An LSP infrastructure to build EMF language servers for Web-deployable model editors*

Roberto Rodriguez-Echeverria[1], Javier Luis Cánovas Izquierdo[2], Manuel Wimmer[3], and Jordi Cabot[4]

[1] Quercus SEG, Universidad de Extremadura, Cáceres, Spain rre@unex.es
[2] UOC, Barcelona, Spain jcanovasi@uoc.edu
[3] CDL-MINT, TU Wien, Vienna, Austria wimmer@big.tuwien.ac.at
[4] ICREA – UOC, Barcelona, Spain jordi.cabot@icrea.cat

Abstract. The development of modern IDEs is still a challenging and time-consuming task, which requires implementing the support for language-specific features such as syntax highlighting or validation. When the IDE targets a graphical language, its development becomes even more complex due to the rendering and manipulation of the graphical notation symbols. To simplify the development of IDEs, the Language Server Protocol (LSP) proposes a decoupled approach based on language-agnostic clients and language-specific servers. LSP clients communicate changes to LSP servers, which validate and store language instances. However, LSP only addresses textual languages (i.e., character as atomic unit) and neglects the support for graphical ones (i.e., nodes/edges as atomic units). In this paper, we introduce a novel LSP infrastructure to simplify the development of new graphical modeling tools, in which Web technologies may be used for editor front-ends while leveraging existing modeling frameworks to build language servers. More concretely, in this work, we present the architecture of our LSP infrastructure, based on LSP4J, to build EMF-based graphical language servers.

Keywords: Domain-Specific Languages, Language Server Protocol, Modeling Editors

1 Introduction

In the last years, there is an increasing interest to move the modeling editing support to the Web (e.g., Eclipse Che[5] or Theia[6]), which gives new opportunities, but at the same time, opens new risks and challenges. In a Web-based solution, the deployment of a decoupled architecture composed by client-server is key to provide a performing and usable solution to users. In this scenario, we believe the use of protocols between clients and servers enables the development of decoupled solutions for graphical modeling

* This work has been supported by the Spanish government (ref. TIN2016-75944-R). Partially funded by the Austrian Federal Ministry of Science, Research and Economy and the National Foundation for Research, Technology and Development.

[5] https://www.eclipse.org/che/

[6] https://github.com/theia-ide/theia

languages. However, current graphical modeling editors are generally monolithic solutions, tailored to a particular language and built upon existing modeling frameworks. As result, they are not distributable, and therefore they cannot be easily deployed in the Web as client editors. Indeed, finding the proper approach to develop and maintain modern Web modeling editors reusing current modeling platforms still remains as a challenge. Additionally, the development of full-fledged graphical modeling tools is a challenging and complex task [14].

A similar conclusion has been drawn for programming IDEs (and, in general, textual editors) last year, thus motivating the creation of the Language Server Protocol (LSP)[7] as a community effort. LSP allows decoupling language IDE development into a client-server architecture, hence a language-agnostic client tool can connect to (potentially) any number of language-specific servers by means of a standard protocol.

Conversely, although the advantages of following the path defined by LSP may be clear for developing IDEs aimed at graphical languages, currently the question about how to do it properly remains open as LSP has been defined without considering this kind of languages. In a previous work [10] we discussed different alternatives of using LSP to decouple graphical modeling language IDEs as well as their pros and cons.

In this paper we present the details of the implementation of our infrastructure to build full-fledged EMF-based graphical language servers by leveraging an LSP reference implementation. Those graphical language servers support the connection of distributable generic client editors, which may be parameterized by a language description. This solution enables creating platform-agnostic client editors as they can be implemented in a platform different from the language server and contains no dependences to modeling frameworks. For illustration purposes, in this paper we present a web-based client that uses our EMF-based graphical language server.

This paper is organized as follows. Section 2 provides a brief introduction to LSP. Section 3 presents an illustrative implementation of our approach as a proof of concept. Section 4 discusses related work. And finally, in Section 5 we outline the main conclusions drawn by this work and point out the future work.

2 The language server protocol

The Language Server Protocol (LSP) is a communication language aimed at standardizing the messages exchanged between language client editors and servers. The LSP is based on an extended version of JSON RPC v2.0[8]. On the lowest level, JSON RPC just sends messages from a client to a server, which can be notifications, requests or responses. The relation between an incoming request and a sent response is done through a request id.

In an LSP-based architecture, clients provide a language-agnostic front-end editing support, thus being responsible for managing editing actions without knowing anything about the semantics of the language; and servers provide language-specific support, which mainly covers language semantics, e.g., they check for the correctness of the

[7] https://langserver.org

[8] https://www.jsonrpc.org/specification

source code and correspondingly send any issue to the client to be presented as syntactic errors. This architecture offers developers the freedom to choose the most suitable technology to implement the client editor and the language server independently of each other.

In its current state, LSP supports only text documents (i.e., textual programming languages). The protocol defines two main concept groups, namely: (1) the editor (or IDE) "data types" such as the currently open text document (i.e., URI), (2) the position of the cursor (i.e., line and column) or a range of selected text in the editor. It is important to note that these concepts are not at the level of a programming language domain model which would usually rely on abstract syntax tree elements and compiler symbols (e.g., resolved types, namespaces, etc.), thus simplifying the protocol significantly. This way LSP allows clients to connect to different language servers in a seamless way, thus easily providing support to new languages.

LSP v3.0 defines more than 40 different message types[9] describing requests, responses and notifications between client and server. These messages are organized according to their operational scope into five different categories: general, window, client, workspace and document (`textDocument`).

In the last months a number of language serversand clientshave appeared, and its number is growing. To develop a LSP-enabled server, several approaches have also appeared. In the context of this work, we will focus on the solution to develop LSP systems in Java, called LSP4J[10], as it is the programming language used to develop the Eclipse Modeling Framework. LSP4J is a Java binding for LSP which enables the implementation of both LSP clients and servers.

3 Approach

We propose an LSP infrastructure to edit EMF-based graphical languages in the Eclipse platform. Figure 1 illustrates our approach (LSP infrastructure is shaded), which includes the following elements: (1) a client, which provides a distributable generic editor for graphical languages; (2) a full-fledged graphical server, which keeps track of every change done in the editor and provides additional capabilities like validation or editing support (e.g., hovering), and (3) the usage of LSP with an Intermediate Representation Format (IRF), which represents the graphical model instances being edited and shared between client and server. Note that we are not extending LSP nor creating a new protocol, and therefore a mapping between editing operations and LSP messages is required[11].

We have created a proof-of-concept prototype to test our LSP infrastructure using a simple graphical language to represent families including concepts (family and member concepts) with names and edges between them (to link a member to a family)[12]. The

[9] `https://microsoft.github.io/language-server-protocol/specification`

[10] `https://github.com/eclipse/lsp4j`

[11] More information about this mapping can be found in a previous work [10] and use cases web page (`http://hdl.handle.net/20.500.12004/1/C/MODELS/2018/423`)

[12] Available at `http://hdl.handle.net/20.500.12004/1/A/LSP4GML/001`

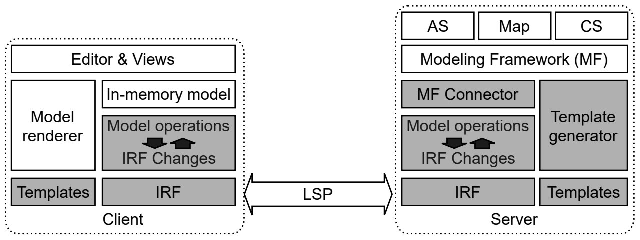


Fig. 1. LSP infrastructure proposed. AS = Abstract Syntax. CS = Concrete Syntax. Map = Mapping between AS and CS. IRF = Intermediate Representation Format.

prototype includes (1) a distributable generic client (Web-deployable model editor) and (2) a graphical language server, which both use the IRF and LSP to represent graphical modeling language instances and communicate with each other, respectively. Next we describe these elements.

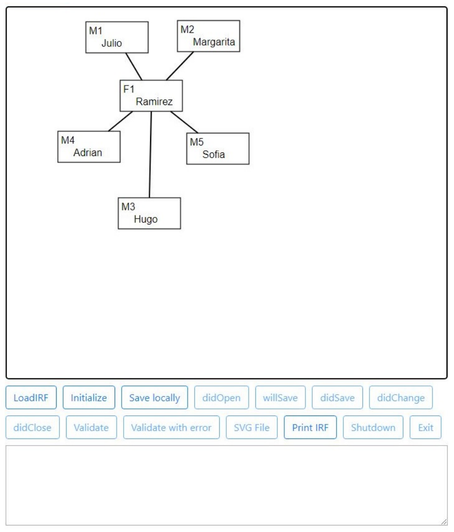
3.1 Distributable generic client

This case illustrates the opposite approach to the monolithic editor commonly found in current solutions (e.g., [6, 15, 4, 13, 11]). A distributable solution includes a (Web-friendly) model-driven tool to edit and manage any graphical language and relies on a server to perform most of the language operations. A generic model editor is parameterized by a language description, which configures the editor including the semantic- and diagram-aware available operations. This option enables the development of platform-agnostic editors as the graphical modeling editor can be implemented in a platform which is different from the language server.

In this work, for testing purposes, the client has been developed as a Web-based application using standard Web technologies (i.e., HTML, CSS, SVG and JavaScript). Figure 2a shows a snapshot of the developed client, including an instance of our example graphical language which models the members (see nodes with identifiers starting with `M`) of the `Ramirez` family (see node with `F1` identifier). The client interface includes a canvas, where the user can edit the elements of the language; a set of buttons to test the LSP messages and a log text box to visualize the status of the editor and communications. Note, however, that this work is focused on the development of graphical language servers.

The client always keeps an instance of the graphical modeling language expressed in IRF. Any change to the instance of the language updates the IRF definition, which is synchronized with the server by means of LSP messages (document category). Likewise, events and editing actions created at the server-side may also update the IRF definition and consequently the instance.

IRF definitions are expressed in JSON and includes three main elements, namely: (1) a metadata section, which includes information about the language being edited; (2) nodes, which represent an element in the graphical model; and (3) edges, which represent a relationship between two nodes in the graphical model. Figure 2b shows an excerpt of the IRF definition representing the instance shown in Figure 2a. It includes



(a)

```
{
  "rootElement": "Family",
  "namedElementPackage": "family.model.family.impl.NamedElementImpl",
  "implPackage": "family.model.family.impl.",
  "nodes": [
    {
      "id": "1",
      "type": "Family",
      "abstract": { "name": "F1", "lastName": "Ramirez" },
      "concrete": { "width": 100, "height": 50 },
      "diagram":  { "x": 182, "y": 117, "z": 0 },
      "editorOptions": { "movable": "True", "editable": "True" }
    },
    {
      "id": "2",
      "type": "Member",
      "abstract": { "name": "M1", "firstName": "Julio" },
      "concrete": { "width": 100, "height": 50 },
      "diagram":  { "x": 127, "y": 23, "z": 0 },
      "editorOptions": { "movable": "True", "editable": "True" }
    },
  "edges": [
    {
      "id": "1",
      "type": "default",
      "origin": "1",
      "target": "2",
      "abstract": {
        "name": "father",
        "multiplicity": "1",
        "containment": "yes"
      },
      "editorOptions": { "movable": "True", "editable": "True" }
    }, ...
}
```

(b)

Fig. 2. (a) Snapshot of the distributable generic client developed in our approach. (b) Excerpt of the IRF definition for the example shown in the snapshot.

the metadata information (see first three lines) and the definitions for two nodes (i.e., the family node, `F1`, and one of its members, `M1`) and one edge (i.e., the relationship between the previous nodes). As can be seen, node elements include information regarding the abstract and concrete syntax of the model elements (see the `abstract` and `concrete` keys in the `node` element), as well as additional information for identifying the node (`id` and `type` keys), layout information (`diagram` key) and behavior information (`editorOptions` key). Likewise, edge elements share part of the node structure but add information to specify the source and target of the relationship (`source` and `target` keys, respectively).

The diagramming support of the client editor relies on JointJS[13], a free diagramming library developed in JavaScript. To enable genericity in the client, the symbols of the language are retrieved from the server, which publishes a set of SVG-based templates for the language concepts via a specific LSP message. The client is able to render language symbols by injecting the information represented in the IRF definition into the SVG templates. Figure 3 shows an example of a SVG-based template for the concrete syntax used for the member nodes of Figure 2a. Thus, `text` SVG elements of Figure 3 are initialized with the values of the corresponding keys of the `abstract` section of the IRF definition. These templates are used to configure the JointJS-based diagram editor, thus the user can drag&drop them to create instances of the graphical language.

[13] `https://www.jointjs.com`

```
<g class="rotatable">
  <g class="scalable">
    <rect y="0" x="0" height="10" width="20" style="color:#000000;opacity:1;vector-effect:none;
       fill:#ffffff;fill-opacity:1;stroke:#000000; stroke-width:0.25;stroke-opacity:1;"/>
  </g>
  <text id="name" y="20" x="5" style="font-family:sans-serif;fill:#000000;fill-opacity:1;"></text>
  <text id="firstName" y="40" x="25" style="font-family:sans-serif;fill:#000000;fill-opacity:1;"></text>
</g>
```

Fig. 3. Example of SVG-based template.

3.2 Graphical language server

A graphical language server comes into play when the client editor does not implement the full set of operations identified for graphical languages. A full-fledged graphical language server presents more functionality than textual language servers, basically because it needs, firstly, to provide a richer editing support to encompass two different syntaxes (abstract and concrete)[14], and secondly, to provide a more complex diagramming support to cover semantic- and diagram-aware specific capabilities (e.g., autolayout).

We consider that a graphical language server must provide (at least) the following stack of functionalities (ordered by dependence): (1) management and validation of abstract model instances; (2) management and validation of concrete model instances; (3) a way to provision of language descriptions (stencil palette & compositional rules) to clients; (4) diagrammatic operations, e.g., autolayout by means of ELK[15]; (5) auxiliary operations, e.g., serialization support. Note that some of these functionalities could also be partially supported by the client, for instance, it may apply compositional rules to keep a basic layout.

Our implementation of a language server relies on the Eclipse Modeling Framework (EMF) and the Graphical Modeling Framework (GMF) to provide model management, validation and storage of graphical languages. While EMF offers the core support to manage and validate models and metamodels, GMF provides the required support to define the concrete syntax of graphical languages. With this solution, any graphical language of the Eclipse ecosystem could be potentially supported.

Figure 4 presents the overall architecture of an EMF graphical language server according to our approach. As shown, three main modules are defined: (1) the LSP4J language server, implementing the core functionality of the language server; (2) the IRF (de)serializer, which reads/writes IRF files or snippets; and (3) the Model Framework (MF) connector, providing a common interface to different modeling frameworks.

The language server is implemented upon the LSP4J API, which requires defining a proper implementation for the following interfaces: *LanguageServer*, *TextDocumentService* and *WorkspaceService*. Those interfaces provide methods to handle LSP messages from different categories, namely *LanguageServer* takes care of general messages (e.g., initialize, shutdown, etc); *TextDocumentService* handles the text document synchronization messages; and *WorkspaceService* implements the methods for command execution and workspace management.

[14] Textual languages are also defined by abstract/concrete syntaxes but the abstraction gap between them is much shorter than the one for graphical languages.

[15] https://www.eclipse.org/elk/

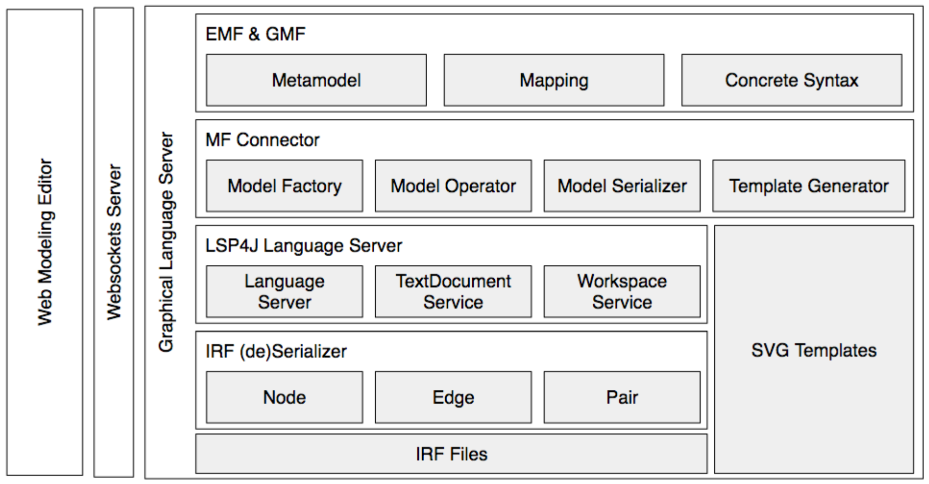


Fig. 4. LSP-based graphical language server architecture

The IRF de(serializer) contains the modules to save/load IRF files from/into objects in memory. Its main modules represent the main elements of an IRF file, such as nodes and edges.

The Model Framework connector needs to be tailored to the concrete modeling framework used for language definition and, also, to the specific language defined. Therefore, this module may bridge the gap between any modeling framework and an LSP-compliant language server. Basically, this connector provides the following relevant modules: a model factory, a model operator, a model serializer and a template generator.

The model factory is used by the language server to produce an EMF-compliant model instance from an IRF file or snippet. Therefore, this modules needs to generate elements that are specific of the language in use. There are two main alternatives to do that: (i) code generation, and (ii) using the reflective EMF API. In our case, we have used the reflective EMF API to generate a model instance from an IRF file. Our algorithm generates a new element from every node, initializing its properties according to the abstract and concrete sections of its IRF specification, and satisfies their relationships by parsing the IRF edges.

The model operator implements a gateway to execute different operations on the EMF model, such as validation or type-based selection. The values returned are correspondingly parsed into LSP messages by the LSP4J modules in order to generate the appropriate response or notification for the Web model editor (client).

The model serializer is in charge of transforming model instances into the IRF, so that they can be managed by the language server and, then, edited remotely by the Web tool. It must consider both abstract and concrete syntax of every model element to generate the proper IRF node representation (abstract and concrete sections). This module is useful to load models serialized in a format that EMF can handle, e.g. XMI.

Finally, taking the concrete syntax as input, the template generator produces SVG specifications of the stencils associated to each model element of the graphical language. Those SVG templates may afterwards be sent to the Web model editor as a response to specific LSP message (executeCommand), hence the Web front-end may update its stencils palette for each model instance according to its graphical language.

As a final note, LSP4J provides standard socket connection, working properly for local inter-process communication, but not adequate for connections through the Web. Note that nowadays language servers and IDEs are usually run in the same machine, mainly because performance and security issues. Therefore, in order to provide remote connection for Web model editors, it is necessary to include a WebSockets server acting as a gateway between the Web client and the graphical language server.

4 Related work

The use of LSP for programming languages is an emerging working area where companies are addressing the development of language servers for well-known languages.There are also similar efforts for the development of language servers for textual domain-specific languages, like the support provided by Typefox using Xtext (i.e., the framework generates the server for any Xtext-based language). However, to the best of our knowledge, little attention has been paid for the development of client/servers for graphical languages.

Only a few initiatives aim at protocol-enabled client-server solutions, namely: Obeo, Sprotty and Eclipse Che. Obeo proposes a solution relying on an extension of LSP for graphical languages. This initiative was released in March 2018 and is still under heavy development. Typefox is developing Sprotty[16], a framework which offers a web-based environment for graphical languages. The tool relies on Xtext and LSP to synchronize a textual language instance between client and server, which is then rendered at client-side to show a diagram projection. The projection is only a view and therefore no edition is allowed. Eclipse Cheis maybe a special case, as it aims at providing an IDE in the Cloud. Eclipse Che incorporates a server which provides the set of IDE capabilities of the original desktop-based Eclipse. The server can also connect to LSP language servers to provide support to additional textual programming languages. Thus, Eclipse Che actually works as a generic client for textual LSP language servers. However, the support for graphical languages is still very limited.

Our work is also related to those approaches developed to deliver modeling functionality as a service [9] (a.k.a., Model as a Service, MaaS). A MaaS approach can cover any modeling functionality (e.g., model storage, using CDO[17] or Morsa [3]; or code generation as described by Crocombe et al. [1]) but we are specially interested in the support for designing and creating graphical models. Thus tools such as AToMPM [12], GenMyModel [2] and WebGME [7] propose client-server solutions where users can create and edit instances of graphical modeling languages. However, all of them rely on proprietary protocols for the client-server communication.

[16] `https://github.com/theia-ide/sprotty`
[17] `https://www.eclipse.org/cdo`

Several works propose the development of stand-alone Web-based modeling environments (e.g., [6, 15, 4, 13, 11]). However, we may consider these kind of works as monolithic editors.

5 Conclusion and future work

In this paper we present the design and implementation of an infrastructure to build EMF-based model language servers relying on standard LSP and a text-based model representation shared between clients and servers. From our point of view, reusing current model-driven technologies is key to language servers affordable implementation. Moreover, we provide a proof-of-concept implementation of a generic Web-based client editor connecting to our EMF language servers. As future roadmap, the following working lines are worth to be highlighted.

Generation of language servers. An automatic approach to generate the required server implementation would promote the adoption of decoupled solutions. This implementation could be derived from a textual definition of the language abstract and concrete syntaxes (e.g., the one used in the Collaboro approach [5]). In our LSP infrastructure, it could be generated: (1) the modeling framework connector for the specific language (i.e., model factory and serializer), and (2) the language description in SVG format (i.e.,template generator).

Performance assessment. LSP-based client-server synchronization in a fully decoupled scenario (i.e., both ends are deployed in different systems) may entail a high bandwidth consumption and therefore significantly impact the editor performance. The fewer the operations handled at client-side the greater the number of messages to send to the server. Therefore, at least for Web-deployable clients, it seems coherent to provide support to as many operations as possible to reduce bandwidth without compromising editor genericity. Further studies in real cases should be carried out to identify performance issues and derive proper optimizations for the LSP infrastructure. Additionally, one of the main bottlenecks of the current implementation is the lack of support of incremental updating of EMF models. Therefore, every time an IRF instance needs to be operated (e.g., for validation) the corresponding EMF model needs to be generated from scratch. This continuous generation of EMF models may have a significant impact on tool performance. We plan to deal with it in future versions of the infrastructure.

IRF evolution. IRF design has been mainly driven for two main forces: (1) the necessity of a representation encompassing abstract and concrete syntaxes properties in a single resource; and (2) the provision of the strictly necessary information to the client side. Its definition has been inspired by OMG's Diagram Interchange Format [8] and the JSON-based protocol used by the AToMPM graphical modeling tool [12]. Nevertheless, further and more complex case studies are needed for a proper evaluation of its completeness and suitability.

Security issues. The development of our Web-deployable model editor raised some security issues, e.g. Cross-Origin Resource Sharing (CORS). Security issues need to be analysed in further works.

References

1. Crocombe, R., Kolovos, D.S.: Code Generation as a Service. In: Int. Workshop on Model-Driven Engineering on and for the Cloud. pp. 25–30 (2015)
2. Dirix, M., Muller, A., Aranega, V.: An Online UML Case Tool. In: Europ. Conf. on Object-Oriented Programming (2013)
3. Espinazo-Pagán, J., Cuadrado, J.S., Molina, J.G.: A Repository for Scalable Model Management. Software and System Modeling **14**(1), 219–239 (2015)
4. Hiya, S., Hisazumi, K., Fukuda, A., Nakanishi, T.: clooca : Web based tool for Domain Specific Modeling. In: Demo at Int. Conf. on Model Driven Engineering Languages and Systems. pp. 31–35 (2013)
5. Izquierdo, J.L.C., Cabot, J.: Collaboro: a Collaborative (Meta) Modeling Tool. PeerJ Computer Science **2**, e84 (2016)
6. Leal, J.P., Correia, H., Paiva, J.C.: Eshu: An Extensible Web Editor for Diagrammatic Languages. In: Symp. on Languages, Applications and Technologies. pp. 12:1–12:13 (2016)
7. Maróti, M., Kecskés, T., Kereskényi, R., Broll, B., Völgyesi, P., Jurácz, L., andÁkos Lédeczi, T.L.: Next Generation (Meta)Modeling: Web- and Cloud-based Collaborative Tool Infrastructure. In: Workshop on Multi-Paradigm Modeling. pp. 41–60 (2014)
8. OMG: UML Diagram Interchange. OMG (2006)
9. Popoola, S., Carver, J., Gray, J.: Modeling as a service: A survey of existing tools. In: Workshop on Model-driven Engineering Tools (MDETools'17). vol. 2019, pp. 360–367. CEUR Workshop Proceedings (2017)
10. Rodriguez-Echeverria, R., Cánovas Izquierdo, J., Wimmer, M., Cabot, J.: Towards a Language Server Protocol Infrastructure for Graphical Modeling. In: Int. Conf. on Model Driven Engineering Languages and Systems. p. (on press) (2018)
11. Rose, L., Kolovos, D., Paige, R.: Eugenia live: a flexible graphical modelling tool. In: Workshop on Extreme Modeling. pp. 15–20 (2012)
12. Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Mierlo, S.V., Ergin, H.: AToMPM: A Web-based Modeling Environment. In: Demo at Int. Conf. on Model Driven Engineering Languages and Systems. pp. 21–25 (2013)
13. Thum, C., Schwind, M., Schader, M.: SLIM - A Lightweight Environment for Synchronous Collaborative Modeling. In: Int. Conf. on Model Driven Engineering Languages and Systems. pp. 137–151 (2009)
14. Vuyović, V., Maksimović, M., Perišić, B.: Sirius: A rapid development of DSM graphical editor. In: Int. Conf. on Intelligent Engineering Systems. pp. 233–238 (2014)
15. Wimmer, M., Garrigós, I., Firmenich, S.: Towards Automatic Generation of Web-Based Modeling Editors. In: Int. Conf. on Web Engineering. pp. 446–454 (2017)