LiquidML: A Web Modeling Language Supporting Fast Metamodel Evolution

Esteban Robles Luna[1], Julián A. García-García[3], Gustavo Rossi[1, 2], José Matías Rivero[1, 2,]
Francisco Domínguez Mayo[3] and María José Escalona[3]

[1]*LIFIA, Facultad de Informática, UNLP, La Plata, Argentina*
[2]*Conicet, Caba, Argentina*
[3]*Web Engineering and Early Testing Group, University of Seville, Seville, Spain*

Keywords: Evolution, Model-based Development, Self-reflective, Web Development.

Abstract: Model Driven development approaches are being adopted by the software industry due to a core set of benefits such as raising the level of abstraction and reducing coding errors. However, their underlying modelling languages tend to be quite rigid, making their evolution hard, specifically when the corresponding metamodel do not support primitives and/or functionalities required in specific business domains. In this work, we present an approach for fast evolution of the modelling language that is "self reflective", allowing modellers to abstract new language concepts from the primitives. The main advantage of our approach is that it provides zero application downtime and automatic tool evolution. As a consequence, applications created with our approach are able to adapt quicker to the business needs than those based on traditional Web modelling languages. We compare our approach with existing modelling languages in a case study providing a proof of its ability to adapt faster than to new business needs.

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 GWT (GWT, 2016), JQuery (JQuery, 2016) to non-relational databases such as MongoDB including tools that help you monitor the running application (to keep the application running 24x7) such as New relic (New Relic, 2016). 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 (Ceri et al, 2000; Escalona et al, 2008; García-García et al, 2014) have been developed in the Academia and have real world application (Escalona et al, 2013). These Web DSLs are generally based on MDE (Model-Driven Engineering) (Schmidt, 2006) or MDD (Model-Driven Development) (Pastor et al, 2008) and thus 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. Through constant communication between stakeholders and software reusability and adaptability to change, these practices have reduced software development costs. The necessity for quick changes has surged due to the huge number of applications that makes harder to acquire and withhold users active in the Web site. As a consequence, the ability to make small but effective changes in a matter of 1 to 3 days became increasingly important and affects the decision of which T&Ls to be used. A clear example of this issue is the introduction of A/B testing techniques to help with the analysis of which design version of a new feature will be implemented in full. 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 the users more active or improves user retention) a simpler decision. Then the design must be implemented in full within the next couple of days.

In this context, a common problem that these DSL share is their inability to evolve fast as the underlying business requirements change, e.g. introducing new modelling primitives for accelerating the implementation of a project with tight deadlines when fine-grain business-related requirements arise. In those cases, where the current metamodel does not support the desired functionality, a set of activities needs to be performed in the development environment to adapt to it, including: Extend the metamodel with new concepts; Extend the transformation engine to derive the desired code; Perform a full round of tests of the new functionality; Adapt the tooling to add the new elements in the user interface; Deploy the new version into modelers' machines.

Depending on the metamodel change, these activities can take an amount of time that a project might not tolerate. A workaround to this problem can be achieved if the modeling environment supports hooks where modelers can introduce pieces of programming code that are taken into account during code derivation (Ceri et al, 2000). However, if that workaround needs to be applied to multiple model elements, it can be time consuming and error prone – implying also a violation of model's abstraction, which is one of the primary advantages of using a Model-Driven process.

Our approach overcomes this problem by allowing modelers to "abstract" concepts from existing models following a modeling by example approach. That is, when an aspect of the application can be captured in a reusable concept, the implementation environment allows modelers to select the elements from the models and abstract a new concept that gets automatically integrated in the language. Additionally, the approach takes care of the whole development lifecycle by including the new concept in the derived application and thus minifying the hassle of evolving the metamodel.

In this paper we present the runtime environment and the implementation of our model-based approach that minifies the difficulties of metamodel evolution. The theoretical foundation of our proposal is described in (Robles et al, 2014) and it is named LiquidML. LiquidML allows building applications that can be modeled using the message-passing paradigm (Hohpe et al, 2003). We present the approach using a simple Web application where a weather component needs to be introduced into the language primitives.

Rest of the paper is structured as follows: in Section 2 we present the LiquidML environment, describing its primitives and its ability to abstract new concepts. We provide details about how these aspects are translated into a production environment by presenting the LiquidML's runtime environment in Section 3. We introduce part of the implementation in Section 4. Finally in Section 5 we describe some related work and in Section 6 we present some conclusions and future work.

2 LiquidML

In the following subsections we introduce the LiquidML modeling environment.

2.1 Overview

LiquidML (Robles et al, 2014) is a modeling language that allows modelers to create applications based on the message passing paradigm (Hohpe et al, 2003). 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 will emphasize the relationship between LiquidML, other modeling languages such as WebML and NDT (Escalona et al, 2008; García-García et al, 2014) and the actual Web Application.

Applications models built with "conventional" approaches (e.g. WebML) are transformed into code that is run inside a Web container such as Apache Tomcat or a PHP server. The basic primitives in these languages abstract high level entities such as Web page, domain objects and usual behaviors such

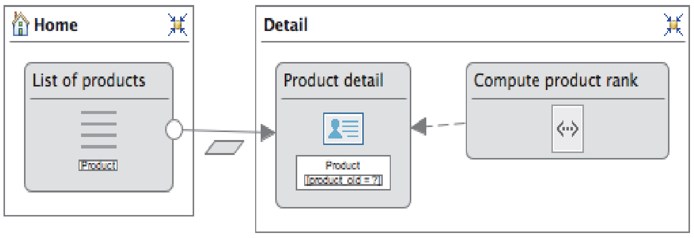


Figure 1: WebRatio model for an E-Commerce application.

as navigation. As an example, consider the WebML model of Fig. 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 allows to see 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".

As will be shown in the following subsections, LiquidML has a finer grained metamodel that focuses on the basic concepts of model passing languages such as message, message source, processor and router (Hohpe et al, 2003). Thus from a level of abstraction point of view, LiquidML seems to be closer to the level of abstraction of code that to the elements in WebML. Also, the focus of LiquidML is completely behavioral, whereas WebML is structural; for instance, the arrows between the elements do not imply a sequence of evaluation but connections such as hyperlinks.

From a development process perspective, LiquidML models can be either derived semi-automatically from high-level models such as WebML and NDT or they can be created manually using a LiquidML editor (Robles et al, 2014). When manipulated, some higher-level concepts can be abstracted as modelers discover them and thus the development metamodel gets enriched interactively within the process (Sect. 2.4). Similar to what happens in the lifecycle when using "traditional" model-driven approaches, a Build/Snapshot is created and then it gets deployed into a server that is capable of running/interpreting the Web Application.

2.2 Basic Concepts

Our main motivation to develop LiquidML has been the lack of behavioral expressiveness in existing Web modeling languages. As a consequence, transforming the behavioral aspects of a Web application (e.g. in WebML) becomes cumbersome and usually extra notations need to be added to (graphically) represent a new expected behavior. 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 behavioral aspects for applications that can be modeled using the message passing paradigm such as Web Applications and thus enables to refine such important and low-level aspects.

A LiquidML model is a composition of Flows; each Flow describes a sequence of steps that need to be applied to the current Web request (called a Message in LiquidML) to obtain a proper response. 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 categorized the Elements using the categories found in (Hohpe et al, 2003) and every element has a different icon that represents it:

Message Source

A message source 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 for incoming requests and generating 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 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.

Processor

A processor may transform, execute or just read information from a message by changing or reading the message's payload and properties. There is a wide variety of processors though the most common one is the *ScriptingProcessor* that allows modeller 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 and 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 modeller.

- *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 transforms the message payload into a XML document using an XSLT definition.
- *Mapping transformer*: it transforms the message payload into a Map/Dictionary using the configured keys and expressions.

Router

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

- *Choice router*: 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 to true is the one that gets activated: the element that the connection reaches is the next one to be evaluated.
- *All router*: it creates a copy of the current message sending it to all the "all connections" evaluating them in parallel and collecting the results.
- *Wiretap router*: it creates a copy of the message and sends it in async way to the wire tap connection. It is an implementation of the EIP (Enterprise Integration Patterns).
- *Chain router*: The chain router evaluates each of the chain connections in sequence. After evaluating the 1st chain connection, the result is passed to the 2nd connection and so on until all the chain connections are exhausted.

Connections

It describes a relationship between 2 elements. The most common connection is the "Next in chain" which specifies that after the "source" element processes the message, then 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 have only 1 next in chain connection while the target element may have multiple.
- *Choice connection*: Choice connections have a configurable condition and the source element can only be a choice router though the target element can be any kind of element. The choice router may have multiple choice connections.
- *All connection*: Similar to choice connections, all connections only apply to source all routers though the target element can be of any kind. The all router may have multiple all connections.
- *Wiretap connection*: It only applies to wiretap routers and a wiretap router can only have 1 wire tap connection that is connected to any kind of element.
- *Chain connection*: Similar to choice connections, chain connections only apply to source chain routers though the target element can be of any kind. The chain router may have multiple chain connections.

To exemplify the concepts, we present a Flow for the product's detail page of an E-commerce web application (Fig. 2). The *Element* with no incoming arrow 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 "Route path" is a *ChoiceRouter*, which behaves like a choice/switch statement and it will route the message to the "Get info" processor if the request comes to a URL starting with "/product/*". The "Get info" is another router that gets information in parallel from multiple sources. It

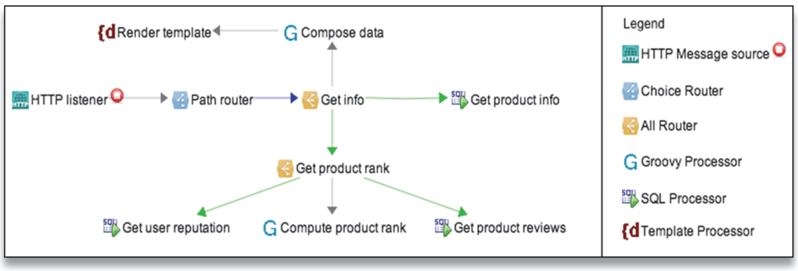


Figure 2: Product details flow.

obtains the product info from the DB: ("Get product info") and triggers the computation of the product's rank, which involves two database queries ("Get user reputation" and "Get product reviews") and a Processor that computes the rank from this information ("Compute product rank"). Finally, the information gets composed ("Compose data") and used for rendering a Web page in the "Render template" processor.

In the following subsection we present the actual metaclasses, templates and the Abstraction class in LiquidML. The components in the models described in Fig. 2 show how a traditional Web page can be splitted in atomic steps, which allows to model and optimize fine-grain aspects of the application. The cost of having such low-level modelling language (in comparison to classic model-based Web development approaches) is compensated with the possibility of easily grouping its components in more abstract elements at modelling time.

2.3 Abstracting Concepts from Examples

When developing applications with models, sometimes a desire business feature requires having a flexible general model that is not always available. For instance, in the E-commerce application, we may want 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 LiquidML's language and as a consequence a workaround is needed to implement it. Therefore the modelers are able to come with a solution that integrates a sequence of processors to perform the external API calls to, for instance, the OpenWeather API (Open weather map, 2006). Basically, by making an IP to City mapping and then looking up for that city in the API we are able to get the information needed for our recommendation systems.

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

Following the same approach that code based environments provide, in LiquidML we have the ability to encapsulate and abstract a new concept from a subgraph. So, modelers are able to select the elements to be abstracted, click 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 shown in the next section

2.4 Models and Templates

LiquidML allows abstracting new metamodel concepts from LiquidML application models. For achieving this, our approach uses a set of template classes that configure the parts of the abstraction (Fig. 3). A special metaclass is required in the metamodel to capture the new abstracted concepts; we call this class Abstraction. A new instance of the Abstraction class is created when a concept is captured (Section 2.3). The instance of the Abstraction class knows how to instantiate its internal pieces since it reference the list of templates to configure each of them (Fig. 3).

The primitives of the language are instantiated by looking into the metamodel implementation and its configuration. To achieve the instantiation, we use a set of template classes that may have a one to one correspondence with the meta classes, however the instances of the template classes are one per model instance while the meta classes are one per concept. Templates are either Simple or Composite; simple templates have a list of properties to configure the instance and composite elements know how they are composed.

2.5 Evolution Process

Once the Abstraction instance is created a set of activities is performed automatically to evolve the development environment and the application under development.

The 1st step is to block the user interaction with the flow under development and checks that the elements selected formed 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.

Once the validation is performed, the abstraction and templates instances are created. The abstraction is set a default icon and the modeler can input a name for the abstraction.

The 3rd step is to remove 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.

3 RUNTIME ENVIRONMENT

As aforementioned, flows define the behavioral part of the Web application. On the contrary to all MDWE approaches, we decided to interpret rather than to derive the code of a Web application. Strong cons and pros of both approaches can be found in (Mellor et al, 2002) and in many informal discussions (The Enterprise Architect, 2016; Executable models, 2016; Webratio, 2016); however, 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. It is true that at a first sight, code-generation seems to be the right option, however interpretation gives us an opportunity to easily modify the behavior in a dynamic way.

As our behavioral models (Flows) are rather simple, the interpreter algorithm is quite simple too. We present a simplified version of the algorithm using a Java-based pseudocode in the next lines: the interpreter works when messages are received (event driven (Hohpe et al, 2003)) on Message sources (e.g. an HTTP message source) (line 1). It finds the next element (*currentElement*) that will handle the message (line 3 and 6) and evaluates it using the message content (line 4). An evaluation returns a Message instance that could be the same as the previous one or a new one depending on the *Element* intent (data transformation, routing, etc.) and it is passed to the next Element until we run out of *Elements* (line 3).

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

4.    while (currentElement != null) {
5.    interpreter.evaluate(currentElement,
                                  message);
6.    currentElement =
      interpreter.getNextInChain(currentElement);}}
```

A special case is handled by the interpreter (line 5) when *currentElement* is an Abstraction. In that case, we follow the same approach as any other programming language behaves by using a stack as Abstraction is basically composed of more primitive elements that may include other processors or Abstractions. So processing an *Abstraction* is processing its internal subflow starting from the initial element.

Interpretation happens while engineers are building the application and when the application is run in every other deployment environment (QA, Staging, Production). Once the models satisfy the requirements, the deployment process to a specific

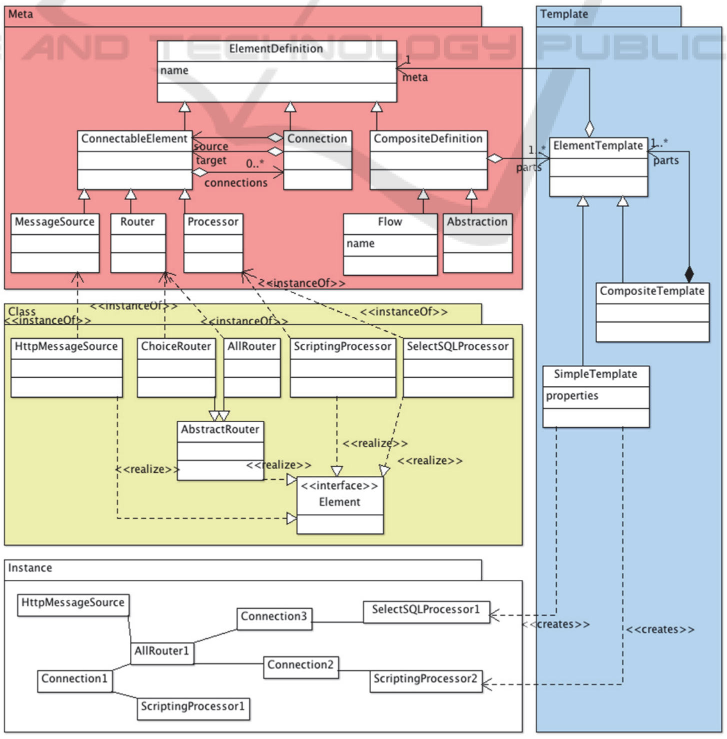
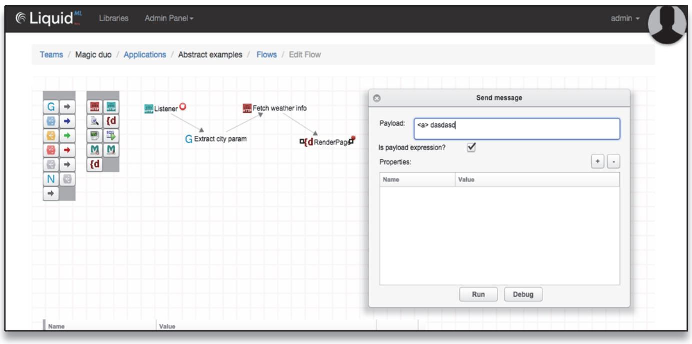


Figure 3: LiquidML models, templates and instances.



Payload: <a> dasdasd

Is payload expression? ✓

Figure 4: Sending messages to specific model elements.

environment occurs. The deployment is an automatic process where a copy of the models is moved 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 diagnosing problems while our models are running.

4 IMPLEMENTATION

The aforementioned concepts have been implemented in an environment that allows modellers to create applications based on the message-passing paradigm. The environment is completely Web based and to the best of our knowledge, the first one to additionally implement the Modelling as a Service paradigm (Toffetti 2012).

One clear advantage of our implementation is that allows modellers to debug the application under development at the model level. In Fig 4 we show how modelers are allowed to send messages to specific model elements.

The LiquidML environment is composed of 2 main applications that can be instantiated multiple times and run in a cluster: an editor and servers. The editor is instantiated in multiple machines running behind a load balancer in Amazon EC2 though it can be installed in any local servers. A brief description of each application is the following: ***LiquidML editor***, which allows defining the applications, modelling Flows and deploying them to LiquidML servers and It also, allows modelers to abstract new concepts from existing models; and ***LiquidML server***, which is responsible for holding the application definitions, the LiquidML interpreter (Robles et al, 2014) 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.

Both the editor and the server have been built using open source technologies of the J2EE stack. We have used 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 built the CupDraw framework 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 source code and the demonstration videos; especially the one demonstrating the "abstraction" feature. A complete version of the Editor without pruning any of the menus and toolbars can be seen in the LiquidML site.

5 RELATED WORK

In (Blair et al, 2009) the idea of holding models at runtime to perform runtime changes is presented. The approach focuses in the representation of the actual requirements as models while the application is running. The applications built under the models@runtime paradigm are from a different domain and seems to have less sophisticated business requirements than a Web application. On the other hand, LiquidML uses models@runtime to have a live representation of how the application is constituted and as a consequence it can be manipulated to be able to abstract new concepts.

Approaches oriented to allow evolving metamodels and co-evolve its models have been studied recently (Cicchetti et al, 2008, Hoisl et al, 2014). They are similar to LiquidML in the need of adding some extra primitives that are not supported by the language. However, in LiquidML anything that could be written with a Scripting processor (> 95% of the cases) can be done with no downtime and the tool is evolved automatically within the abstraction process.

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 (EMF, 2016)), we have to discard it as a solution for making an easy to evolve environment.

6 CONCLUSIONS

In this paper, we have presented the LiquidML, a Web modelling language that supports the fast evolution of its metamodel and supporting tools. By capturing the abstraction concept, which references the templates to configure its parts, we are able to build concepts from existing model solutions following a modelling by example approach which mimics with the approach use to build frameworks from existing pieces of code. The environment is fully functional, and is the first one to implement the modelling as a service approach, so it is fully reproducible and available for researchers, engineers and modellers to experiment.

The LiquidML language is formally defined and we do not expect to see much changes in that regards, however the implementation environment still needs some improvement regarding its usability such as allowing modellers to change icons, input and output parameters (implicit right now), documentation and the release process of new abstractions to the community. From a conceptual point of view, models do not provide a way of being tested, so we plan to formalize a testing framework that will allow modellers to test flows and provide tools such as flow coverage which will give confidence to modellers when releasing a new version of the application. Finally, we plan to include a “concepts” market where people can consume concepts that a different modeller team are using and thus creating a community around LiquidML.

ACKNOWLEDGEMENTS

This research has been supported by MeGUS project (TIN2013-46928-C3-3-R) of the Ministerio de Ciencia e Innovación (Ministry of Science and Innovation, Spain, TIN2013-46928-C3-3-R).

REFERENCES

Blair G., Bencomo N., France R. B., "Models@ run.time," Computer, vol. 42, no. 10, pp. 22-27, October, 2009.

Ceri S, Fraternali P, Bongio *A. Web Modeling Language (WebML): a modeling language for designing* Web sites. Comput. Networks, vol.33, pp.137–157, 2000.

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.* IEEE Computer Society, Washington, DC, USA, 222-231.

EMF. Website: http://www.eclipse.org/modeling/emf. Last access: 2016.

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.

Escalona MJ, Garcia-Garcia JA, Mas F, Oliva M, Valle C. Applying model-driven paradigm: CALIPSOneo experience. *Conference on Advanced Information Systems Engineering 2013, vol.1017*, pp. 25-32. 2013.

Executable models vs code-generation vs model interpretation. Website: modeling-languages.com/ executable-models-vs-code-generation-vs-model- inter pretation-2/. Last access: 2016.

García-García J.A., MJ Escalona, F Domínguez-Mayo, A. Salido. “*NDT-Suite: A metodological tool solution in the Model-Driven Engineering Paradigm*”. DOI: 10.4236/jsea.2014.74022. 2014.

GWT. *Development toolkit*. Website: www.gwtproject. org. Last access: 2016.

Hohpe G. and Woolf B., Enterprise Integration Patterns: *Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley Professional, 2003, p. 736.

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.

Mellor SJ, Balcer M. Executable UML: A Foundation for Model-Driven Architectures. *Addison-Wesley Longman Publishing, Inc.,* Boston, MA, USA. 2002.

New Relic. Website: www.newrelic.com. Last access 2016.

Open weather map. Website: openweathermap.org. Last access: 2016.

Pastor O, España S, Panach JI, Aquino, *N. Model-driven development. Informatik-Spektrum*, 31(5), 394-407. 2008.

The Enterprise Architect. Website: http://www. theenterprisearchitect.eu/archive/ 2010/ 06/ 28/ model-driven-development-code- generation- or- model- inter pretation. *Last access*: 2016.

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*. Springer-Verlag, Berlin, Heidelberg, 5-19 2012.

Robles E, Rivero JM, Urbieta M, Cabot J. Improving the scalability of Web applications with runtime transformations" *in Proceedings of the 14th International Conference in Web Engineering*. 2014.

Wimmer M, Schauerhuber A, Kargl H. On the Integration of Web Modeling Languages: Preliminary Results and Future Challenges. *Proceedings of the 3rd International Workshop on Model-Driven Web Engineering*, 2007.

Webratio. Website: blog.webratio.com. Last access: 2016.

JQuery. Website: https://jquery.com/. *Last access*: 2016.

Schmidt DC. Model-Driven Engineering. *IEEE Computer, Computer Society, vol. 39, no. 2*, pp. 25-31, 2006.