A Domain-Specific Language for Modeling Web User Interactions with a Model Driven Approach

Carlos Eugênio Palma da Purificação / Paulo Caetano da Silva
Salvador University (UNIFACS)
Salvador, Brazil
email: carloseugenio@gmail.com / paulo.caetano@pro.unifacs.br

***Abstract*-Domain Specific Languages have many applications. They provide a way to abstract domain concepts and express these concepts in a more expressive way when compared with general-purpose languages. Recently the Object Management Group has released the beta version of its Interaction Flow Modeling Language standard to model user interaction in applications. We contribute with a proposal of a textual domain-specific language, to use this model as a base for modeling user interaction in web applications, along with Model Driven Software Development techniques and a set of tools and components, to propose a generative approach to model user interaction in web applications. The approach allows the definition of user interaction flow models using a textual form, architecture reuse, improved code quality and speed in development.**

Keywords-Model Driven Software Development; Domain Specific Languages; Web Applications.

I. INTRODUCTION

Researches in software reuse show that in order to achieve significant results in this field, a paradigm shift towards the use of software families rather than individual systems is required [18]. In order to achieve this goal, many technics have been proposed. In this work, we propose a tool-set for modeling user interactions in applications using a textual Domain-Specific Language (DSL), called EngenDSL [12], modified for accommodate most *Interaction Flow Modeling Language* (IFML) [11] concepts. The language is used along with a set of tools and libraries that realize the Model-Driven Software Development (MDSD) [10] generative software approach to create applications, to generate a small application as a proof of concept of the methodology.

We have used the IFML standard [11] to base our user interaction model, adding some important features such as the possibility to define and reuse the layout and style for views, specify presentation libraries, attaching behavior semantics and chaining to user actions. Whilst previous researches in this area [5] provide some kind of support to user interaction modeling, until recently, there was not a formal specification for modeling user interactions such as the OMG IFML Standard.

The rest of this paper is structured as follows. In the next section, concepts and background information related to main topics of this paper are presented. Section 3 describes the EngenDSL language meta-model and constructs. Section 4 discusses the model transformation problem. Section 5 provides an example application that shows language constructs and some development tools used in the approach. Section 6 presents some related work. Section 7 provides some conclusion and planning for future work.

II. CONCEPTS AND BACKGROUND

The MDD – *Model-Driven Development* is a technique that gives software models a high importance role in software development process.

Following the concept of prioritization of models and their use as key artifacts during all phases of software development (specification, analysis, design, implementation and testing), is the Model-Driven Software Development (MDSD) [10]. This concept uses an agile approach to software development, along with generative techniques to deliver software based on a software product lines – SPL approach.

As in all areas of science and engineering, there are always specific and generic approaches to solve a given problem [3]. The description of a problem in a language developed specifically for its domain tends to be a more optimized and direct solution, possibly caring greater expressiveness to describe the domain concepts. Domain-specific Languages are languages created for a particular domain. They are also called specialized languages, problem-oriented or special purpose languages [9]. Deursen et al [3], defines DSL as programming languages or executable specification languages that offer great power to express, with notations and abstractions usually focused on a restricted domain, a particular problem.

IFML [11] is a modeling standard developed by OMG designed for expressing the content, user interaction and control behavior of software applications front-end. It allows software practitioners to model and describe the main parts of an application front-end. Therefore, the specification divides these parts in the following dimensions: (i) **View** – the view part of the application composed of containers and view components; (ii) **State and Business Logic** – the representation of objects that carries application state and the business logic that can be executed; (iii) **Data and Event Binding** – the binding of the view components to data objects and events; (iv) **Event Control Logic** – the logic responsible to determining the action execution order after in response to an event; (v) **Architecture Distribution** – the distribution of control, data and business logic at the application tiers.

 ISBN: 978-1-61208-412-1

The general IFML meta-model is presented in [11]. The *IFMLModel* is the core container of IFML meta-model elements and contains a *InteractionFlowModel* which is the user view of an IFML application. It is composed of *InteractionFlowModelElements* – an abstract class that represents a generalization of every element of a *InteractionFlowModel* (the view elements); *DomainModel* – model elements that represents the business domain or data; *ViewPoints* – represents only specific aspects of the system to facilitate comprehension of referencing *InteractionFlowModelElements*.

The *DomainModel* represents the business domain of the application, the content and the behavior that is referenced by the *InteractionFlowModel*. The *DomainModel* contain *DomainElements*. These are specialized in terms of concepts (*DomainConcept*), properties (*FeatureConcept*), behaviors (*BehaviorConcept*) and methods (*BehaviorFeatureConcept*).

III. THE ENGENDSL

The EngenDSL extension was modeled to aid in developing applications by abstracting user interaction concepts based on IFML standard. The main objective was to define a textual DSL, which could be used to streamline user interaction modeling. The foundation for this was the IFML standard and in Figures 1, 2 and 3, we present the main concepts and extensions provided by EngenDSL to the IFML standard.

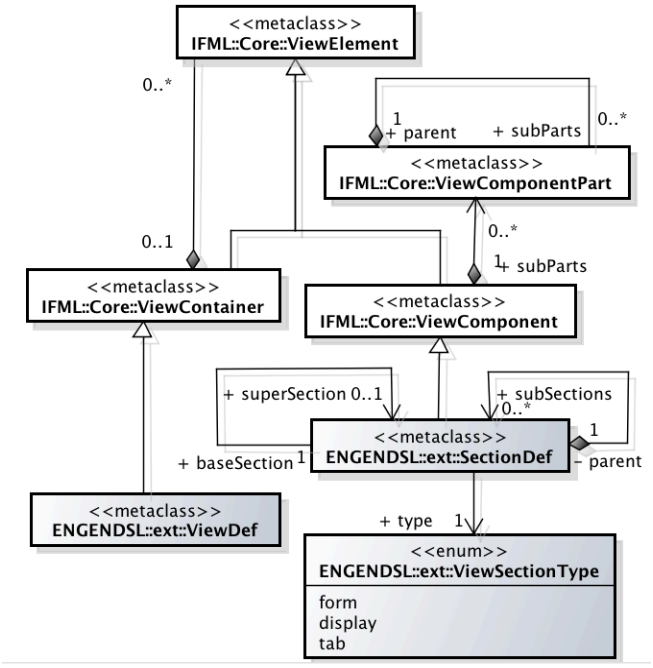

Figure 1. The EngenDSL IFML Extension Package for View Elements

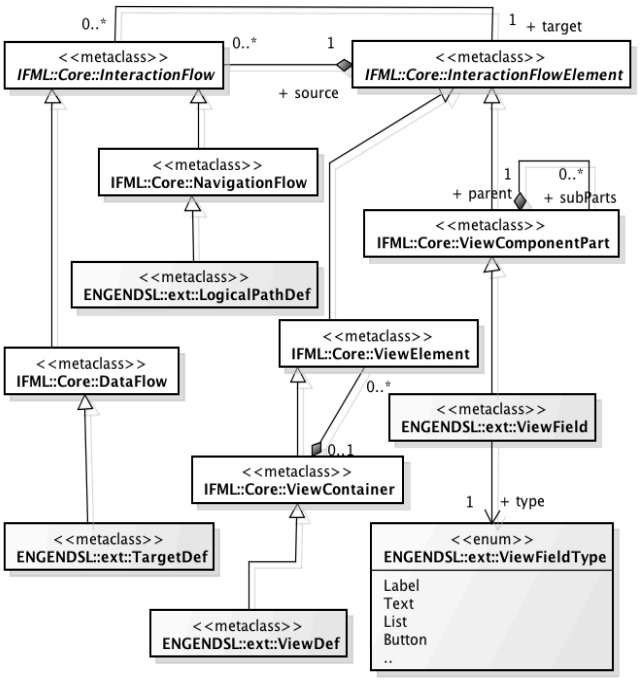

Figure 2. The EngenDSL IFML Extension package for Interaction Flow Elements

Therefore, this EngenDSL language extension was designed to comply with the IFML standard as much as possible, while giving the possibility to define other architectural aspects not covered by the standard. Figure 4 summarizes the mapping between IFML and EngenDSL main elements.

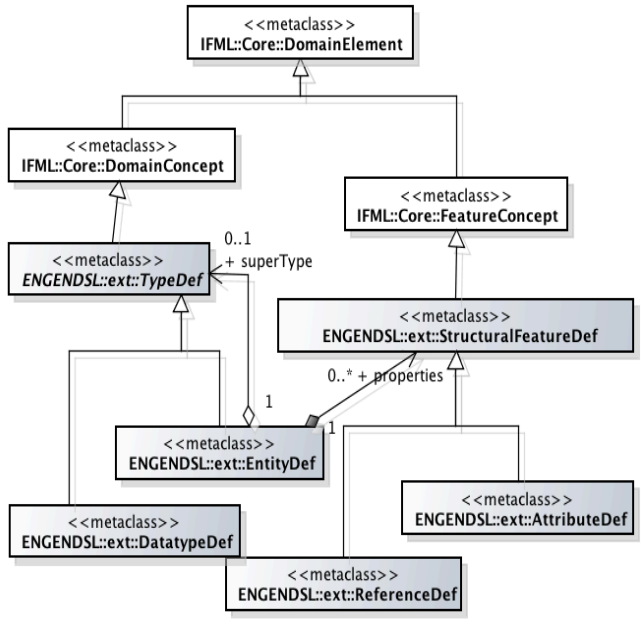

Figure 3. EngenDSL IFML Extensions Package for Domain Elements

IFML Concept	EngenDSL Core Concepts
ViewContainer	ViewDef
ViewComponent	SectionDef
ViewComponentPart	ViewField
Event	ViewEventDef
ActivationExpression	FieldDecoratorDef
Module	ModuleDef
InputPort, OutputPort	-
DomainConcept	TypeDef
	EntityDef
	DatatypeDef
FeatureConcept	StructuralFeatureDef
	AttributeDef
	ReferenceDef
Action	ControllerDef
Navigation Flow	LogicalPath
Data Flow	TargetDef
Parameter, Parameter Binding, Parameter Binding Group	ViewFieldPropertyDef, ViewFieldName
-	LayoutDef
-	LayoutSectionDef
-	LayoutStyleDef

Figure 4. EngenDSL-IFML Elements mapping table

We briefly describe these elements and show examples on how they are used in application models. The examples and further details can be found in Section 5, and the complete model for the approach can be seen in the GitLab public repository for this project [4].

The IFML standard defines *ViewContainers* as containers to other *ViewElements* like other *ViewContainers* and *ViewComponents*. In the proposed language, the same semantics apply. In the language, *Views* are model elements that represent the container IFML concept. Structurally, in the language, *Views* may be composed of other *Views* and *Sections* (see *Section* element). *Sections* are components that compose *Views* and display content. *Section* definitions are laired and composed inside *View* definitions and have a type - *ViewSectionType*. A *Section* definition, in the proposed language, represents the IFML concept of *ViewComponent*. Therefore, sections are used to present or capture any user data, which conforms to the IFML standard. They have a name, a type, a namespace, and may be composed of other *Sections* or *Fields* – which are called subsections. Within *Sections,* we have *Fields*. They represent the IFML *ViewComponentPart* concept. *Fields* have a name, a type – *ViewFieldType* and properties that define specific field semantics like its ID, Label, etc. The Field's type semantically defines its behavior and presentation characteristics such as *"button"* or *"select"* field types. The *EngenDSL* language extends the IFML *ViewComponent* concept, in that a *Section* definition can extend, or in-line include another *Section* definition.

When developing business applications, as a rule, we usually have present the idea of model entities that represent real-world concepts. In the *EngenDSL*, the *Entity* concept is derived from the Model concept in Model View Controller (MVC) [8] and the Domain Concept in IFML.

The EngenDSL directly uses the concept of *Controllers* representing, according to MVC standard, the components that mediate data input and output for both the MVCs *Model* and *View*. The concept of *LogicalPath* is directly associated with the concept of navigation between *Views* and *Controllers*. A *Controller*, at the end of processing, redirects the user to a path. Defining this path is the function of a *LogicalPath*, which is an abstraction for the concept of a redirect in a web application. However, as a *Controller* can resolve to trigger another *Controller* at processing end, *Views* and *Controller*s represent, and are extensions of *LogicalPath*s within *EngenDSL*. Therefore, a *Controller* may determine, in accordance with its implementation logic, redirect the user to a path – *LogicalPath*, which will ultimately render a web page to the application user, or send the application flow to another *Controller*. Note that, at some point, the last *LogicalPath* in a flow within the application logic will be a *View*.

Figure 5 shows an example of such a navigation flow, detailed later in this work in Section 5 using the proposed language. Here the *ListProductsCtrl*, a controller whose type is *Search* (will find data in repository and return the list found), has one of its *LogicalPaths* (*success* – the predefined default logical path) pointing to the View *ListProductsView*. After the selection event is triggered on an item in the list in this view, the *ProductListTable* field *target* property defines the next item in the interaction flow: the *ViewProductCtrl*, which is a controller of type *View*. These controllers will find an item in persistence storage and return the item data. After performing the action, the controller will use its default "*success*" *LogicalPath* to determine the next element in the interaction flow - *ViewProductVw*.

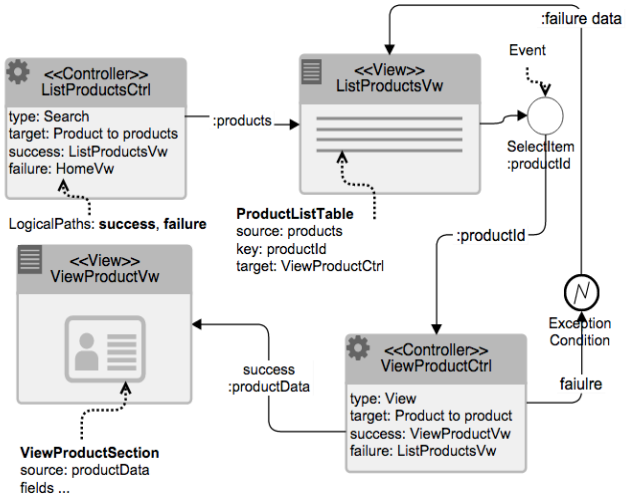

Figure 5. Navigation Flow Example

IV. MODEL TRANSFORMATIONS

In consonance with Model-Driven Software Development technics, the proposed approach uses intermediate model-to-model transformation phase, or phases, before final code generation and integration in the target platform architecture. We use this technique to enable the specification of requirements and accidental concerns, which are foreign to the proposed DSL. The proposed approach uses an intermediate model – *Engen Intermediate Model* (EIM), which can be partially seen in Figure 6. The entire model can be seen in the project GitLab repository [4].

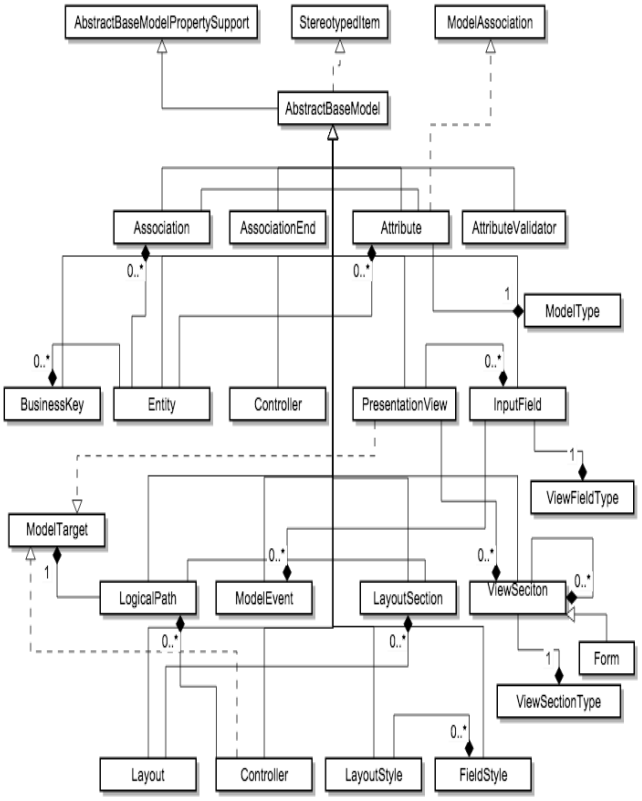

Figure 6. The Engen Intermediate Model (EIM)

The solution was implemented using the XText Framework [19] and other components. The overall elements included in the solution are shown in Figure 7. One of these components is the *EngenGenerator Plugin*. This component, in turn, uses a set of rules specified by the internal *MTMTransformer* interface.

The rule set is configured in a per project configuration file. Each line in this file defines a component (usually a Java class) that implements the *MTMTransformer* interface and its purpose is to transform one or more elements from the given EMF model to the EIM model already shown in Figure 6. The triggering of M2M transformation component can be done by any Java component.

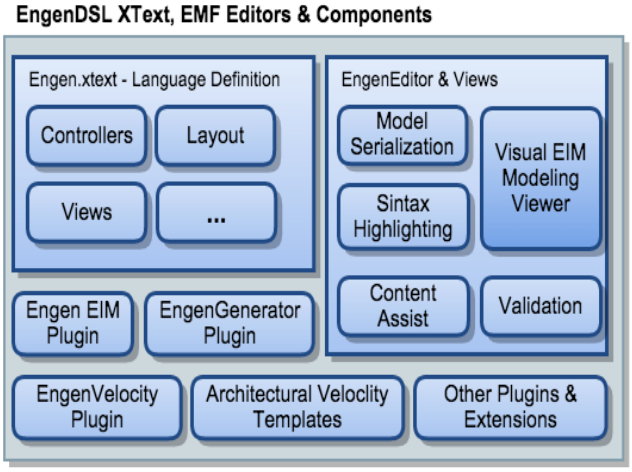

Figure 7. EngenDSL Components

The output from M2M transformation is a XML file that serializes the generated intermediate model. This model can then be further customized by using a technique called model weaving [14]. The proposed implementation of this concept consists in augmenting the original model by setting the value of some properties in separated configuration files.

The rationale for using this technique is that after each model transformation and serialization, attached custom information is not lost. This is automatically done by the provided tooling as shown in Figure 8, in which we can see some properties available for customization after a M2M transformation phase. These properties come from the previously defined intermediate model already shown in Figure 6, and usually have a default value that can be changed by setting the related property of the model.

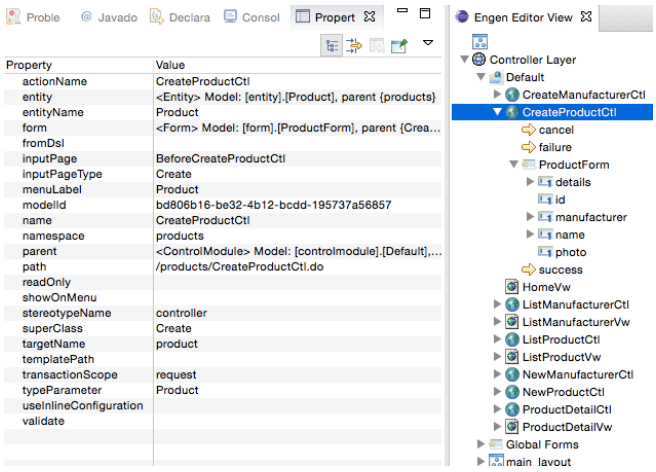

Figure 8. Model weaving customization in tool

The system developer should use this model to fine-tune the created model representation of the DSL concepts. After this step, the M2T transformation can take place. This last step is similar to M2M transformation but require templates targeted to the final application specific platform and

architecture. These templates are responsible to transform the augmented intermediate model into target platform code.

In Figure 9, we show a small portion of a template written in Apache Velocity [15] that receives a model element, identified as *$field* in the template, and outputs an HTML date text field with an associated action.

```
#set ( $accessKey = "")
<input
    name="${field.name}"
    id="${field.name}"
#if (${field.disabled})
    disabled
#end
    size="${field.size}"
    maxlength="${field.maxDisplaySize}"
    accesskey="$!{accessKey}"
    tabindex="${field.ordinalPosition}"
    class="${field.style}"
    value='${${form.name}.map.${field.name}}'
    placeholder='dd/mm/yyyy'
 />


<!-- The JQuery date picker for field ${field.name} -->
<script type="text/javascript">
    $(function() {
        $("#${field.name}").datepicker({ dateFormat: 'dd/mm/yy' });
    });
</script>
```

Figure 9. Apache Velocity template for a text field

Therefore, in conjunction with templates, tailored to generate the artifacts to the target architecture, this intermediate step, allows to complete the system configuration, build and generation of the final solution, which will be based on the target architecture specified by intermediate and final transformations, model weaving and templates.

V. AN APPLICATION EXAMPLE

In this section, we show a portion of an application modeled using the proposed method. The full examples can be found on the project GitLab public folder [4].

The example is a small shopping application. Some of the *Entities*, along with properties and constraints are shown in the example in Figure 10.

The domain model represented by the *Entity* instances is straightforward. They define the domain model elements along with their properties and constraints. Following are the controllers, for example the *ListProductCtl*. The definition specifies a controller model element. In this case, the *Search* type is defined, areas in the *CreateProductCtl* a *Create* controller type is defined. The "*Search"* and "Create" controller types (after the colon in the controller definition) configuration binds the model element to an external component that performs a search in the persistence storage for the domain model element - specified by the *target* property, using the provided criteria. In this specific example, if this controller is accessed from the home page, all elements in persistence storage should be returned, since there are no criteria defined.

On the *ListProductVw* view element in Figure 11, we can see how *Sections* can be nested and how the search for the products can be further customized.

manufacturer.vdsl

```
entity Manufacturer {
    namespace products
    name string
    address string
    constraints {
        required name, address
    }
}
```

CreateProductCtrl.vdsl

```
controller CreateProductCtl : Create {
    namespace products
    target Product to product
    success ListProductVw
    failure NewProductVw
    cancel ListProductVw
}
```

product.vdsl

```
entity Product {
    namespace products
    name string
    details string
    manufacturer Manufacturer
    photo image
    constraints {
        required name, details,
    }
}
```

ListProductsCtrl.vdsl

```
controller ListProductCtl : Search {
    namespace products
    target Product to product
    label "View All Products"
    success ListProductVw
    failure HomeVw
}
```

Figure 10. Entities and Controllers definitions

While when the *ListProductsCtl* controller was accessed without any search context information the controller performed a database search for all products, without any search criteria, the search triggered by the *NewSearch* button, on the *RefineSearchSection* form, will use the context information from the input fields (*name* AND *manufacturer*) as the criteria for the controller to further filter the results.

```
view ListProductVw {
    label "Products found"
    section productListSection {
        field productTable : table {
            target ProductDetailCtl
            source "returnCollection"
            key "id"
            property photo
            property name
        }
    }
    section RefineSearchSection : form {
        target ListProductCtl
        label "Search Products"
        section SearchFields {
            field name : text {
                property name "Name:"
            }
            field manufacturer : select {
                id manufacturer
                label "Manufacturer:"
                property manufacturer "Manufacturer"
            }
            field NewSearch : button {
                id newSearchButton
                label "Search Products"
                target ListProductCtl
            }
        } // End of SearchFields
    } // End of RefineSearchSection
}
```

Figure 11. The ListProductVw View Definition

 ISBN: 978-1-61208-412-1

This is automatically done by the ad-hoc *Web Framework* we are using. We can see from the *NewSearch* button configuration, that the *target* property points to the *ListProductCtl* controller. This information corresponds to the event model definition for this element. In the proposed model, this information is sufficient for templates to interpret the event model: the target controller for the action (along with its fields containing the criteria for the query entered by the user).

Figure 12 shows a screenshot for the generated HTML running in an application server for the *NewProductVw View*.

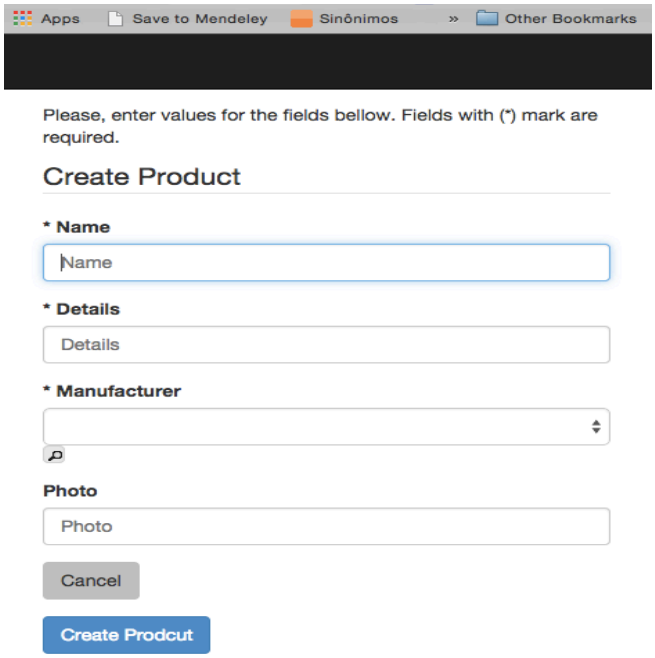

Figure 12. Running Application Screenshot for the Create Product View

As stated before, the proposed approach includes a way to define and reuse application layout. In the DSL, we declare the partial layout for the application as shown in Figure 13.

The layout name appears after the *layout* keyword. The layout specifies a *template* property that will be parsed by the M2T engine, and will produce the file defined by the *path* property. A small section for the specified template is shown in Figure 14.

This template defines some markup instructions and delegates the layout subsections parsing to another template called *parseLayoutSections*. Each section defined by the main layout element is a composition of URLs, templates and libraries. For example, the *head* section shown in Figure 12, defines a template to be parsed before and other after the main template, for this section (which will be by default the *section* name if the *template* attribute is not defined). This information is just stored in the model, not hardcoded in the solution, so they can be used by the templates and M2T components as they find applicable.

```
layout main_layout {
    // This is relative to the architectural style path.
    // The layout must already be loaded when Views are parsed
    template: "new_bootstrap.layout.vm"
    path: "/WEB-INF/jsp/tiles/layouts/layout.jsp"
    // These sections are detailed later in this file.
    section: head
    section: body
    section: menu
    // Defines the overall styles for this layout
    style: bootstrap_tabular
}

layout_section head {
    before: "head_init.vm"
    // This will allow templates to act on generic libraries.
    // Note: if a library is not referenced here it will not be
    // parsed by the model transformer
    library: jquery
    library: bootstrap
    library: customStyle
    after: "head_end.vm"
}
```

Figure 13. Partial Layout definition

```
##################################################################
## File: new_bootstrap.layout.vm
##################################################################
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
<%@ taglib prefix="shiro" uri="http://shiro.apache.org/tags" %>
<%@ taglib uri="http://struts.apache.org/tags-tiles" prefix="tiles" %>
<!-- Init new_bootstrap.layout.vm -->
<!DOCTYPE html>
<html lang="en">
## Aside from inserting basic html tags, parsing the parseLayoutSections
## is the onlything this template should be doing.
#parse("${rootControlPath}/parseLayoutSections.vm")

</html>
<!-- End of new_bootstrap.layout.vm -->
```

Figure 14. Section Layout template

VI. Related Work

Several works have been proposed for modeling and specification of Web systems. *WebML* [16] and *WebDSL* [17] are examples. This work starts from these approaches to specify a textual and declarative form of describing the various aspects involved in this type of application. However, none of them is based in an independently defined standard like IFML.

The *WebML* proposed in Ceri et al [16], also defines a model that uses other work as the basis for the definition of Web applications. *WebML* uses *UML* and *XML* in the data dimensions (structural model), pages (composition model), topology of links between pages (navigation model), the graphics and layout requirements for the pages (presentation model) and customization needs (customization model) for delivering content to the user.

Visser proposal [17] is more similar to the one presented in this work, called *WebDSL*. It defines a language for describing web applications, covering its various aspects. However, in a different direction than the one presented in this article, the language goes beyond the simple statement of these concepts to specify conditional structures, functions and methods, including algorithms, which are used to define

 ISBN: 978-1-61208-412-1

the behavior of certain parts of the application architectural structure and components. While this approach has some advantages, it possibly creates a greater complexity in the development process.

VII. CONCLUSION AND FUTURE WORK

Domain Specific Languages are a technological trend. Its application in Web systems is perfectly feasible and even recommended. Not only because of productivity that this application can achieve, but also by all other benefits that a development approach based on models and automatic generation of artifacts can bring to Web Systems Engineering. The benefits include rapid construction and prototyping, the reduction of failures, standard architecture and solutions, suitable and adaptable technology approach, in addition to allow a safe evolution between different technologies and frameworks that constantly arise for this type of application.

This work demonstrated a DSL, developed for web specific domain, in consonance with an adopted OMG standard for defining user interactions – IFML. This can improve reusability of models, view interactions modeling understanding and collaboration since the text nature of the solution, enables better handling of models, inclusive when related to version control repositories and tools.

We are now looking into Business Rule integration into the DSL to allow advanced rules verification, along with the basic constraint based rules. We also intend to implement a visualization interface to the model using standard IFML notation.

REFERENCES

[1] J. Bettin, “Best Practices for Component-Based Development and Model-Driven Architecture”, 2003. Available from: http://s23m.com/publicwhitepapers/best-practices-for-cbd-and-mda.pdf. [retrieved: May, 2015]

[2] K. Czarnecki and S. Helsen. “Classification of Model Transformation Approaches". OOPSLA’03 Workshop on Generative Techniques in the Context of Model-Driven Architecture. 2003, pp. 1–17. Available from: http://s23m.com/oopsla2003/czarnecki.pdf. [retrieved: May, 2015].

[3] A. V. Deursen, P. Klint and J. Visser. “Domain-Specific Languages: An Annotated Bibliography.”. ACM SIGPLAN Notices. Volume 35, Issue 6. 2000, pp 26-36. http://www.st.ewi.tudelft.nl/~arie/papers/dslbib.pdf.

[4] Engen, 2015. “Engen Project Site.” Available at https://gitlab.com/engen/public/wikis/home [retrieved: May, 2015].

[5] V. Estêvão, S. Souza, R. D. A. Falbo, and G. Guizzardi. “A UML Profile for Modeling Framework-Based Web Information Systems.” In Proc of the 12th International Workshop on Exploring Modeling Methods in Systems Analysis and Design. 2007, pp. 149–58.

[6] JM. Favre. “Towards a Basic Theory to Model Model Driven Engineering.” In Procs. of the 3rd Int. Workshop in Software Model Engineering (WiSME). 2004, pp. 262-271.

[7] F. Fondement and R. Silaghi. "Defining Model Driven Engineering Processes". Third International Workshop in Software Model Engineering (WiSME). held at the 7th International Conference on the Unified Modeling Language (UML), 2004.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. "Design Patterns: Elements of Reusable Object-Oriented Software". 2nd ed. Boston - 1995.

[9] M. Mernik, J. Heering and A. M. Sloane. “When and How to Develop Domain-Specific Languages.” Journal ACM Computing Surveys (CSUR), Volume 37, issue 4, 2005, pp. 316–44. DOI:10.1145/1118890.1118892.

[10] N. Moreno, J. R. Romero and A. Vallecillo. “An Overview of Model-Driven Web Engineering and the MDA". Web Engineering: Modelling and Implementing Web Applications Human-Computer Interaction Series 2008, pp. 353-382.

[11] OMG 2014. Interaction Flow Modeling Language. Available from: http://www.ifml.org. [retrieved: May, 2015].

[12] C. E. Purificação and P. C. Da Silva. “EngenDSL - A Domain Specific Language for Web Applications". 10th International Conference on Information Systems and Technology Management – CONTECSI 10. 2013. pp. 879–99. doi:10.5748/9788599693094-10CONTECSI/PS-474.

[13] T. Stahl, M. Voelter and K. Czarnecki. "Model-Driven Software Development: Technology, Engineering, Management". 2006, John Wiley & Sons. ISBN:0470025700.

[14] M. Völter. “Model-Driven Software Development", 2006. Available from: http://www.voelter.de/data/books/mdsd-en.pdf. [retrieved: May, 2015].

[15] Apache Velocity, 2009. Available from: http://velocity.apache.org. [retrieved: May, 2015].

[16] S. Ceri, P. Fraternali and A. Bongio, "Web Modeling Language (WebML): a modeling language for designing web sites". Computer Networks 33 (1-6), 2000, pp. 137-157.

[17] E. Visser. "WebDSL: A case study in domain-specific language engineering, generative and transformational techniques in software engineering". GTTSE, Lecture Notes in Computer Science, Springer (2008). Tutorial for International Summer School GTTSE, 2008, pp. 1-60.

[18] K. Czarnecki. "Overview of generative software development. In J.-P. Bantre et al,, Ed., Unconventional Programming Paradigms (UPP’04), Lecture Notes in Computer Science, vol. 3566, 2004, pp. 313–328, Mont Saint-Michel, France.

[19] XText, 2009. Available from: http:// www.xtext. org. [retrieved: May, 2015].

 ISBN: 978-1-61208-412-1