Universidad Autónoma
de Madrid

Biblos-e Archivo
Repositorio Institucional UAM

Repositorio Institucional de la Universidad Autónoma de Madrid
https://repositorio.uam.es

Esta es la **versión de autor** del artículo publicado en:
This is an **author produced version** of a paper published in:

Computer Languages, Systems & Structures 49 (2017): 133-151

DOI: https://doi.org/10.1016/j.cl.2017.04.002

Copyright: © 2017. This manuscript version is made available under the CC-BY-NC-ND 4.0 licence http://creativecommons.org/licenses/by-nc-nd/4.0/

El acceso a la versión del editor puede requerir la suscripción del recurso
Access to the published version may require subscription

An Approach to Build XML-based Domain Specific Languages Solutions for Client-Side Web Applications

Enrique Chavarriaga, **Francisco Jurado**, **Fernando Díez**
Universidad Autónoma de Madrid, 28049 Madrid, Spain
enrique.chavarriaga@inv.uam.es, {francisco.jurado, fernando.diez}@uam.es

Summary. Domain-Specific Languages (DSLs) allow for the building of applications that ease the labour of both software engineers and domain experts thanks to the level of abstraction they provide. In cases where the domain is restricted to Client-Side Web Applications (CSWA), XML-based languages, frameworks and widgets are commonly combined in order to provide fast, robust and flexible solutions. This article presents an approach designed to create XML-based DSL solutions for CSWA that includes an evaluation engine, a programming model and a lightweight development environment. The approach is able to evaluate multiple XML-based DSL programs simultaneously to provide solutions to those Domain Specific Problems for CSWAs. To better demonstrate the capabilities and potential of this novel approach, we will employ a couple of case studies, namely *Anisha* and *FeedPsi*.

Keywords: Domain-Specific Languages; XML Interpreter; JavaScript; Web Application; XML Programing Language.

1 Introduction

Domain-Specific Languages (DSLs) provide a high-level of abstraction in order to model specifications, structures and functionalities that solve domain-specific problems. The goal of a DSL is to make the design, definition, and implementation of systems simpler by allowing domain experts to better perform their tasks while building high quality and reliable systems in order to provide Domain-Specific solutions [52]. Succinctly, Fowler [16] describes a DSL as *«a computer programming language of limited expressiveness focused on a particular domain»*.

The development of a DSL implies the use of scanners, parsers and code generation tools to evaluate domain-specific models to perform the associated functionality. However, when working with an XML standard, XML-based DSLs (XML-DSL) can take advantage of general-purpose parsers like the Simple API for XML (SAX) and the Document Object Model (DOM) [53]. These APIs are incorporated into most programming languages where they allow programmers to access and to modify both the structure and the content across XML languages. In this way, XML allows for both the storage and exchange of auto-documented structured information in the language that best fits what needs to be modelled [13][55], as well as specifying and extending grammars with associated functionality that can be readily implemented through the use of standard APIs. Thus, XML-DSLs provide straightforward DSLs specification that are extensible and easy to combine, and whose associated functionalities can be rapidly implemented using existing available APIs for the corresponding target programming language.

Technological changes over the past decade have revolutionized the capabilities and the types of web-based information systems that can be delivered. These changes require a new point of view when providing solutions for Client-Side Web Applications (CSWA). Currently, the design and implementation of these kinds of solutions are exploratory topics [46] based on building dynamic

web pages and their related technologies [36][54]. In addition, Web 2.0 [1] provides the technologies, services and tools to build functional, pleasant and usable web pages, with the added value to be able to deploy them across multiple platforms.

However, in spite of the growing relevance of CSWA and the demonstrated interest in DSL by the scientific and industrial community, as far as we know, there are few works that allow creating DSL solutions for CSWA. Thus, in this context we want to address the creation and implementation of XML-DSLs for CSWA, which are not covered by the few existing solutions. Accordingly, this article proposes the *Programmable Solutions Interpreter* (Psi), an approach to building CSWA solutions that includes an evaluation engine, a programming model and a lightweight development environment, which we have called *PsiEngine*, *PsiModel* and *PsiEnvironment* respectively. Our approach allows for the creation and evaluation of XML-DSLs for CSWA that encapsulates functionality and integrates with other web widgets and frameworks in order to create fast, robust and flexible solutions.

The central piece of the *PsiEngine* is the *PsiXML Interpreter* (*PsiXML* for short), a JavaScript XML interpreter able to evaluate XML-DSLs programs. *PsiXML* is able to register several XML-DSLs and to evaluate several programs written in the registered XML-DSLs. XML-DSLs programs are a set of programmable tags that have specific associated functionality. When executing an XML-DSL program it consists of evaluating the functionality of each XML tag. In addition, an XML-DSL specified and implemented in the *PsiEngine* is able to link and interchange heterogeneous information (in XML and JSON formats) and to apply up-to-date security policies and good programming practices [26][60], in this way, it will be able to develop flexible XML-DSLs for CSWA.

The *PsiModel* establishes a programming model for lead programmers to generate JavaScript code and accompanying documentation, mainly oriented toward the creation of XML-DSLs and reusable JavaScript components. Finally, the *PsiEnvironment* is the lightweight environment that employs the *PsiModel* and uses the *PsiEngine*.

With these three elements (*PsiEngine*, *PsiModel* and *PsiEnvironmen*) the main goal of this article is to set up the foundations to apply Model Driven Engineering (MDE) while building CSWA. MDE is a software methodology centred in defining models to simplify the creation of information systems [49]. It combines concepts like DSL, XML-DSL and DSVL (Domain-Specific Visual Language [28]) together with transformation engines and code generators. In this context the *PsiEngine* and the *PsiModel*, which we detail in this article, are the cornerstone for the aforementioned goal.

In order demonstrate the implementation of *PsiEngine* and its capabilities, while operating the *PsiModel* and the associated *PsiEnvironment*, we will provide two case studies, namely: *Anisha* and *FeedPsi*. The goal of *Anisha* is to build two XML-DSLs for animating basic frames. It is a running example that details the interpretation and execution of multiple programs as well as information binding. Meanwhile, *FeedPsi* is a News Aggregation CSWA designed to evaluate the integration with other web components, RSS services and frameworks. In both cases we have followed the qualitative case study methodology suggested in [59] and adapted it for software engineering [2]. With these two case studies we will cover the most relevant features for programming XML-DSL solutions with our approach.

The rest of the article is structured as follows: section 2 will highlight related cutting edge works; section 3 will provide a general overview and the most relevant features of our approach, including the *Anisha* running example; in section 4 we will explain the *FeedPsi* case study, summarizing the validation details of *PsiEngine*; section 5 will detail the results we have obtained; finally, section 6 will close the article with some concluding remarks and future works.

2 Outline and related works

The term Domain-Specific Language (DSL) is not rigorously defined in the literature. As mentioned above, Fowler [16] defined it as *«a computer programming language of limited expressiveness focused on a particular domain»*. In [51], Spinellis says that a *«DSL is a programming language tailored specifically to an application domain: rather than being for a*

general purpose, it captures precisely the domain's semantics». Meanwhile, Mernik et al. [38] said that *«domain-specific languages (DSLs) are languages tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use compared with general-purpose programming languages in their domain of application»*.

For its part, [12] studies the languages linguistic composition and assists in classifying DSLs while taking into account: language extension, language restriction, language unification, self-extension, and extension composition. Accordingly, our work focuses on building self-extension languages and extension composition for web-clients.

There appears to be a consensus that a DSL is a programming language devoted to solving specific problems in particular domains. As such, the abstraction of their syntax and semantics are determined by the problem domain, and are aimed to implement information systems that provide solutions to specific problems. According to [11][16][17][23][25][31][38], DSLs have become a unique Software Engineering research area and a keystone Software Engineering methodology. Thus, DSLs have been proved in Software Factories [18], Generative Programming [9] and the MDE [10][49]. Particularly, from a MDE point of view, in [10] we can see a study about software products, platforms and transformations tools to build modelling languages.

In [32] we can find a Systematic Mapping Study (SMS) on DSLs to identify research trends in the period 2006-2012. Their authors looked for possible open issues and an analysis on what they called demographics of the literature. In their SMS study, the authors observed that the DSL community appears to be more interested in the development of new techniques and methods that support the different phases of the development process (analysis, design and implementation) of DSLs, rather than researching new tools, and only a small portion of studies focus on validation and maintenance. In addition, the authors observed that most of the works do not indicate the tools they utilized for the implementation. This is really meaningful because to build a DSL solution involves the use of tools for the implementation of interpreters and compilers like Lex and Yacc [4] or Flex and Bison [34] to create them. Currently, widely used Integrated Development Environments (IDE), such as Eclipse and Visual Studio .NET, provide tools and languages specifically designed to implement DSL. Even general purpose programming languages allow for the exploitation of their features to accomplish the task. Examples of IDE tools abound, we can highlight several plugins for the Eclipse environment, such as Stratego/XT [5], LISA [39], Spoofax [27], Antlr [42], Xtext [3] and Eclipse Modelling Project [19], as well as the DSL Modeling [7] and Boo [45] for the .NET platform (see [38] for up to 20 development tools). In [43] the authors developed Onto2Gra to build DSLs following an ontological approach. General programming language tools for building DSLs include those [3] for Java, [30] C#, [17][14] Ruby, [37] Python, [25] Clojure, [17][11] Groovy, [47] Haskell and [29] aspect-oriented programming.

So far, we can see that currently there are a lot of IDEs and tools that exist to implement DSLs. However, which is the preferred solution when we focus on building DSLs (and particularly XML-DSLs) for web-client? The mentioned SMS does not make explicit reference to the creation of DSL solutions for web-clients. But, among the few examples of tools to build solutions for web-clients, we can mention Jison [6] a JavaScript parser generator, which has been used to implements the CoffeeScript language [33]. However, CoffeeScript is not a DSL, but a little programming language that compiles to JavaScript thanks to Jison.

When seeking DSL solutions for a web-client, there are only a few solutions, and all of them are oriented to build only the user interface. Thus, we can mention [21], [15] and [50] as examples. In these works, the authors define ad-hoc XML-DSLs for a web-client, although they do not give any advice about the tools the authors have used in the building process. In [21] we can see a work about an XML-DSL for a clinical application, to define and describe user interfaces where different users must interact. In [15] the authors propose the DUI language (Distributed Language Interfaces) to describe user interfaces, task composition and web augmenters. Finally, in [50] a XML-DSL is described to define different 3D animations in web-client.

With these two drawbacks, i.e., the need for researching the building of new DSL tools for web-clients, and, the need for DSLs support in web-client solutions, our research focuses on implementing and executing XML-DSLs solutions for CSWA.

3 Approach to Build XML-DSL for Client-Side Web Applications

In this section we will present our approach detailing how to manage XML-DSL solutions that can be directly interpreted and evaluated in CSWAs. Accordingly, this will document the central ideas related to the *PsiEngine* (*Programmable Solutions Interpreter Engine*), and how to implement it, evaluate it, while interpreting and executing code within the web-client. By using a running example, we will describe several concepts related to the approach, namely: the *PsiGrammar*, the *PsiLanguage* and the *PsiComponent*. Finally, we will detail the programming model this approach outlines, which we have named *PsiModel*, and the environment we have developed to apply and test it.

3.1 The *PsiEngine*

Fig. 1 shows an outline of our approach. The main idea is to fetch *PsiCode*, written in the most appropriate XML-DSL, together with the necessary *Resources* to evaluate and interpret them within the *PsiEngine*. Once the *PsiCode* and *Resources* are available, the *PsiEngine* parses and transforms them to JavaScript by interpreting their DOM and executing the JavaScript code associated to each DOM element. As a consequence we obtain the *PsiObject* that solves a specific problem within a CSWA. In addition, the corresponding error and warning messages are reported in case they need to be managed.

Fig. 1 shows the central piece of this approach: the *PsiEngine*. It has been fully developed using web-client technologies, combining HTML5, CSS3, JavaScript and DOM, with technologies, services and tools from Web 2.0. This way we have an engine that completely runs in the web-client, and given the definition of specific XML-DSLs, it is able to build web components, web widgets and/or dynamic web pages that provide solutions to specific problems provided in a CSWA.

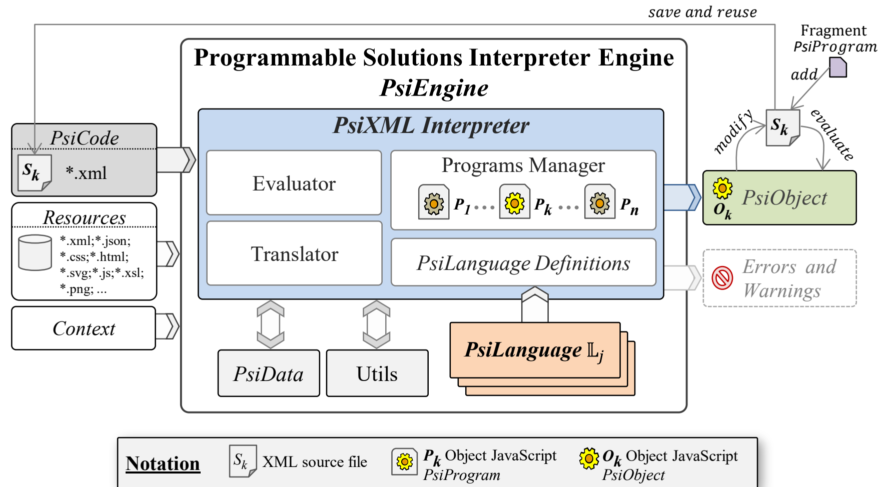


Fig. 1. *PsiEngine* (Programmable Solutions Interpreter Engine) to execute *PsiCode* in a web-client.

The *PsiEngine's* core is the *PsiXML Interpreter* whose main goal is to evaluate and interpret the *PsiCode*, to generate the JavaScript code and finally to execute it. Additionally, it has a common shared area called *PsiData*, which allows for the exchange of information, functionalities and objects among running *PsiPrograms*.

In Fig. 1 we can also see how the *PsiXML* manages several *PsiLanguages* $\Psi = \{\mathbb{L}_1, \mathbb{L}_2, \dots, \mathbb{L}_m\}$, i.e., those XML-DSLs it will be able to evaluate, interpret and execute. To do so, the *PsiXML* registers a set *PsiLanguages Definitions*. For each different *PsiCode* S_k written in a *PsiLanguage* $\mathbb{L}_j$, a *PsiProgram* P_k is created and managed by the *Programs Manager*. The *PsiProgram* P_k is responsible for converting the *PsiCode* S_k to a *PsiObject* O_k by performing a syntactic and semantic analysis based on the grammar of a *PsiLanguage* $\mathbb{L}_j$ (transformation carried

out by the *Translator* component of the *PsiXML*). This transformation consists of processing the DOM from the *PsiCode*, validating each DOM element and executing the corresponding functionality (performed by *Evaluator*). In addition, the *Programs Manager* manages all the *PsiPrograms* $\mathbb{P} = \{P_1, P_2, \ldots, P_n\}$ that the *PsiXML* is able to evaluate.

PsiLanguages have similarities with other XML-based languages. To specify and use the different XML-DSLs in the web-client, we follow the same approach of other XML-based languages, such us XSL, SVG, MathML, etc. In summary, all of them are intended to enrich the content of web pages by defining an XML grammar, where each XML element has its own semantic implemented by the corresponding associated functionality to achieve its objective once interpreted by the web-client. However, in spite of native code the interpreter into current web browsers or by means of plugins, our approach performs all the analysis directly with the *PsiEngine* using the language specifications that come from the server and that can dynamically change. In this way, the *PsiEngine* easily manages new XML-DSLs, associates the functionality corresponding to their semantic, and provides a working environment that facilitates their evaluation.

On the other hand, unlike other XML-based languages, *PsiLanguages* have the ability to associate the *PsiCode* with external resources (both XML and JSON) in order to use and modify information at runtime. To associate information implies that an XML containing the data exists as a DOM (XML-data DOM), so that any element of the *PsiCode* S_k can have a reference to elements from the XML-data DOM. Similarly it is possible to associate JSON information with elements of *PsiCode* S_k.

3.1.1 Defining *PsiLanguage*

To specify a *PsiLanguage* so that the *PsiEngine* can manage it, we need to define its corresponding grammar. A *PsiGrammar* $\mathbb{G}$ for a *PsiLanguage* will be defined by a tuple:

$$\mathbb{G} = \langle \mathbb{T} | Root | \Delta \rangle \qquad (1)$$

where $\mathbb{T} = \{Tag_1, Tag_2, \ldots, Tag_n\}$ is the set of tags, $Root$ (for some $Tag_j \in \mathbb{T}, 1 \leq j \leq n$) is the root element of the grammar and Δ is an object that the language structure defined as:

$$\Delta = \{Tag_i : \Delta_i | Tag_i \in \mathbb{T}\} \qquad (2)$$

where $\Delta_i \in \Delta$ is an object specified by:

$$\Delta_i = \{\text{TAG}: v_T, \text{CLASS}: v_C, \text{CHILDREN}: v_H, \text{MULTIPLICITY}: v_M, \text{STRICT}: v_S, \text{VALIDATOR}: v_V\} \qquad (3)$$

where v_T is the tag name, v_C is the associated class name, v_H are the child tags nodes (default is *null*), v_M is the multiplicity the child tags (possible values: “0..1”, “1..1”, “0..n” ó “1..n”), v_S is the strict validation of the children tags (default is *true*), and v_V specifies validation tag attributes (default is *null*).

It is commonly accepted that the use of DTD and XML Schema easily describe the structure, the grammatical constraints and the data types of XML documents. When developing the *PsiEngine* we analysed the use of these to validate *PsiCode*. As previously stated, data and programs can be separated in our approach. Although the *PsiCode* is written in XML and therefore can be validated using DTD or XML Schema, XML and JSON data are loaded and linked at runtime. Therefore an alternative to validate the code and data under this perspective is required. The couple composed by the *PsiLanguage Structure Diagram* (*PsiLSD*) and the *PsiGrammar Validator Attributes* (*PsiGVA*) solves this problem.

The *PsiLSD* shown in Fig. 2(a) is a graphical representation of the language structure related to a *PsiGrammar*. For its part, Fig. 2(b), shows the *PsiGVA* and its specific syntax for validating tag attributes v_V. *PsiLSD* and *PsiGVA* dramatically simplify the design and development of grammars for the *PsiXML*.

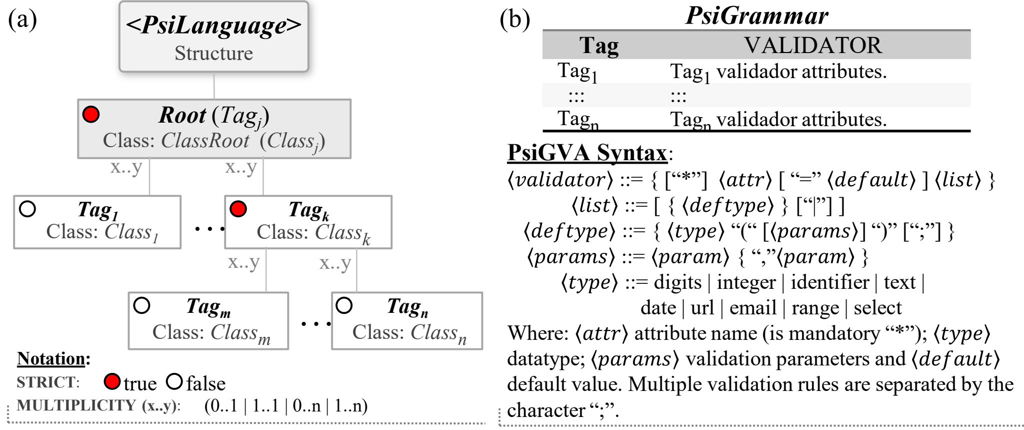


Fig. 2. (a) *PsiLanguage Structure Diagram*; (b) *PsiGrammar Validator Attributes*.

Once the *PsiGrammar* has been defined, it is necessary to implement its semantic, i.e., to code the functionality associated to each tag $\mathbb{T} = \{Tag_1, Tag_2, \ldots, Tag_n\}$ from the *PsiLanguage*. These functionalities are implemented as a set of classes from a reusable JavaScript component (see [8][58]), and then matched with their corresponding tags. The evaluation and execution of all functionalities solves a domain specific problem in a CSWA.

Thus, a *PsiLanguage* $\mathbb{L}$ is defined as a tuple:

$$\mathbb{L} = \langle \mathbb{G} | \mathbb{K} | \mathbb{T} \leftrightarrow \mathbb{C} \rangle \tag{4}$$

where $\mathbb{G}$ is the *PsiGrammar* definition for the *PsiLanguage* (as defined by (1)), $\mathbb{K}$ is the reusable JavaScript software component (called *PsiComponent*), $\mathbb{C} = \{Class_1, \ldots, Class_n\}$ is the subset of classes implemented in $\mathbb{K}$ and finally, $\mathbb{T} \leftrightarrow \mathbb{C}$ is the association between Tag_k and $Class_k$, for each $Tag_k \in \mathbb{T}$ and $Class_k \in \mathbb{C}$, respectively.

Fig. 3 shows the association between the tags from the $\mathbb{G}$ grammar and their related classes from $\mathbb{C}$. Because a tag in a *PsiLanguage* may be defined for more than one part of the grammar structure, different functionalities can be associated. Similarly, a class can be associated to different tags. In Fig. 3(a) we can see how *PsiLSD* associates the corresponding JavaScript class name (i.e., $v_c \in \Delta_i$ in (3)) with its related tag name in the XML grammar.

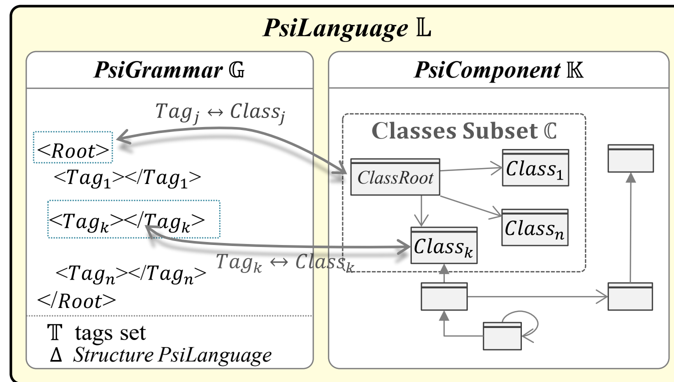


Fig. 3. Association diagram between tags set and classes set to define a *PsiLanguage*.

TEMPLATE 1 shows how to develop a *PsiComponent* $\mathbb{K}$, where the classes that manage the desired functionality and their associated tags from the *PsiGrammar* are implemented.

TEMPLATE 1. JavaScript template to define a *PsiComponent*

```
var MyComponentPsi = (function () {
  // Tag set for the PsiGrammar
  var ID_TAG_T1 = "T1", …, ID_TAG_Tn = "Tn";
  // Language structure
  var ID_TAGS = {};
```

```
  ID_TAGS[ID_TAG_T1]={TAG:"Tag1", CLASS:"Class1", CHILDREN:…, MULTIPLICITY:…,
                      STRICT:…,  VALIDATOR:"…"};
  …
  ID_TAGS[ID_TAG_Tn]={TAG:"Tagn", CLASS:"Classn", CHILDREN:…, MULTIPLICITY:…,
                      STRICT:…, VALIDATOR:"…"};
  // PsiGrammar definition
  var GRAMMAR = {
    NAME: "GrammarName",
    TAGS: ID_TAGS,
    ROOT: ID_TAG_Tj // Root element
  };
  // Associated classes for the PsiGrammar
  function Class1(){}; Class1.methods({...}); // Class Class1 implementation
  ...
  function Classj(){}; Classj.methods({...}); // ClassRoot implementation
  ...
  function Classn(){}; Classn.methods({...}); // Class Classn implementation

  // Aditional programming elements for the PsiComponent
  // PsiLanguage Register
  PsiXML.registerLanguagePsi("GrammarAlias", GRAMMAR, ParsePsi);
  return {
    // PsiComponent interface
  };
})();
```

It should be noted that those classes that relate ℂ to a *PsiLanguage* inherit from an abstract class (*PsiElement* from the *PsiLanguages Definitions*). This abstract class implements the abstract behaviour for each element. In particular it binds each DOM element (i.e. each tag) from the XML source code, manages the grammatical structure, and performs attributes and child elements validation. Furthermore, it has the ability to execute (by calling the corresponding JavaScript functions) the functionality related to each specific component through an events set. Also, the *PsiElement* abstract class is able to natively manage two languages, namely:

i. *PsiLanguage to Link Information* (*PsiLI*) in charge of associating information stored in XML and JSON formats from the *Context* or the *PsiData* to the corresponding class.

ii. *PsiLanguage Code Attribute* (*PsiCA*) in charge of evaluating "inline" code in tag attributes. The finality of this inline code is to allow setting the attribute value with the associated information.

Because these languages have been implemented in the *PsiElement* abstract class, the use of the *PsiLI* and *PsiCA* languages can be incorporated into every element of a *PsiLanguage*. In this way, the approach provides the capability to associate external XML/JSON information resources with the implementation of every *PsiLanguage* natively. By isolating the external resource storage along with the *PsiPrograms*, resources can be reused among components, frameworks and web applications. On the one hand, *PsiLI* allows *PsiLanguages* to define their association to external data resources; on the other hand, *PsiCA* provides the semantics needed to access, use and modify those resources. This is an innovative feature in creating XML-DSLs as it provides heterogeneous XML/JSON information to any XML-DSL, unlike other approaches such as XLink (XML Linking Language), which adds XML elements and resources by cross-linking documents, images and files through to the web.

More details about the *PsiEngine* are explained in http://hilas.ii.uam.es/api. This website includes interactive class diagrams for the *PsiXML*, detailed *PsiLanguages* definition and additional programming elements.

3.1.2 Running Example: The *Anisha* Case Study

In order demonstrate how this approach works in a systematically defined development process, we will first define the grammar, then detail how the related functionality is associated to each grammatical element, and conclude by showing how the interpreter executes the source code for each language. Thus, we will utilize a running example we have called *Anisha*. *Anisha* combines

two *PsiLanguages*, namely *ShapesPsi* and *AnimaPsi*. *ShapesPsi* allows for the creation of circles and rectangles based on SVG, while *AnimaPsi* let move and stop the graphical elements defined in *ShapesPsi*.

▪ **The *ShapesPsi* Language**

FRAGMENT 1 shows an example code written in *ShapesPsi* language. In this code we can see how to create a red circle with centre at point (100, 100), label "Circle", radius 30 (all units in pixels), and a blue rectangle with centre at point (250, 100), label "Rectangle", weight 20 and high 10. Evaluation can be seen in http://hilas.ii.uam.es/Anisha/basic.

FRAGMENT 1. Example code written in *ShapesPsi* language (basic.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<Shapes name="Basic example ShapesPsi">
  <Circle id="F1" center="100,100" radius="30" label="Circle"
          style="fill:red;stroke:gray;stroke-width:2px"/>
  <Rectangle id="r1" center="250,100" width="70" height="50" label="Rectangle"
             style="fill:blue;stroke:gray;stroke-width:2px"/>
</Shapes>
```

As previously explained, *ShapesPsi* is a DSL that follows an XML-based grammar for creating circles and rectangles. Fig. 4(a) shows the *PsiLSD* of the *ShapesPsi* Language, in which the root tag *Shapes* is the beginning of the program. Then, we can define the necessary grammar elements: circles (multiple *Circle* tags) and rectangles (multiple *Rectangle* tags).

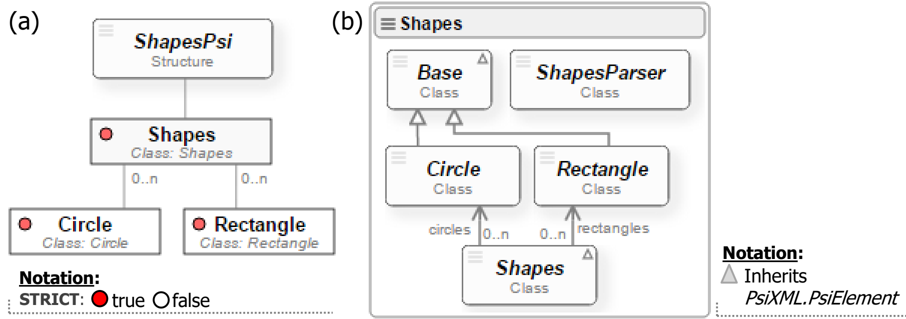


Fig. 4. (a) *PsiLSD* of the *ShapesPsi* Language. (b) Class diagram for the **Shapes** Component.

The UML Class diagram for **Shapes** component is shown in Fig. 4(b). The **Shapes** class defines a set of circles and rectangles. The *Base* abstract class defines a generic figure, while the *Circle* class paints a circle with a text in the middle on a SVG canvas, and the *Rectangle* class paints a rectangle with a text in the middle. All the coded circles and rectangles are stored in *PSI Data* so that other programs and languages can use them. The reader is encouraged to look up the detailed implementation of the **Shapes** component at http://hilas.ii.uam.es/Anisha/api.

Thus, we can specify the *ShapesPsi* Grammar, as follows:

$$\mathbb{G}_{ShapesPsi} = \langle \mathbb{T}_{ShapesPsi} | Shapes | \Delta_{ShapesPsi} \rangle$$

being the tags set

$$\mathbb{T}_{ShapesPsi} = \{\text{Shapes}, \text{Circle}, \text{Rectangle}\},$$

tag root is Shapes, and $\Delta_{ShapesPsi} = \{t_i : \Delta_i | t_i \in \mathbb{T}_{ShapesPsi}\}$.

Its implementation follows the previously defined template shown in FRAGMENT 2. The Shapes tag has a compulsory attribute called *name* and contains *Circle* and *Rectangle* tags. The *Circle* tag has attributes to specify an *id*, the circle *center* and *radius* (10px by defect), a *label* and CSS *style*. The *Rectangle* tag has attributes to specify an *id*, the initial *point*, the *width* and *height* (both with 10px by default), a *label* and CSS *style*. FRAGMENT 2 shows the $\mathbb{T} \leftrightarrow \mathbb{C}$ association (as defined by (4)), the association between tags from *ShapesPsi* grammar and classes the *PsiComponent* for the *ShapesPsi* Language, using TEMPLATE 1. The full JavaScript code can be viewed at http://hilas.ii.uam.es/js/psi/Anisha.js. Additionally, the source code for Anisha case study

will be available in http://github.com/echavarriaga/Anisha.

FRAGMENT 2. The *ShapesPsi* language implementation (**Shapes** component) using the *PsiComponent* definition TEMPLATE 1.

```
var Shapes = (function () {
  // Tag set for the Anisha Grammar
  var ID_TAG_SHAPES = "shapes", ID_TAG_CIRCLE = "circle", ID_TAG_RECTANGLE = "rectangle";
  // Language structure
  var ID_TAGS = {};
  ID_TAGS[ID_TAG_SHAPES] = {
    TAG: "Shapes", CLASS: "Shapes", MULTIPLICITY: "1..1", VALIDATOR: "*name:text",
    CHILDREN: { "Circle": ID_TAG_CIRCLE, "Rectangle": ID_TAG_RECTANGLE }
  };
  ID_TAGS[ID_TAG_CIRCLE] = {
    TAG: "Circle", CLASS: "Circle", MULTIPLICITY: "0..n",
    VALIDATOR: "*name:identifier|*center:text|radius=10:integer|label:text|style:text"
  };
  ID_TAGS[ID_TAG_RECTANGLE] = {
    TAG: "Rectangle", CLASS: "Rectangle", MULTIPLICITY: "0..n",
    VALIDATOR: "*name:identifier|*point:text|width:10:integer|height=10:integer| label:text|style:text"
  };
  // ShapesPsi Grammar definition
  var GRAMMAR = { NAME: "ShapesPsi", TAGS: ID_TAGS, ROOT: ID_TAG_SHAPES };
  //Classes implementation
  function Shapes(){}; MiniProgram.methods({...}); //Shapes Class
  function Base(){}; Base.inherits(PsiXML.PsiElement); Base.methods({...}); //Base Class
  function Circle(){}; Circle.inherits(Base); Circle.methods({...}); //Circle Class
  function Rectangle(){}; Rectangle.inherits(Base); Rectangle.methods({...});//Rectangle Class
  function ParseShapesPsi(){}; //Parser Shapes Class
  // PsiLanguage Register
  PsiXML.registerLanguagePsi("ShapesPsi", GRAMMAR, ParseShapesPsi);
  return {
    // PsiComponent interface
  };
})();
```

Once we have shown how to implement the JavaScript functionality for the Shapes Component, we can illustrate how to use the *PsiLI* and *PsiCA* languages. To do so, we will detail how to specify two new programmable tags (Circle and Rectangle) for the *ShapesPsi*. We will make use of two information sources: an XML document and a JavaScript object.

What follows is a simple game based on two characters. First, FRAGMENT 3 shows how the information from characters is extracted from a XML file. Second, FRAGMENT 4 presents how the *Context* object is defined by the characters' features and the XML file (FRAGMENT 3) is loaded and registered by *PsiData*.

FRAGMENT 3. File "characters.xml"

```
<?xml version="1.0" encoding="utf-8"?>
<Characters>
  <Character id="p1" first="Luke" last="Skywalker"/>
  <Character id="p2" first="Obi-Wan" last="Kenobi"/>
</Characters>
```

FRAGMENT 4. Context for people from file "character.xml"

```
var context = {
  features: {
    p1: {point: "50,50", color: "black"},
    p2: {point: "250,200", color: "green"}
  }
}
PsiData.document.register("characters", "xml/characters.xml");
```

FRAGMENT 5 shows a *ShapesPsi* Program based on *PsiLI* and *PsiCA* languages. The Circle tag defines the data-document attribute, whose value is written in the *PsiLI* language and is interpreted by *PsiXML*. As a result, the *PsiData* object looks for a character with an *id* value equal

to `p1` (jQuery selector `Character[id=p1]`). It also assigns the result to the variable *info* that is added to the instance of the *Circle* class. Additionally, the *data-context* attribute recovers the property *features* from the *Context* object, identified as `p1`, and assigns this value to the variable *qt* of the class instance. It is worth mentioning that *data-context* (*Context* object information) and *data-document* (*PsiData* information) attributes are recorded, and both are natively available for all tags in any *PsiLanguage*.

Moreover, FRAGMENT 5 shows how the *label* attribute, from the Circle tag, uses the *PsiCA* language in order to obtain information from *info*. In same way, the *center* and *style* attributes use the *PsiCA* language to obtain information from *qt*.

FRAGMENT 5. Using *PsiLI* and *PsiCA* languages in *ShapesPsi* programs (characters.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<Shapes name="the characters ...">
  <Circle data-document="characters:info=Character[id=p1]" data-context="features:qt=p1"
       id="cp1" center="{{qt.point}}" radius="30" label="{{info.first}}"
       style="stroke:black;stroke-width:1px;fill:{{qt.color}}"/>
  <Rectangle data-document="characters:info=Character[id=p2]" data-context="features:qt=p2"
       id="cp2" center="{{qt.point}}" width="60" height="40" label="{{info.first}}"
       style="stroke:black;stroke-width:1px;fill:{{qt.color}}"/>
</Shapes>
```

At this point, we highlight again the capability to associate heterogeneous information sources (XML or JSON) with a program element natively including *PsiLI* and *PsiCA*. The class bound to the element receives the information and processes it according to its needs. This enriches programming, because an attribute of the element can be transformed into an Object or DOM element.

- **The *AnimePsi* Language**

Continuing the *Anisha* running example, Fig. 5 shows the implementation of *AnimePsi* language that allows moving figures defined with *ShapesPsi*. This language has two tags: *Animates* and *Move*. In Fig. 5(a) we can see the *PsiLSD* for the *AnimePsi* language and Fig. 5(b) shows its *PsiGVA*.

According to Fig. 5(a), the *Animates* root tag specifies a set of figures movements, i.e. multiple *Move* tags. The *Move* tag defines a reference to the figure (*key* attribute), a path for the movement (*path* attribute), the number of steps in the *path* (*steps* attribute), the number of seconds for movement (*seconds* attribute), the initial waiting time (*wait* attribute, default 0) and if the movement must be repeated (*repeat* attribute, default "no").

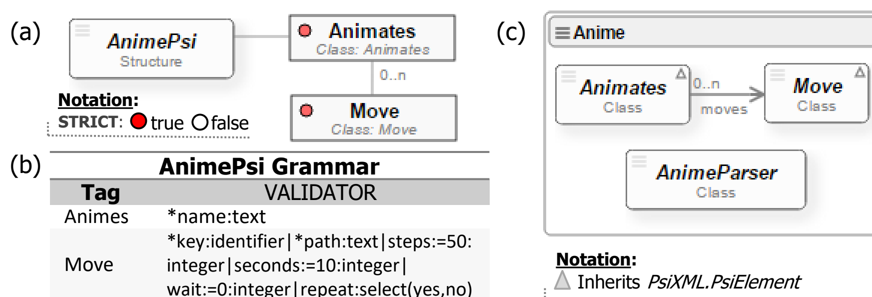


Fig. 5. (a) *PsiLSD* for the *AnimePsi* Language. (b) *PsiGVA* for the *AnimePsi* Language. (c) Class diagram for the **Anime** Component.

The **Anime** component implements the functionality related to the *AnimePsi* language, as shown in the class diagram of Fig. 5(c). This component has two classes: *Animates* and *Move*, bound to the *Animates* and *Move* tags respectively. The Animates class starts the animation program, while the *Move* class implements the movement bound to the *Move* tag and based on the definition of the SVG *path* tag. The reader is encouraged to look up the detailed implementation of the Shapes Component at http://hilas.ii.uam.es/Anisha/api.

In FRAGMENT 6 we can see the characters motion defined in the previous fragment. In

evaluating this *AnimePsi* program, "Luke" approaches "Obi-Wan" that runs off. Evaluation can be seen in http://hilas.ii.uam.es/Anisha/runs.

FRAGMENT 6. Game for moving characters written through the *AnimePsi* language

```
<?xml version="1.0" encoding="utf-8"?>
<Animates name="runs, Obi-Wan, runs">
  <Move key="cp1" steps="50" seconds="3" path="L100,50L180,200"/>
  <Move key=" key="cp2" steps="50" seconds="5" wait="4"
        path="F250,100 400,100 400,200 S550,300 550,200 F550,50 100,50 100,200 L100,300
L250,300 l250,200" />
</Animates>
```

The main objective of the *Anisha* case study has been to illustrate a few features of *PsiEngine*. Firstly, this example validates the binding of the heterogeneous source information (XML and JSON). This feature provides versatility to the XML-DSLs in *PsiEngine*. Secondly, *Anisha* tests the coexistence of multiple *PsiLanguages* as well as the ability to run multiple *PsiPrograms* with rapid information exchange and cross functionality. In general, *PsiEngine* allows for the creation of straightforward XML-DSLs with a high level of abstraction, which are extensible and combinable.

3.2 The Psi Programming Model and Its Lightweight Environment

Currently, code generation is a core activity in software engineering with particular focus in Model-driven Engineering [49]. Code generation leads to timesaving, greater efficiency, higher quality and more standardization when building information systems [44][49]. In this context, utilizing the approach as previously described, to simplify the creation of *PsiLanguages* $\mathbb{L}$, we propose the *PsiModel*. The *PsiModel* allows for the definition of *PsiGrammars* $\mathbb{G}$ and the implementation of the *PsiComponents* $\mathbb{K}$, as well as other JavaScript components.

The *PsiModel* separates the *PsiLanguage* specification from the *PsiComponent* implementation and their association by means of code-behind techniques. To do so, the *PsiModel* involves two *PsiLanguages* as shown in Fig. 6, namely:

- *MPsi Specification Language:* a *PsiLanguage* to specify the programming elements, i.e. the programmable tags.
- *MIPsi Implementation Language:* a *PsiLanguage* to implement the programming elements specified by means of the *MPsi* language.

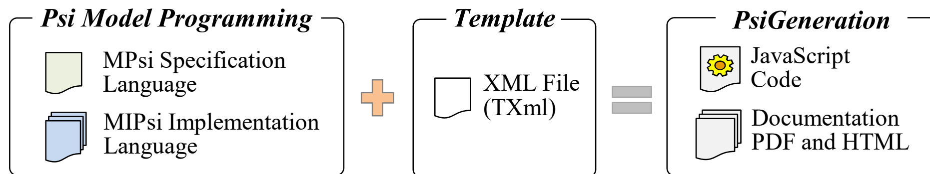


Fig. 6. *PsiModel* Scheme to generate JavaScript code and documentation.

Using this method, *MPsi* and *MIPsi* respectively allow for the specifying and implementation of the programming elements including: *Var*, *Object*, *Enum*, *Script*, *Class* (using the class definition given in [8][58]), *Component* (using the component or module definition given in [8][58]) and *ComponentPsi* (which defines and implements *PsiGrammar* and *PsiComponent*, to generate a *PsiLanguage*). For further details on the specification and implementation of *MPsi* and *MIPsi* languages, please consult http://hilas.ii.uam.es/psimodel.

In order to apply the *PsiModel* including *MPsi* and *MIPsi* while building *PsiLanguages*, we have implemented the lightweight development environment called *PsiEnvironment*. In Fig. 7 we can see a snapshot of the *PsiEnvironment* accessible with the URL:

http://hilas.ii.uam.es/PsiXML/viewer.html?**psifile**=<*name*> (5)

being <*name*> the name of the project or component. Additionally, the source code for each project or component will be available in http://github.com/echavarriaga/PsiSource/.

As an example, in Fig. 7 the *Anisha* case study is shown (psifile=*Anisha*). The "*Specification*" tab contains the *MPsi* specification, whilst the "*Implementation*" tab contains the *MIPsi* implementation.

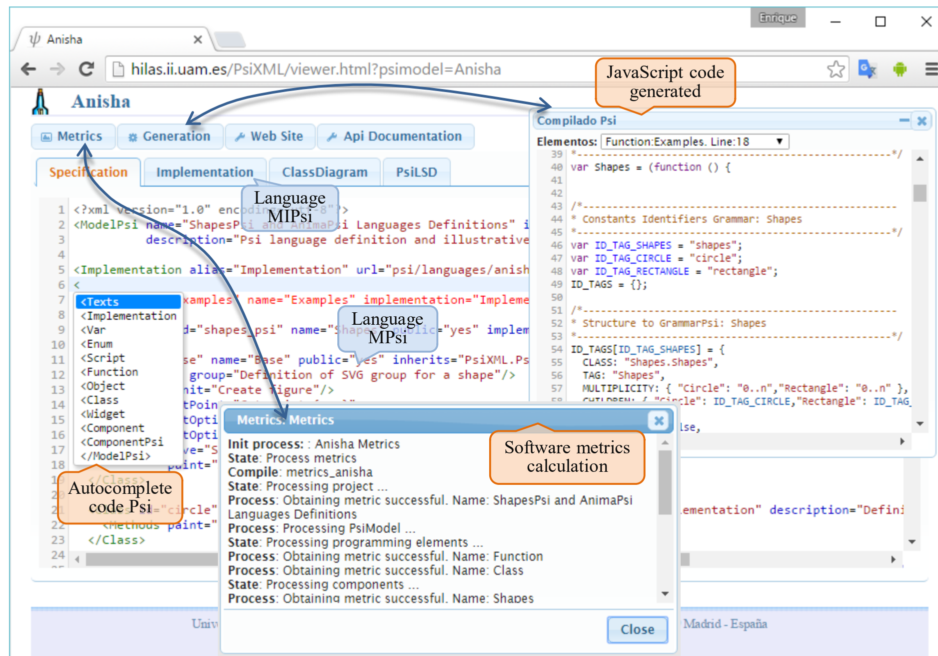


Fig. 7. Lightweight development environment to apply the *PsiModel*.

The *PsiEnvironment* implements several features including code autocompleting for *MPsi*, *MIPsi*, JavaScript, XML, HTML and CSS languages. It also has a visual component for online display of *PsiLSD* diagrams, UML Class diagrams as well as source code. Once the *PsiLanguage* is defined and implemented, the *PsiEnvironment* can automatically generate the JavaScript code.

An additional feature in *PsiEnvironment* is the process of calculating of a set of software metrics obtained for the generated JavaScript code and for the *PsiModel* languages. The software metrics are: lines of code [40] (SLOC, LLOC and CLOC), cyclomatic complexity [35], Halstead metrics [20] and maintainability index [41]. To perform these calculations we made use of Excomplex [48], a package implemented on NodeJS (nodejs.org).

3.3 Final Comments

Throughout this section we have presented a method for creating XML-DSLs, which we have called *PsiLanguages*, using the *PsiEngine*. With the *Anisha* running example we have illustrated the creation of *PsiLanguages* along with the relevant features of the *PsiEngine*. Additionally we have presented the *PsiModel*, a programming model and its corresponding lightweight programming environment to facilitate the language specification and *PsiCompoment* implementation. The design and implementation for the *PsiComponents* can be summarized according the following steps:

i. Create the *PsiLSD* and specify *PsiGVA*.
ii. Create the UML Class Diagram for the *PsiComponent*.
iii. Implement the *PsiComponent* functionality based on the *PsiLSD* and the *PsiGVA* using the PsiEnvironment.
iv. Perform functional tests for the *PsiComponent.*

The following section is devoted to develop *FeedPsi*, a web application developed entirely with the proposed method. The final section is dedicated to validating the *PsiEngine* using a qualitative case study methodology.

4 *FeedPsi* web application case study

In order to test our approach, the aim of this case study is to illustrate the use of *PsiEngine* for the specification and implementation of a web application we have called *FeedPsi*. Briefly, it is a web application executed with the goal of creating and deploying a dynamic web page, which aggregates content from several web sources in order to build a News Portal. These sources can be formatted in the two most commonly used formats, namely: RSS [57][22] and Atom [56][24] both written in XML.

Throughout this section, we will present the complete cycle from problem analysis to implementation while utilizing the *PsiEngine*. This cycle begins with how to build the *FeedPsi Language* and encapsulate it in a *PsiComponent*, followed by, evaluating the *PsiProgram* and how the dynamic website is generated as a result.

4.1 Analysis and design

In Fig. 8 we can see the design of the *FeedPsi* web application. It manages RSS feeds, allowing us to choose, display, store by category and/or dispose news. In addition, the figure shows the *FeedPsi* Language, which is implemented in *Feed Component* and evaluated on the *Interpreter PsiXML* shown. *FeedPsi* will need to manage the list of RSS feeds (RSS multiple tags), the list of categories (multiple Category tags) and the list of current news selected by the user (one *MainNews* tag). When a RSS feed is read, the headlines are available in a floating submenu (see Fig. 8). Once labelled, news can be displayed and/or stored in any category for later reading.

The proposed web application does not require server-side programming, making it possible for the entire application deployment to take place directly on the web-client, with minimal computational cost on the server.

In order to better exemplify the use of our proposal, the following section will show another case of study where the approach has been used, namely, *FeedPsi*, a dynamic web application completely developed using our approach.

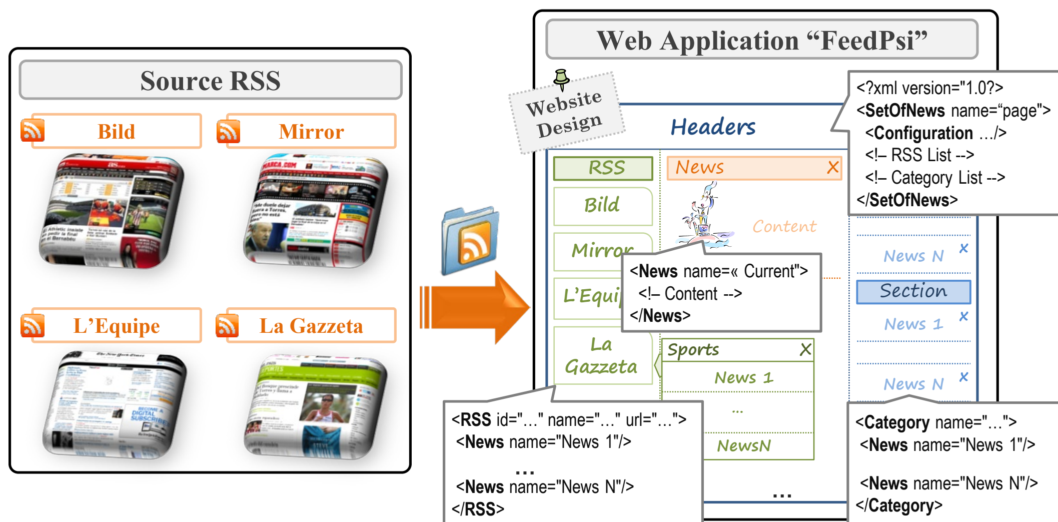


Fig. 8. Design of a web application to aggregate news via RSS feeds, based on *PsiLanguage*.

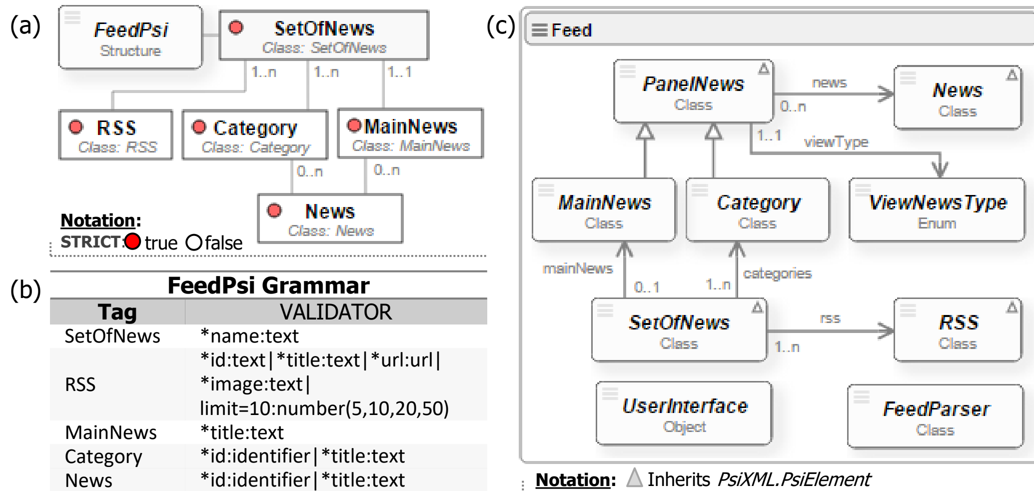
4.2 Feed Component Specification

The implementation of the web application for managing and displaying RSS feeds following our DSL approach starts with the specification of the corresponding *Feed Component* as previously stated. Next, we will detail the steps.

Fig. 9 shows the *PsiLSD* and Fig. 9(b) shows the *PsiGVA* for the *FeedPsi* Language. The *SetOfNews* tag is the element root in the *FeedPsi Program* evaluation. It contains the list of RSS sources (*RSS* tag), the list of Categories (*Category* tag) and the news currently displayed

(*MainNews* tag).

The *RSS* tag has an identifier, a name and the URL for the RSS source. It has the capability to connect via AJAX to fetch the data from the RSS news service. The *Category* tag has lists of news stored by category. It has an identifier and the category title. The *MainNews* tag contains list of news currently displayed to the user. Finally, the *News* tag contains the news obtained from the RSS feed, which has a unique identifier and it is displayed if it falls under the *MainNews* tag.



FeedPsi Grammar

Tag	VALIDATOR
SetOfNews	*name:text
RSS	*id:text|*title:text|*url:url|*image:text|limit=10:number(5,10,20,50)
MainNews	*title:text
Category	*id:identifier|*title:text
News	*id:identifier|*title:text

Fig. 9. (a) *PsiLSD* of the *FeedPsi* Language. (b) *PsiGVA* of the *FeedPsi* Language.
(c) Class diagram for the Feed Component.

Fig. 9(c) shows the class diagram for the Feed Component. The detailed implementation of the Feed Component is shown at the URL: http://hilas.ii.uam.es/FeedPsi/api, by class diagrams and documentation tabs. The full JavaScript code for the Feed Component can be accessed at http://hilas.ii.uam.es/js/psi/*FeedPsi*.js. Additionally, the source code will be available in http://github.com/echavarriaga/FeedPsi.

In the class diagram (Fig. 9(c)) we can see the functionality associated with the set of news (*SetOfNews* class) where we can add and remove both RSS feeds and categories. Each RSS source (*RSS* class) has the ability to update its content making an AJAX call by using the associated URL, creating a news entry, and creating a sub-cathegory of headlines with the help of the *SetOfNews* class.

In addition, categories (*Category* class) and current displayed news (*MainNews* class) can be managed (add, delete or verify) by using the *PanelNews* class. The *MainNews* class has been implemented to visualize the news chosen from the RSS sources or from the different categories. Meanwhile, the *News* class defines news with a unique *id* and it is in charge of displaying the news. The *UserInterface* is charged with processing and displaying the interface for the *FeedPsi* application.

4.3 Running the *FeedPsi* web application

In order to show how the Feed Component runs together with the *FeedPsi* Language to implement the ***FeedPsi*** web application (http://hilas.ii.uam.es/*FeedPsi*), we will show how it works with the RSS source for the following periodicals: *Bild, Mirror, AS, La Gazzeta, L'Equipe y El Tiempo* (see Fig. 10).

Thus, FRAGMENT 7 shows the *FeedPsi* source code that implements the *FeedPsi* web application (the news aggregator).

FRAGMENT 7. *FeedPsi* source code for sports (sports.xml)[1].

```
<?xml version="1.0" encoding="utf-8"?>
```

[1] http://hilas.ii.uam.es/FeedPsi/source/sports.xml

```
<SetOfNews name="RSS Sports Psi Style">
  <RSS id="as" title="Germany: Bild Sport" image="images/rss/bild.png"
       url="http://rss.bild.de/bild-sport.xml"/>
  <RSS id="theguardian" title="England: The Guardian"
       image="images/rss/mirrorsport.png"
       url="http://feeds.theguardian.com/theguardian/uk/sport/rss"/>
  <RSS id="as" title="Spain: AS" image="images/rss/as.png"
       url="http://as.com/rss/tags/ultimas_noticias.xml"/>
  <RSS id="lagazzetta" title="Italy: La Gazzetta" image="images/rss/lagazzetta.png"
       url="http://www.gazzetta.it/rss/home.xml"/>
  <RSS id="lequipe" title="France: L'Equipe" image="images/rss/lequipe.png"
       url="http://www.lequipe.fr/rss/actu_rss.xml"/>
  <RSS id="eltiempo" title="Colombia: El Tiempo" image="images/rss/eltiempo.png"
       url="http://www.eltiempo.com/deportes/rss.xml"/>
  <MainNews title="Latest Sports News"></MainNews>
  <Category id="football" title="Football"/>
  <Category id="cycling" title="Cycling"/>
  <Category id="tennis" title="Tennis"/>
  <Category id="motor" title="Motor"/>
</SetOfNews>
```

Fig. 10 shows the resulting web application after the *PsiXML* processes the source code. At http://hilas.ii.uam.es/FeedPsi/index.html?list=⟨name⟩ we can evaluate other *FeedPsi* programs, where ⟨*name*⟩ can be *science*, *spain*, *sport-es*, *technology* and *journal*.

Finally it should be noticed that the objective of the presented case study is not to deploy a fully debugged web application, but to better show how to use the concept of *PsiComponent*, *PsiLanguage* and the *PsiXML* for building web applications.

Fig. 10. *FeedPsi* Web Application.

5 Results and validation

To perform the validation of the *PsiEngine* we followed the qualitative case study methodology suggested in [59] and adapted for software engineering [2]. According to this methodology, a case study in a web-client is translated into one or more *PsiLanguages* developed to create a set of programs written in these languages that have the necessary resources to implement solutions to specific problems. Some results may include, for example, new components, widgets or web applications.

The multicase methodology described in [59] validates the relevant aspects of a research

objective. These objectives are formulated as a set of criterions to validate. This is done by means of case studies that covers different parts of the whole set. The total cases should cover the full criterions. A criterion can be validated with more than one case study. Overall, the evidence created from the multicase methodology is considered robust and reliable, but can also be extremely costly in time and execution requirements [2]. In this case our objective is to validate the *PsiEngine* and *PsiModel*, and so the criterions list to validate this are the following:

F1. Implement the *PsiEngine* and its *PsiXML*.

F2. Allow *PsiPrograms* execution: (S) simple-simple (one program of a *PsiLanguage*); (P) multiple-simple (multiple programs of a *PsiLanguage*); (M) multiple-multiple (multiple programs written with multiple *PsiLanguages*).

F3. Associate *PsiPrograms* with heterogeneous data-sources: (X) XML; (J) JSON.

F4. Provide objects and information sharing: (S) among different programs written in the same *PsiLanguage*; (M) among different programs written with different *PsiLanguages*.

F5. Implement a dynamic CSWA using *PsiProgram*(*s*).

F6. Implement the *MPsi* and *MIPsi* languages of the *PsiModel*.

F7. Define and implement the *PsiModel* within the *PsiEnviroment*: (C) JavaScript Component; (P) *PsiComponent* (development of *XML-DSLs*).

Criterions F1-F5 allow us to validate the *PsiEngine* while criterions F6 and F7 allow us to validate the *PsiModel*.

The next section will summarize and analyse several software metrics obtained for *PsiEngine* and it interpreter. The *PsiXML* together with the *Anisha* case study and the *PsiModel* will serve to validate the set of criterions listed above. Because the *PsiEnvironment* is based on the use of *PsiLanguages*, it is further treated as an additional case of study.

5.1 *PsiEngine*: validation of the implementation

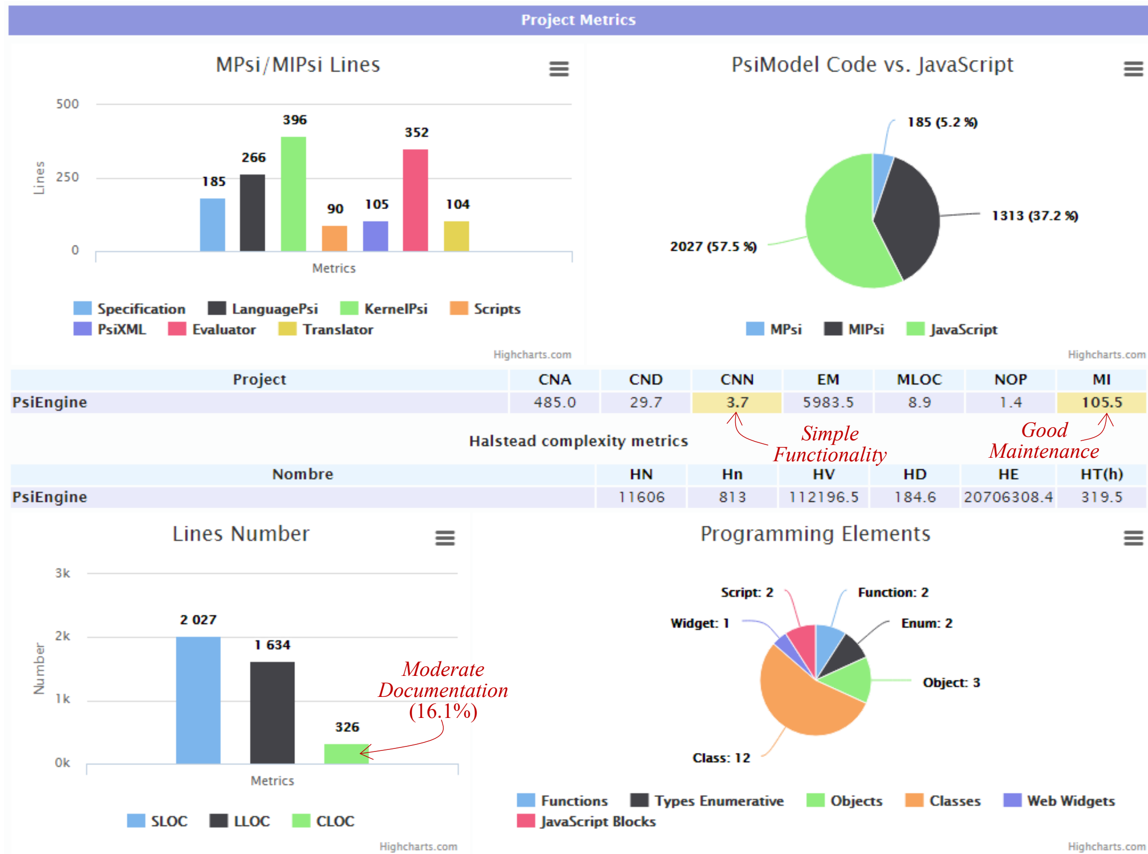
PsiEngine itself and its main component, the *PsiXML*, have also been defined and implemented by using the *PsiEnvironment* (use (5) being "psifile=*PsiEngine*" for project details). Table 1 summarizes the *PsiEngine* project files. There are 7 files written in *PsiModel* Languages, with up to 1,498 lines of code in *PsiLanguage* that generate a total of 2,027 lines in JavaScript in the *PsiEngine*.js file. This means a conciseness ratio of 1.4.

Table 1. *PsiEngine* project components summary.

	PsiModel				**JavaScript Generated code**		**Conciseness**
Components	MPsi	MIPsi	Diagram	PSILOC	File	SLOC	PSILOC/SLOC
PsiXML*, *PsiData*, *PsiLanguage	185(1)	1,313(6)	☑	1,498	PsiEngine.js	2,027	1.4

Note: X(Y) **MPsi**/**MIPsi**; X: Number of Psi lines, Y: Number of Psi files; **Diagram**: Class diagram; **PSILOC**: total Psi lines of code; **SLOC**: JavaScript generated lines.

Fig. 11 shows a snapshot of the *PsiEngine* metrics automatically generated by the *PsiEnvironment*. We would like to add that the average cyclomatic complexity CNN for the functions/methods of the project gives a value of "simple functionalities" (CNN<10 according to [35]), the Maintainability Index MI is appropriated (MI>85 according to [41]) and the commented lines of code CLOC are moderate (16.1%). Moreover, the time needed to implement or to understand a program rounds to 40 days (319.5 hours according to Halstead Time [20]).



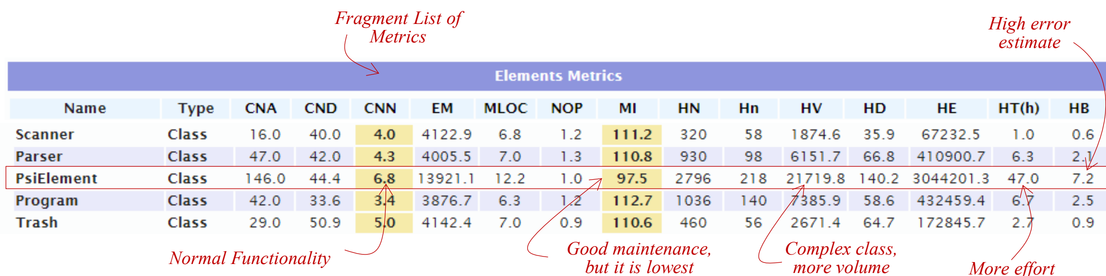
Project	CNA	CND	CNN	EM	MLOC	NOP	MI
PsiEngine	485.0	29.7	3.7	5983.5	8.9	1.4	105.5

Halstead complexity metrics

Nombre	HN	Hn	HV	HD	HE	HT(h)
PsiEngine	11606	813	112196.5	184.6	20706308.4	319.5

Fig. 11. *PsiEngine* metrics summary provided by the *PsiEnvironment*.

Fig. 12 shows a snapshot of the software metrics listed for some of the programming elements of the *PsiEngine* generated with *PsiEnvironment*. Delving into the results, the abstract class *PsiElement* of *PsiXML* is the most complex of the *PsiEngine* programming elements. This fact is predictable because it is the cornerstone of *PsiLanguages*, having the ability to process both *PsiLI* and *PsiCA* languages. CNN has an averaged functionality. Although with a good MI it is the one with the lowest value. *PsiElement* is the class that requires greater effort to be implemented. The error estimate exceeds the limit advised implying that a revision of the class is needed. By and large, the *PsiEngine* presents averaged numbers regarding cyclomatic complexity and error estimates, both metrics are within acceptable limits, which validates the implementation facts of F1.



Elements Metrics

Name	Type	CNA	CND	CNN	EM	MLOC	NOP	MI	HN	Hn	HV	HD	HE	HT(h)	HB
Scanner	Class	16.0	40.0	4.0	4122.9	6.8	1.2	111.2	320	58	1874.6	35.9	67232.5	1.0	0.6
Parser	Class	47.0	42.0	4.3	4005.5	7.0	1.3	110.8	930	98	6151.7	66.8	410900.7	6.3	2.1
PsiElement	Class	146.0	44.4	6.8	13921.1	12.2	1.0	97.5	2796	218	21719.8	140.2	3044201.3	47.0	7.2
Program	Class	42.0	33.6	3.4	3876.7	6.3	1.2	112.7	1036	140	7385.9	58.6	432459.4	6.7	2.5
Trash	Class	29.0	50.9	5.0	4142.4	7.0	0.9	110.6	460	56	2671.4	64.7	172845.7	2.7	0.9

Fig. 12. Software metrics snapshot for the programming elements of the *PsiEngine* generated by *PsiEnvironment*.

5.2 Results for case studies

Table 2 details the data computed by the *PsiEnvironment* for the *Anisha, FeedPsi* and *PsiModel* case studies. In the table two discernible parts can be distinguished: the *PsiModel* and the corresponding generated JavaScript code. By using the URL (5) replacing <*name*> with the name of the case study we can see the *PsiModel* code, the UML Class diagram and the *PsiLSD* diagram,

and can even generate the JavaScript code to analyse the corresponding software metrics. It is worth highlighted that the *PsiModel* serves as a case study and it is autogenerated in order to be used within the *PsiEnvironment*.

Table 2. *PsiEngine* case studies list.

Case Study	***PsiModel***			**JavaScript generated code**			**Conciseness**
	MPsi	MIPsi	PSILOC	Fichero	CLOC	SLOC	SLOC/PSILOC
Anisha	60 (1)	157 (1)	217	*Anisha*.js	20.6%	510	2.4
FeedPsi	54 (1)	160 (1)	214	*FeedPsi*.js	18.8%	505	2.4
PsiModel	341 (1)	825 (6)	1,166	MPsi.js	17.0%	2,339	2.0
Total	455 (3)	1,142 (8)	1,597		19.1%	3,354	2.1

Note: X(Y) **MPsi/MIPsi**; X: Number of Psi lines, Y: Number of Psi files; **PSILOC**: Total Psi lines of code; **SLOC**: JavaScript generated lines. **CLOC**: Comments percentage.

Summarizing the three projects of *PsiComponents* analysed, we developed 11 *PsiModel* files. Of these 11, 27% (3 files) are Psi specification files and the remaining 73% (8 files) are Psi implementation files. We wrote a total of 1,597 lines of *PsiModel* source code that generated a grand total of 3,354 lines of JavaScript code. This means a conciseness ratio of 2.1, whilst *Anisha* and *FeedPsi* have a conciseness ratio of 2.4. Lines of comments represent 19.1% (average for the three case studies), which is a moderate rate.

In Fig. 13 we summarize some results of the metrics computed for the *Anisha* and *FeedPsi* case studies along with the *PsiModel* component. The averaged cyclomatic complexity CNN is low for the three cases (CNN<10), which means a "simple functionality". The Halstead Time HT needed to implement or to understand the *PsiModel* is greater than the corresponding time for the rest, due to its complexity and functionality. Its program duration (11,703 HN) and vocabulary size (594 Hn) is six times greater than the average length and vocabulary of *Anisha* and *FeedPsi* case studies. The average estimated errors HB is under the recommended limits (<2). The class *StructureElement* of the *PsiModel* is the only that overtake this limit (2.1 HB).

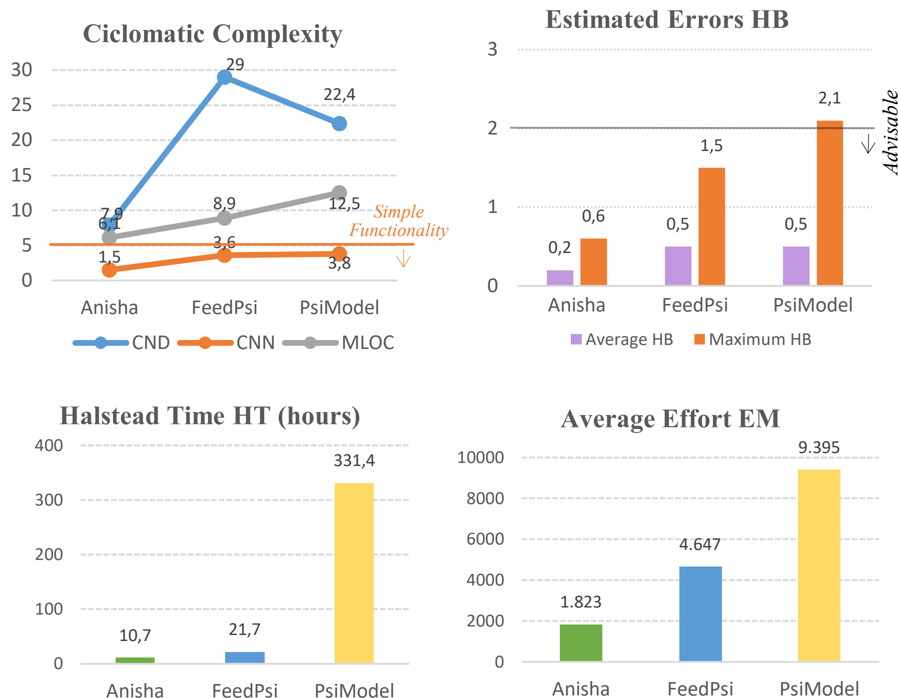


Fig. 13. Summary of metrics for the *Anisha* and *FeedPsi* case studies and the *PsiModel* component.

Finally, Fig. 14 displays the Maintainability Index MI for the *Anisha*, *FeedPsi* and *PsiModel* case studies in addition to those of the components and frameworks used for these cases. As can be seen, all components and frameworks have good Maintainability (>85). The components developed

from *PsiModel*, such as *PsiEngine* and *FeedPsi*, are at the same level as renowned frameworks like CKEditor, Codemirror and Highcharts. The computing of the MI metric for different frameworks (CKEditor, Codemirror, Highcharts and jQuery) has been made with the JSComplexity tool (http://jscomplexity.org).

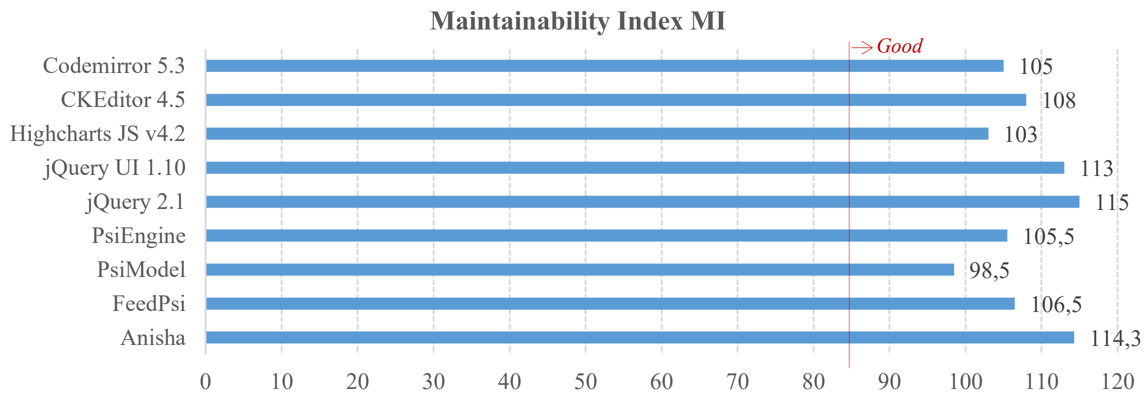


Fig. 14. MI for *Anisha*, *FeedPsi* and PsiModel case studies and components and frameworks used.

In summary, the case studies and the *PsiModel* accomplish the recommended values for the software metrics as specified in the literature. In addition, with these case studies and the *PsiModel* we have validated functionalities F6 and F7.

5.3 Analysis of results

The *Anisha* case study validates the *PsiPrograms* execution (F2) twofold: (P) multiple-simple (multiple programs of a *PsiLanguage*); (M) multiple-multiple (multiple programs written with multiple *PsiLanguages*). For the (P) case we wrote four programs *ShapesPsi,* each of which draws a circle or a square (http://hilas.ii.uam.es/Anisha/ms). For the (M) case we used the aforementioned four programs along with four additional programs AnimePsi. The result is a figures race in the web-client (http://hilas.ii.uam.es/Anisha/mm). In addition, *Anisha* also validates the *Heterogeneous information association* XML and JSON (F3) by means of the *PsiLI* language, as shown in section 3.3. (F4) is also verified because *ShapesPsi* and *AnimePsi* programs share objects and information between them.

Otherwise, the case study *FeedPsi* validates the simple-simple (one program of a *PsiLanguage*) *PsiProgram* execution (F2, type S) and the web application dynamic generation based on *PsiProgram*(*s*) (F5), by means of the Feeds website. Moreover, it allows for the association of dynamic information with RSS (F3, type X) existing as a fully server independent application. Table 3 summarizes the number of lines needed for the *FeedPsi* application. A total of 1,151 lines of code comprising static code (66%) HTML, CSS, XSL, XML and JS source code (excluding the lines of *PsiEngine* and jQuery), and 44% generated code from the PsiModel. At this point it should be emphasized that the conciseness ratio of a *FeedPsi* program is about 52. This means that a *FeedPsi* program it seems to be more of a configuration file than a common program. *FeedPsi* is a novel case study that represents a high level XML-DSL to produce a website web application.

Table 3. *FeedPsi* source code lines summary.

	Application *FeedPsi*						***FeedPsi* Program**	**Conciseness**
	Static Code				Generate Code	**Total**		
	HTML	CSS	XML	XSL	*FeedPsi*.js	#LP	#LP	#LC/#LP
Files	1	1	1	4	1	8	1	
Lines	52	78	246	270	505	1.151	22	**~ 52**

Note: **#LC**: Source code lines (HTML+CSS+XSL+XML+JS); **#LP**: *FeedPsi* source code lines.

Finally, the *PsiModel* component constantly validates both the *PsiEngine* and interpreter

criterions set (F1, F2, F3, F4 and F5).

5.4 Validation summary

Table 4 summarizes the *PsiEngine* set of criterions set validations. F1 criterion has been validated threefold: by means of the software metrics, the through *Anisha,* and the *FeedPsi* case studies and *PsiModel* component that validate it behaviour.

F6 and F7 validation requires using *PsiModel* and its associated *PsiEnvironment*. To do so we developed JavaScript components (C=3) and *PsiComponents* (P=5).

The three case studies validate the three possible ways to execute *PsiPrograms* including on the *PsiEngine* (F2), heterogeneous information association (F3) and information sharing (F4). Moreover, we create the *FeedPsi* case study to validate the dynamic generation of web applications deployed completely through a *PsiProgram*.

In conclusion, helped by the qualitative methodology multicase suggested in [59] and adapted by [2], and the criterions validation summary shown in Table 4, we validated the goals *PsiEngine* and *PsiModel*.

Table 4. The *PsiEngine* and *PsiModel* criterions list and the validations summary.

	Criterions	***PsiEngine***	***Anisha***	***FeedPsi***	***PsiEnviroment***
F1	Implement the *PsiEngine* and its *PsiXML*	☑	☑	☑	☑
F2	Allow *PsiPrograms* execution: (S) simple-simple (one program of a *PsiLanguage*); (P) multiple-simple (multiple programs of a *PsiLanguage*); (M) multiple-multiple (multiple programs written with multiple *PsiLanguages*).	☒	PM	S	M
F3	Associate *PsiPrograms* with heterogeneous data-sources: (X) XML; (J) JSON	☒	XJ	X	☒
F4	Provide objects and information sharing: (S) among different programs written in the same *PsiLanguage*; (M) among different programs written with different *PsiLanguages*.	☒	SM	N/A	SM
F5	Implement a dynamic a dynamic CSWA using *PsiProgram*(s).	☒	☒	☑	☑
F6	Implement the *MPsi* and *MIPsi* languages of the *PsiModel*.	☒	☒	☒	☒
F7	Define and implement the *PsiModel* within the *PsiEnviroment*: (C) JavaScript component; (P) *PsiComponent* (development of *XML-DSLs*).	C=3	P=2	P=1	P=2

6 Conclusions

Domain-Specific Languages allow for the building of software applications with greater speed by increasing the productivity of both software engineers and domain experts thanks to the level of abstraction they provide. Building a DSL solution involves the use of tools for the implementation of interpreters and compilers. However, as we have shown, there are few existing approaches able to build client-side web DSL alternatives.

To address this, alongside this article, we have formalized and validated the *Programmable Solutions Interpreter* (Psi), our approach to work with XML-based DSL solutions for CSWA, which includes an evaluation engine, a programming model and a lightweight development environment. Thus, we have detailed the *PsiEngine* and described its main component: the *PsiXML*. *PsiEngine* combines HTML5, CSS3, JavaScript and DOM languages, with Web 2.0 technologies, services and tools, and the definition of a kind of XML-DSLs, which we called *PsiLanguages*. This combination permits *PsiEngine* to build web components, web widgets and/or dynamic web pages that provide solutions to specific problems in web applications. Whilst, *PsiXML* is an interpreter able to evaluate programs coded in XML-DSLs to provide solutions to specific domain problems within a CSWA.

To facilitate the implementation of *PsiLanguages* following our approach and using the *PsiEngine,* we have created the *PsiModel* based on two *PsiLanguages*: *MPsi* (specification

language) and *MIPsi* (implementation language). Additionally, to more accurately apply the *PsiModel* we implemented the *PsiEnvironment*, a lightweight development environment that includes useful features such as code autocompleting, software metrics computation, diagrams displaying, etc.

To better demonstrate the capabilities and potential of our approach, we have presented two case studies to detail the whole process (analysis, design and implementation) of developing web applications, including reusable components creation, language specification and combination, the use of frameworks, etc. following a DSL perspective. The first case study *Anisha* implements two languages and has been used as a running example throughout the explanation while exploring the different concepts related to our approach. The second case study, *FeedPsi*, a web application for RSS aggregation, has demonstrated that it is possible to develop client-side web applications dynamically by using our approach. Moreover, as a conclusive proof we have implemented the *PsiEngine* itself by using the *PsiModel* in order to validate it.

Therefore, we have probed how our approach allows for the building XML-based DSL solutions for CSWA by using *PsiLanguages* specification. In addition, once the *PsiEngine* is running the *PsiLI* and *PsiCA* languages make possible to define and deploy new DSL solutions as required.

With the *PsiEngine* and the *PsiModel*, we want to establish the foundations to apply the MDE paradigm to build CSWA, so that every engineer can take advantage from this paradigm.

As future works, we are building a set of *PsiLanguages* to build graphical models from serialized DSVL for CSWA. They have the capability to generate diagrams automatically with heterogeneous XML/JSON data sources and synchronizing information among diagrams and data sources.

Acknowledgments

This work has been partially supported by the DSVL-B2T research and development department from the B2T-Concept Company (http://www.b2tconcept.com/).

References

[1] Anderson, P. Web 2.0 and Beyond: Principles and Technologies. London: Chapman and Hall/CRC, 2012.

[2] Baxter P, Jack S. Qualitative Case Study Methodology: Study Design and Implementation for Novice Researchers. The Qualitative Report;2008:13-4.

[3] Betinni L. Implementing Domain-Specific Languages with Xtext and Xtend. Birmingham: Packt Publishing Ltd; 2013, p. 1-11.

[4] Brown D, Levine J, Mason T. Lex & Yacc. 2nd ed. New York: O'Reilly Media, Inc; 1992.

[5] Bravenboer M, Trygve K, Vermaas R, Visser E. Stratego/XT: A language and toolset for program transformation. Science of Computer Programming. 2008; 72(1–2), p.52-70. http://dx.doi.org/10.1016/j.scico.2007.11.003.

[6] Carter Z. JS sequence diagrams. http://jison.org; 2009 [accessed 30.10.16].

[7] Cook S, Jones G, Kent S, James D. Domain-Specific Development with Visual Studio DSL Tools. Denver: Addison-Wesley Professional; 2007, p. 1-23.

[8] Crockford D. JavaScript: The Good Parts. Sebastopol: O'Reilly Media, Inc; 2008, ch. 5.

[9] Czarnecki K, Eisenecker U. Generative Programming: Methods, Tools and Applications. Denver: Addison-Wesley; 2000.

[10] Da Silva AR. Model-driven engineering: a survey supported by a unified conceptual model. Computer Languages, Systems & Structures. 2015; 43, p.139-155.

[11] Dearle F. Groovy for Domain-Specific Languages. Birmingham: Packt Publishing Ltd; 2010, ch. 1.

[12] Erdweg S, Giarrusso PG, Rendel T. Language composition untangled. Proceedings of the 12th Workshop on Language Descriptions, Tools, and Applications, LDTA 2012.

[13] Fawcett J, Quin L, Ayers D. Beginning XML. 5th ed. Wrox Press; 2012.

[14] Fanagan D, Matsumoto Y. The Ruby Programming Language. Sebastopol: O'reilly Media, Inc.; 2005, ch. 8.

[15] Firmenich S, Rossi G, Winckler M, Palanque P. An approach for supporting distributed user interface orchestration. International Journal of Human-Computer Studies, 2014;72:1: p. 53-76.

[16] Fowler, M. Domain Specific Languages. Denver: Addison-Wesley Professional; 2010, p. 21-27.

[17] Ghosh, D. DSLs in Action. Greenwich: Manning Publications, 2010, p. 9-15.

[18] Greenfield J, Short K. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. New Jersey: Wiley Publishing; 2004.

[19] Gronback RC. Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Denver: Addison-Wesley Professional; 2009.

[20] Halstead M. Elements of Software Science. New York: The Computer Science Library; 1977.

[21] Herzberg D et al. Specifying computer-based counseling systems in health care: A new approach to user-interface and interaction design. Journal of Biomedical Informatics. 2009; 42:2: p. 347-355.

[22] Holzner S. Secrets of RSS. Berkeley: Peachpit Press.; 2006, ch. 3.

[23] Hudak P. Building domain-specific embedded languages. ACM Comput. 1996;196:4.

[24] Johnson D. RSS and Atom in Action: Web 2.0 Building Blocks. Greenwich: Manning Publications Co.; 2006, p. 70-77.

[25] Kelker R. Clojure for Domain-specific Languages. Birmingham: Packt Publishing Ltd.; 2013, ch. 1.

[26] Kern C. Securing the tangled web. Communication ACM 2014;57:9: p. 38-47.

[27] Kats L, Kalleberg K, Visser E. Domain-Specific Languages for Composable Editor Plugins. Elsevier: Proceedings of the Ninth Workshop on Language Descriptions, Tools, and Applications (LDTA'09), Electronic Notes in Theoretical Computer Science 2009;253:7, p. 149-163.

[28] Kelly S, Tolvanen J. Domain-Specific Modeling: Enabling Full Code Generation. Wiley-IEEE Computer Society Press 2008.

[29] Kniesel G, Winter V, Siy H, Zand M. Making aspect-orientation accessible through syntax-based language composition. IEEE: IET Software 2009;3:1, p. 1-13.

[30] Kourie, D.G., Fick, D. & Watson, B.W. (2008). Virtual machine framework for constructing domain-specific languages. *IET Software*. IEEE, 3, 3, 219-237.

[31] Kosar T, et al. Comparing general-purpose and domain-specific languages: An empirical study. Computer Science and Information Systems 2010;7:2, p. 247–264.

[32] Kosar T, Bohra S, Mernik M. Domain-Specific Languages: A Systematic Mapping Study. Information and Software Technology 2016;71, p. 77-91.

[33] Lee P. CoffeeScript in Action. Greenwich: Manning Publications Co.; 2014, ch. 1.

[34] Levine J. Flex & Bison. Sebastopol: O'Reilly Media, Inc.; 2009.

[35] McCabe T. A Complexity Measure. IEEE Transactions on Software Engineering 1976;SE-2:4, p.308-320.

[36] McDaniel A. HTML5: Your visual blueprint™ for designing rich web pages and applications. Indianapolis: Jhon Wiley & Sons, Inc.; 2011, ch. 3-8.

[37] McGuire P. Getting Started with Pyparsing. Sebastopol: O'Reilly Media, Inc.; 2007.

[38] Mernik M, Heering J, Sloane AM. When and how to develop domain-specific languages.

ACM Comput. Surv. 2005;37:4, p. 316–344.

[39] Mernik M, Lenic M, Avdičaušević E, Zumer V. LISA: An Interactive Environment for Programming Language Development. Springer Berlin Heidelberg, Lecture Notes in Computer Science 2002;2304, p. 1-4.

[40] Nguyen V, Deeds-Rubin S, Tan T, Boehm B. A SLOC Counting Standard. University of Southern California, Center for Systems and Software Engineering, http://sunset.usc.edu/csse/TECHRPTS/2007/usc-csse-2007-737/usc-csse-2007-737.pdf; 2007 [accessed 13.06.16].

[41] Oman PW, Hagemeister J, Ash D. A Definition and Taxonomy for Software Maintainability. Moscow: Technical Report, University of Idaho, Software Engineering Test Laboratory; 1991.

[42] Parr T. The Definitive ANTLR 4 Reference. 2nd ed. Raleigh: Pragmatic Bookshelf; 2013, part 1.

[43] Pereira MJ, Fonseca J, Henriques PR. Ontological approach for DSL development. Computer Languages, Systems & Structures. 2016; 45:1, p.35-52.

[44] Prout A, Atlee JM, Day NA, Shaker P. Code generation for a family of executable modelling notations. Software and Systems Modeling; 2012;11:2, p. 251-272.

[45] Rahien A. DSLs in Boo: Domain-Specific Languages in .NET. Greenwich: Manning Publications Co.; 2010, ch. 3-4.

[46] Sajja PS, Akerkar R. Intelligent Technologies for Web Applications. Minneapolis: CRC Press; 2012, ch. 1.

[47] Serrano, A. Beginning Haskell A Project-Based Approach. New York: Apress; 2014, part 4.

[48] Stilwell J. npm: Escomplex, v 1.3. https://www.npmjs.com/package/escomplex; 2014 [accessed 15.07.16].

[49] Schmidt DC. Model-driven engineering. Computer-IEEE Computer Society 2006;39:2:25.

[50] Shim H, Kang B, Kwag K. Web2Animation - Automatic Generation of 3D Animation from the Web Text. IEEE/WIC/ACM: Proceedings of the 2009 International Joint Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT '09).

[51] Spinellis D. Notable design patterns for domain-specific languages. Journal of Systems and Software 2001;56:1, p. 91-99.

[52] Voelter M, et al. DSL Engineering: Designing, Implementing and Using Domain-Specific Languages. Dslbook.org; 2013, p. 23-38.

[53] Document Object Model (DOM) Level 3 Core Specification, http://www.w3.org/TR /2004/REC-DOM-Level-3-Core-20040407/; 2004 [accessed 30.06.16].

[54] Web Design and Applications, http://www.w3.org/standards/webdesign/; 2014 [accessed 28.06.16].

[55] XML Technology, http://www.w3.org/standards/xml/; 2014 [accessed 27.06.16].

[56] Wittenbrink H. RSS and Atom. Birmingham: Packt Publishing; 2005, ch. 4.

[57] Woodman M. How to Build an RSS 2.0 Feed. Sebastopol: O'Reilly Media, Inc.; 2006.

[58] White A. JavaScript Programmer’s Reference. Indianapolis: Wiley Publishing, Inc.; 2009, ch. 10.

[59] Yin RK. Case Study Research: Design and Methods. 5st ed. London: Sage Publications, Inc.; 2014.

[60] Yue C, Wang H. A measurement study of insecure javascript practices on the web. ACM Transaction Web 2013;7:2, p. 1-39.