionary capabilities of existing modeling languages and LiquidML in a real world language extension.

Key words: Evolution, Model based development, self-reflective, Web development.

1. Introduction

During the last couple of years a myriad of technologies and languages (T&L) have been developed to simplify and speed up the process of Web application development and maintenance. These T&L range from development frameworks such as AngularJS [18], GWT [19], JQuery [20] to non-relational databases such as MongoDB [21] including tools that help to monitor the running application (to keep the application running 24x7), such as New relic [34]. Most of these T&Ls have been developed in the industry and are based on coding activities while only a few domain specific languages (DSL) for Web application development [2, 4] have been developed in the Academia and have real world application [13, 14, 17]. These Web DSLs are generally based on MDE (Model-Driven Engineering) [15] or MDD (Model-Driven Development) [16] and thus, they require a model to code transformation in order to obtain a running application.

In addition to these T&L, the industry has shifted from traditional cascade development approaches to agile practices. These practices have reduced software development costs [35] through constant communication between stakeholders and software reusability and adaptability to change. The necessity for quick changes has emerged due to the large number of applications that make harder to acquire and withhold active users on the Website. Consequently, the ability to make small but effective changes in a matter of one to three days becomes increasingly important and affects the decision of which T&Ls

should be selected. A clear example of this issue is the introduction of A/B testing techniques[a] to help with the analysis of which design version of a new feature will be totally implemented. To achieve it, each version of the feature is partially implemented and presented in a production environment while usage data is recorded. Afterwards, a usage analysis report is generated, making the choice of which design suits better (e.g. makes users more active or improves user retention) a simpler decision. Then the design must be fully implemented within the next couple of days.

A recent study [46] shows that the main aspect for selecting the development process and T&Ls involved in a project has changed from how an initial prototype can be developed to how easy it is to maintain, monitor and find root causes of the problems. The study conducts a survey to identify the main aspects taken into account for the adoption of agile practices and the discoveries are related to the natural ability to constant change that Web applications are currently suffering. The study clearly differentiates two groups:

- Static aspects to modify or adapt the Web application to support a change or new requirement. These aspects may include changing the code or DSL to support a new requirement, modify a DSL transformation to derive new code, add a new library, framework or component or extend the DSL to improve its expressiveness.
- Dynamic aspects that affect the Web application that is running 24x7 and need to be investigated, fixed and deployed. These aspects include monitoring how the servers are running, if CPU and memory are in acceptable levels and the throughput is enough, if there are any memory leaks, exceptions or alerts and how the application may be reconfigured dynamically to fix any of these issues.

Nowadays, many of the code base technologies can easily accommodate many of the static and dynamic aspects needed in modern Web applications. With regards to the static aspects, almost any modern programming language such as Java [37], PHP [23], .NET [38], Ruby [39], Scala [40] can be easily manipulated to change the behavior of the application at design time, either by coding the solution or by reusing existing solutions in the forms of components, libraries or frameworks. Together with the usage of Continuous delivery [41] and QA Automation practices a new version of the application can be safely deployed to hundred of servers in a short acceptable period of time. Furthermore, the dynamic aspects, such as monitoring, become simple by using New Relic, which provides the necessary information to keep servers and Web applications healthy. However, most of these languages do not provide ways to be reconfigured dynamically and thus a whole building and deploying process is needed to fix a problem (there is no dynamic way to fix them).

On the other hand, static aspects in Web DSLs can be solved easily and fast depending on the requirement to be implemented. If the expressiveness of the DSL can accommodate the requirement, then we only need to express it and generally, by means of model transformations, we obtain a new version of the Web application we can deploy. Sometimes, we can reuse existing solutions in the DSL, whenever they are stored in a shared repository (e.g. a model repository). If the DSL does not support the requirement, a series of activities need to be performed, to evolve the language in order to satisfy the business requirements as fast as demanded [46]. For example, let's suppose that we need to add a weather component (Section 3.1) to our application; this component will let the engineers make decisions based

[a] A/B testing (also known as split testing or bucket testing) is a method of comparing two versions of a webpage or app against each other to determine which one performs better. AB testing uses data & statistics to validate new design changes and improve your conversion rates [36].

on the weather information of a particular city (e.g. the city where the person is accessing the application). None of the existing Web modelling languages has this element present in their metamodel and as a consequence they need to evolve the metamodel, the transformations into code and its supporting tools. With regards to the dynamic aspects mentioned in [46], as DSLs are generally based on models, their transformation into lower level entities increases the gap between the space where problems are identified (code and new relic) and the space where the application is created and modified (the DSL). As a consequence, tools such as New Relic becomes a non-viable solution and proposals to fix this gap are barely covered [7]. Therefore, though DSLs for Web application development have increased attraction [4, 44, 45], these problems can limit their adoption in medium to large-size companies [46] because static aspects can be partially covered and the dynamic ones are far from being solved.

In [7, 48] we have briefly introduced the LiquidML language with its core features with strong emphasis in its support for runtime transformations to solve dynamic changes and in [49] we have defined our initial ideas about how LiquidML can provide fast metamodel evolution for the static ones. In this work we provide a full description of the language in addition to details about how the fast metamodel evolution is defined and implemented. We provide a comparison on how evolution in Web modelling languages is implemented and used, in order to prove that LiquidML is a better solution to the others languages in this aspect.

The paper is structured as follows: in Section 2 we present the background of this work: the full definition of the LiquidML language. In Section 3, we show how the language allows the abstraction of new concepts within the same language. In Section 4, we introduce some implementation details and in Section 5 we compare LiquidML with existing Web modelling languages. Finally, Section 6 describes some related work and Section 7 presents some conclusions and future work.

2. Background

The following subsections describe the LiquidML modeling language and environment. In Section 2.1 we offer an overview of LiquidML and its relationship with other modeling languages and activities. And in Section 2.2 we provide a full description of the language including its concepts and some of the already defined primitives. We present LiquidML metamodel and an instantiation with a small real example.

2.1 Overview

LiquidML [7] is a modeling language that allows modelers to design applications based on the message-passing paradigm [5]. We have chosen the message-passing paradigm, as it is well understood for expressing behaviour and information flow in several different types of applications. LiquidML was originally conceived to support changes at runtime using runtime transformations [7] thus it main benefit is that it allows engineers fixed production problems at runtime with 0 application downtime [7, 48].

There are many different subtypes of applications that can be built using LiquidML including Web and Integration applications. This work focuses on the Web aspects of LiquidML and as a consequence, we emphasize the relationship between LiquidML, other modeling languages such as IFML [17] and NDT [4].

Applications models built with “conventional” approaches (e.g. IFML) are transformed into code that is run inside a Web container such as Apache Tomcat [22] or a PHP server. The basic primitives in

these languages abstract high level entities such as Web page, domain objects and usual behaviours such as navigation. As an example, consider the IFML model of Figure 1 that describes a simplified E-Commerce application that has 2 pages: the “Home” and the “Detail”. The boxes inside the pages are instances of model elements that allow listing the Products and allow seeing the actual details of a specific one. In addition, the other box allows computing a ranking for the product to be displayed, e.g. “Ranked #2 in Computers”.

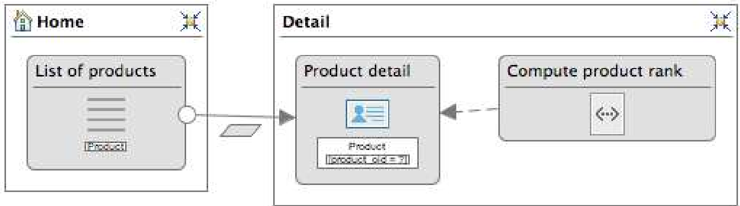


Figure 1. IFML model for an E-Commerce application

From a development process perspective, LiquidML models can be either semi-automatically derived from high-level models such as IFML and NDT (an overview is presented in [7]), or they can be designed manually using a LiquidML editor [7]. When manipulated, some higher-level concepts can be abstracted as modelers discover them and thus the development metamodel gets enriched interactively within the process (Section 3). Similarly to what happens in the lifecycle when using ‘traditional’ Model-driven approaches, a Build/Snapshot is executed and then deployed into a server that is capable of running/interpreting the Web Application. Figure 2 shows a summary of these relationships.

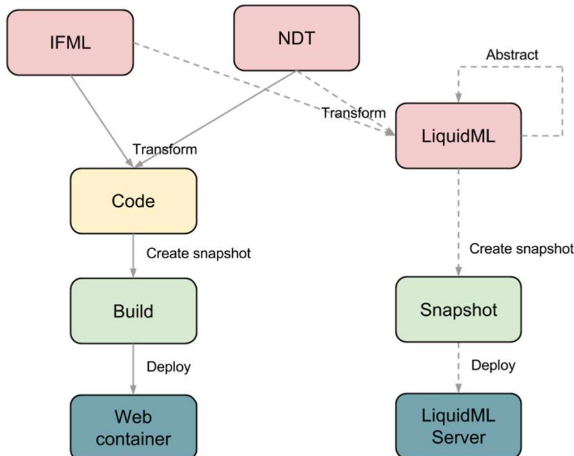


Figure 2. Relationships among modelling languages, IFML and their activities

2.2 Basic concepts

In order to introduce the LiquidML concepts, Figure 3 presents the transformed IFML (E-commerce product list and detail pages) model of Figure 1 in LiquidML using the transformations presented in [7]. The *Element* with no incoming arrow (HTTP listener) represents the *Message* source listener that will

receive incoming requests – in this case, it will receive HTTP request and will transform them into Messages. The Element connected to the *Message* source named "Path router" is a *ChoiceRouter* (instance of *Router*), which behaves like a choice/switch statement and it will route the message to the "Get info" element if the request comes to a URL starting with "/product/*". The "Get info" is another router (instance of Router) that gets information in parallel from multiple sources. It obtains the product info from the DB: "Get product info" (instance of Processor) and triggers the computation of the product's rank (instance of Router), which involves two database queries ("Get user reputation" and "Get product reviews" (instances of *Processor*)) and a *Processor* that computes the rank from this information ("Compute product rank"; instance of *Processor*). Finally, the information gets composed ("Compose data" *Processor*) and used for rendering a Web page in the "Render template" processor.

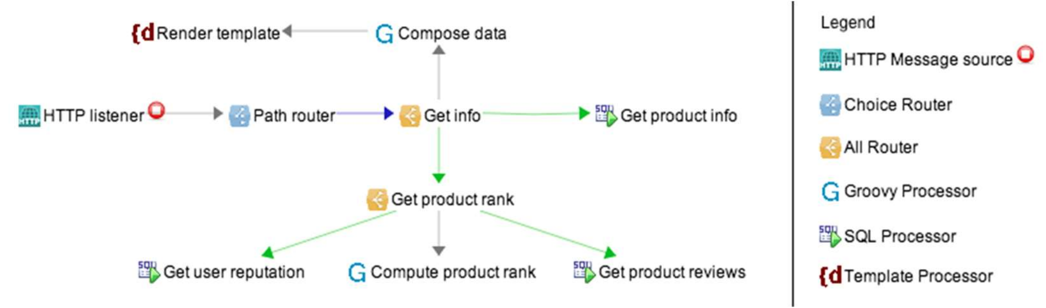


Figure 3. Product details flow

Formally, a LiquidML model is a composition of *Flows*, where each one describes a sequence of steps that need to be applied to the current Web request (called a Message in LiquidML) in order to obtain a proper response. Typically, if the application defines 3 or 4 different Web requests (one per business requirement) each is designed in a different flow. We can also define everything in one huge flow but it will be hard to read and maintain.

A message has a payload (body) and a list of properties, while each step is visually identified by an icon and constitutes an Element of the Flow. Communication between Elements occurs by means of message interchanges. The way in which messages are moved from one Element to another is defined by the Connections between them. We have classified the Elements using the categories found in [5] and every element has a different icon that represents it:

Message source. It is responsible for creating instances of messages based on different conditions. There can be many different types of message sources, for instance one of the HTTP message source listens to incoming requests and generates messages from them. Another example is the Queue message source that listens to a Data queue and creates a new message when the queue is filled. Some other message sources include:

- Cron message source: Creates a message every time a cron expression[b] evaluates to true.
- FTP message source: Creates a message for each file that is read from a remote FTP server.
- File message source: Creates a message for each file that is read from the local file system.

[b] A cron expression is a string consisting of six or seven subexpressions (fields) that describe individual details of the schedule.

Processor. It may transform, execute or just read information included in a message, by changing or reading the payload and properties. There are a wide variety of processors though the most common one is the *ScriptingProcessor* that allows modelers to write scripting code for custom complex logic. Some other common processors are:

- Log processor: It reads information from the message and generates log information based on its configuration.
- Select SQL processor: It uses a Select SQL statement to fetch information from a database as well as sets the list of rows recovered into the message payload.
- Change SQL processor: It executes an Insert/Update/Delete statement in a database and stores the number of affected rows in a property configured by the modeler.
- Dust processor: It converts the message payload using a dust template into processed HTML that can be rendered in a Web browser.
- JSON transformer: It transforms the message payload into a JSON object.
- XML transformer: It changes the message payload to a XML document using an XSLT definition.
- Mapping transformer: It transforms the message payload into a Map/Dictionary through the configured keys and expressions.

Router. It moves the message among *Elements* depending on type and condition. For instance, a *ChoiceRouter* routes the message to a specific *Element* of its list following a Boolean condition.

- Choice router: It behaves in the same way as a 'switch' statement in a procedural programming language. It evaluates the conditions of each of the choice connections in sequence, and the first one that evaluates as true is the one that gets activated; the element that the connection reaches is the next one to be assessed.
- All router: It creates a copy of the current message sending it to 'all connections' by evaluating them in parallel and collecting the results.
- Wiretap router: It designs a copy of the message and sends it in an asynchronous way to the wiretap connection. It is an implementation of the Wiretap pattern of [5].
- Chain router: It evaluates each of the chain connections in sequence. After testing the first chain connection, the result is passed to the second connection and so on until all the chain connections are exhausted.

Connection. It describes a relationship between 2 elements. The most common connection is the 'Next in chain', which specifies that once the 'source' element processes the message, the 'target' element will be the next to process the message. The complete list of connections is:

- Next-in-chain: It describes a sequence between a source element and a target element. The source element can only have one next-in-chain connection, while the target element may have multiple ones.
- Choice connection: It has a configurable condition. The source element can only be a choice router, although the target element can be any kind of element. The choice router may have multiple choice connections.
- All connection: It behaves similarly to the previous one. In this case, all connections only apply to source all routers, though the target element can be of any nature. The all router may have multiple all connections.

E. Wiretap connection: It only applies to wiretap routers and these can only have one wiretap connection to any kind of elements.

Φ. Chain connection: Like choice connections, chain connections only apply to source chain routers, thus the target element can be of any nature. The chain router may have multiple chain connections.

Figure 4 summarizes the metamodel of LiquidML; every Element (ElementDefinition class) has a name and it has many subclasses. The aforementioned Message sources, routers and processors are subclasses of ConnectableElement, which are elements that can be connected between each other. A ConnectableElement can have multiple connections and a Connection has a source and a target element. A Flow and an Abstraction (described in the following section) are CompositeDefinition that allow composing elements in LiquidML.

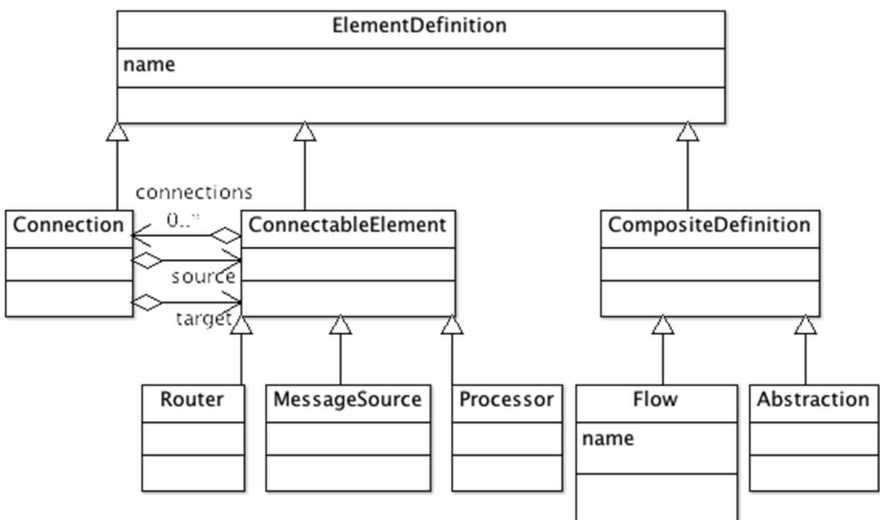


Figure 4. LiquidML metamodel

The following section presents how the fast metamodel evolution is defined in LiquidML; we present it using an exemplar Weather element that we want to add the modelling language.

3. Fast metamodel evolution in LiquidML

In this section we present how the evolution is defined in LiquidML. In Section 3.1 we present an example that will serve through the rest of the paper, to show how the evolution is defined. In Section 3.2 we show how the metamodels are changed at design time, how it affects the application under development and we provide details about how flows are modified with the abstracted concepts.

3.1 Adding a Weather element

When developing applications with models, a desire business feature sometimes requires having a flexible general model that is not always available. For instance, in an E-Commerce application, we

would like to present and change the behavior of the application based on the weather conditions where the user is located. This functionality was not natively included in the LiquidML language and as a result, a workaround is needed to implement it. Therefore, modelers are able to find out a solution that integrates a sequence of processors to perform the external API calls, for instance, to the OpenWeather API, basically by making an IP to City mapping and then, looking up for that city in the API that enables us getting the information needed for our recommendation systems. Figure 5 shows the sequence to integrate our Flow with the OpenWeather API. The 2 toolbars on the left side, show some of the predefined elements of the LiquidML language

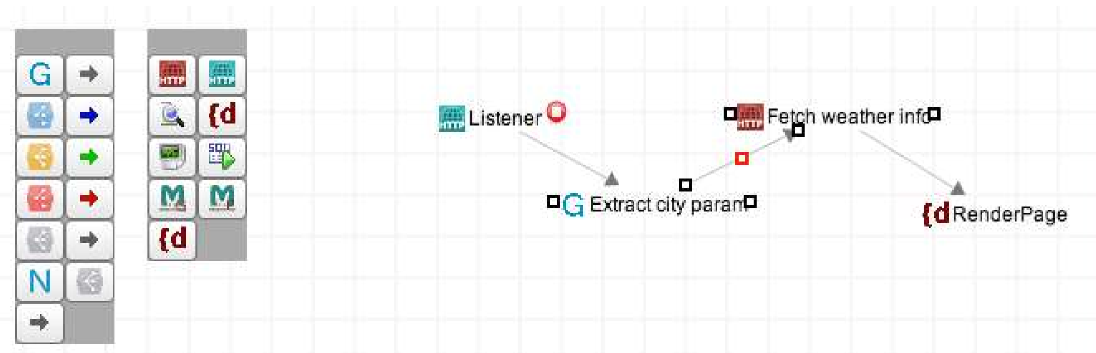


Figure 5. OpenWeather workaround

This solution works fine for one specific flow. Nonetheless, the E-Commerce application is composed of a set of applications where each one may contain multiple flows. In several of these flows we may have to use Weather information, and applying this workaround everywhere is not clearly a feasible solution.

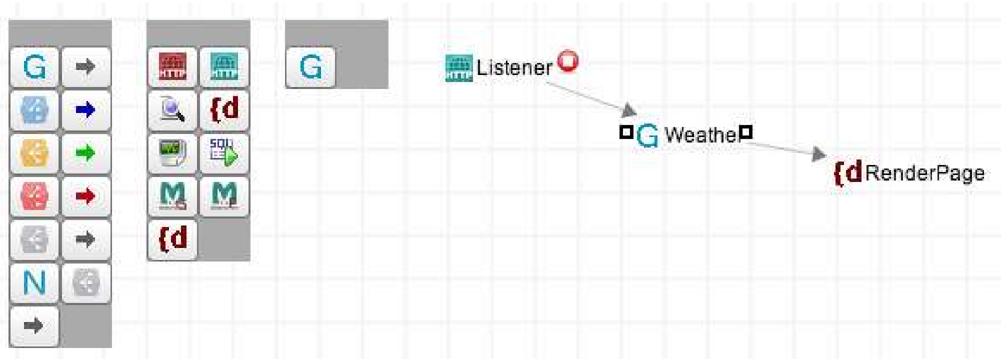


Figure 6. Weather element abstracted

LiquidML enables encapsulating and abstracting a new concept from a subgraph, just following the same approach that code-based environments provide. Thus, modelers can select the elements to be abstracted, by clicking the 'Abstract' button and the environment automatically reconfigures the Flow with the new abstracted element. The environment then creates an instance of the Abstraction metaclass as the next subsection shows. Finally, a new icon is added to the third toolbar that represents the new

Weather concept (Figure 6). From now on, modelers can select the same element from the toolbar and drop it wherever they need it.

3.2 Models and templates

LiquidML allows abstracting new metamodel concepts from LiquidML application models. For instance, the Weather element of the previous section has a definition of 3 connected elements: an element that extracts the city name, a connection and an HTTP request to fetch the information from the Open weather API. This abstraction abstracts these connected elements and can be instantiated anywhere; that is, creating copies of these 3 elements multiple times and with different configurations (e.g. API key, response times, etc). The class hierarchy that captures this configuration is the ElementTemplate hierarchy of Figure 7.

An Abstraction (Meta package of Figure 7) has a name and references a root ElementTemplate element that will configure its pieces; in the Weather example, the template will create the Scripting element, the connection and the HTTP requester element. The configuration of the Abstraction, is then stored in the ElementTemplate instances, e.g. the API key. In addition, SimpleTemplate instances also need to know which primitive class they represent in the Class package (Figure 7). For example, when creating the HTTP requester, the SimpleTemplate instance needs to know the class name: HTTPRequesterProcessor, and all the properties it needs to configure to make it work properly.

Once the Abstraction and template instances are created, they formed a new Abstraction that can be instantiated anywhere. To sum up, a detailed process of the elements being used when abstracting new elements is the following:

1. An instance of an Abstraction is created (Meta package) e.g. the weather element.
2. The instance references a Root template object (Template package) that knows how to configure its internal pieces (Template composite hierarchy) e.g. a weather template configuration.
3. Each Simple template will reference the class to be used for instantiation in the Class package and will store the property values that will be used to configure it. e.g. a Scripting and a HTTP simple templates may store API keys, the script to be used, the URL of the open weather API, etc.
4. Then when the abstraction is used in a flow it can be,instantiated by following a recursive process over the Template hierarchy as follows:
    a. each SimpleTemplate instantiates a Class and configures it. E.g. Scripting and HTTP instances
    b. each CompositeTemplate composes the instantiation result of Simple or Composite templates.

Figure 7 summarizes the Meta, class and instance level while showing the template package cross cutting the 3 of them.

From the editor point of view, once the Abstraction instance is created a set of activities is performed automatically to evolve the development environment and the application under development. Figure 8 describes the list of activities that are automatically performed.

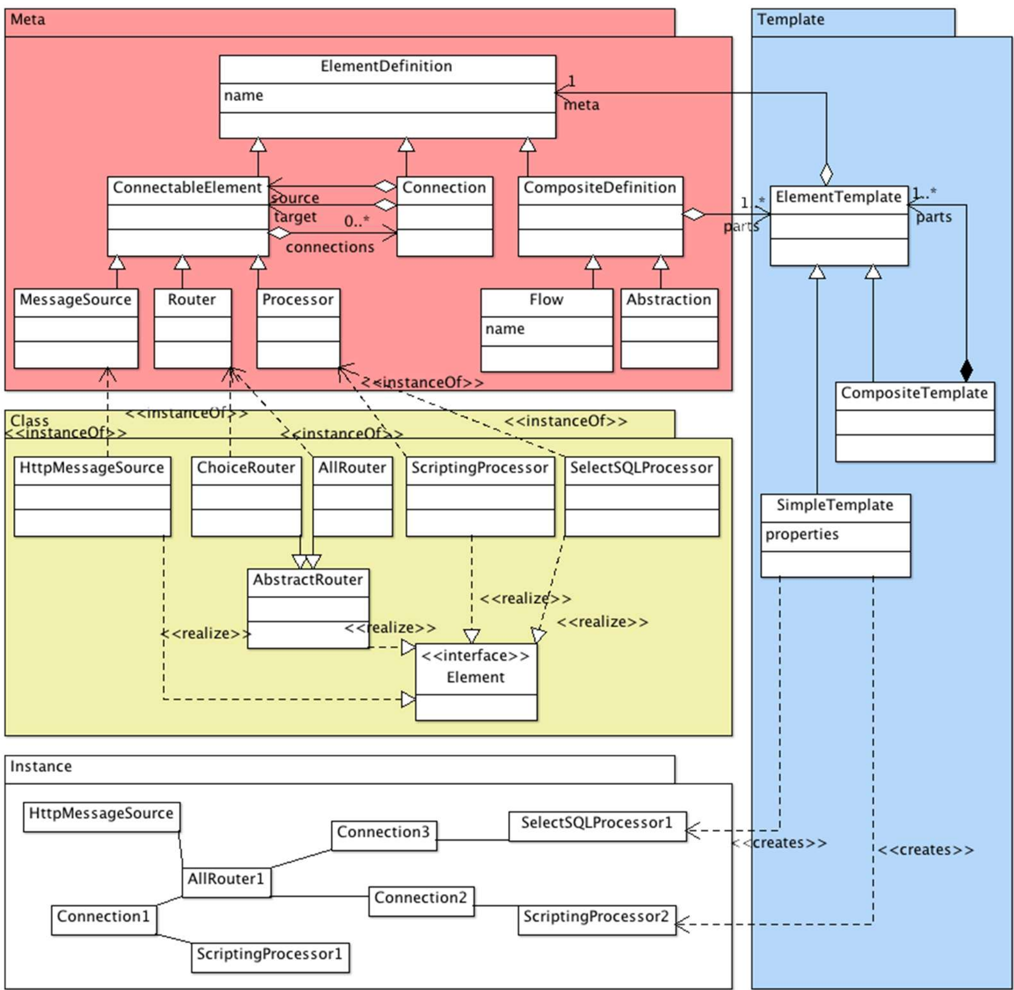


Figure 7. LiquidML models, templates and instances

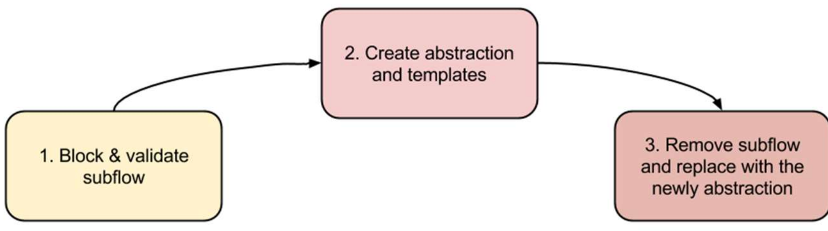


Figure 8. Evolution process

1. The editor blocks the user interaction with the flow under development and check that the elements selected form a single connected subgraph with a root element (an element that

does not have an incoming arrow). This is because the semantics of message passing require an initial element to delegate the behavior.

2. Once the validation is performed, we deal with creating the abstraction and templates instances. The abstraction sets a default icon and the modeler can input a name for the abstraction.
3. The final step consists in removing the elements from the flow under development, creating a template that refers to the new abstraction and hooking up the incoming/outcoming arrows to the abstraction created.

4. Implementation

In this section we present some technical challenges we have faced during the implementation of LiquidML (Section 4.1) and the technologies used to implement it (Section 4.2).

4.1 Technical challenges

As aforementioned, flows define the behavioral part of the Web application. On the contrary to all MDWE approaches, we decide to interpret rather than to derive the code of a Web application. Strong cons and pros of both approaches can be found in and in many informal discussions [25, 26, 27]. Nonetheless, we do not expect to find a definite answer to this matter, but rather present the advantages we found for Web application development in our model-based approach. Code-generation is likely to be the right option at first sight, although interpretation gives us an opportunity to easily modify the behavior in a dynamic way. We have taken into account the following aspects to choose interpretation over code generation:

- We want to manipulate the application at runtime [7, 48] in a controlled way: this aspect can be easily achieved if we have control over the interpreter. In code-generated environments where code has to be compiled, changing the application at runtime can't be generally done in a safe way (e.g. C++, Java).
- We want to avoid having gaps between development and running environments. For instance, diagnosing a production problem in an IFML application is hard, because the code is derived to Java and the traceability links are hard to connect back to the models. In LiquidML, we can measure performance over elements, find elements in the flows with errors, etc. without having to trace back any links, as the elements are the ones interpreted by the interpreter.

As our behavioral models (Flows) are rather simple, the interpreter algorithm is quite simple as well. We present a simplified version of the algorithm using a Java-based pseudocode in the next lines. The interpreter works when messages are received on message sources (e.g. an HTTP message source) (line 1). It finds the next element (*currentElement*) that handles the message (line 2 and 5) and evaluates it using the message content (line 4). An evaluation returns a *Message* instance, which could be the same as the previous one or a new one depending on the *Element* intent (data transformation or routing, among others), and it is passed to the next *Element* until we run out of Elements (line 3).

```
1. OnMessageReceived(MessageSource msgSource, Message message): {
2.     Element currentElement = interpreter.getNextInChain(msgSource);
3.     while (currentElement != null) {
```

```
4.          interpreter.evaluate(currentElement, message);
5.          currentElement = interpreter.getNextInChain(currentElement);
6.     }
7. }
```

A special case is handled by the interpreter (line 4) when the currentElement is an *Abstraction*. In that case, we follow the same approach as any other programming language behaves by using a stack when an *Abstraction is encountered*. So processing an *Abstraction* is processing its internal subflow starting from the initial element. Figure 9 shows an example on how the interpreter dives into 2 abstractions for the complete execution of the flow.

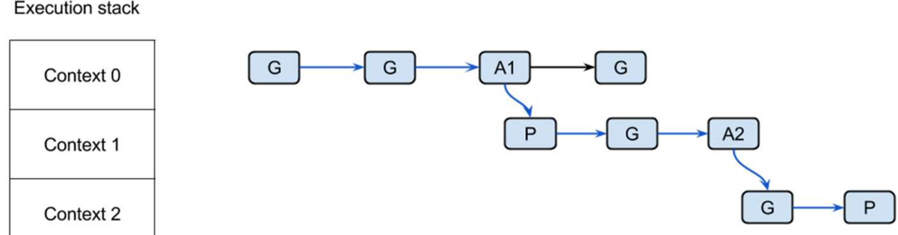


Figure 9. Evaluation of Abstractions using an Execution stack

Interpretation happens while engineers are building the application and when the application is run in every other deployment environment (QA, Staging or Production). Once the models satisfy the requirements, the deployment process to a specific environment occurs. It is an automatic process consisting in moving a copy of the models to the servers where they can start receiving messages. As aforementioned, unforeseen problems may appear in a production environment, thus in the following subsection, we present two tools to help to diagnose problems while our models are running.

4.2 Java implementation

The aforementioned concepts have been implemented in an environment that allows modelers to create applications based on the message-passing paradigm. It is a completely Web-based environment and to the best of our knowledge, the first one to additionally implement Modeling as a Service paradigm [8].

One clear advantage of our implementation is that help experts to debug the application under development at the model level. Figure 10 displays how they are allowed to send messages to specific model elements and Figure 11 represents how they can set breakpoints and evaluate expressions dynamically in the Web editor.

The LiquidML environment is composed of 2 main applications that can be instantiated multiple times and run in a cluster: editor and servers. The editor is instantiated in multiple machines working behind a load balancer in Amazon EC2[c], though it can be installed in any local servers. Below, we briefly describe each application:

- LiquidML editor: it allows defining the applications, modelling Flows and deploying them to LiquidML servers. It also, allows modellers to abstract new concepts from existing models.

---

[c] http://aws.amazon.com/ec2/

E. LiquidML server: Is responsible for holding the application definitions, the LiquidML interpreter [7] and notifying the editor about how applications are running. In addition, it regularly checks if it has any pending deployments and if so, it fetches the Application and automatically deploys it.

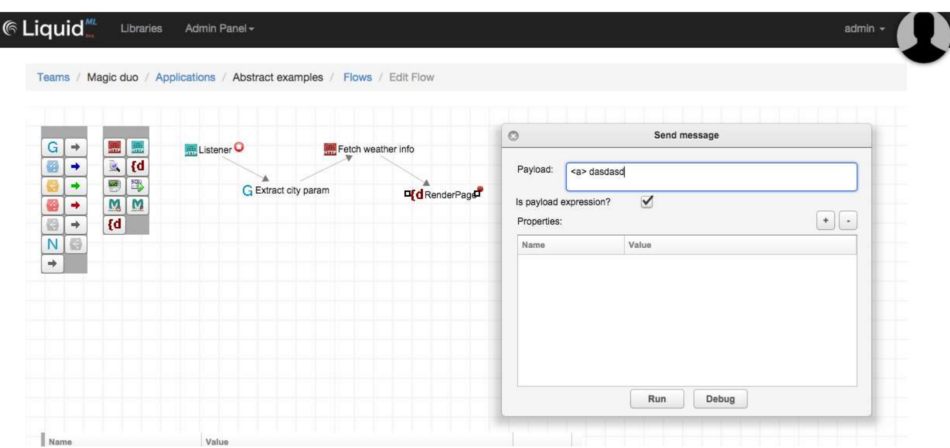


Figure 10. Sending messages to specific model elements

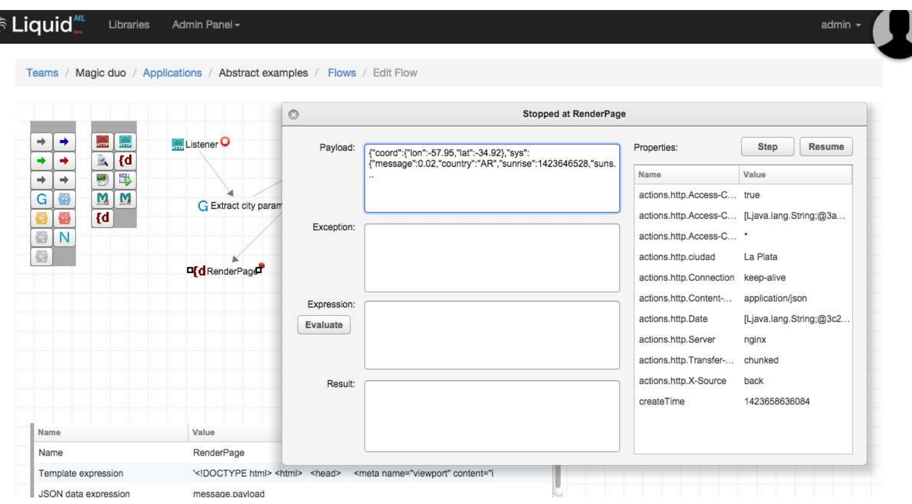


Figure 11. Debugger stopped at model breakpoint

Both editor and server have been built using open source technologies of J2EE stack, particularly the following ones: Spring and Hibernate for basic service and ORM mapping; Spring MVC and Twitter bootstrap for UI; and Jersey for the LiquidML API. As part of this development, we have created the CupDraw framework[d] for building Web diagram editors, which is publicly available. For the technical readers, we invite them to visit the LiquidML site http://www.liquidml.com and check the project's

---

[d] https://github.com/estebanrobleslunacupDraw

source code and the demonstration videos; especially the one demonstrating the 'Abstraction' feature. Figure 12 shows a complete version of the Editor without pruning any of the menus and toolbars.

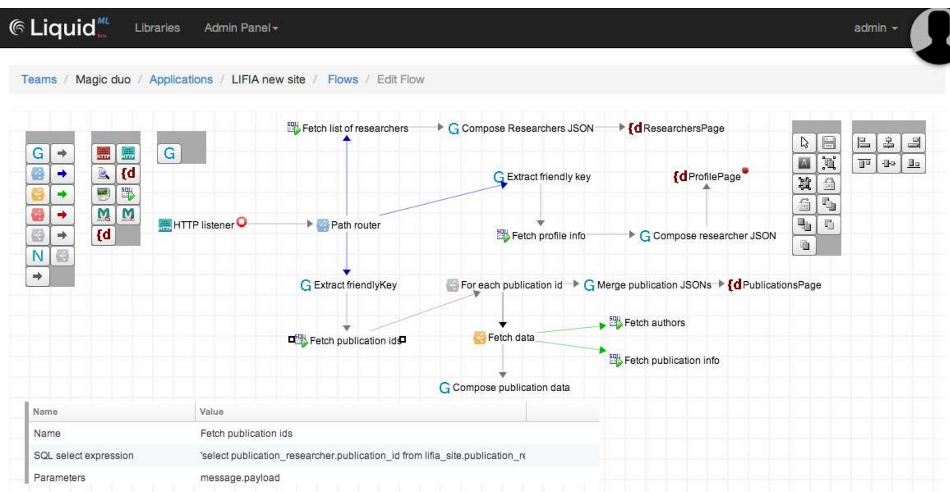


Figure 12. A complete flow editor view

5. Evaluation: Implementing a new abstraction

Analyze the:	introduction of a new element in the language
For the purpose of:	evaluating the LiquidML approach
With respect to:	time, effort and flexibility
From the viewpoint of:	the development team in charge of building a new element in the language
In the context of:	the development of a real-world useful extension for MDWE languages

Table 1. Goal of the proposed experiment

In this section we present an evaluation based on the Weather element introduced in Section 3.1 in order to confirm that our approach works better than existing modelling languages. We have measured time, effort and flexibility of implementing a new abstraction in two of the most modern and popular

Interaction Flow Modeling Languages (IFML and NDT) and we have compared it with LiquidML. We made both an empirical and then a metric-based comparison to assess LiquidML advantages. For the metric-based comparison, We used the Goal-Question-Metric approach [47] to define the goal of the empirical study and to drive the experiment itself. Basically, GQM defines a certain goal, refines this goal into questions, and specifies metrics that should provide the information to answer these questions. A GQM goal template for the evaluation is described in Table 1. In the next subsections we will describe in detail the evaluation study we conducted.

5.1 Definition Stage: Questions and metrics

As required by the GQM method, we defined a set of questions to achieve the proposed goal. With this purpose in mind, the following set of questions was defined:

- Question 1: Does the LiquidML approach allow to be extended with new language concepts faster than IFML and NDT?
- Question 2: Does the LiquidML approach require less effort than IFML and NDT to be extended with new language concepts?

In order to answer these questions, we defined the following metrics:

- Questions 1
  - Metric BT (Building Time): The average time (measured in men-hours) to build the extension to the language.
- Question 2
  - Metric LoC (Lines of Code): The amount of lines of code required to write the parts of the extension that cannot be defined directly through models or configuration
  - Metric AI (Artifacts involved): The amount of artifacts of configuration/code that have to be defined or configured to make the extension work.

5.2 Methodology and Data Collection

Participants. For building the extension, we chose three participants (two from the National University of La Plata, Argentina, and the other one from the University of Seville, Spain) with at least 5 years of expertise in MDWE. One participant worked on the IFML extension building, the other one was in charge of the NDT implementation and the last one LiquidML version. All three participants have not being involved in the development of the approaches and they were trained accordingly. At the same time, authors of this paper (from now on, the Organization Team) tracked the process, answered any doubt related to the requirements that the extension to be implemented must fulfil and also tracked the time spend in that task – to compute the BT metric.

Apparatus. For the IFML case, the WebRatio[e] tool was used. WebRatio is the official tool implemented by IFML authors and contains all the functionalities required to build language extensions out of the box. In the NDT case, the NDT-Suite was used, since this is the official tool for NDT as WebRatio is to IFML. On the other hand, for LiquidML, the web tooling already introduced in this paper was used. The

[e] WebRatio - https://www.webratio.com/

requirements for the extension to be implemented were described textually in a formal document and the Organization Team clarified any further doubt on-site.

Procedure. In both cases, the detailed definition and requirements of the Weather element that has to be added to the language (IFML, NDT or LiquidML, depending on the case) was provided to the corresponding developer/modeller. After the developer/modellers expressed that they understood with clarity the requirements that have to be fulfilled, they started the building task. The Organization Team was present during the entire task to answer any requirements-related question and also to track the time spent in the task using a chronometer. After that, the same team analysed the obtained software artefacts (code files, models, configuration specs, etc.) and computed the metrics introduced in the past subsection to answer the defined questions.

5.3 Implementation

In this section we will describe the underlying details and the steps that have to be accomplished by developers/modelers to build the implementations of the Weather extension in IFML, NDT and LiquidML languages.

5.3.1 LiquidML implementation

The LiquidML implementation is similar to what is has been shown in Section 3.1 and we will avoid duplicating the same content in this section. Instead we will emphasis the process followed by the modeller in order to create the Weather element. The modeller performed the following steps:

1. Read open weather API and obtain API keys.
2. Test the API calls using a browser.
3. Create a HTTP requester in LiquidML editor and test it using the "send message" tool shown in Figure 10.
4. Create a Scripting processor to obtain the city name from the message.
5. Connect the 2 of them in a chain and test it again with the "send message" tool.
6. Abstract the concept in a new element.
7. Test the abstracion in a different flow.

As shown, the full process requires only 7 steps and it was done fast in regards with time consumption as the modeller had experience with the LiquidML tools.

5.3.2 The NDT methodology and implementation

NDT (Navigational Development Techniques) is a Web methodology that offers a complete model-driven support for each phase of the software development lifecycle. Both metamodels of NDT and transformation rules between them have been implemented as the suite named NDT-Suite which is deployed in Enterprise Architect (EA) in order to provide a real environment to apply NDT to a business context. For example, metamodels of NDT are implemented using UML-profile (what is known as NDT-profile) and the transformation rules as the NDT-Driver [12] (which is included in NDT-Suite).

This overall NDT allows a better understanding on how this methodology can implement the proposed example. For this purpose, it is necessary to update NDT-profile in order to take into account

the CityWeatherStatus concept (and its attributes). After manually updating the NDT-profile, EA is already able to interpret these changes in the profile of NDT and any user can use this new concept to design her/his models and information systems. The new primitive can be used to model the CityWeatherStatus as shows Figure 13. Three attributes are included in this new service: city, country and location.

After defining the service model, it is necessary to set the location of the information associated with the service within a user interface prototype. NDT defines Visualization Prototype (PV) models which allow specifying what information will be displayed by the final system and how users can navigate and interact with them.

In our evaluation, it is necessary to show the information returned by the new service (CityWeatherStatus). Therefore, NDT-profile must allow linking service models and PVs. For this purpose, NDT has also been extended to allow this relationship.

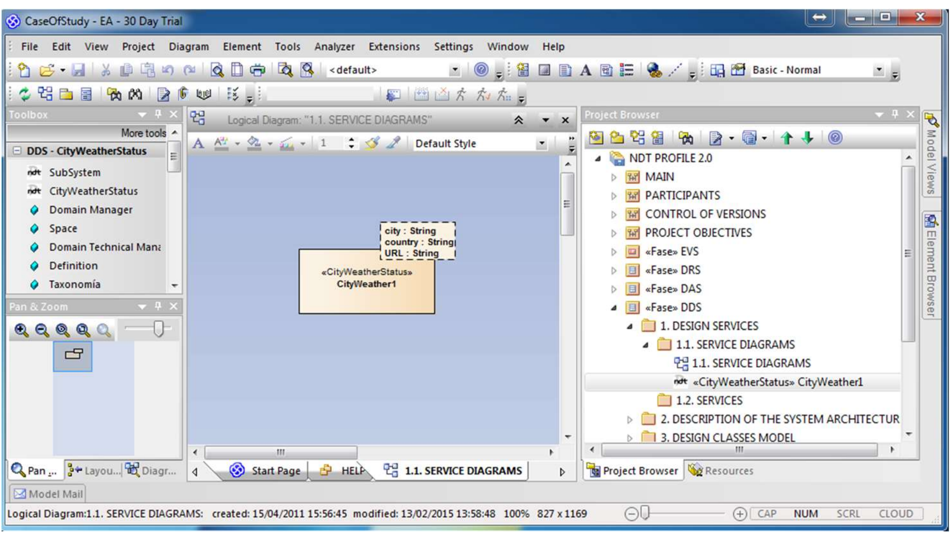
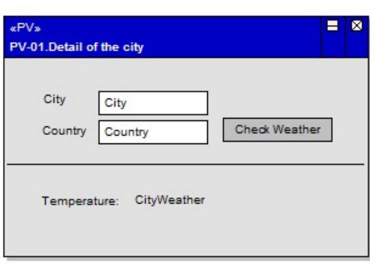
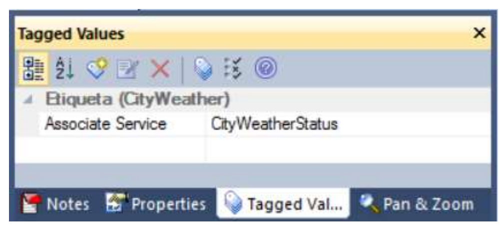


Figure 13. The new primitive service can be instantiated using the toolbox of EA.



(A) Interaction Model



(B) Tagged value of the 'CityWeather' label

Figure 14. Relationships between 'Service' and PV artefacts in NDT.

Figure 14 (A) shows how our CityWeather prototype has been designed using NDT. It comprises four labels and two text fields (which are editable fields to indicate city and country values). However, the most interesting component is the 'CityWeather' label, which is responsible for displaying (in non-editable format) the climate of the given city. This value is returned by the 'CityWeatherStatus' service (see previous figure). This relationship is established using a tagged value in EA for NDT in the 'CityWeather' label (see Figure 14 (B)). In the present case study, this tagged value is called 'Associated Service' and it is automatically generated when a label instance is created since it has been included in NDT-profile.

Once service and interaction models are modeled according to NDT rules, it is possible to generate Java code by means of transformation rules. At the beginning, code generation was not supported by NDT, but we have extended it and we have defined model-to-text transformation rules to generate code from models. In our case of study, NDT implements each service modeled as a REST service that is consumed in our PV using Ajax and the jQuery framework. Figure 15 presents a simplified version of the final code.

```
1.  <html>
2.    <head>
3.      <title>PV-01.Detail of the city</title>
4.      <script type="text/javascript"
5.        src="http://ajax.googleapis.com/ajax/libs/jquery/1/jquery.min.js">
6.      </script>
7.    </head>
8.    <body>
9.      City: <input type="text" name="city" id="city"/>
10.     Country: <input type="text" name="country" id="country"/>
11.     <button type="button" id="WeatherBtn">Check Weather</button>
12.     <div id="result"> Temperature: <span></span></div>
13.     <script type="text/javascript">
14.       jQuery("#WeatherBtn").click(function(){
15.         jQuery.get("http://localhost:8080/CityWeather1/",
16.            { city:jQuery("#city").val() ,
17.             country:jQuery("#country").val() ,
18.            },function(result){
19.                jQuery("#result span").text(result)})})
20.    </script>
21.   </body>
22. </html>
```

Figure 15. Code lines of the implementation using the NDT-Suite tool support.

5.3.3 IFML modelling language and WebRatio implementation

IFML is the new standard adopted by OMG (the Object Management Group) for UI modelling. It was converged from IFML [2] and it currently supports the platform independent description of graphical user interfaces for Web applications. The description is focused on the structure and behavior of the application as perceived by the end user. It is built on a very few, highly composable concepts, which can be used to assemble complex Web applications for publishing or updating content. The key aspect of IFML is the capability of defining a Web Model consisting of pages, content components and operation components, linked to form a specification of the required content publishing or management functions. The IFML specification [17] consists of:

E. The IFML metamodel (MOF).
Ф. The IFML UML profile.
Г. The IFML visual syntax (defined through Diagram Definition and Diagram Interchange Specification).
H. The IFML XMI model exchange format.

WebRatio is a platform that gives support for the core components provided by IFML modeling language. WebRatio includes the notion of custom components to build customized software components for Web applications development. A custom component is a content component or an operation component defined by the developer, and not included in the WebRatio standard installation. It is both similar and different to the standard components:

- Like standard components, a custom component can be placed in a hypertext diagram, linked to other components, with input or output links and the components properties can be defined using the Property View; if the component is a content component, it can also be placed in the grid using the Grid View, and a Component template can be chosen to display its content.
- Unlike a standard component, a custom component has a behavior completely defined by the component designer. Components that performed virtually any tasks, from sending emails to publishing XML documents, in order to interact with Web services, can be implemented.

The overall architecture of WebRatio distinguishes two major elements:

- WebRatio Development Environment, which supports the design-time activities of application development, including the specification of the application site views and the generation of the application code.
- WebRatio Runtime, which enables the execution of Java Web applications generated by WebRatio, by offering a feature-rich execution framework for page templates and business components.

In order to define and execute the weather component both design-time and run-time issues had to be addressed. Firstly, a new Component definition to the component library (mandatory) was added. The Component definition was defined in the unit library (mandatory). For this aim, Figure 16 shows that it is necessary to edit the Unit.xml file using the tool provided editor. It instructs WebRatio to build the proper commands for placing the custom Component in the Web Model. Simultaneously, the custom Component is linked to other Components defining the coupling of input and output parameters.

Secondly, a set of rules has to be added to validate the usage of the custom Component in the Web Model and to generate error and warning reports. This means to create and edit the WebModel.template file, writing a fragment of Groovy code. In this case, it was not necessary to define any kind of errors and warnings.

Then, a set Groovy template files (Input.template, Output.template, Logic.template) has to be added to produce the runtime XML descriptors associated with the custom Component. In this case, three Component templates was added as shown in Figure 17, which indicates how the unit has to be rendered in a page (this feature is mandatory if the custom Component is a content Component). In the input template, the Input value is described as the city name from which the weather information is obtained. In the Output template, we describe the output, which in this case is the temperature of the city. Finally, in the Logic template, we specify the class that implements the actual service.

WeatherComponent

General Properties

General Information

16x16 Icon:

32x32 Icon:

Instance Icon

Name: WeatherComponent

ID Prefix:

unit

Name Prefix:

WeatherComponent

Label:

Weather Component

Flows

Flow Source

Flow Target

Default Intra-Page Flow:

Default Flow To Operation:

OK Flows

OK Flow Source

OK Flow Target

Multiple OK Flows

OK Flow Codes Script:

Type

View Component · Operation · Both

KO Flows

KO Flow Source

KO Flow Target

Multiple KO Flows

General | Sub-Elements | Logic | Layout | Documentation | Version

Figure 16. The Unit.xml file viewed by the provided editor of WebRatio.

Input.template

```
#?delimiters <%,%>,<%=,%>
<%
setXMLOutput()
def unitId = unit.valueOf("@id")
%>

<InputParameters>
    <InputParameter name="<%=unitId%>.cityName" label="City name" />
</InputParameters>
```

Output.template

```
#?delimiters <%,%>,<%=,%>
<%
setXMLOutput()
def unitId = unit.valueOf("@id")
%>

<OutputParameters>
    <OutputParameter name="<%=unitId%>.temperature" label="Temperature" />
</OutputParameters>
```

Logic.template

```
#?delimiters <%,%>,<%=,%>
<%
setXMLOutput()
def unitId = unit["id"]
%>

<Descriptor service="com.webratio.units.custom.weathercomponent.WeatherComponentUnitService">
   <% printRaw(serializeXML(unit, false)) %>
</Descriptor>
```

Figure 17. Groovy template files for associated descriptors.

Finally, Figure 18 shows the implementation class described in the Logic template. This class performs the HTTP request to the open weather API, obtaining the information from the input parameters, making the API calls and adding the output values to the context.

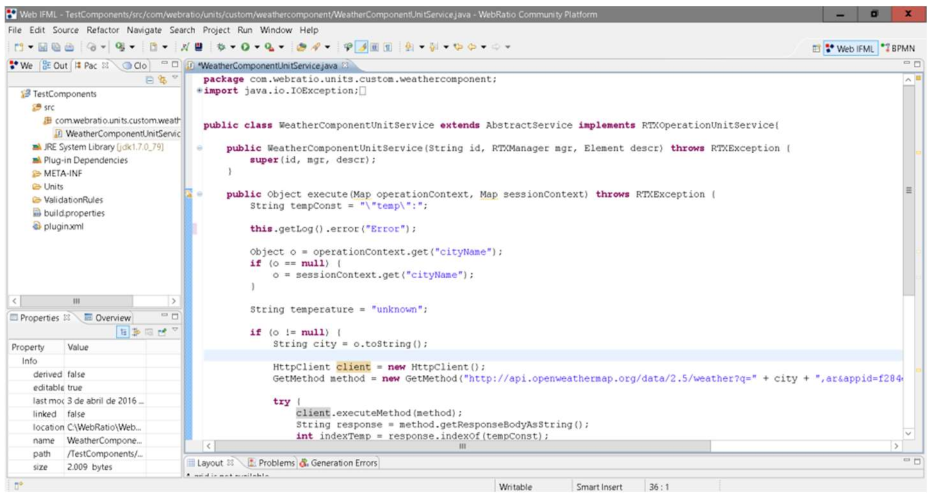


Figure 18. Implementation of the runtime java class for the custom Component.

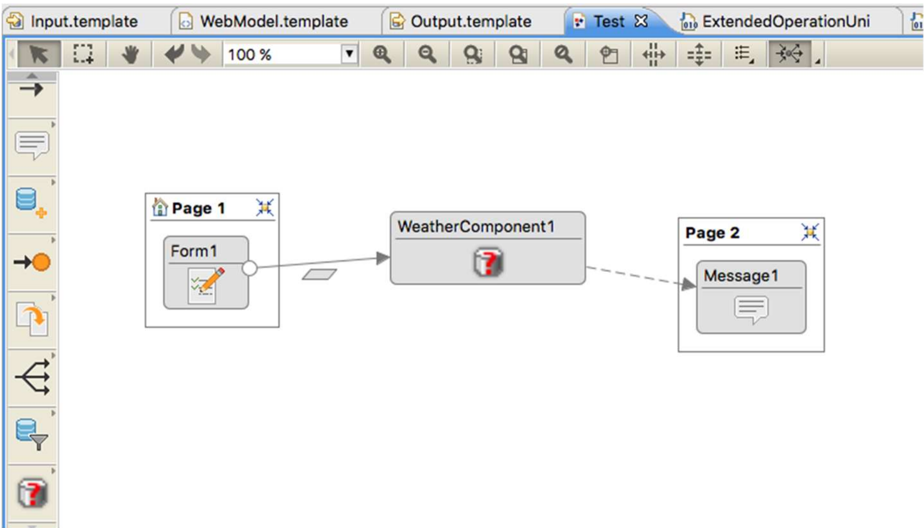


Figure 19. Test for the custom Component in the WebRatio tool support.

In return of these activities, WebRatio lets you use your Components in the Web Model of every Web Project and these Components will cooperate perfectly with the standard Components. In Figure

19, the WeatherComponent is included in the IFML model. The city name is passed as an input parameter from Page1 and then the Weather component computes the temperature and is output to Page2 for rendering.

5.4 Empirical Discussion

The main motivation to develop LiquidML has been the lack of behavioural expressiveness in existing Web modelling languages. As a consequence, transforming the behavioural aspects of a Web application becomes cumbersome and usually extra notations need to be added to (graphically) represent a new expected behaviour. Additionally, the high level (and mostly structural) nature of these languages completely hides the way in which lower level Web requests are processed (e.g. in which sequence) by just ignoring them, thus hindering important spots for introducing performance improvements, for instance, to obtain better application scalability. LiquidML provides a way of representing the behavioural aspects for applications that can be modelled through the message passing paradigm such as Web Applications. Therefore, it enables refining such important and low-level aspects.

As regards the use of LiquidML is concerned, some features related to customizing, developing, debugging and deploying new components were found:

- It offers a good ability to create components from smaller units using abstractions. The environment allows debugging step by step (this feature is allowed for any other application component in LiquidML).
- It is important to note that the weather component code was already working, so, developing and debugging tasks did not take long more than 1 hour. In addition, debugging costs were minimal and they were resolved like any other Java application. In addition, it is possible to inspect objects, their values and evaluate expressions using LiquidML.
- Once the component is abstracted, it is automatically added to the toolbox and it can be used by any other application. The process of abstraction is encapsulated in the modelling environment itself without having to know the underlying issues.
- The components that can be abstracted are those that can be expressed by Groovy code or other components. In case you need a library in particular, it must be requested that it be included within the environment manually. Aspect that takes time and is difficult to estimate its cost.

As regards the use of NDT methodology and NDT-Suite tool support are concerned, some features related to customizing, developing, debugging and deploying new components were found:

- It is necessary to update the NDT-profile (which is defined in Enterprise Architect extending the UML profile) in order to define a new service artefact. This task was really easy to perform with Enterprise Architect and it didn't take so many time (no more than 30 minutes).
- In order to define the view of the component, the NDT-Profile had to be extended to allow relationships between Service artefacts and PV artefacts. This task was also easy to perform with Enterprise Architect and it didn't take so many time (no more than 30 minutes).
- Once modelled service and interaction models according to the NDT rules, it is necessary to generate Java code using transformation rules. In this case, this code generation was not supported by NDT, so, we had to extend and define the model-to-text transformation rules to generate code

from models. In addition, we had to improve the NDT-Driver in order to support this functionality. (it took us 1 hour).
- Finally NDT-Suite doesn't support an environment to debug and deploy the code. So, all these tasks must be done independently.

As regards the use of IFML language modelling and the WebRatio tool support are concerned, some features related to customizing, developing, debugging and deploying new components were found:
- It not seems to be extremely difficult to create a project based on custom Components for any Java developer. Add a library it is equal and easy as in other Java project.
- Every time a custom Component is created, it is placed automatically on the toolkit (it can be dragged and dropped). So, you need to create an Eclipse project in particular and Java classes but It is an environment based on aware components and, in the particular case of the weather Component, the code was already running so just we had to create a component and understand the methods that had to be implemented (it took us 3 hours).
- The process requires creating a new component instead of abstracting from simpler Components.
- The community version of WebRatio does not allow deploying and debugging the application locally, so it made it almost impossible to find the problems. It could be solved by trial and error to make it work. This task took us a long time, about 1 hour. Including that the application had to be deployed in the cloud and it couldn't be done locally.

After testing all supporting tools (LiquidML, IFML and NDT-Suite) and after discussing and analysing differences between them, Table 2 summarize a representing value for activities (customizing, developing, debugging and deploying of new components) based on the time, effort and flexibility that offer each tool support. These values are defined using these notations: "-", "+", and "++" (from a lower to a higher value).

	Customizing	Developing	Debugging	Deploying
LiquidML	++	++	++	++
WebRatio	++	++	+	+
NDT-Suite	++	+	-	-

Table 2. Value for activities using LiquidML, WebRatio and NDT-Suite for extending new components

5.5 Metric-based analysis results

Following the GQM framework, based the data collected during the case study executed we computed values for the metrics defined before in order to answer the questions that help us to evaluate LiquidML. The computation methods and results where the following:

- Metric BT (Building Time): time spent by modelers was measured by observation and using a chronometer by the Organization Team. The results were the following: 4 hours for IFML, 6 hours for NDT, 1 hour for LiquidML. We can observe that LiquidML clearly performed better than IFML and NDT from this point of view.
- Metric LoC (Lines of Code): it was measured by analyzing the models/source code files after the implementation of the new language element was defined. Results were 120 lines of code for IFML, over 250 lines of code for NDT, 20 lines of code for LiquidML. We can assess that LiquidML performed better in the amount of lines of code required to implement the component.
- Metric AI (Artifacts involved): it was measured by analyzing the models and source code stubs, counting how many variables or values have to be set in every approach to correctly get the new language element working within the language/tooling. Results were 1 model, 6 new files, and 120 lines of code for IFML, 2 model, 3 new files, over 250 lines of code for NDT, 1 model and 20 lines of code for LiquidML.

After computing this metrics we can conclude that:

- Question 1 can be answered positively, accordingly to the results obtained from computing the only metric assigned for that question: BT.
- Question 2 can be answered positively too, accordingly to the results obtained from computing the two metrics that were defined for it (LoC and CV)

Since all the questions have been answered positively, according to the GQM scheme we can conclude that the goal has been reached and we can affirm that LiquidML performs better than IFML and NDT regarding time and effort required for extending the language.

5.6 Conclusions and threats to validity

In the evaluation described in this section, despite we were not able to do a detailed statistical tests, empirical and metric-based analysis suggested that LiquidML performed better that IFML and NDT regarding time and effort required to introduce language extensions. Below we enumerate a list of threats to validity that we considered for our evaluation and we also comment what has been done to mitigate them.

- Modeler's unbalance. Having previous experience in developing or working with APIs like the one that was used in the evaluation to build a language extension may result in one modeler (the one with more experience) working faster than the rest. The same applies with modelers with more years of experience in general in comparison with the rest. We mitigated this threat including a set of questions in the interviews to assess that modelers have no experience in the domain and also checked that all the modelers have the same years of experience in the field.
- Extension complexity. The extension chosen to be included in the languages may have been chosen to favor LiquidML in the study. To avoid this bias, we included a very simple and emblematic widget (present in a lot of websites and even in mobile environments), which requires no persistence (to avoid evaluating that aspect of the approaches), defines a basic UI

and interacts with external services (two basic features that are omnipresent in almost all modern web software actually).

6. Related work

In [1] the idea of holding models at runtime to perform runtime changes is presented. The approach is based on the representation of the actual requirements as models while the application is running. The approach was initially applied for autonomous systems where domain experts are not able to access them. The applications built under the models@runtime paradigm belong to a different domain and seems to have less sophisticated business requirements than a Web application. As a result, their metamodels tend to be quite static and do not change that often. In addition, LiquidML uses models@runtime to have a live representation of how the application is constituted. The consequence is that they can be manipulated to be able to abstract new concepts. This allows modelers to come up with quick solutions to problems where the metamodel was not initially intended to be used.

Approaches oriented to allow evolving metamodels and co-evolving their models have been studied recently [6, 3]. These approaches focus on the aspects related to evolving the underlying classes, transformations and tools that were obtained from the metamodel by looking at the changes that the metamodel has suffered. They are feasible solutions for model-based environments where transformations are applied and application downtime and model migration are allowed. This has some similarities and differences, if compared with the approach followed in LiquidML. They are similar to LiquidML in the need of adding some extra primitives that the language does not support. However, in LiquidML anything that could be written with a Scripting processor (> 95% of the cases) can be done without any downtime and the tool evolves automatically within the abstraction process.

In [30] authors present a model-based approach that leverages evolutionary computation to automatically generate, at run time, target system models that balance tradeoffs between functional and non-functional requirements in response to run-time monitoring of environmental conditions. Specifically, that approach generates graph-based representations of architectural models for potential target system configurations. That approach is named Plato-MDE which allows generating target system models at run-time in response to changing requirements and environmental conditions. Each target system model represents a potential system configuration that may be reached through a sequence of reconfiguration steps. Plato-MDE also evaluates each generated target system model to determine its suitability given current system conditions.

In [31] authors propose a solution for automatically evolving the system model by integrating new variants and periodically validating the existing ones based on updated quality parameters. This proposal is based on a BPEL-based framework which uses a service composition model to represent the running system. This system allows leveraging the advantages of service-oriented architectures: flexibility, loose coupling and ease of integration. The framework estimates quality of service (QoS) values based on information provided by a monitoring mechanism, ensuring that changes in QoS are reflected in the system model, called service-level objectives (SLO). Authors also show how the evolved model can be used at runtime to increase the system's autonomic capabilities and delivered QoS. In addition, the framework presented in [31] provides a monitoring mechanism that allows gathering information on the running system and monitoring service execution. Based on monitoring information, the framework detects violations of the system SLOs and invalidates the violating variants. In this way, subsequent SLO

violations are prevented and the system can maintain the required quality. The system model is updated with QoS values that are estimated based on monitoring information; the updated QoS values are used to select the runtime configuration. By using the model at runtime, changes in the model are immediately reflected in the running system.

With regards to approaches related with performance and specifying the behaviour of the application is concerned, the problem has been treated in different works [50]. In Gambi et al. [51], an approach to predict the performances of a Web application is described. The authors explain how the information of MDWE solutions can be used to automatically obtain accurate representations of the running application in terms of layered queue networks (LQNs). Specifically, the paper explains how a MDWE methodology can be exploited to generate such performance models and presents a proof of concept. Compared to other works, the work showed in [51] is quite similar, this approach translates pieces of WebML into LQN sub models and it uses LQN for the exploration of parameters. In [32] authors propose a doctoral thesis which expects to contribute a model-driven, architecture-based approach to developing runtime adaptable and evolvable distributed collaborations, basing on a collaboration abstraction called medium. Although this paper presents a very initial research work can be interesting to know the authors' proposal. They adopt a collaboration abstraction called *medium* that is a collaboration abstraction represented as a software entity. An application is built by interconnecting some functional components with a medium that represents the collaboration of the functional components. The approach is to generalize the refinement process in order to obtain all planned evolutionary architecture variants from a collaboration abstraction, then compose these variants into an adaptable medium that can dynamically select a proper running variant at runtime.

In [33] authors propose the use of high-level abstraction models for the evolution system behaviour in runtime. These behavioural models describe the routine tasks that users want to be automated by the system. This paper proposes the use of models at runtime to evolve the behaviour of Ambient Intelligence (AmI) systems with which environments are electronically enriched to make them sensitive to user needs. One of the most important goals of building such environments is to serve people in their everyday lives and free them to a large extent from tedious routine tasks.

The main problem solved by our approach comes from applying model-driven techniques to industry applications that require time constraints. As a consequence and due to the rigid features of traditional metamodeling approaches (like Eclipse Modeling Framework [29]), we have to discard it as a solution for making an easy to evolve the environment.

7. Conclusion and future work

This paper has presented the extensions to the LiquidML language to support fast metamodel evolution: an increasingly important aspect of modern Web application development. The newly introduced abstraction and template classes support the language's fast metamodel evolution. Using these classes we are able to abstract new language elements from the language primitives. We have shown (Section 5) that the language provides faster evolution than the existing modelling language such as IFML and NDT in a real world evaluation. The LiquidML environment is fully functional, and it is the first one to implement modeling as a service approach making it completely reproducible and available for researchers, engineers and modelers to experiment.

Although the LiquidML language is formally defined, there are several aspects that require further work. The implementation environment still needs improvement concerning its usability such as allowing modelers/designers to change the icons of the new abstractions, their input and output parameters (implicit right now), documentation of new abstracions and the release process of new abstractions to the community. A testing framework similar to JUnit is necessary to provide testing support for flows and abstractions is necessary and needs to be implemented in the near future. In addition to the testing framework, coverage tools: to measure which flows/abstractions are being tested and which are not, is going to be introduced soon. Finally, we plan to include a market for community created abstractions, where people can consume new elements that a different modeler team is using, thus creating a community around LiquidML.

Acknowledgements

This research has been supported by the Pololas project (TIN2016-76956-C3-2-R), by the SoftPLM Network (TIN2015-71938-REDT) of the Spanish Ministry of Economy and Competitiveness and by the VPPI of the University of Seville.

References

1. Blair G., Bencomo N., France R. B., "Models@ run.time," Computer, vol. 42, no. 10, pp. 22-27, October, 2009
2. Ceri S., Fraternali P., and Bongio A., “Web Modeling Language (WebML): a modelling language for designing Web sites,” Comput. Networks, vol. 33, no. 1–6, pp. 137–157, Jun. 2000.
3. Cicchetti A., Di Ruscio D., Eramo R., and Pierantonio A. Automating Co-evolution in Model-Driven Engineering. In Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference (EDOC '08). IEEE Computer Society, Washington, DC, USA, 222-231.
4. Escalona M.J. and Aragon G., “NDT. A Model-Driven Approach for Web Requirements,” IEEE Trans. Softw. Eng., vol. 34, no. 3, pp. 377–390, May 2008.
5. Hohpe G. and Woolf B., Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Professional, 2003, p. 736.
6. Hoisl B., Hidaka S., Hu Z., Towards Co-Evolution in Model-driven Development via Bidirectional Higher-Order Transformation. 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2014), Lisbon, Portugal, January 7-9, 2014.
7. Robles Luna E., Rivero J. M., Urbieta M. M., and Cabot J., “Improving the scalability of Web applications with runtime transformations” in Proceedings of the 14th International Conference in Web Engineering. p 430-439. 2014.
8. Toffetti G.: Web engineering for cloud computing (web engineering forecast: cloudy with a chance of opportunities). In Proceedings of the 12th international conference on Current Trends in Web Engineering (ICWE'12). Springer-Verlag, Berlin, Heidelberg, 5-19 (2012).
9. Wimmer M., Schauerhuber A., and Kargl H., “On the Integration of Web Modeling Languages: Preliminary Results and Future Challenges,” in Proceedings of the 3rd International Workshop on Model-Driven Web Engineering, 2007.
10. Mellor, S.J., Balcer, M.: *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (2002)

11. García-García J.A., Escalona, M.J.; Domínguez-Mayo, F.J.; Salido, A. "NDT-Suite: A metodological tool solution in the Model-Driven Engineering Paradigm". Journal of Software Engineering and Applications, vol. 7. Núm. 4. Pag. 206-217. DOI: 10.4236/jsea.2014.74022. 2014.
12. García-García, J.A., Cutilla, C.R., Escalona, M.J., Alba, M., Torres, J. "NDT-Driver, a Java Tool to Support QVT Transformations for NDT". In the 20th International Conference on Information Systems Development (ISD 2011), Reflections, Challenges and New Directions Pooley, R.J.; Coady, J.; Linger, H.; Barry, C.; Lang, M.; Schneider, C. (Eds.). DOI 10.1007/978-1-4614-4951-5_8, pp. 170-176. , 2012.
13. Salido, A.; García-García J.A.; J. Gutiérrez; J. Ponce. "Tests Management in CALIPSOneo: A MDE Solution", Journal of Software Engineering and Applications, Vol.7 No.6, PP. 506-512. DOI: 10.4236/jsea.2014.76047. 2014.
14. Escalona M.J., Garcia-Garcia J. A., Mas F., Oliva M., Del Valle C. Applying model-driven paradigm: CALIPSOneo experience. Proceedings of the Industrial Track of the Conference on Advanced Information Systems Engineering 2013 (CAiSE'13), vol. 1017, pp. 25-32. 2013.
15. Schmidt, D. C. Model-Driven Engineering. IEEE Computer, Computer Society, vol. 39, no. 2, pp. 25-31, 2006.
16. Pastor, O., España, S., Panach, J. I., & Aquino, N. Model-driven development. Informatik-Spektrum, 31(5), 394-407. 2008.
17. IFML, Interaction Flow Modeling Language. Website: http://www.omg.org/spec/IFML/. Last access may 2016.
18. AngularJS. JavaScript MVW Framework. Website: https://angularjs.org/. Last access may 2016.
19. GWT. Development toolkit for building and optimizing browser-based applications. Website: www.gwtproject.org. Last access may 2016.
20. jQuery. JavaScript Library. Website: https://jquery.com/. Last access may 2016.
21. MongoDB. Website: www.mongodb.com/. Last access may 2016.
22. Tomcat apache. Website: http://tomcat.apache.org/. Last access may 2016.
23. PHP. Website: http://www.php.net/. Last access may 2016.
24. OpenWeather API. Website: http://openweathermap.org/. Last access may 2016.
25. Den Haan J. Model Driven Development: Code Generation or Model Interpretation?. Website: http://www.theenterprisearchitect.eu/archive/2010/06/28/model-driven-development-code-generation-or-model-interpretation. Last access may 2016.
26. Cabot J. Executable models vs code-generation vs model interpretation. Website: http://modeling-languages.com/executable-models-vs-code-generation-vs-model-interpretation-2/. Last access may 2016.
27. Blog webratio. Why Code Generation is better than Model Interpretation (from our customers' point of view). Website: http://blog.webratio.com/?p=368. Last access may 2016.
28. HL7. Introduction to HL7 Standards. Website: www.hl7.org/implement/standards. Last access may 2016.
29. Eclipse Modeling Framework. Website: http://www.eclipse.org/modeling/emf/. Last access may 2016.
30. Ramírez, Andres J.; CHENG, Betty HC. Evolving models at run time to address functional and non-functional adaptation requirements. En Proceedings of the 4th International Workshop on Models at Runtime. 2009.
31. Mosincat A, Binder W, Jazayeri M. Achieving runtime adaptability through automated model evolution and variant selection, Enterprise Information Systems, 8:1, 67-83, DOI: 10.1080/17517575.2012.691182. 2014.

32. Phung-Khac, An, et al. A Model-driven Architecture-based Approach to Runtime Adaptable and Evolvable Distributed Collaborations. MODELS Doctoral Symposium. 2008.
33. Serral, E; Valderas, P; Pelechano, V. Addressing the evolution of automated user behaviour patterns by runtime model interpretation. Software & Systems Modeling, vol. 14, no 4, p. 1387-1420. 2015.
34. New Relic. Website: www.newrelic.com. Last access may 2016.
35. Huo, Ming, et al. Software quality and agile methods. En Computer Software and Applications Conference, COMPSAC 2004. Proceedings of the 28th Annual International. IEEE, pp. 520-525. 2004.
36. Glossary. What Is A/B Testing? Website: https://www.optimizely.com/ab-testing/. Last access may 2016.
37. Bloch, Joshua. Effective java (the java series). Prentice Hall PTR, 2008.
38. PLATT, David S. Introducing Microsoft. Net. Microsoft press, 2002.
39. Learn Ruby the Hard Way, 3rd Edition. Website: http://learnrubythehardway.org/book/. Last access may 2016.
40. Scala, Object-Oriented Meets Functional. Website: Http://www.scala-lang.org/. Last access may 2016.
41. HUMBLE, Jez; FARLEY, David. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader). Pearson Education, 2010.
42. Koch, N., Knapp, A., Zhang, G., Baumeister, H.: UML-based Web Engineering: an approach based on standards. In: Rossi, G., Pastor, O., Schwabe, D., Olsina, L. (eds.) WE, pp. 157–191. Springer. 2008.
43. Rossi, G., Schwabe, D.: Modeling and implementing web applicactions with OOHDM. In: Rossi, G., Pastor, O., Schwabe, D., Olsina, L. (eds.) WE, pp. 109–155. Springer. 2008.
44. ACERBIS, Roberto, et al. Model-Driven Development of Cross-Platform Mobile Applications with Web Ratio and IFML. En Mobile Software Engineering and Systems (MOBILESoft), 2015 2nd ACM International Conference on. IEEE, 2015. p. 170-171. 2015.
45. Martínez Y, Cachero C, Meliá S: MDD vs. traditional software development: A practitioner's subjective perspective. Information & Software Technology 55(2). pp 189-200. 2013.
46. Papatheocharous E, Andreou A. Empirical evidence and state of practice of software agile teams. Journal of Software: Evolution and Process. Volume 26, Issue 9, pages 855–866, 2014.
47. V. Basili, G. Caldiera, D. Rombach, The Goal Question Metric approach, 1994.
48. Robles Luna E., Rivero J.M., Urbieta M. “LiquidML: A Model Based Environment for Developing High Scalable Web Applications” in Proceedings of the 14th International Conference in Web Engineering. p 519-522. 2014.
49. Robles Luna E., García-García J.A., Rossi G., Rivero J.M., Domínguez Mayo F.J., Escalona M. J. “LiquidML: A Web Modeling Language Supporting Fast Metamodel Evolution”. in Proceedings of WEBIST (1) 2016: 318-326.
50. Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: A survey. IEEE Trans. Softw. Eng. 30(5), 295–310 (2004).
51. Gambi A., Toffetti G., Comai S. Model-Driven Web Engineering Performance Prediction with Layered Queue Networks. In: Daniel F., Facca F.M. (eds) Current Trends in Web Engineering. ICWE 2010. Lecture Notes in Computer Science, vol 6385. Springer, Berlin, Heidelberg. 2010. DOI: 10.1007/978-3-642-16985-4_3