



An approach to build JSON-based Domain Specific Languages solutions for web applications

Enrique Chavarriaga^{a,*}, Francisco Jurado^b, Francy D. Rodríguez^c

^a Uground Global S.L., Madrid, Spain

^b Universidad Autónoma de Madrid, Madrid, Spain

^c Universidad Católica de Ávila, Ávila, Spain

ARTICLE INFO

Keywords:

Domain-Specific Languages
JavaScript
JSON
JSON-DSL
Web applications
Templates engine

ABSTRACT

Because of their level of abstraction, Domain-Specific Languages (DSLs) enable building applications that ease software implementation. In the context of web applications, we can find a lot of technologies and programming languages for server-side applications that provide fast, robust, and flexible solutions, whereas those for client-side applications are limited, and mostly restricted to directly use JavaScript, HTML5, CSS3, JSON and XML. This article presents a novel approach to creating DSL-based web applications using JSON grammar (JSON-DSL) for both, the server and client side. The approach includes an evaluation engine, a programming model and an integrated web development environment that support it. The evaluation engine allows the execution of the elements created with the programming model. For its part, the programming model allows the definition and specification of JSON-DSLs, the implementation of JavaScript components, the use of JavaScript templates provided by the engine, the use of link connectors to heterogeneous information sources, and the integration with other widgets, web components and JavaScript frameworks. To validate the strength and capacity of our approach, we have developed four case studies that use the integrated web development environment to apply the programming model and check the results within the evaluation engine.

1. Introduction

Domain-Specific Languages (DSLs) provide a high level of abstraction to model, specify and define structures, specifications and functionalities that solve domain-specific problems. The goal of a DSL is to ease the process of implementing a system or part of it, allowing domain experts to be involved in the development process of reliable, robust and high-quality systems to provide solutions to specific problems [1,2].

The DSL deployment implies using parsers, analyzers and code generators to evaluate and execute the code behind the DSL specification. Also, to favor the deployment of DSLs, we can find Integrated Development Environments (IDEs), such as Visual Studio, Eclipse, NetBeans and WebStorm, among others, that provide utilities and dedicated languages and frameworks to design and implement DSLs. Focusing on web application development, when we must define grammars that are easy to integrate into the building and deployment of web applications, there are two widely adopted de facto standards: those based on XML and those based on JSON.

Thus, on the one hand, when a DSL based on the XML standard [3, 4], i.e. the DSL follows an XML grammar (XML-DSL), general purpose parsers such as Document Object Model (DOM) [5] can be used for the specification of domain-specific solutions and the evaluation and execution of the DSL. Besides, when the approaches use languages such as

HTML5, SVG [6], MathML [7] and XSLT [8], the solutions are enhanced on the client side, both visually and functionally. As an example, we can mention the work in [9], where we can find the PsiEngine, a XML-DSL execution engine for web clients, and a set of tools that facilitate the development and running of those DSLs. In [9,10] the authors show case studies of DSL with XML-based solutions that use the PsiEngine for different domain-specific problems.

On the other hand, the JSON standard [11] focuses on information exchange both on the server and client side. Thus, we can mention JSON for Linked Data (JSON-LD) [12,13], which allows the exchange of structured information that can be read and shared automatically. However, several issues arise when we specify a DSL that follows a JSON grammar (JSON-DSL), namely: how is the JSON-DSL grammar defined, what parsers, analyzers and code generation tools can we use to run the DSL, how is evaluated a program written in JSON-DSL, and whether can multiple programs and multiple JSON-DSLs interact. This paper proposes an approach that satisfies all these issues.

Despite the growing relevance of web applications and the interest shown by the scientific and industrial communities in using this kind of application, there are few research works in the literature dealing with the specification and evaluation of JSON-DSL at both server- and

* Corresponding author.

E-mail addresses: echavarriaga@uground.com (E. Chavarriaga), francisco.jurado@uam.es (F. Jurado), fdiomar.rodriguez@ucavila.es (F.D. Rodríguez).

client-side web applications. Literature on this topic focuses on JSON-DSLs for solving domain-specific problems and not on tools neither approaches for implementing JSON-DSLs in general. Most of the works address the specification of a JSON-DSL and how it works, regardless of desktop applications, server- or client-side web applications. Thus, to mention a few, Canis [14] is a high-level language that allows JSON specifications for data-driven graph animations, JSON-P [15] shows a case study on the development of a player for simple human-machine dialogs, JS4Geo [16] is a canonical JSON schema for geographic data stored in NoSQL databases, and JSON-LS [17] facilitates cross-linking with BioThings APIs for knowledge exploration.

Therefore, in this article we propose an architecture for building JSON-DSL, called RhoArchitecture, named after our previous PsiArchitecture [9]. This architecture includes: (a) a JSON-DSL Rho Evaluation Engine (RhoEngine for short), which is the JavaScript component able to run multiple programs written in different JSON-DSLs; (b) the Rho Programming Model (RhoModel for short) that establishes a programming model to add JavaScript functionality and support the corresponding code generation and documentation; and (c) an integrated web development environment known as Web Integrated Development Environment for Rho (WebIDERho for short) to allow specifying, implementing and deploying NodeJS-based server-side and client-side projects, as well as visualize the class diagram. Our approach allows: (i) the specification and evaluation of JSON-DSL; (ii) the implementation of JavaScript components that can interact with the DSL; (iii) the application of JavaScript Templates Engine that can serve as a way for programmers to effectively and efficiently generate strings coded in HTML, JavaScript, CSS, etc.; and (iv) the connection to heterogeneous information sources (JSON, XML, and Text) to embed data and integrate it with other widgets, components, and web frameworks. With all these features, the goal is to create fast, robust, and flexible solutions for both server- and client-side web applications.

With these three pieces of the RhoArchitecture (RhoEngine, RhoModel and WebIDERho), we try to establish the base for applying Model-Driven Engineering (MDE) in the specification, implementation, and deployment of JSON-DSL. MDE is a software engineering paradigm focused on defining Domain Models to simplify the building of information systems [18]. Thus, by combining the concepts of JSON-DSL with code generation and transformation engines, we set a solid foundation for applying MDE for web applications.

We will provide four case studies to demonstrate the capability of JSON-DSL specification and evaluation and the implementation of JavaScript components in our RhoArchitecture. The first case study is the classic “Hello World” to show the implementation and execution of a JSON-DSL. The second case study highlights the ability to manage multiple heterogeneous information sources (XML, JSON and Text) integrated. The third case study aims to validate programming at the server-side with the creation of a web service that includes the specification of a JSON-DSL, the use of Templates Engine and the design of web pages. The last case of use, which we called DrawRho, validates in an integrated way all the features proposed for the RhoArchitecture, including the interface with other frameworks. In all these case studies we have followed the qualitative case study methodology suggested by [19] and adapted it for Software Engineering in [20] to validate the most relevant characteristics of our approach.

The rest of the article is structured as follows: Section 2 highlights related works; Section 3 provides an overview of RhoArchitecture and the relevant features of our approach; Section 4 shows the four case studies; Section 5 details the results we have obtained; and finally, Section 6 closes the article with some conclusions and future work.

2. Outline and related works

The term Domain-Specific Language (DSL) is not defined rigorously in the literature. Fowler [2] defines it as «*a computer programming language of limited expressiveness focused on a particular domain*». In [21–23] the authors agree that a DSL is a programming language that is

targeted at a specific problem, such that its syntax and semantics contain the same level of abstraction that the problem domain offers, and its objective is to facilitate the design, definition and implementation of information systems that provide a solution to the problem domain. In addition, according to [2,23] the DSL provides a suitable grammar so that domain experts can perform these tasks more efficiently and produce systems of higher quality and reliability. On the other hand, the work presented in [24] studies the grammar composition of languages and helps to classify DSLs by considering: language extension, language restriction, language unification, self-extension and extension composition.

In [25], we can find a Systematic Mapping Study (SMS1) to determine the most popular domains (in this order: web, network, data intensive apps, control systems, low-level software, parallel computing, visual languages, embedded systems, real-time systems, dynamic systems, among others) where DSLs have been applied, using publications before 2011. In addition, they perform several research to list techniques, methods and/or processes dealing with DSLs. Lastly, SMS1 makes a comparative analysis between types of research versus domains.

In [26] we can find another Systematic Mapping Study (SMS2) 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 SMS2, 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 used for the implementation.

Moreover, we can find in [27] a last Systematic Mapping Study (SMS3) to identify and map the tools and IDEs (what authors call Languages Workbenches LW) in publications between 2012–2019. They identified 59 tools (9 under commercial licenses and 41 non-commercial) after analyzing more than 230 papers, and concluded that the tools largely cover the features proposed in [26] (grouped in the categories: notation, semantics, editor, validation, testing, and composability). Furthermore, in SMS3 the authors observed that the developers adopt a type of textual or graphical notation to implement their DSL.

The implementation of a DSL involves the use of parsers, analyzers, and code generation tools to obtain the functionality to run the DSL. Throughout time, most interpreters and compilers are based on Lex & Yacc [28] or Flex & Bison [29]. In addition, current IDEs provide specialized tools, plugins and languages for ease the design and implementation of DSLs. For example, Visual Studio has the Software Development Kit (SDK) for building model-based DSLs [30] and Eclipse provides a variety of specialized plugins for building DSLs like Stratego/XT [31], LISA [32], Spoofox [33], Antlr [34], Xtext [35,36] and Eclipse Modeling Project [37]. From the point of view of MDE [18, 38,39], a survey on software products, platforms and transformation tools for building modeling languages can be found in [40]. Likewise, using general purpose programming languages, together with specific design patterns and methodologies, we can build internal DSLs, e.g., for Java [23,41], C# [42], Scala [23,43], Ruby [23], Kotlin [44], Rust [45], Groovy [23,46], Python [47], Clojure [48], and Haskell [49].

Thus, as far as we know, there are many tools for creating DSLs that are mainly focused on the creation of textual or graphical DSLs. However, there are no solutions to implement JSON-DSL. The mentioned SMS1, SMS2 and SMS3 do not explicitly refer to the creation of JSON-DSL, neither the creation of DSL for web client-side. The papers [14–17] mentioned above, describe how their specification and the implementation of the functionality of the JSON-DSL are explicitly performed *ad-hoc*.

With these two drawbacks – the need to make tools for building JSON-DSL available; and the need to have an execution engine to run

programs written with JSON-DSL at both at server-side and client-side of web applications – our work focus on covering these needs and creating case studies to validate our proposal.

3. Approach to building JSON-based Domain Specific Languages

This section presents a detailed approach to the specification and implementation of JSON-based Domain Specific Languages (JSON-DSL) solutions for web applications, both on the server and client side. This section shows the core ideas related to RhoArchitecture and its three pieces: RhoEngine, RhoModel, and WebIDERho. Thus, we will start providing a brief overview of the approach, and afterwards, we will break down it.

3.1. Brief overview

RhoArchitecture's cornerstone is the RhoEngine, a JavaScript component that can run multiple programs written in different JSON-DSLs. A JSON-DSL is a programming language that follows a JSON grammar, while RhoLanguage is a JSON-DSL plus the JavaScript classes that implement grammar elements' functionality. The running of a JSON-DSL program evaluates the functionality of the nested grammar symbols, starting from the root and drilling down based on grammar definition. JSON-DSLs running in RhoEngine can connect and exchange heterogeneous information, use template engines, use components and web components, apply security policies, and follow good programming practices [50,51], making their code more functional, reliable, and robust.

To allow a JSON-DSL work with the RhoEngine, we must follow the RhoModel, which establishes a programming model for generating JavaScript code and documentation. As we will detail, the RhoModel focus on specifying the JSON-DSL and implementing the JavaScript components and other necessary reusable web components at both the server- and client-side. The RhoModel is based on our previous PsiModel [9].

Finally, the WebIDERho uses the RhoModel as programming model and the RhoEngine as the execution environment. WebIDERho enables us to define projects at the server and client-side, visualize the class diagram, automatically generate documentation, and deploy NodeJS-based server and client-side web applications.

3.2. The RhoArchitecture

To follow our explanation, Fig. 1 presents the software architecture that defines, at a conceptual level, the components involved in our approach to work with JSON-DSLs, i.e., the RhoArchitecture. The main goal of this architecture is to facilitate next two steps:

- i. *Specifying a JSON-DSL*: in this step, a JSON-DSL is defined as a RhoLanguage using the RhoModel (in the lower-central part of Fig. 1). To do so, first we must define the JSON-DSL grammar and then implement the functionality related with the elements (terminal and non-terminal symbols) of the grammar. In the RhoModel, this functionality must be provided as a set of JavaScript Classes by inheriting from the RhoLanguage base classes to ease the programmer's task. The functionality associated to each element of the grammar is implemented in what we call Components (at the bottom of Fig. 1). After providing the grammar and the related functionality, the RhoModel can generate the JavaScript code when evaluating the corresponding JSON-DSL code.
- ii. *Evaluating the JSON-DSL*: the main goal of this step is to fetch a RhoCode program (on the left side of Fig. 1) written in a RhoLanguage together with the necessary resources (i.e., HTML fragments, images, videos, css, svg, etc.) and then to run the RhoCode using RhoEngine. To do so, the RhoEngine generates a

RhoProgram by transforming the RhoCode into a JavaScript object (the RhoObject on the right side of Fig. 1). This RhoObject is generated by executing the functionality of each nested element of the grammar, starting from the root element, and drilling down according to the definition of its grammar. The obtained RhoObject solves a specific problem and can run within a web application, either on server- or client-side. While performing this step, compilation and execution errors will be reported for processing.

At the top of Fig. 1, the Data Sources components encapsulate the functionality to manage information in JSON, XML and Text format. Thus, any object or class defined in RhoArchitecture can use these Data Sources components to fetch external information and link one or more of its properties at runtime. Thus, the proposal allows us to bring heterogeneous information and assign it to the class with the appropriate parameterization. Therefore, the approach can decouple the data sources from the JSON-DSLs, adding versatility and power when implementing solutions to specific problems within a web application.

At the bottom of Fig. 1, we can see Templates. They are pre-designed text containing variable labels that can be dynamically tuned to produce customized text output. These templates can be used to generate HTML, SVG, JSON, XML, and JavaScript code, which speeds up the development of web applications. To custom manage the use of the templates, on the left-hand of Fig. 1, we can see the Templates Engine. A Template Engine is a JavaScript component that enables the creation of custom Templates using different specific syntaxes (usually precompiled in memory) to generate strings quickly and efficiently. There is a variety of Template Engines in the market, and in particular, we can mention EJS [52], Handlebars [53,54], and Hogan [55] engines. These engines are added to RhoArchitecture as plugins and are used easily by RhoEngine. Additionally, RhoArchitecture implements an easy native Template Engine we have called Plain, with limited functionality. Using these Template Engines when creating JSON-DSL, together with the Data Sources binding, exponentially increases the power and versatility of these languages.

At the bottom of Fig. 1, we can see Web Components [56–59], which are widgets or reusable components built in HTML, DOM, JavaScript and CSS, and are deployed within the web application. RhoArchitecture generates the code of web components automatically by using the Templates Engine. To incorporate these Programming Components and Tools into our programming models, we have implemented the Factory Web Components and Plugins Templates Engine (see Fig. 1 on left of the RhoModel box).

As can be inferred so far, the main idea behind our approach is a strong emphasis in code generation, which is a well-established field in software engineering with a particular focus on model-driven engineering [60,61]. Code generation brings time savings, increased efficiency, higher quality and standardization when building information systems [62,63]. Thus, in this context, to simplify the specification of RhoLanguages (which we will detail later in Section 3.4), we need the RhoModel (see Fig. 1). This allows the definition of RhoGrammars \mathbb{G} and the implementation of Template Engines, Web Components and JavaScript Components based on RhoEngine. RhoModel separates specification from implementation following our findings from previous research work on the former PsiModel [9] and automatic code generation.

To sum up, with the RhoArchitecture, any JSON-DSL implemented for RhoEngine is able to connect to heterogeneous data sources (XML, JSON, text, etc.), to use Templates Engine, to work with Web Components in order to add versatility and functionality to the language, and if security policies and good programming practices are applied then quite reliable and robust executions can be obtained.

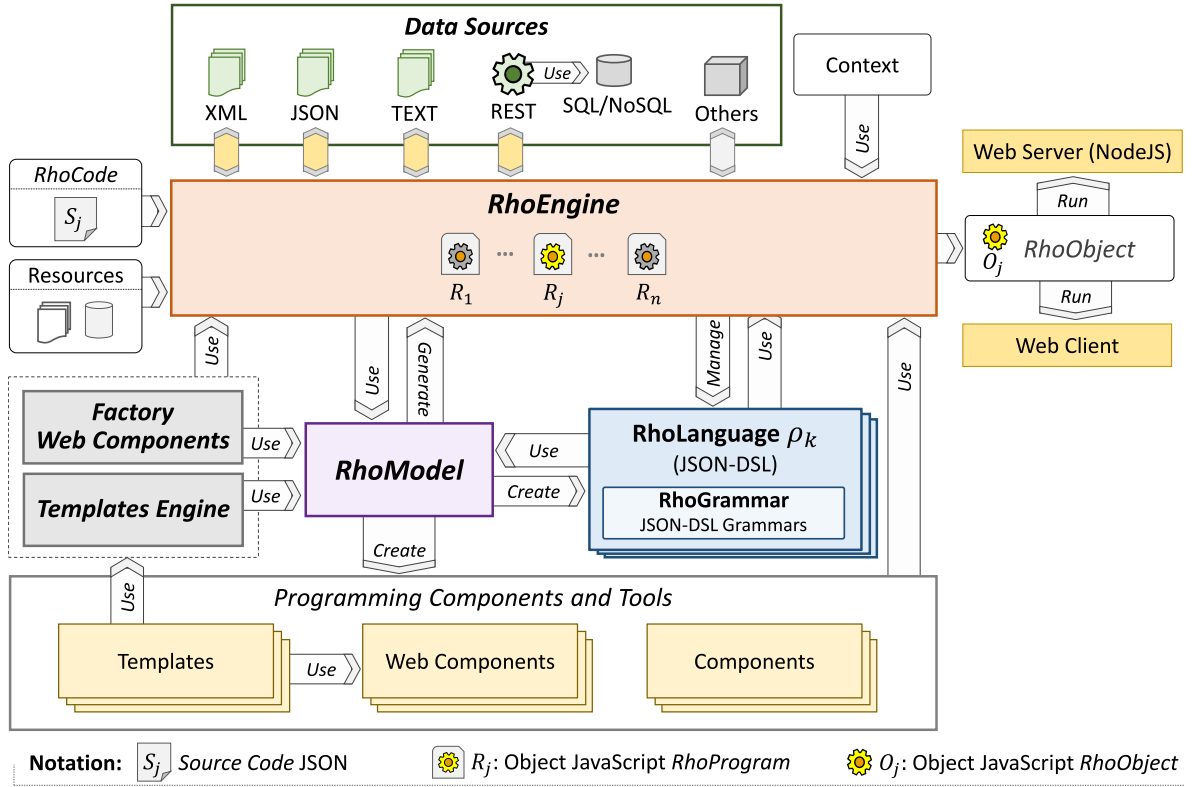


Fig. 1. Diagram of functional blocks for the RhoArchitecture.

3.3. The RhoEngine

Within the RhoArchitecture, the RhoEngine is the evaluation engine for JSON-DSLs. Formally, RhoEngine manages a set of RhoLanguages defined as $P = \{\rho_1, \dots, \rho_k, \dots, \rho_m\}$. That is, it can handle several JSON-DSLs, and in this way, to interpret and evaluate jointly several programs written in JSON to creating components for web application.

Each RhoLanguage $\rho_k \in P$ is registered with an alias. Thus, when executing a RhoCode S_j , the alias of the corresponding ρ_k is loaded. The RhoEngine gets the S_j and the alias for ρ_k , creates a RhoProgram R_j and adds it to the list of running programs $\mathbb{R} = \{R_1, \dots, R_j, \dots, R_n\}$. Then, each R_j converts S_j into a RhoObject O_j . This is what we call evaluation of JSON-DSL, which ends up executing S_j using a language ρ_k .

To execute S_j , RhoEngine gets the grammar of the language ρ_k and starting from the root element of the grammar in S_j , it validates and evaluates the functionality for that element. Then, it goes deep into the nested elements of S_j and analyzes them against the structure of the grammar of ρ_k , so that, for each nested element it validates and evaluates the associated functionality. The execution ends when RhoEngine has finished going through all the nested elements of S_j . As a result of the execution, the RhoProgram R_j returns a reference to an object O_j , which solves a specific problem (or part of it) within a web application, either at web server or web client side.

Formally, a solution to a specific problem for a W_{APP} web application can be seen as a set of RhoLanguages programs to be executed:

$$W_{APP} = \{S_{jk} | 1 \leq j \leq n, S_{jk} \text{ coded in } \rho_k \in P, 1 \leq k \leq m\}$$

and the execution of a W_{APP} web application, either at client or server side, can be expressed as the set of executions:

$$\text{Exec}(W_{APP}) = \{O_{jk} | 1 \leq j \leq r, O_{jk} \text{ object reference of } R_{jk} \in \mathbb{R}, 1 \leq k \leq m\},$$

The execution of an application W_{APP} can coexist with the executions of other W_{APP} written in different $\rho_k \in P$, i.e., different JSON-DSL.

Due to RhoEngine evaluates the JSON-DSL code directly, and thanks to the dynamic nature of JSON, the source program (RhoCode S_j) can change during its execution by modifying the RhoObject O_j . To allow reusing the modified code, RhoEngine can serialize the changes and create a new S_j . This same JSON feature would also allow us to combine RhoCode fragments to build dynamic programs, obtaining versatility, flexibility, and adaptability to changes in a web application. Furthermore, unlike other JSON-DSLs, RhoLanguages can associate external resources (XML, JSON or text) to use and modify information at runtime. Additionally, they have the ability to use JavaScript Web Components and Template Engine. The RhoEngine base class has built-in support for all these features, which can be directly incorporated into a JSON-DSL specification.

3.4. The RhoLanguage

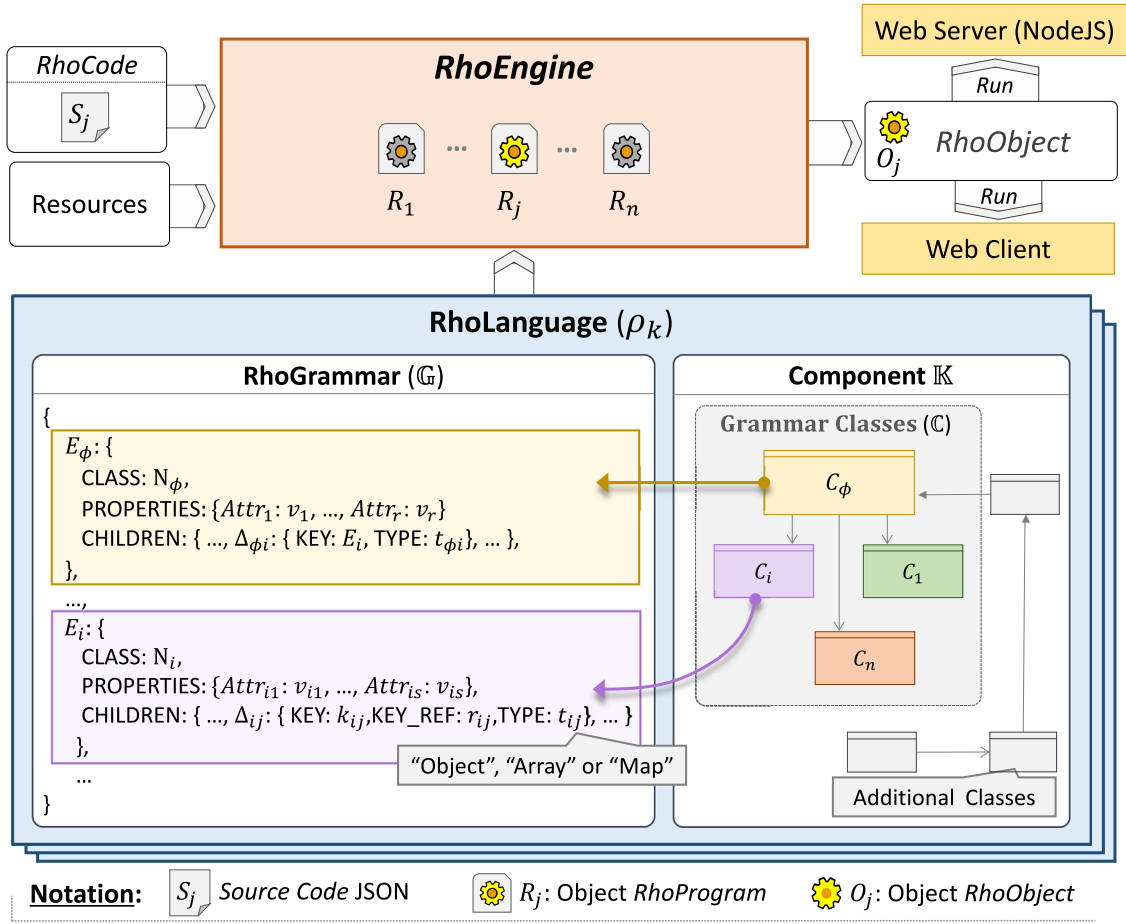
As mentioned above, a JSON-DSL is a programming language written using JSON grammar, and the associated functionality is implemented with JavaScript programming, both client-side and NodeJS-based server-side [64–66].

From this assertion, these JSON-DSLs can be specified and built within RhoModel as a JavaScript component (see Fig. 2). We have denoted these JSON-DSLs by RhoLanguages. For a particular RhoLanguage ρ_k , a RhoGrammar \mathbb{G} is defined by a duple:

$$\mathbb{G} = \langle \mathbb{E} | E_\phi \rangle \quad (1)$$

where $\mathbb{E} = \{E_1, E_2, \dots, E_n\}$ is the set of available objects or elements of the grammar, that is, the functionality associated to the elements of the language defined via an instance object E_i . The object E_ϕ is the root object of the grammar (for any $E_\phi \in \mathbb{E}$, $1 \leq \phi \leq n$). Each $E_i \in \mathbb{E}$ has the next structure:

$$E_i = \{\text{CLASS} : N_i, \text{PROPERTIES} : P_i, \text{CHILDREN} : H_i\} \quad (2)$$

Fig. 2. Association diagram between *RhoGrammar* \mathbb{G} and component \mathbb{K} .

where N_i is the name of the related class, P_i are the properties or attributes for the object E_i , and H_i defines the children or nested objects (see Fig. 2), where:

$$H_i = \{\Delta_{ij} | 1 \leq j \leq m, \Delta_{ij} = \{\text{KEY: } k_{ij}, \text{KEY_REF: } r_{ij}, \text{TYPE: } t_{ij}\}\} \quad (3)$$

been m is the number of nested objects, and Δ_{ij} the definition of the nested object (where k_{ij} is the reference to an element of the grammar that depends on the attribute r_{ij} , and t_{ij} is the type of nested object, which can be “Object”, “Array” or “Map”).

Once defined the *RhoGrammar* \mathbb{G} , we must implement its semantic, i.e., to code the functionality associated to each object (i.e., the instance of the corresponding element) of the grammar \mathbb{G} that is referenced in \mathbb{E} (see Fig. 2). This functionality is implemented in the grammar's set of classes $\mathbb{C} = \{C_1, \dots, C_n\}$. The \mathbb{C} classes are stored in the package \mathbb{K} of reusable Component coded in JavaScript [8,58] that implements the entire *RhoLanguage* ρ_k . The execution of a program S_j written with the grammar of ρ_k , is the evaluation of all nested objects in the S_j .

Formally, a *RhoLanguage* ρ_k is defined with the tuple:

$$\rho_k = \langle \mathbb{G} | \mathbb{K} | \mathbb{E} \leftrightarrow \mathbb{C} \rangle \quad (4)$$

where *RhoGrammar* \mathbb{G} is the grammar of language ρ_k , as defined in (1), \mathbb{K} is the reusable JavaScript component that implements the functionality of the language. ρ_k , \mathbb{C} is the subset of classes that define the grammar's functionality \mathbb{G} , and finally, $\mathbb{E} \leftrightarrow \mathbb{C}$ is the association between E_i and C_i , for each $E_i \in \mathbb{E}$ and $C_i \in \mathbb{C}$, respectively. Note that a class $C_i \in \mathbb{C}$ can be associated with multiple elements of the grammar \mathbb{G} , and for an object $E_i \in \mathbb{E}$ has only one class defined C_i .

TEMPLATE 1 and TEMPLATE 2 shows a possible implementation of a JavaScript Component for a *RhoLanguage* ρ_k and the grammatical definition for its execution in *RhoEngine*. The TEMPLATE 1 defines the

Component \mathbb{K} of Fig. 2, and contains: (i) the set of grammar classes \mathbb{C} (where each class inherits *RHO.JSNDL.Base*, see Section 3.7), (ii) the additional classes of the Component \mathbb{K} , and (iii) the public interface of the Component \mathbb{K} .

TEMPLATE 2 shows: (i) defines the Grammar \mathbb{G} of Fig. 2 according to (1), (ii) the register language ρ_k in *RhoEngine*, and finally, (iii) example of the execution of a programmer S_j of language ρ_k in *RhoEngine*.

3.5. The *RhoModel*

Automatic code generation is one of the cornerstones of software engineering, leading to time savings, greater efficiency, higher quality, and standardization for building information systems [63,67]. In this context and as mentioned in the previous section, *RhoModel* (*Rho Programming Model*) allows the specification and implementation of *RhoLanguages*, *Templates Engines*, *Web Components* and *JavaScript Components*. *RhoModel* separates specification from implementation based on our previous works on the *PsiModel* [9] and using code-behind techniques. *RhoModel* allows to specify and implement the following: (i) basic programming elements (*Const*, *Var*, *Object*, *Enum*, and *Function* tags), (ii) classes (*Class* tag, using the definition given in [64,67]), (iii) references to external classes (*ExternalClass* tag), (iv) *JavaScript* components (*Component* tag, using the definition of component or module given in [67,68]), and (v) definition of *DSL's* (*DSL* tag).

More details about *RhoEngine*, *RhoLanguages* and *RhoModel* can be found in the “*Rho API*” submenu of <http://www.devrho.com>. This website includes *RhoModel* code, automatic documentation, generated code, and interactive diagrams, of all libraries and projects generated with *RhoModel* (“*Docs*” submenu) in its lightweight development environment *WebIDERho*.

TEMPLATE 1. JavaScript template to define Grammar Classes \mathbb{C} of a *RhoLanguage*.

```
// Definition Component RhoLanguage K
var MyComponentK = (function () {
  // Associated classes for the RhoGrammar G
  function Class1 (def, parent, ref) { }; // Class Class1 implementation
  Class1.prototype = Object.create(RHO.JSONDSL.Base.prototype);
  ...
  function ClassI (def, parent, ref) { }; // Class ClassI implementation
  ClassI.prototype = Object.create(RHO.JSONDSL.Base.prototype);
  ...
  function ClassN (def, parent, ref) { }; // Class ClassN implementation
  ClassN.prototype = Object.create(RHO.JSONDSL.Base.prototype);

  // Component additional classes
  function ClassOther1 (def, parent, ref) { };
  ...
  function ClassOtherM (def, parent, ref) { };

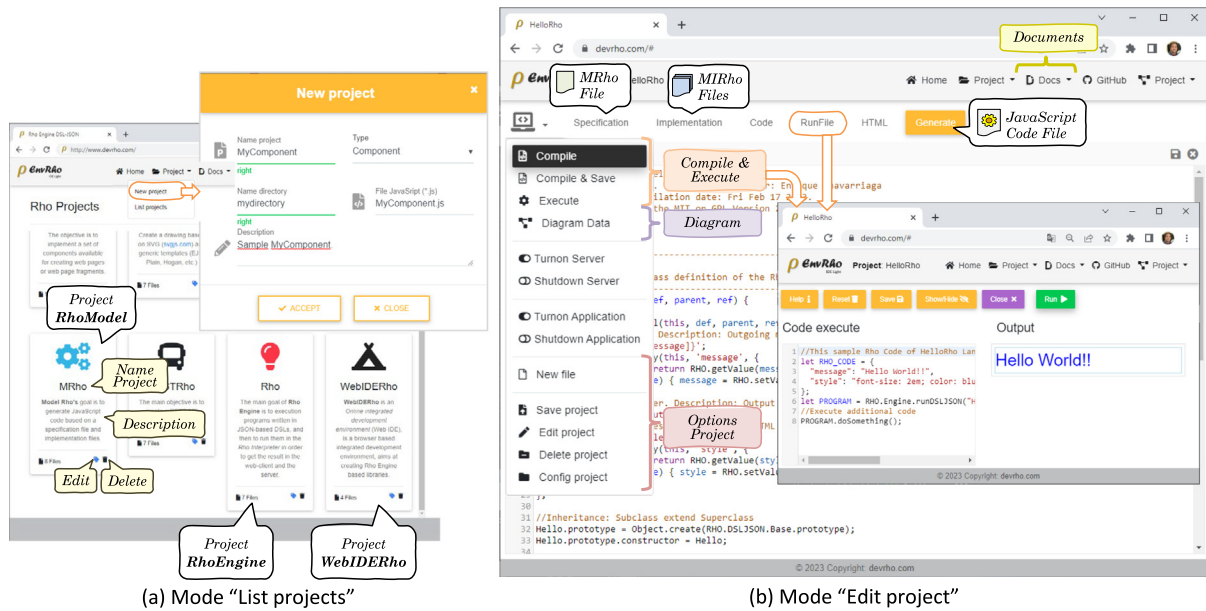
  return {
    Class1: Class1, ...
    ClassI: ClassI, ...
    ClassN: ClassN,
    ... //Others programming elements.
  }; // MyComponentK interface
})();
```

TEMPLATE 2. JavaScript template to define RhoGrammar \mathbb{G} of a *RhoLanguage*.

```
// Definition RhoGrammar G
var MY_RHO_GAMMAR = {
  NAME: "MyNameRho",
  ELEMENTS: {
    "Eroot": {
      CLASS: "MyComponentK.ClassRoot",
      PROPERTIES: {"Attr1": {VALID: true}, ..., "AttrR": {VALID: true}},
      CHILDREN: {
        ..., "NameOi": {KEY: "Ei", TYPE: "<Type>"}, //<Type>: Object, Array or Map
      }
    },
    "E1": {
      CLASS: "MyComponentK.Class1", PROPERTIES: { ... }, CHILDREN: { ... }
    },
    ...,
    "Ei": {
      CLASS: "MyComponentK.ClassI",
      PROPERTIES: {"AttrI1": {VALID: true}, ..., "AttrIS": {VALID: true}},
      CHILDREN: {
        ...,
        "DeltaIJ": {KEY: "Ej", TYPE: "<Type>"}, //<Type>: Object, Array or Map
        ...,
        "DeltaIK": {KEY: ["Ek1", ..., "Ekm"], KEY_REF: "AttrIK", TYPE: "<Type>"},
        ...
      }
    },
    ...,
    "En": {
      CLASS: "MyComponentK.ClassN", PROPERTIES: { ... }, CHILDREN: { ... }
    }
  },
  ROOT: "Eroot" // Root name element
};

// RhoLanguage Register
RHO.registerRhoLanguage ("MyNameRho", MY_RHO_GAMMAR);

//Execute program Sk
var RHO_CODE_Sk = { ... };
PROGRAM = RHO.Engine.runJSONDSL("MyNameRho", "Sk", {json: RHO_CODE_Sk});
```

Fig. 3. Integrated web development environment **WebIDERho**.

3.6. The WebIDERho

The WebIDERho Web Integrated Development Environment is implemented to allow developers to create projects at the web server and web client side, visualize class diagrams, create documentation automatically, and deploy NodeJS-based web servers and client applications. WebIDERho manages group of projects called project list. Fig. 3(a) shows the projects list “_core” (RhoArchitecture core projects), each project shows its name, description, number of files, and the edit and delete buttons. In WebIDERho, we can create three kinds of projects: Empty, RhoLanguage (specification of JSON-DSL’s) and Component (creation of JavaScript components), all based on RhoModel.

On the other hand, Fig. 3(b) shows the editor for a WebIDERho project. The following summarizes the project menu options:

- i. **Open Projects List:** opens projects list by using www.devrho.com?projects=<list> in the web browser, where <list> is the name projects list. By default, the RhoModel sample projects (<list> = _samples) are displayed, which contains the case study projects detailed in this article.
- ii. **Compile & Execute** compiles the specification file (MRho File) and the implementation files (MIRho Files) to generate the JavaScript code. On the other hand, WebIDERho has a simple environment for the execution of the component.
- iii. **Diagram:** builds and displays the class diagram of the project (see Fig. 5). It is possible to create as many diagrams as needed.
- iv. **Server & Application:** deploys NodeJS-based web services and applications.
- v. **Options Project:** manages the Rho project (save all, edit and delete). It allows the creation of multiple file types, as it uses CodeMirror [69] as editor.
- vi. **Documents:** in this submenu we access the utilities for the automatic documentation of all available RhoEngine components and examples.

Particularly, Fig. 3(a) shows the projects set with “<name> = _core”, among the list is the implementation of our RhoEngine engine (Rho project, type Component), the RhoModel programming model (MRho project, type Component), and this development environment (WebIDERho project, type Component).

3.7. Implementation summary

Based on the RhoArchitecture block diagram of Fig. 1, Fig. 4 shows the RhoArchitecture components diagram, and, Fig. 5 details the RhoArchitecture class diagram, where RHO is the name of the main component, and the subcomponents that make it up are summarized below:

- **Engine** is the component that implements our execution engine *RhoEngine*.
- **JSONDSL** is the JavaScript Component that implements the basis for the creation of *RhoLanguage*, i.e., the foundations for the building and implementation of JSON-DSLs.
- **DS, Templates y Factory** are the components that implement the management of *Data Source*, *Template Engine* and *Factory Web Components*.
- **MRho** is the external component that implements our programming model *RhoModel*.

WebIDERho diagrams are based on Diagram Programming Generate DPG [70], a JSON-DSL that allows to specify programmable diagrams based on PsiDiagram [71] for web applications. It is possible to have multiple views of the diagram and to see the source code of each RhoModel programming element. As future works, WebIDERho will be able to support both textual and visual programming DSLs. The class diagram in Fig. 5 is a DPG diagram, and the detailed help generated by WebIDERho can be found at www.devrho.com?doc=rho.

3.8. Final comments

Throughout this section, we have presented the RhoArchitecture that defines how to specify and implement JSON-DSLs, as well as how they are evaluated by the RhoEngine. In summary, a JSON-DSL is a programming language written with JSON grammar. The grammar of the language and its functionality (denoted by RhoLanguage) can be specified and built using the Rho Programming Model (named RhoModel). Finally, the integrated web development environment, WebIDERho, enable the use of the RhoModel and allows defining web server and web client projects, visualize the class diagram, create automatic documentation, deploy servers, and web applications based on NodeJS.

Next section focuses on developing case studies to validate the whole approach.

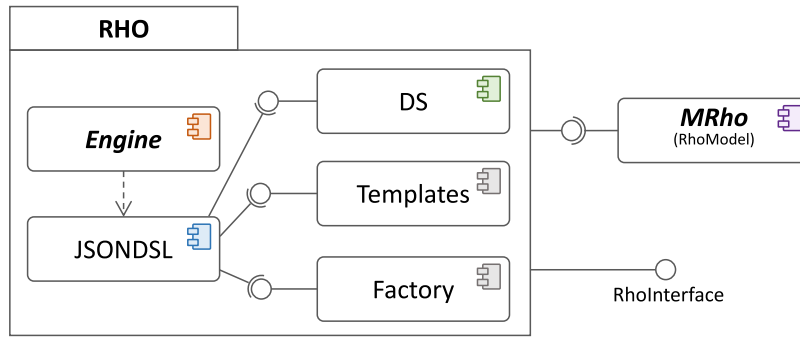


Fig. 4. RhoArchitecture components diagram.

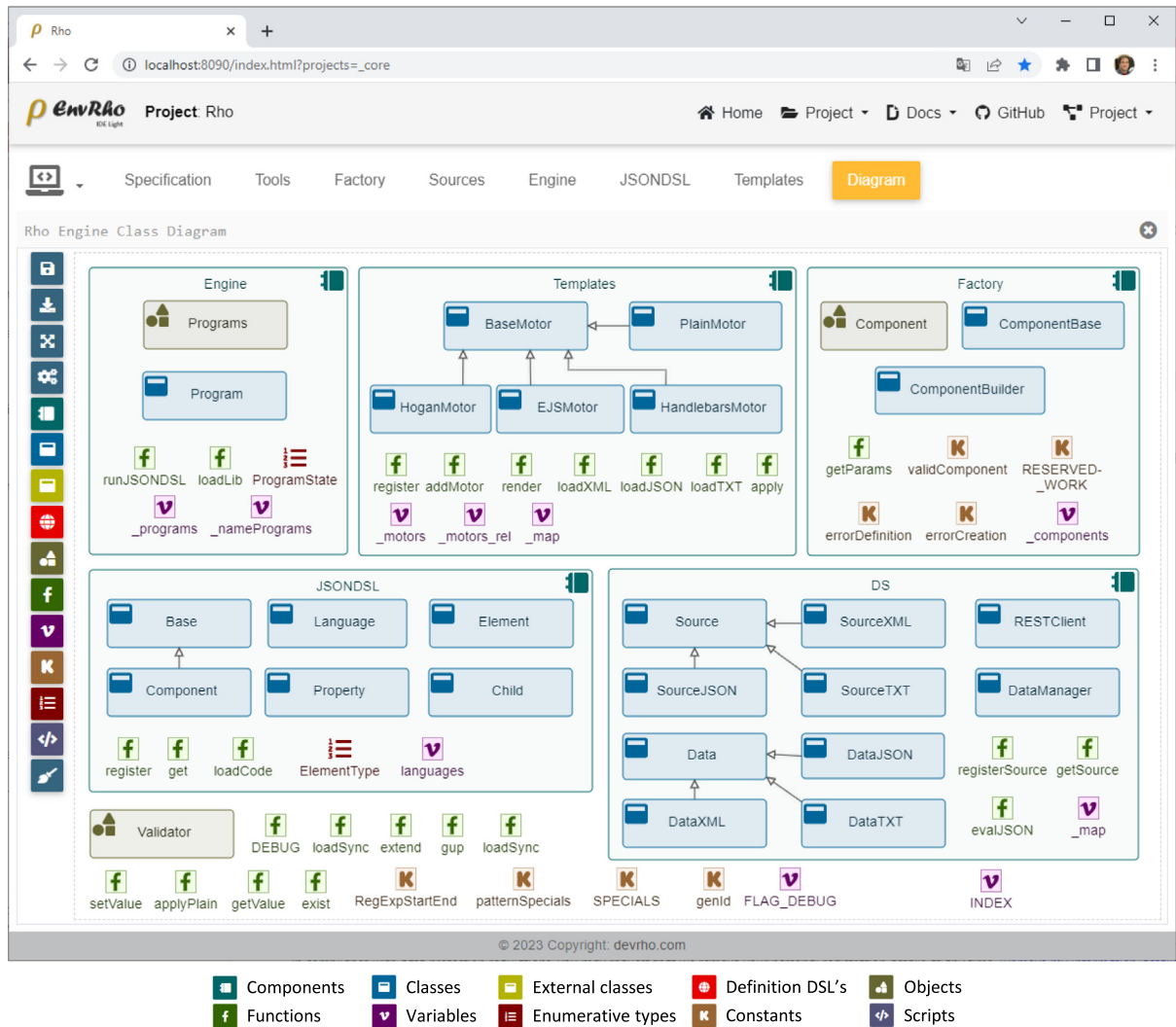
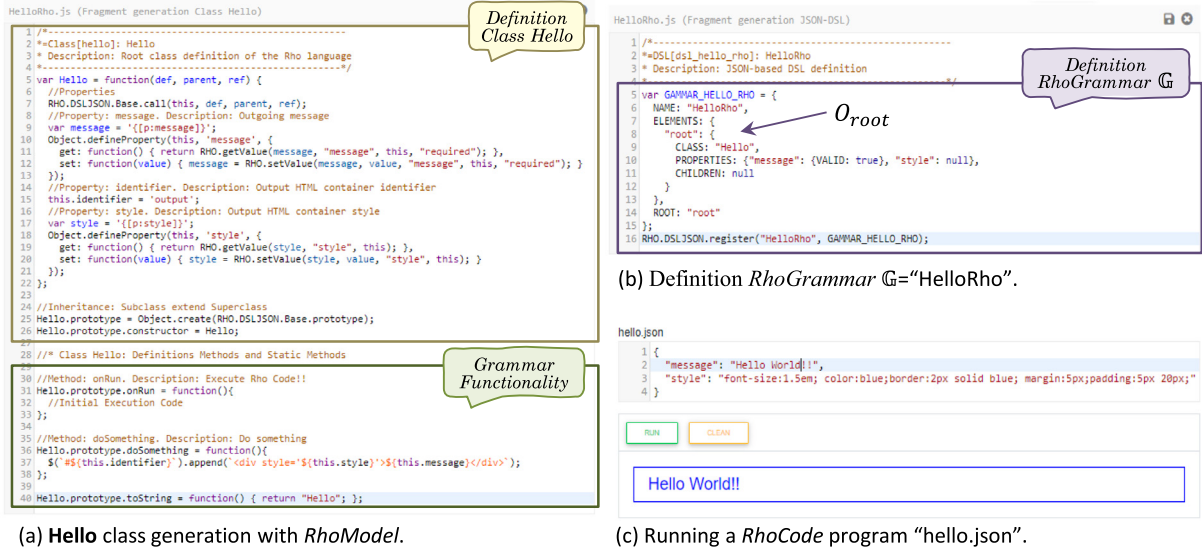


Fig. 5. RhoArchitecture class diagram. Visual language for RhoModel visualization based on DPG.

4. Cases of studies

In this section, we will detail how to apply and validate our whole approach. Thus, the aim of the case studies is to illustrate the use of RhoArchitecture and the RhoEngine. The first case is the classical “Hello World”, where we create a very simple JSON-DSL to show in a straightforward way how to specify and deploy a RhoLanguage. The second case study highlights the use of multiple heterogeneous data

sources (XML, JSON and Text) implemented in a single class. The third case study deals with validating the capabilities to the RhoEngine for server-side applications, using a RhoLanguage for implementing a REST service for the back-end, creating Templates Engine, and designing web pages that use Material Design for the front-end (for example, by using MDBootstrap [72]). The last case study fully validates the most relevant features of the RhoArchitecture, namely: the specification of a RhoLanguage, the implementation of Components and Web Components, the

Fig. 6. Program HelloRho “hello.json” and its execution in *RhoEngine*.

use of Templates Engine, the exchange of heterogeneous information, and the integration with other frameworks.

4.1. Implementing the HelloRho JSON-DSL

HelloRho is a JSON-DSL that aims to show a simple specification of a RhoLanguage. In Fig. 6(a), we can see how HelloRho implements the functionality for the grammar in only one class $\mathbb{C} = \mathbb{O} = \{Hello\}$. Also, RhoModel uses Template 1 (see Section 3.4) as a guide for the code generation, and to implement the reusable JavaScript Component \mathbb{K} in “HelloRho.js”. Also, we can observe that the functionality of this grammar (at the bottom of Fig. 6(a)) consists of adding a DIV element in a container with an identifier (attribute `this.identifier`), and then modifying the text Content (attribute `this.message`) and the style (attribute `this.style`).

For its part, Fig. 6(b) shows the definition of the grammar. As we can see, the name of the grammar is “HelloRho”, it has only the root element containing two properties: message (mandatory property with `{VALID: true}`) and style. The JavaScript class that implements its functionality (“Hello”), and no children as nested elements. That is, we have the grammar for HelloRho $\mathbb{G} = \langle \mathbb{O} | O_{root} \rangle$ specified according to (1), where the HelloRho language is defined by:

$$HelloRho = \langle \mathbb{G} | \mathbb{K} | \mathbb{O} \leftrightarrow \mathbb{C} \rangle$$

Finally, in Fig. 6(c), we can see result for the execution of the program written in the RhoCode ($S_j = \text{“hello.json”}$ at the bottom of Fig. 6(c)). This program can be modified and executed as desired. This example is available at devrho.com, in the menu option “Samples>Hello world!!”.

4.2. Associate heterogeneous data sources of information

The *TestSources* case study aims to validate the ability to associate heterogeneous Data Sources in XML, JSON and Text formats for Objects and Classes defined in RhoArchitecture. This example consists in displaying the character’s information coming from different Data Sources (see Fig. 7(a)), as described below: (i) the first and last name comes from a JSON file “actors.json”; (ii) the age, email and image reach from an XML file “details.xml”; and (iii) the description is loaded from tags in a Text file “descriptions.txt”. The information from the different Data Sources can be accessed through a Key identifier, as shown in Fig. 7(a).

Fig. 7(b) shows how RhoModel can define attributes within an *Object* or *Class*, whose source links to a *Data Source* (XML, JSON

and Text). For instance, the attribute *Name*, takes the information of “`{{info:First}}`”; the variable *info* links to the source ACTORS, in turn, registered in the RHO.DS component. Another remarkable detail is how the tags are configured (using JavaScript regular expressions) in the *desc* attribute to obtain the *enumerate* of the Actor.

Finally, Fig. 7(c) shows the execution that displays the information of an Actor through its identifier. Note that RhoModel’s Plain template engine and MDBootstrap [72] were used for the template design.

In this case study, it is worth noting that the Actor class brings together the three available data sources. In perspective, if the data sources are linked to SOAP Services [73], REST services [74], or to non-relational databases, the Actor class could directly update the information through these services, being transparent to the class. If we add the use of template engines, and the design of web pages with material design, then RhoArchitecture is an interesting alternative in implementing and deploy web applications, both at client- and server-side. This case study is available at devrho.com, in the menu option “Samples>Star Wars Actors”.

4.3. Creating a simple “web service”

ComicSpeech is a RhoModel project that aims to validate the server-side programming of RhoEngine in the following aspects: (i) specification of a web service, (ii) creation of a server-side RhoLanguage and (iii) the use of the different Templates Engine with material design for Bootstrap [72].

Fig. 8(a) shows a RhoCode that manages the list of speeches and their search. Also, Fig. 8(a) outline the methods available on the web service “/comicspeech”. Thus:

- (i) the method “/list” (shows the available list of speeches), and
- (ii) the method “/speech” (search for a list of comma-separated speeches).

Fig. 8(b) shows the output for the execution of this method for: *Ouch*, *Hey* and *ZZZ* “/speech?search = Ouch,Hey,ZZZ”. This case study is available on devrho.com, under the menu option “Samples>Comic Speech”.

4.4. Creating an SVG diagramming JSON-DSL

DrawRho is a RhoLanguage that aims to paint reusable graphical elements based on an SVG library. DrawRho is based on the creation

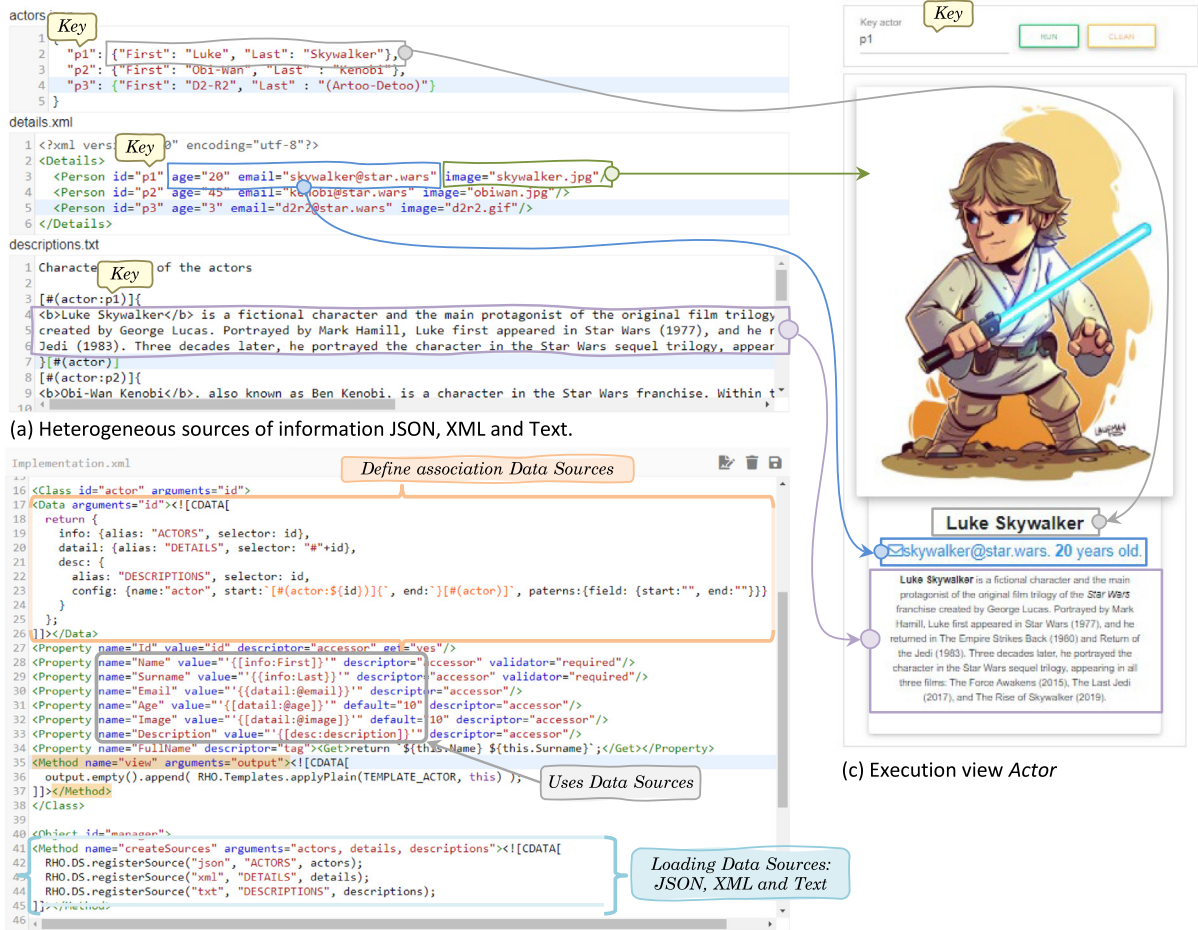


Fig. 7. Linking different heterogeneous data sources.

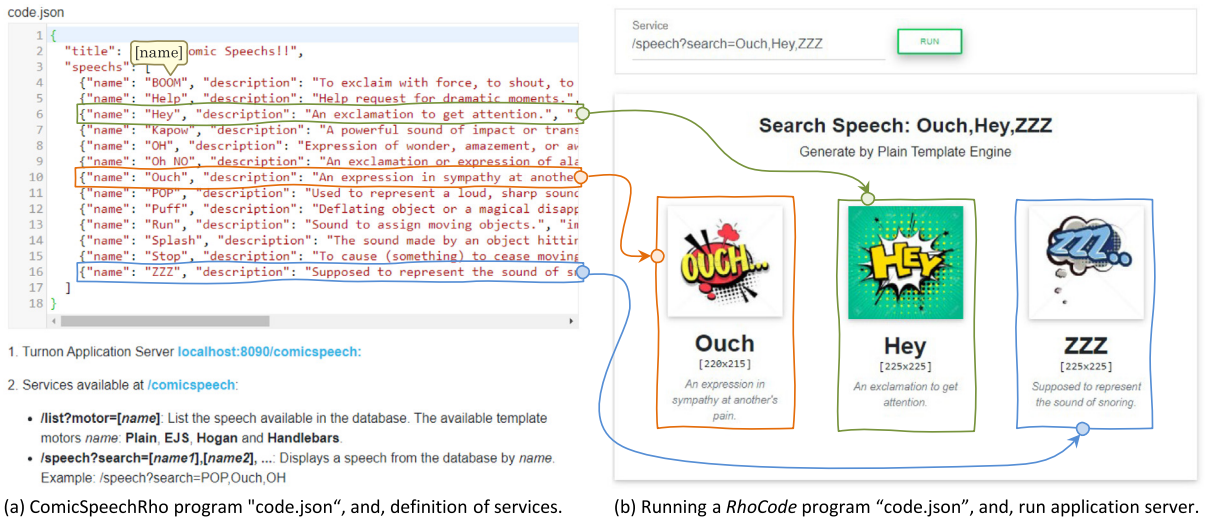


Fig. 8. Example of web services using JSON-DSL and the Templates Engine.

of Web Components with the help of the SVGJS framework [75], the Handlebars templates [54] and the Draggable plug-in [76] to move the graphical elements. Thus, DrawRho implements the following graphical elements:

- (i) *layer*: defines the concept of a graphical layer where it contains a list of *shapes*, *lines* and *containers*. The graphical layer refers to the depth at which the set of elements in the layer is located,

that is, whether it is at the bottom (defined as the first layer) or at the front (defined as the last layer). It is defined with the SVG element grouping tag (*g* tag).

- (ii) *shape*: is the graphical representation of a figure, object or entity. It is defined by the grouping of SVG elements (*g* tag) such as: rect, circle, ellipse, image, line, polyline, polygon, text, path, etc. (i.e., all available SVG elements).

- (iii) *line*: is the description of a path with a beginning and an end, at the edges are defined markers to represent arrows, joints, connectors, etc. A *line* can be defined with an SVG line, polyline or path element, and text can be added with the SVG text tag.
- (iv) *container*: represents a grouping of graphical elements such as *shapes*, *lines* or other *containers*. If a *container* is moved, all its elements move with it. It is also defined with the SVG element grouping tag (g tag).

As previously stated, the case study in this section comprehensively validates the most relevant features of our RhoArchitecture, such as: the creation of a JSON-DSL (with multiple executions), the creation of Components and Web Components, the use of Templates Engine and the Data Sources information exchange connection (in JSON format).

Fig. 9(a) shows a couple of *RhoCode* programs with grammar *DrawRho*. The first program (file “StarWars.json”) draws the containers with the roles in Star Wars. The second program (file “Speechs.json”) draws the nodes with images from speech. Also, Fig. 9(a) shows the library of SVG graphical elements defined in the file “templates.xml”, and the file “sources.json” contains two *Data Sources* in JSON format: *Actors* (list of actors) and *ComicSpeech* (list of speeches). Fig. 9(b) shows the output of the execution.

The way it works for each graphical element is as follows: (1) determine the type of element (attribute *type* = “*layer|shape|line|container*”); (2) search in the graphical library for the corresponding template (using the *lib* attribute); (3) apply the changes to the information in the template for the graphical element (attribute *key*, *x*, *y*) for each SVG element (attribute *key*) by modifying their attributes, and links the corresponding information (attribute *source*) for the *Data Source* of *Actors*; and then, (4) add the SVG canvas to the template; finally, (5) links the necessary events.

In particular, Fig. 9 shows the “StarWars.json” once it works. Briefly, it searches a *shape* with *Node* identifier specifying a circle and text (full name of a *Actor*) and places it in the position (150,60). We want to remark that the creation of a diagram can be done through a set of independent programs, providing versatility when creating diagrams and allowing experts in niche parts of the specific domain to collaborate with each other. Languages of this kind can be used in the creation of collaborative diagrams. This case study is available on devrho.com, under the menu option “Samples>Draw Rho”.

4.5. Other examples

There are other interesting case studies on devrho.com, under the menu option “Samples” (BPM Tester, MDB Tester), which are not included in the validation study results. The BPM Tester project includes the implementation of a RhoLanguage called BPMERho to execute a BPMN 2.0 [77] designed in the BPMEPSi visual tool [71]. The MDB Tester project also includes the implementation of a RhoLanguage called MDBRho that aims to create Material Design Bootstrap MDB components, forms, navigation, dialogue boxes, block design, etc. [72].

5. Results and validation

The research has been conducted according to the **qualitative case study methodology** suggested by [19] and adapted for Software Engineering in [20]. Therefore, a case study for RhoArchitecture must seek to validate the most relevant characteristics or issues, namely: the creation and execution of JSON-DSL's, the creation of Components and Web Components, the use of Templates Engine, and the exchange of heterogeneous information. For short, the case study must allow us to validate RhoEngine and RhoModel as a whole. WebIDERho is included as the implementation of RhoModel for validation.

The methodology described in [19] corresponds to the **multi-case type**. The multi-case type in this context can be expressed as the set of characteristics and/or functionalities to be validated. Each case of

study covers part of the set, and the total number of cases must cover the whole set of characteristics. A feature can be validated by more than one case study. In general, although it can be extremely costly in time and execution [20], the evidence created from the multi-case type is considered robust and reliable.

5.1. Defining relevant characteristics of RhoArchitecture

Below, the list of the most relevant characteristics and/or functionalities for RhoArchitecture are described:

- C1. Implementation and execution of RhoEngine as a reusable JavaScript component and working at both web server (W) and web client (C) level.
- C2. Implementation and use of the MRho, MIRho and EditorRho languages (code editor based on CodeMirror [69]) of RhoModel for WebIDERho.
- C3. Implementation of case studies: HelloRho, Start Wars, Comic-SpeechRho, DrawRho.
- C4. Capacity to create of *RhoLanguages* and execution of RhoPrograms: (S) simple-simple (a program of a *RhoLanguage*); (P) multiple-simple (multiple programs of one *RhoLanguage*); (M) multiple-multiple (multiple programs written with multiple *RhoLanguages*).
- C5. Capacity to accept heterogeneous data sources: (X) XML; (J) JSON; (T) Text.
- C6. Creation and use of (C) *Components* y (W) *Web Components*.
- C7. Definition and use of *Templates Engine*: (P) Plain; (E) EJS; (H) Handlebars; and (O) Hogan.

It is worth highlighting that the components RhoEngine, MRho, MIRho, WebIDERho and EditorRho were also implemented using RhoModel, and thus, they also act as case studies for validation. Hence, with these components and the four case studies previously presented, the most relevant characteristics and/or functionalities of RhoArchitecture will be covered.

On the one hand, we will use several software metrics to validate the quality of the implementations of RhoEngine, RhoModel and the Case Studies, and in this way, to appraise characteristics C1–C3. On the other hand, characteristics and/or functionalities C4–C7 allow us to validate the RhoModel programming model and the RhoEngine functionality.

5.2. Validating the implementation of RhoArchitecture and case studies

In the field of Software Engineering, a software metric represents an objective measure to know or estimate a feature of an information system. Although there are a lot of software metrics in the literature, [78–80] present systematic reviews focused on software quality, reliability, documentation, and complexity, among others.

5.2.1. Software metrics used to validate JavaScript modules

In the context of our proposal, software metrics should focus on the analysis of JavaScript. In [81], software metrics are redesigned for prototype-based languages, such as JavaScript. In particular, the analysis will be performed on the generated JavaScript code using RhoModel, and we will compare them with recognized JavaScript components or modules such as: Bootstrap (bootstrap.com), CodeMirror [69], jQuery (jquery.com), Material Design Bootstrap [72], among others.

The software metrics that we have chosen to use in our analysis are:

- (i) *Lines Codes* [82] (LOC, physical lines code SLOC, logical lines code LLOC, comment lines code CLOC, and, blank lines code BLOC);
- (ii) *Cyclomatic Complexity* [83] (average per-function cyclomatic complexity CNN, and, cyclomatic complexity density for module CND [84]);

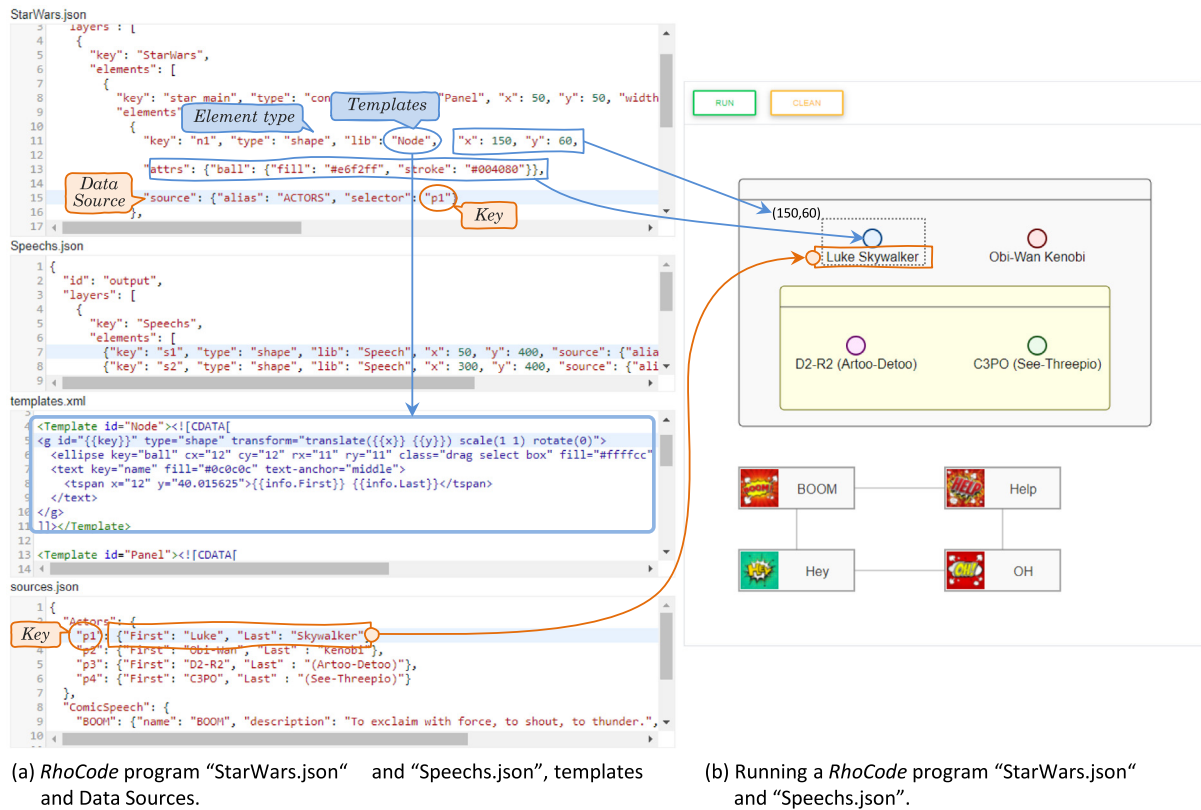


Fig. 9. Example of a DrawRho program for displaying a diagram.

- (iii) *Halstead Metrics* [85] (Halstead vocabulary size per-module HS, Halstead difficulty per-module HD, Halstead volume per-module HV, Halstead effort per-module HE, Halstead bugs per-module HB, Halstead time per-module HT, and, Average Halstead effort per-function HEF); and
- (iv) *Maintainability Index* MI [86].

We have used the following two tools to calculate the metrics above: *Excomplex* [87] implemented on NodeJS, provides an analysis of the software complexity of JavaScript Abstract Syntax Trees (ASTs); and *FrontEndART SourceMeter* for JavaScript [88,89], which is a tool for source code analysis that can perform a deep static analysis of complex JavaScript source code (sourcemeter.com).

5.2.2. Analysis of JavaScript code generated with RhoModel

Table 1 summarizes the JavaScript metrics of the code generation for the Components of the RhoArchitecture (noted by CRA, in yellows): RhoEngine, RhoModel, WebIDERrho and EditorRho. The four components were developed as projects that follow the RhoModel. There were created 20 files, of which 20% (4 files) are intended for the MRho specification and the remaining 80% (16 files) for MIRho implementation files. In these files, we find a total of 4422 lines of RhoModel source code that generated a total of 7808 lines of JavaScript code. This means a conciseness ratio of 1.77. It is worth noting that RhoEngine's conciseness ratio of 2.19 is the highest because the programming model supports code generation at both the web client and web server sides. On the other hand, CLOC comment lines represent 27% (average of the four projects), which is a very high rate of documented capacity (>25%, according to [40]), whereas RhoEngine has an average of 31%. This highlights the quality of existing RhoArchitecture documentation.

In details the code generation summary of the Case Studies (denoted by CAS, blue color) presented in this document. Ten files were written (40% MRho files and 60% MIRho files). The average conciseness ratio of 1.68 is acceptable, and on average it is 29%, which implies that the case studies are also well documented.

Table 2, details the code generation summary of the Case Studies (denoted by CAS, blue color) presented in this document. Ten files were written (40% MRho files and 60% MIRho files). The average conciseness ratio of 1.68 is acceptable, and on average it is 29%, which implies that the case studies are also well documented.

Table 3 shows the most relevant metrics for the frameworks used in WebIDERrho (denoted by FUW, in purple). We retrieve the source code of each framework in the version shown. If we analyze SLOC, MDB PRO has 53% and jQuery 17% of a total of 41,541, i.e., 70% of lines of code. On the other hand, RhoEngine (CLOC = 22%) and jQuery (CLOC = 17%) have "moderate documentation", the remaining frameworks are "poorly documented". An interesting finding is that 36% of lines in JQuery are CBLOC and BLOC and it has the lowest percentage of physical lines of code, SLOC = 64%.

Fig. 10 presents the percentage of SLOC, LLOC, CLOC and BLOC, for the total number of LOC lines, for all the components/frameworks analyzed in this article. From the total number of 58,926 lines of code, 85% are FUW, 14% are CRA, and 1% are CAS. Furthermore, the "good documentation" of CRA and CAS for FUW is ratified. BLOC is between 10%–20% in most components/frameworks, while SLOC is between 50%–80%. It is worth noting that the highest percentage of LLOC is for WebIDERrho with 72%.

Table 4 provides a summary of the software metrics (cyclomatic complexity, Halstead metrics and maintainability index) distributed over the three sets CRA, FUW and CAS.

Overall, according to Table 4, the software metrics obtained for CAS are very good. On the one hand, the cyclomatic complexity reflects "simple functionality, without too much risk" (CNN < 5 according to [83]). On the other hand, and according to [81,85], CAS have the following characteristics in average values: low complexity and lower associated functionality (HS = 120.5, HV = 5922), lower difficulty level (HD = 31), lower implementation time (HT = 15,970), minimum error estimation (HB = 2.0) and low understanding effort (HEF = 66).

Table 1

[CRA] Components of the RhoArchitecture: JavaScript generation code summary.

Project	File	RhoModel			JavaScript generated code					Conciseness RHOLOC/SLOC
		MRho	MIRho	RHOLOC	LOC	SLOC	LLOC	CLOC	BLOC	
RhoEngine	Rho.js (client)	237(1)	1,578(6)	1,815	2,375	1,347	1,350	714	314	2.19
	Rho.js (server)				1,598	847	809	510	241	
RhoModel	MRho.js	277(1)	949(6)	1,226	2,140	1,518	1,055	516	106	1.75
WebIDERho	WebIDERho.js	129(1)	910(3)	1,039	1,232	812	886	287	133	1.19
EditorRho	EditorRho.js	44(1)	298(1)	342	463	303	218	103	57	1.35
Total		687(4)	3,735(16)	4,422	7,808	4,827	4,318	2,130	851	1.77

Note: X(Y) MRho/MIRho; X: Number of Rho lines, Y: Number of Psi files; RHOLOC: total Rho lines of code; LOC: JavaScript generated lines; SLOC: Physical executable code lines; LLOC: Logical executable code lines; CLOC: Comments code lines; BLOC: Blank code lines.

Table 2

[CAS] Components Case Studies: JavaScript generation code summary.

Case Study (*.js)	RhoModel			JavaScript generated code					Conciseness SLOC/RHOLOC
	MPsi	MIPsi	RHOLOC	LOC	SLOC	LLOC	CLOC	BLOC	
HelloRho.js	13 (1)	30 (1)	43	67	34	35	22	11	1.56
StarWars.js	25 (1)	57 (1)	82	127	76	90	35	16	1.55
ComicSpeechRho.js	18 (1)	109 (1)	127	172	99	101	46	27	1.35
DrawRho.js	55 (1)	205 (3)	260	493	276	297	149	68	1.90
Total	111 (4)	401 (6)	512	859	485	523	252	122	1.68

Note: X(Y) MRho/MIRho; X: Number of Rho lines, Y: Number of Psi files; RHOLOC: total Rho lines of code; LOC: JavaScript generated lines; SLOC: Physical executable code lines; LLOC: Logical executable code lines; CLOC: Comments code lines; BLOC: Blank code lines.

Table 3

[FUW] Components/Frameworks used in WebIDERho: JavaScript code summary.

Framework/Component	File (*.js)	Version	JavaScript code				
			LOC	SLOC	LLOC	CLOC	BLOC
Bootstrap 4	bootstrap.js	4.5.0	4,421	3,225	2,644	314	852
CodeMirror 5	codemirror.js	5.1	9,807	7,802	6,180	1,016	989
jQuery	jquery.js	3.5.1	10,872	6,906	4,421	1,890	2,026
MDB jQuery PRO	mdb.js	4.19.2	23,117	22,064	9,146	799	254
Psi Engine	PsiEngine.js	3.0	2,024	1,514	1,023	440	88
Total			50,259	41,541	23,414	4,459	4,259

Note: LOC: JavaScript generated lines; SLOC: Physical executable code lines; LLOC: Logical executable code lines; CLOC: Comments code lines; BLOC: Blank code lines.

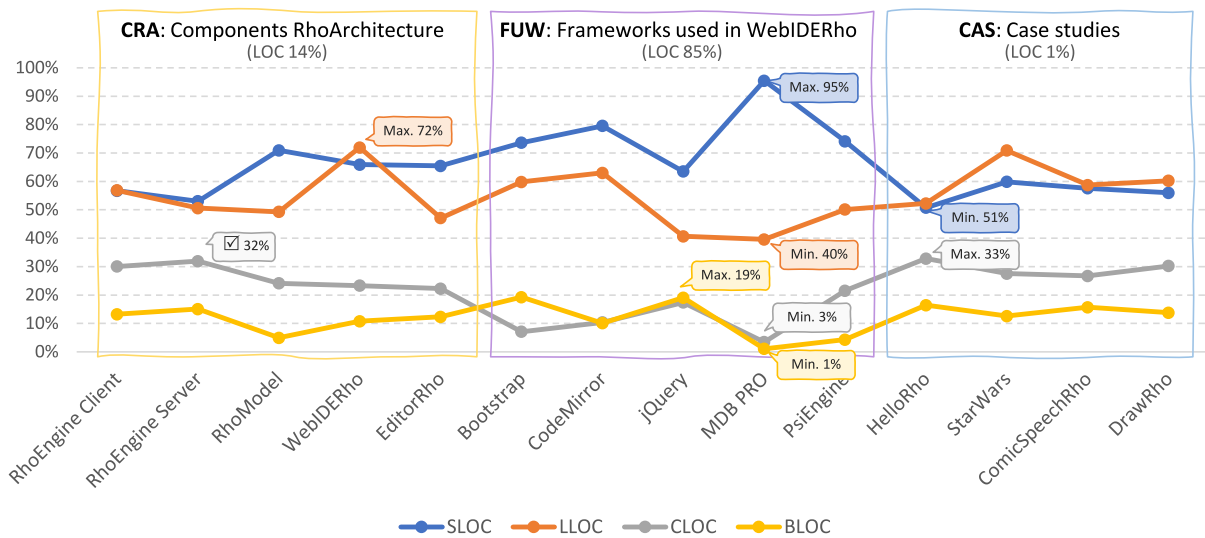


Fig. 10. Comparison of the SLOC, LLOC, CLOC and BLOC metrics between Components RhoArchitecture CRA, Frameworks used WebIDERho FUW and Case Studies CAS.

Hereafter, we will focus on analyzing and comparing the software metrics obtained for CRA and FUW. This analysis will help us to determine the quality, reliability, and complexity of our proposal.

Fig. 11 shows the graph of the software measures of Cyclomatic Complexity for the CRAs and FUW. For all components, it reflects that the functions/methods have a “simple functionality, without much

risk” (CNN < 5 according to [83]). On the other hand, [84] shows that the lower the CNN value, the simpler the productivity and software maintenance becomes, it is worth noting that the CNN value of CRA is 50% lower than that of FUW. Additionally, we highlight that our RhoEngine engine has better productivity, better maintainability, and less risk, than any FUW component.

Table 4

Summary of software metrics (cyclomatic complexity, Halstead metrics and maintainability index) from: RhoArchitecture, case studies and frameworks used.

Framework/Component	CNN	CND	HS	HD	HV*	HE ⁺	HEF	HB	HT*	MI
Rho Engine v1.1 (client)	1.9	14.3	610	154	68.9	10.63	2,663	23.0	590.4	114.3
Rho Engine v1.1 (server)	1.8	13.7	413	140	41.2	5.78	2,082	13.7	321.4	116.6
Rho Model v1.0	3.5	20.1	626	176	95.2	16.73	11,073	31.7	929.4	97.9
Web IDE Rho v1.1	1.7	9.4	575	118	54.8	6.45	2,336	18.3	358.5	114.2
Editor Rho v1.0	1.4	8.4	246	71	14.4	1.02	1,350	4.8	56.8	122.1
Bootstrap v4.5.0	2.4	18.2	1,296	166	174.1	28.97	3,487	58.0	1,609.7	109.6
CodeMirror v5.1	3.5	28.1	2,736	317	663.0	209.97	7,382	221.0	11,665.3	104.7
jQuery v3.5.1	3.5	31.6	1,801	259	334.8	86.65	6,003	111.6	4,814.0	107.4
MDB PRO 4.19.2	2.4	25.2	6,760	526	2,515.0	1,322.00	5,144	838.3	73,444.6	114.2
Psi Engine v3.0	3.7	29.7	813	185	112.2	20.71	5,984	37.4	1,150.4	105.5
HelloRho v1.0	1.0	2.9	57	10	1.2	0.01	246	0.4	0.7	140.9
StarWars v1.1	1.0	1.2	103	18	3.4	0.06	553	1.1	3.4	129.4
ComicSpeechRho v1.0	1.6	7.3	129	33	4.3	0.14	1,628	1.4	7.9	114.9
DrawRho v1.0	1.2	4.4	193	63	14.8	0.94	1,435	4.9	52.0	123.9

Note: CNN: Cyclomatic complexity average per-function; CND: Cyclomatic complexity per-module; HS: Halstead vocabulary size per-module; HD: Halstead difficulty per-module; HV: Halstead volume per-module (*thousands); HE: Halstead effort per-module (+millions); HEF: Average Halstead effort per-function; HB: Halstead bugs per-module; HT: Halstead time per-module (*thousands); MI: Maintainability Index.

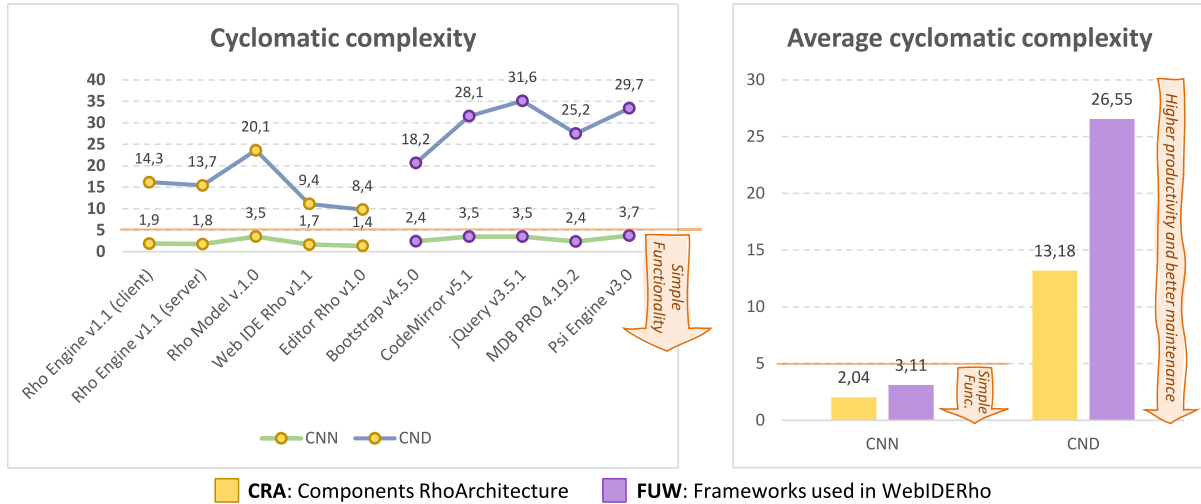


Fig. 11. Cyclomatic complexity graphs for the Components RhoArchitecture CRA, and the Frameworks used WebIDERho FFW.

Fig. 12 displays the graph of the Halstead metrics HS, HV, HD, HT, HB and HEF for the CRA and FFW components. Each metric defines an explanation note, the average per group and its respective interpretation.

The average AHS = 494 implies that all CRA components are less complex than FFW, and the average AHV = 55, implies that CRA has little associated functionality. In addition, 41.3% of HV is from MDB PRO, i.e., it is a rather complex framework, with a lot of associated functionality, also the difficulty level is high (HD = 526), has a long implementation time (HT = 53,444,000) and with high estimated error (HB = 838.3).

In comparison, the AHD difficulty level is 31% of the CRA components and 69% of the FFW components, i.e., approximately one-third. The difficulty level of RhoEngine is among the lowest (HD = 140, HD = 154), and it should be noted that it is a JSON-DSL execution engine. The implementation and understanding time (AHT = 451) are low for CRA and is approximately 41 times lower than the FFW components. Similarly, the estimated error (AHB = 18) of CRA is approximately 14 times lower than the estimated error of FFW.

Finally, the HEF values, i.e., the understanding effort for RhoEngine, EditorRho and WebIDERho are less than 3000, requiring less understanding than the FFW components, which are between 3000–8000. The effort to understand RhoModel is the highest (HEF = 11,073), and makes sense since we are establishing a new programming model.

In general terms, the CRA components are low in complexity, with little associated functionality that reduces complexity, low level of difficulty, low implementation and understanding, and low estimation of errors.

Finally, **Fig. 13** plots the Maintainability Index (MI) for CRA, CAS and FFW. As can be seen, all components and frameworks have a “good maintainability” (>85, according to [86]). Components developed from RhoModel, such as CRA and CAS, are higher than the average FFW. It is worth mentioning that RhoEngine and WebIDERho, cornerstone of our proposal, have better maintainability than recognized frameworks such as MDB PRO, jQuery, CodeMirror and Bootstrap. This contributes to validating the quality of RhoModel’s design and code generation.

In summary, the Components RhoArchitecture CRA and the CAS Case Studies succeed in obtaining the recommended values for the software metrics as specified in the literature. With this software metrics analysis, we have validated the relevant characteristics of C1–C3.

5.3. Results for case studies and components RhoArchitecture

This section is aimed at validating the relevant C4–C7 characteristics and/or functionalities of RhoArchitecture, aided by the Case Studies (HelloRho, Start Wars, ComicSpeechRho, DrawRho), and the RhoArchitecture Components (RhoEngine, RhoModel, WebIDERho and

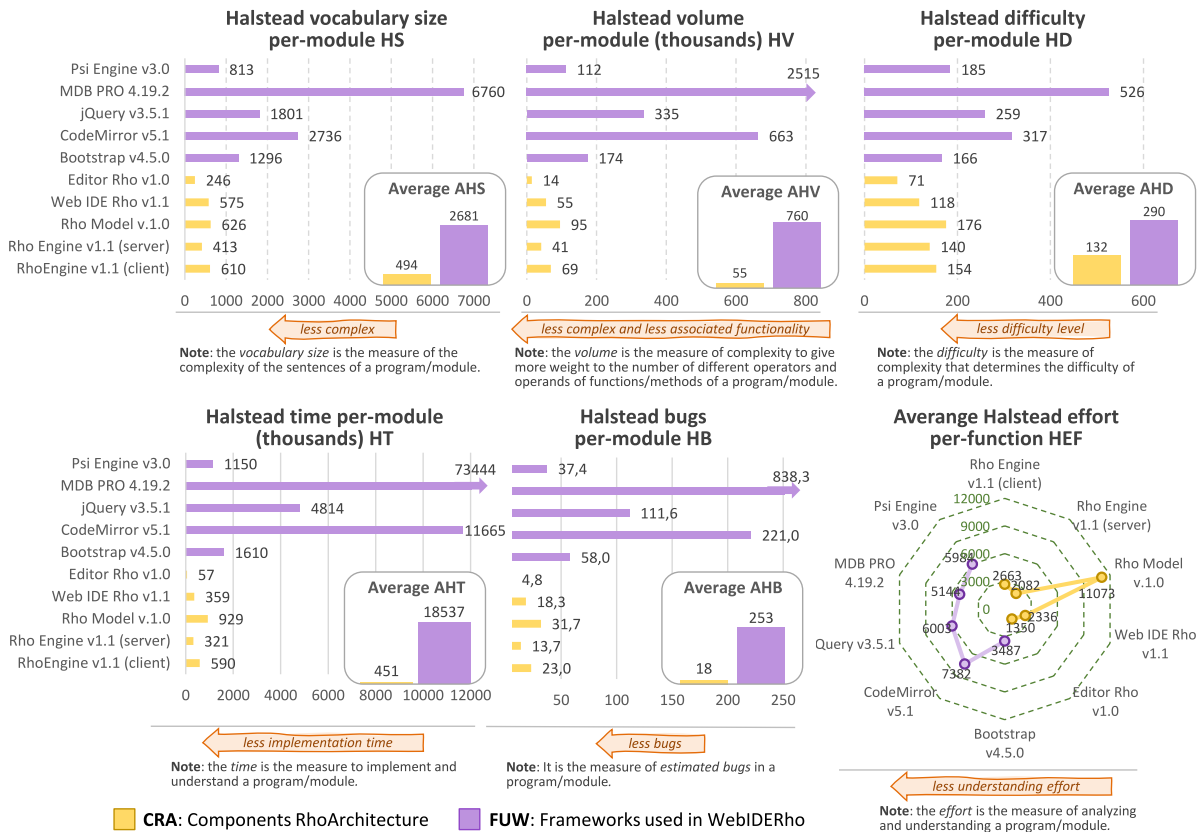


Fig. 12. Graphs of the Halstead metrics HS, HV, HD, HT, HB and HEF for the Components RhoArchitecture CRA, and the Frameworks used WebIDERho FUW.

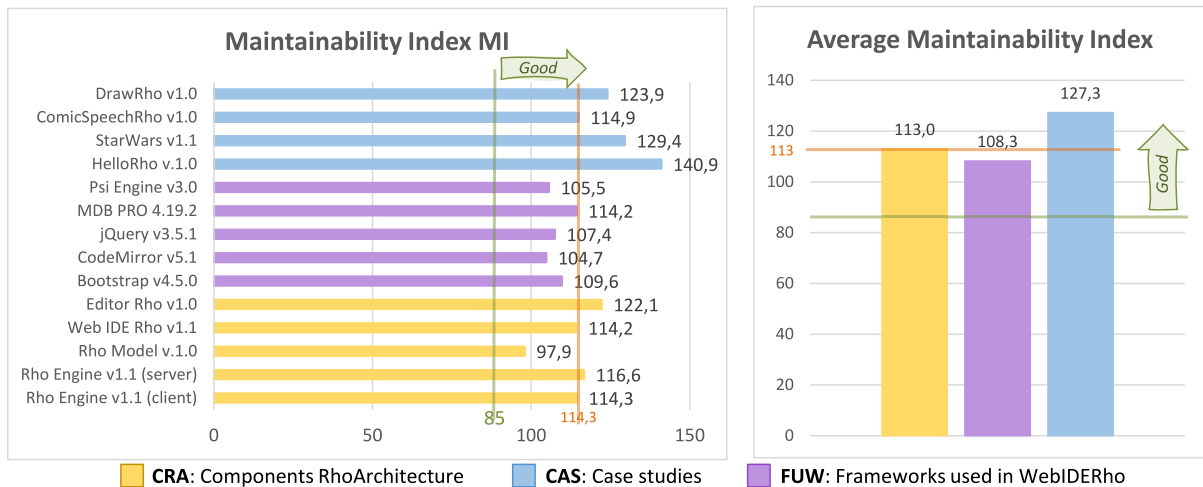


Fig. 13. Maintainability Index MI for RhoArchitecture CRA, Frameworks used WebIDERho FUW, and Case Studies CAS.

EditorRho). For each characteristic, it will be summarized how its objectives have been achieved.

○ Feature C4: Capacity to create RhoLanguages and execution RhoPrograms:

- (S) *simple-simple* (a program of a RhoLanguage): HelloRho is a RhoLanguage, where the program “hello.json” is written and executed. ComicSpeechRho is a RhoLanguage, where its program “code.json” is executed at the server level. Finally, ProjectRho is a RhoLanguage dedicated to the administration of a project in WebIDERho, it executes a program when a project is edited.

- (P) *multiple-simple* (multiple programs of a RhoLanguage): DrawRho is a RhoLanguage, where multiple programs (e.g. StarWars.json, Speechs.json) can be written and run on the same SVG canvas, with the opportunity to generate endless graphics.
- (M) *multiple-multiple* (multiple programs written in multiple RhoLanguages): MRho and MIRho are RhoLanguages of RhoModel, when you run the compilation and code generation of a project in WebIDERho, internally you are running one MRho program and multiple MIRho programs.

○ Feature **C5**: Capacity to accept heterogeneous XML, JSON and Text data sources:

- (X) XML: in RhoModel the templates for code generation are stored in XML. Similarly, in ComicSpeech the templates are saved in “templates.xml” file for page generation.
- (J) JSON: in WebIDERho the information of all projects grouped by user is stored in a JSON file.
- (XJ) XML/JSON: in DrawRho, the Actors and ComicSpeech information used in the diagram are stored in the file “sources.json”, and the templates of the graphic library “templates.xml” are stored in an XML file.
- (XJT) XML/JSON/Text: the Star Wars case study illustrates the flexibility of RhoModel to associate in the same class information stored in XML (“details.xml” file), JSON (“actors.json” file) and Text (“descriptions.txt” file).

○ Feature **C6**: Creation and use of Components and Web Components:

- (C) Components: RhoModel-based components have been created for all the projects studied in this analysis.
- (W) Web Components: DrawRho uses the SVG templates of graphical elements specified in XML in the “templates.xml” file, and the Handlebars template engine for the generation of graphical elements on an SVG canvas. On the other hand, in RhoEngine, you have the base classes for the creation of Web Components.

○ Feature **C7**: Definition and use of Templates Engine:

- (P) Plain: in RhoModel this engine is used to generate the JavaScript code of a project. While in Star Wars this engine is used to display the Star Wars actors’ information.
- (H) Handlebars: In WebIDERho, this engine is used to generate the development environment of a Rho project, a novel approach that may allow for multiple project types in the future. On the other hand, in DrawRho, this engine is used to create SVG graphic elements.
- (PEHO) All engines: the ComicSpeechRho case study is focused on using all available engines for verification.

Based on the above findings, we have qualitatively validated the relevant characteristics C4–C7.

5.4. Validation summary

Table 5 summarizes the validations for the list of relevant characteristics of RhoArchitecture. On the one hand, with the analysis of software metrics on the RhoArchitecture Components and the Case Studies, the relevant characteristics C1, C6–C7 were validated. On the other hand, the relevant characteristics C2–C5 were validated with the Case Studies.

To conclude, following the qualitative case study methodology suggested by [19] and adapted by [20], and using the validation summary of the relevant characteristics shown in Table 5, we provide a validation of the most relevant characteristics or aspects of RhoArchitecture, such as: the RhoModel programming model, the creation and execution of JSON-DSL’s, the creation of Components and Web Components, the use of Templates Engine, and the exchange of heterogeneous XML, JSON and Text information.

5.5. Validity threats

This section summarizes the threats to the internal and external validity of the performed work on the specification and execution of JSON-based Domain Specific Languages. Thus, we will discuss four identified threats to internal validity.

The first two threats are related to the evaluation using the multi-case type in different scenarios, and the consistency and precision of the results. Although the case studies allowed us to globally evaluate the features of the RhoArchitecture to solve domain-specific problems, more case studies are required to determine the efficiency, precision and reliability of JSON-DSLs in aspects such as: (i) concurrent execution of RhoCode program written in several RhoLanguages; (ii) run-time memory requirements; (iii) interaction between different RhoLanguages; (iv) capability to work with large volumes of data; and (v) load tests. On the one hand, these threats are mitigated with the quality of the JavaScript component associated with JSON-DSL and evaluated through different software metrics. On the other hand, performance and reliability are guaranteed by the browser and the NodeJS server.

The other two internal threats identified are related to error and exception evaluation, and semantic validation. The case studies in Section 4 help us control these threats, but again, it depends on the quality of the JavaScript component.

Regarding the external validity of the work, we want to discuss two threats. The first threat is related to the generalization of JSON-DSLs compared to Textual DSLs. For domain experts, it can be difficult or confusing to write solutions in JSON format compared to any other textual language. This threat can be mitigated with a good documentation and illustrative examples of the JSON-DSL. The second relevant threat refers to the usability of WebIDERho for the creation of JSON-DSL. Currently, WebIDERho is based on the PsiModel [9] implementation model, which uses code-behind techniques separating the specification from the implementation. To mitigate this threat, our future works include to extend the functionality of the Class Diagram based on DPG (Diagram Programming Generate [70,71]), so that the creation of a JSON-DSL can be performed in a visual way, that is, we will work in a Domain-Specific Visual Language for the specification and creation of JSON-DSL. Also, it should include the automatic generation of documentation and a debugging and execution area. We are aware that WebIDERho is limited, but it offers the necessary tools and features for our purpose, the construction and execution of JSON-based Domain Specific Languages.

6. Conclusions

Domain-specific languages enable the construction of software applications with high speed by increasing the productivity, of both software engineers and domain experts, due to the level of abstraction they provide. Building a DSL solution involves the use of tools for implementing interpreters and compilers. However, as we have shown, few approaches can create DSL alternatives with JSON grammar for web applications at both the web server and web client levels.

To address this initiative, in this article we have formalized and validated an architecture that allows us to work with JSON-based Domain Specific Languages (JSON-DSL) solutions to address domain-specific problems at both the web server and web client level, called RhoArchitecture. RhoArchitecture includes a RhoEngine evaluation engine, a RhoModel programming model and a WebIDERho lightweight web development environment. Our approach allows the creation and evaluation of JSON-DSLs, JavaScript Components and Web Components, Templates Engine, and the use of connectors to heterogeneous information sources (JSON, XML, Text formats) to encapsulate functionality and integrate them with other web widgets, components and/or frameworks, to create fast, robust, and flexible solutions for a web application. In this context, we have formally defined RhoLanguages and its grammar RhoGrammar to implement JSON-DSLs, which can be implemented with RhoModel and executed with RhoEngine.

To demonstrate the capabilities and potential of our approach, we have presented four Case Studies to validate the most relevant characteristics or aspects, such as the creation and execution of JSON-DSLs, the creation of Components and Web Components, the use of Templates Engine, and the exchange of heterogeneous information. In the first

Table 5
The RhoArchitecture relevant characteristics list and the validations summary.

	Relevant characteristics	RhoEngine	RhoModel	WebIDERho	EditorRho	HelloRho	Start Wars	ComicSpeechRho	DrawRho
C1	Implementation and execution of RhoEngine as a reusable JavaScript component and working at both web server (W) and web client (C) level.	IWC	EC	EC	EC	EC	EC	EW	EC
C2	Implementation and use of the MRho, MIRho and EditorRho languages of RhoModel for WebIDERho.	✓	✓	✓	✓	✓	✓	✓	✓
C3	Implementation of case studies: HelloRho, Start Wars, ComicSpeechRho, DrawRho.	✗	✗	✗	✗	H	S	C	D
C4	Capacity create RhoLanguages and execution RhoPrograms: (S) simple-simple; (P) múltiple-simple; (M) múltiple-múltiple.	✗	M	S	✗	S	✗	S	P
C5	Capacity to accept heterogeneous data sources: (X) XML; (J) JSON; (T) Text.	✗	X	J	✗	✗	XJT	X	XJ
C6	Creation and use of (C) Components y (W) Web Components.	CW	C	C	C	C	C	C	CW
C7	Definition and use of Templates Engine: (P) Plain; (E) EJS; (H) Handlebars; and (O) Hogan.	✗	P	H	✗	✗	P	PEHO	H

case, HelloRho allows us to create and execute a RhoLanguage. The second case study, Start Wars, highlights the use of multiple heterogeneous information sources (XML, JSON, and Text) implemented in a single class. The third case study, ComicSpeechRho, validates programming at the web server level, with the creation of a RhoLanguage, a web service, the use of Templates Engine and the design of web pages with material design. The last case study, DrawRho, comprehensively validates the most relevant aspects of RhoArchitecture.

In the field of Software Engineering, a software metric represents an objective measure to know or estimate a characteristic of an information system. The analysis presented in this article allows us to affirm that the RhoArchitecture Components (RhoEngine, RhoModel, WebIDERho and EditorRho) are not very complex: they have little associated functionality that reduces their complexity; their level of difficulty, implementation and understanding are low; error estimation is also low; and they have good maintainability. It is worth noting that these values were compared with recognized frameworks such as MDB PRO, jQuery, CodeMirror and Bootstrap, and that, in many of the cases, Components RhoArchitecture obtained better-recommended values in these software metrics.

As part of our future work, we aim to incorporate Domain Specific Visual Languages capabilities to the approach that is based on Diagram Programming Generate (DPG). In addition, we plan to develop new RhoLanguages such as BPMERho, which will allow the execution of BPMN 2.0, MDBRho for creating components, forms, navigation, dialogue boxes, block designs, etc. using Material Design Bootstrap, and RESTRho for specifying and executing a REST API through a JSON-DSL.

CRediT authorship contribution statement

Enrique Chavarriaga: Conceptualization, Methodology, Software, Validation, Data curation. **Francisco Jurado:** Conceptualization, Writing – original draft, Writing – review & editing. **Francy D. Rodríguez:** Investigation, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article

Acknowledgment

The validation of this was carried in collaboration with the research department (I+D+i) of UGround Global S.L. (<http://www.uground.com/>).

References

- [1] M. Voelter, *DSL Engineering: Designing, Implementing and using Domain-Specific Languages*, Dslbook, 2013.
- [2] M. Fowler, T. White, *Domain-Specific Languages*, Addison-Wesley Professional, Denver, 2010.
- [3] W3C, Extensible Markup Language (XML) Version 1.1, W3C Recomm, 2006, <https://www.w3.org/standards/xml/> (accessed August 30, 2021).
- [4] J. Fawcett, L. Quin, D. A. Beginning XML, fifth ed., Wrox Press, 2012.
- [5] W3C, Document Object Model (DOM) Level 3 Core Specification Version 1.0, W3C Recomm, 2004, <https://www.w3.org/TR/DOM-Level-3-Core/> (accessed September 3, 2021).
- [6] W3C, Scalable Vector Graphics (SVG) 2, W3C Candidate Recomm, 2018, <https://www.w3.org/TR/SVG2/> (accessed August 30, 2021).
- [7] W3C, Mathematical Markup Language (MathML) Version 3.0, W3C Recomm, 2014, <https://www.w3.org/TR/MathML3/> (accessed September 1, 2021).
- [8] W3C, XSL Transformation (XSLT) Version 2.0, W3C Recomm, 2021, <https://www.w3.org/TR/xslt20/> (accessed September 1, 2021).
- [9] E. Chavarriaga, F. Jurado, F. Díez, An approach to build XML-based domain specific languages solutions for client-side web applications, *Comput. Lang. Syst. Struct.* 49 (2017) <http://dx.doi.org/10.1016/j.cl.2017.04.002>.
- [10] E. Chavarriaga, F. Jurado, F. Díez, PsiLight: A lightweight programming language to explore multiple program execution and data-binding in a web-client DSL evaluation engine, *J. Univers Comput. Sci.* 23 (2017) 953–968.
- [11] ECMA, ECMA-404: The JSON Data Interchange Syntax, first ed., 2018, <https://www.ecma-international.org/publications-and-standards/standards/ecma-404/> (accessed September 2, 2021).
- [12] W3C Recommendation, JSON-LD 1.1: A JSON-Based Serialization for Linked Data (W3C Recommendation 16 July 2020), 2020, <https://www.w3.org/TR/json-ld/>.
- [13] Web Payments Working Group, JSON for Linking Data, 2022, <https://json-ld.org/>.
- [14] T. Ge, Y. Zhao, B. Lee, D. Ren, B. Chen, Y. Wang, Canis: A high-level language for data-driven chart animations, *Comput. Graph. Forum* 39 (2020) 607–617.
- [15] A. Sarasa-Cabezuelo, J.-L. Sierra, Grammar-driven development of JSON processing applications, in: 2013 Fed. Conf. Comput. Sci. Inf. Syst., 2013, pp. 1557–1564.
- [16] A.A. Frozza, R. Mello, S. dos, JS4Geo: a canonical JSON schema for geographic data suitable to NoSQL databases, *Geoinformatica* 24 (2020) 987–1019.
- [17] J. Xin, C. Afrasiabi, S. Lelong, J. Adesara, G. Tsueng, A.I. Su, et al., Cross-linking BioThings APIs through JSON-LD to facilitate knowledge exploration, *BMC Bioinformatics* 19 (2018) 1-N.PAG..
- [18] D.C. Schmidt, Model-driven engineering, *Comput* 39 (2006) 25–31.
- [19] R.K. Yin, *Case Study Research: Design and Methods*, fifth ed., Sage Publications, Inc., London, 2014.
- [20] P. Baxter, S. Jack, Qualitative case study methodology. Study design and implementation for novice researchers, *Qual. Rep.* (2008) 13–17.

- [21] M. Mernik, J. Heering, A.M. Sloane, When and how to develop domain-specific languages, *ACM Comput. Surv.* 37 (2005) 316–344.
- [22] D. Spinellis, Notable design patterns for domain-specific languages, *J. Syst. Softw.* 56 (2001) 91–99.
- [23] D. Ghosh, *DSLs in Action*, Manning Publications, Greenwich, 2010.
- [24] S. Erdweg, P.G. Giarrusso, T. Rendel, Language composition untangled, in: *Proc. 12th Work. Lang. Descr. Tools, Appl. LDTA 2012*, 2012, <http://dx.doi.org/10.1145/2427048.2427055>.
- [25] L.M. do Nascimento, D.L. Viana, P.A.S. Neto, D.A. Martins, V.C. Garcia, S.R. Meira, A systematic mapping study on domain-specific languages, in: *Seventh Int. Conf. Softw. Eng. Adv. (ICSEA 2012)*, 2012, pp. 179–187.
- [26] T. Kosar, S. Bohra, M. Mernik, Domain-specific languages: A systematic mapping study, *Inf. Softw. Technol.* (2016) 71, <http://dx.doi.org/10.1016/j.infsof.2015.11.001>.
- [27] A. Jung, J. Carbonell, L. Marchezan, E. Rodrigues, M. Bernardino, F.P. Basso, et al., Systematic mapping study on domain-specific language development tools, *Empir. Softw. Eng.* 25 (2020) 4205–4249.
- [28] D. Brown, J. Levine, T. Mason, *Lex & Yacc*, second ed., O'Reilly Media, New York, 1992.
- [29] J. Levine, *Flex & Bison*, O'Reilly Media, Sebastopol, 2009.
- [30] Microsoft, Modeling SDK for visual studio - Domain-specific languages, 2022, <https://docs.microsoft.com/en-us/visualstudio/modeling/modeling-sdk-for-visual-studio-domain-specific-languages?view=vs-2022>.
- [31] M. Bravenboer, K.T. Kalleberg, R. Vermaas, E. Visser, Stratego/XT 0.17. A language and toolset for program transformation, *Sci. Comput. Program.* 72 (2008) 52–70.
- [32] M. Mernik, M. Lenič, E. Avdičaušević, V. Žumer, LISA: An interactive environment for programming language development, in: *Int. Conf. Compil. Constr.*, 2002, pp. 1–4.
- [33] L.C.L. Kats, K.T. Kalleberg, E. Visser, Domain-specific languages for composable editor plugins, *Electron. Notes Theor. Comput. Sci.* (2010) 253, <http://dx.doi.org/10.1016/j.entcs.2010.08.038>.
- [34] H. Rajan, *ANTLR: A brief review*, 2022.
- [35] L. Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend*, Packt Publishing, 2013.
- [36] M. Toussaint, T. Baar, Enriching Textual Xtext-DSLs with a Graphical GEF-Based Editor, in: *LNCS*, vol. 10742, Springer Verlag, 2018.
- [37] R. Gronback, *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*, Addison-Wesley Professional, Denver, 2009.
- [38] M. Brambilla, J. Cabot, M. Wimmer, L. Baresi, *Model-Driven Software Engineering in Practice*, second ed., 2017.
- [39] A. Diez, N. Nguyen, F. Diez, E. Chavarriaga, MDE for enterprise application systems, in: *Model. 2013 - Proc. 1st Int. Conf. Model. Eng. Softw. Dev.*, 2013.
- [40] A.R. Da Silva, Model-driven engineering: A survey supported by the unified conceptual model, *Comput. Lang. Syst. Struct.* 43 (2015) 139–155.
- [41] L. Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend*, Packt Publishing Ltd, 2016.
- [42] D.G. Kourie, D. Fick, B.W. Watson, Virtual machine framework for constructing domain-specific languages, *IET Softw.* 3 (2009) 1–13.
- [43] D. Pollak, V. Layka, A. Sacco, DSL and Parser Combinator. *Begin. Scala* 3, Springer, 2022, pp. 237–245.
- [44] V. Subramaniam, *Programming DSLs in Kotlin*, Pragmatic Bookshelf, 2021.
- [45] K. Segeljak, *A Scala DSL for Rust code generation*, 2018.
- [46] F. Dearle, *Groovy for Domain-Specific Languages*, packt Publishing Ltd, 2015.
- [47] P. McGuire, *Getting Started with Pyparsing*, O'Reilly Media, Inc., 2007.
- [48] R.D. Kelker, *Clojure for Domain-Specific Languages*, Packt Publishing, 2013.
- [49] N. Valliappan, R. Krook, A. Russo, K. Claessen, Towards secure IoT programming in Haskell, 2020.
- [50] C. Yue, H. Wang, A measurement study of insecure javascript practices on the web, *ACM Trans. Web* 7 (2013) 1–39.
- [51] G. Czech, M. Moser, J. Pichler, Best practices for domain-specific modeling. A systematic mapping study, in: *2018 44th Euromicro Conf. Softw. Eng. Adv. Appl.*, 2018, pp. 137–145.
- [52] M. Eernisse, *Embedded JavaScript Templating (EJS)*, 2012, <https://ejs.co> (accessed November 15, 2021).
- [53] A. Mardan, *Template Engines: Pug and Handlebars. Pract. Node. js*, Springer, 2018, pp. 113–163.
- [54] Y. Katz, *Handlebars: minimal templating on steroids*, 2021, <https://handlebarsjs.com/> (accessed November 10, 2021).
- [55] Velasco A. Hogan, *JavaScript templating from Twitter*, 2021, <http://twitter.github.io/hogan.js/>.
- [56] *WEBCOMPONENTS.ORG*, Discuss & share web components, 2021, <https://www.webcomponents.org/> (accessed October 11, 2021).
- [57] W3C, *Introduction to Web Components*. W3C Work Gr Note, 2014, <https://www.w3.org/TR/components-intro/> (accessed October 1, 2021).
- [58] A. Gupta, M. Ahirwar, R. Pandey, Creating website as a service using web components, *Int. Res. J. Eng. Technol.* 6 (2019).
- [59] P.J. Molina, Quid: prototyping web components on the web, in: *Proc. ACM SIGCHI Symp. Eng. Interact. Comput. Syst.*, 2019, pp. 1–5.
- [60] K. Lano, Q. Xue, S. Kolahdouz-Rahimi, Agile specification of code generators for model-driven engineering, in: *ICSEA 2020*, 2020, p. 19.
- [61] T. Barth, I.P. Fromm, Modeling and code generation for safety critical systems, in: *Embed. World Conf.*, Vol. 2020, 2020.
- [62] G. Sebastián, J.A. Gallud, R. Tesoriero, Code generation using model driven architecture: A systematic mapping study, *J. Comput. Lang.* 56 (2020) 100935.
- [63] A. Prout, J.M. Atlee, N.A. Day, P. Shaker, Code generation for a family of executable modelling notations, *Softw. Syst. Model.* (2012) 11, <http://dx.doi.org/10.1007/s10270-010-0176-6>.
- [64] O. Fundation, Node.js: JavaScript runtime built on Chrome's V8 JavaScript engine, 2021, <https://nodejs.org/> (accessed October 30, 2021).
- [65] J. Wexler, *Get programming with Node. js*. Simon and Schuster, 2019.
- [66] B. Griggs, *Node Cookbook: Discover Solutions, Techniques, and Best Practices for Server-Side Web Development with Node. js* 14, Packt Publishing Ltd, 2020.
- [67] D. Flanagan, *JavaScript: The Definitive Guide: Master the World's Most-Used Programming Language*, seventh ed., 2020.
- [68] R. Ferguson, *JavaScript and Development Tools. Begin. JavaScript Ultim. Guid. to Mod. JavaScript Dev*, A Press, Berkeley, CA, 2019, pp. 11–24, http://dx.doi.org/10.1007/978-1-4842-4395-4_2.
- [69] M. Haverbeke, *CodeMirror*, 2017, <https://codemirror.net/>.
- [70] F. Rani, P. Diez, E. Chavarriaga, E. Guerra, J. de Lara, Automated migration of eugenia graphical editors to the web, in: *Proc. 23rd ACM/IEEE Int. Conf. Model Driven Eng. Lang. Syst. Companion Proc.*, 2020, pp. 1–7.
- [71] E. Chavarriaga, *Modelo Programable Para la Serialización y Evaluación de Mod-elos Heterogéneos en Clientes Web (Doctoral Thesis)*, Repository Autonomous University of Madrid, 2017.
- [72] *MDBootstrap*, Material Design for Bootstrap V5 & V4, 2022, <https://mdbootstrap.com/> (accessed June 28, 2021).
- [73] W.J. Wang, Y.W. Luo, X.L. Wang, X.P. Liu, Z.Q. Xu, Web services based framework for spatial information and services integration, 28 (2005) 1213–1222.
- [74] F. Rademacher, M. Peters, S. Sachweh, Design of a Domain-Specific Language Based on a Technology-Independent Web Service Framework. Vol. 9278, Springer Verlag, 2015, pp. 357–371, http://dx.doi.org/10.1007/978-3-319-23727-5_29.
- [75] U.-M. Schäfer, *SVG.js*, 2012, <https://svgjs.dev/> (accessed February 28, 2021).
- [76] J. Doyle, *GreenSock: Engaging the internet*, 2021, <https://greensock.com/> (accessed February 10, 2021).
- [77] *OMG*, *Business Process Model and Notation (BPMN)*, Version 2.0.4, 2014, <https://www.omg.org/spec/BPMN> (accessed February 20, 2023).
- [78] A. Tahir, S.G. Mac Donell, A systematic mapping study on dynamic metrics and software quality, in: *IEEE Int. Conf. Softw. Maintenance, ICSM*, 2012, <http://dx.doi.org/10.1109/ICSM.2012.6405289>.
- [79] M. Riaz, E. Mendes, E. Tempero, A systematic review of software maintainability prediction and metrics, in: *2009 3rd Int. Symp. Empir. Softw. Eng. Meas.*, 2009, pp. 367–377.
- [80] A. Jatain, Y. Mehta, Metrics and models for software reliability: A systematic review, in: *2014 Int. Conf. Issues Challenges Intell. Comput. Tech.*, 2014, pp. 210–214.
- [81] S. Ahsan, F. Hayat, M. Afzal, T. Ahmad, K.H. Asif, H.M.S. Asif, et al., Object oriented metrics for prototype based languages, *Life Sci. J.* 9 (2012) 63–66.
- [82] V. Nguyen, S. Deeds-Rubin, T. Tan, B. Boehm, A SLOC Counting Standard, *Univ South California, Cent Syst Softw Eng*, 2007.
- [83] T. McCabe, A complexity measure, *IEEE Trans. Softw. Eng.* SE-2 (1976) 308–320, <http://dx.doi.org/10.1109/TSE.1976.233837>.
- [84] G.K. Gill, C.F. Kemerer, Cyclomatic complexity density and software maintenance productivity, *IEEE Trans. Softw. Eng.* 17 (1991) 1284.
- [85] M. Halstead, *Elements of Software Science*, Comput Sci Libr, 1977.
- [86] P.W. Oman, J. Hagemester, D. Ash, A Definition and Taxonomy for Software Maintainability, *Univ. Idaho, Softw. Eng. Test Lab*, 1991.
- [87] J. Stilwell, *Escomplex Version 2.0.0-alpha*, 2021, <https://www.npmjs.com/package/escomplex> (accessed November 2, 2021).
- [88] *SourceMeter: Version 9.2*, Front Softw Ltd, 2021, <https://www.sourcemeter.com/> (accessed October 15, 2021).
- [89] R. Ferenc, L. Langó, I. Siket, T. Gyimóthy, T. Bakota, Source meter sonar qube plug-in, in: *2014 IEEE 14th Int. Work. Conf. Source Code Anal. Manip.*, 2014, pp. 77–82.