"This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 813884".

Project Number: 813884

Project Acronym: Lowcomote

Project title: Training the Next Generation of Experts in Scalable Low-Code Engineering Platforms

D3.1. Cloud-Based Low-Code Engineering Editor - Interim Version

---

Project GA: 813884

Project Acronym: Lowcomote

Project website: https://www.lowcomote.eu/

Project officer: Dora Horvath

Work Package: WP3

Deliverable number: D3.1

Production date: November 16th 2020

Contractual date of delivery: November 30th 2020

Actual date of delivery: November 30th 2020

Dissemination level: Public

Lead beneficiary: Universidad Autónoma de Madrid

Authors: Lissette Almonte, Juan de Lara, Fatima Rani

Contributors: The Lowcomote partners

HISTORY OF CHANGES		
Version	**Publication date**	**Change**
1.0	November 16th 2020	▪ Initial version
1.1	November 25th 2020	▪ Revised version

Project Abstract

Low-code development platforms (LCPD) are software development platforms on the Cloud, provided through a Platform-as-a-Service model, which allows users to build completely operational applications by interacting through dynamic graphical user interfaces, visual diagrams and declarative languages. They address the needs of non-programmers (so called citizen developers) to develop personalised software, and focus on their domain expertise instead of implementation requirements.

Lowcomote will train a generation of experts that will upgrade the current trend of LCPDs to a new paradigm, Low-code Engineering Platforms (LCEPs). Our envisioned LCEPs will be:

- *open*, allowing to integrate heterogeneous engineering tools;
- *interoperable*, allowing for cross-platform engineering;
- *scalable*, supporting very large engineering models and social networks of developers, and
- *smart*, simplifying the development for citizen developers by machine learning and recommendation techniques.

This vision will be achieved by injecting in LCDPs the theoretical and technical framework defined by recent research in Model Driven Engineering (MDE), augmented with Cloud Computing and Machine Learning techniques. This is possible today thanks to recent breakthroughs in scalability of MDE performed in the EC FP7 research project MONDO, led by Lowcomote partners.

The 48-month Lowcomote project will train the first European generation of skilled professionals in LCEPs. The 15 future scientists will benefit from an original training and research programme merging competencies and knowledge from 5 highly recognised academic institutions and 9 large and small industries of several domains. Co-supervision from both sectors is a promising process to facilitate agility of our future professionals between the academic and industrial world.

Executive Summary

This document describes the design and early prototype of a cloud-based, low-code engineering editor, with unified support of heterogeneous technologies. The document also describes the design of a system to facilitate the construction of backends for recommender systems, which can be integrated with editors for domain-specific languages. In further deliverables (D3.3), these two parts will be integrated, including a front-end based on natural language to access the recommendation.

Table of contents

1. Introduction

The present document is a deliverable of the Lowcomote project (Grant Agreement n°813884), funded by the European Commission Research Executive Agency (REA), under the Innovative Training Networks Programme of the Marie Sklodowska Curie Actions (H2020-MSCA-ITN-2018). The purpose of this document is to provide an overview of the design decisions and first prototypes for a cloud-based low-code engineering editor, with unified support for heterogeneous technologies, and for customized recommendations.

Figure 1 shows a high-level structure of the architecture. Both the graphical editor and the recommender systems will be the front-ends of the envisioned LCEP, called *Lowcomotive*. Both components can be tailored to specific domains – since the goal is that they can be reused to create LCEP in arbitrary domains – and be deployed on a cloud infrastructure. Such components need to interact with the model repository (designed in WP4). The graphical editor is the focus of the work of ESR2, while the recommender is the focus of ESR1. Both components will be analysed in the next two sections.

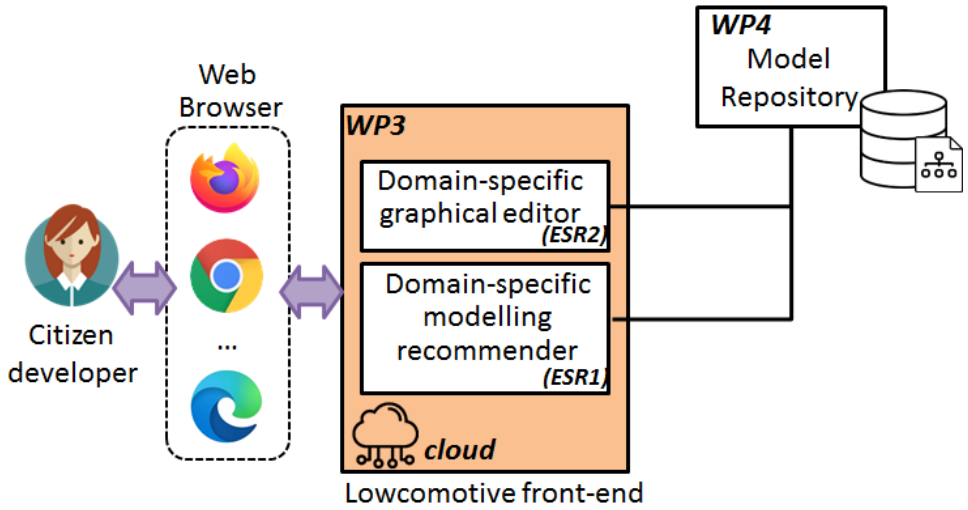


Figure 1.: High-level overview of the architecture

2. Cloud-Based Domain-specific Graphical Modelling Environments

As the project proposal mentions, LCDPs allow describing different aspects of an application using graphical models. However, when the targeted application is complex or encompasses many concepts, their models become large and, without appropriate tool support, they get difficult to create, reuse, navigate, and comprehend. Hence, mechanisms to make modelling more scalable are needed.

There are a few domain-specific modelling frameworks for web-based editing, but creating web-based graphical editors with existing frameworks is still hard and time-consuming due to their low-level code nature. Moreover, the created editors are not scalable beyond tens of elements, are tied to a modelling technology, do not enable rich modelling of editor aspects (e.g., domain-specific abstractions), or do not connect different languages through views.

To alleviate these problems, the Lowcomote project proposes a novel approach to ease the creation of multi-view graphical editors for the Cloud. Instead of relying on low-level JavaScript graphical frameworks, our proposal is founded on language engineering principles. This way, all aspects of the editor (abstract and concrete syntax, user interaction, view definitions and applicable abstractions) will be described through models. The graphical front-ends will be decoupled from the back-end modelling technology, to enable heterogeneous cross-modelling solutions e.g. based on Eclipse EMF, JSON, Ontologies or proprietary knowledge-based representations like the one supported by UGROUND's ROSE [DNF+20]. To enable more scalable modelling, the approach will provide extensible libraries of model abstractions and graph summarization techniques, to support creating more succinct model views. A Cloud-based modelling environment will be ideal for this purpose, to provide enough computation power to perform complex abstractions (enabling better model comprehension and navigation) over large models.

Another goal is being able to profit from existing editors built for e.g., the Eclipse ecosystem. This way, automated migration from such desktop scenarios into a web editor would be desirable, to enable their integration with LCEPs.

In the rest of this section, we first provide an state-of-the-art revision of some of the current main frameworks and approaches, and then report on the approach we have taken (partly based on [RDC+20]).

2.1 Web-based graphical modelling environments: state of the art

Environments to automate the development of graphical DSLs (DSLs) have existed since the end of the 90s. Tools like KOGGE [EWD+96], DOME [BGS+10], GME [LMK+02], Diagen [M02], MetaEdit+ [KT08] or AToM^3 [dLV02], have laid the foundations of some of the tools in use today.

The second wave of tools for graphical DSL definition started with the emergence of model-driven engineering approaches to software development [BCW17], and especially with the popularization of the Eclipse framework. This lead to a plethora of tools targeting the generation of editors for this environment, like Tiger [BEE+08], the Graphical Modelling Framework (GMF) [GMF20] which is based on the Eclipse Graphical Editing Framework (GEF) [GEF20], EuGENia [KGR+17], Spray [GB16], Graphiti [Gra20], and Sirius [Sir20].

Graphical DSLs also play a fundamental role in LCDEs. However, because LCDEs are based on cloud infrastructure, therefore require web-based editors. Hence, there is a third wave of tools for automating the creation of web-based graphical editors, which are the most interesting

for our project, since they can be integrated into LCEP. We review next the most representative ones. Please note that we focus on high level frameworks for their creation (i.e., based on software language engineering principles) and not on low-level frameworks based on JavaScript libraries, since we want to compare editor features.

WebGME [MKK+14] is a web-based evolution of the GME [LMK+02] environment. WebGME is a tool to create graphical DSLs directly in the browser. It is based on software language engineering principles, using a UML class diagram-based meta-models to specify the modelling concepts, relationships and attributes. It also supports model versioning and collaboration on the cloud.

AToMPM [SVM+13] is a web version of AToM^3 [dLV02]. It allows defining graphical DSL editors that run on the web, and to specify DSL semantics using graph transformations [KEP+06]. It supports two types of collaboration mechanisms in real-time. On the one hand, *screenshare* allows two or more clients to share exactly the same drawing area: any modification made to a model (abstract or concrete syntax) is replicated on all observing clients. On the other, *modelshare* only shares the abstract syntax of a model between clients.

Eclipse Theia [The20] is an open-source IDE platform that runs in browsers and on desktops. Theia provides three main elements: First, a customizable "workbench" supporting view, editors, menus, toolbars, etc. This provides the frame to embed modelling-related features, such as graphical editors, code generators and so on. Second, a flexible extension mechanism to add custom features, but also to reuse existing modules provided by frameworks. Third, based on this extension mechanism, the tool makes available a collection of reusable generic features, such as Git integration, a file explorer or a search feature.

Sprotty [Spr20] is an Eclipse project that enables adding diagrams to web applications with little effort. It is a framework -- at much lower level than tools such as AToMPM or WebGME -- based on SVG for rendering and CSS for styling. However, we review it here, since it has been integrated with Eclipse Theia to provide support for diagrammatic views. Sprotty's reactive architecture makes possible to distribute the execution of a diagram arbitrarily between a client and a server, which matches the scenario of the Language Server Protocol (LSP, see below).

EMF.cloud [EC20] is a project – still under development – aiming at making EMF-based technologies accessible via the cloud, including graphical editors, based on Eclipse Theia. Its central component is the model server, which provides a set of APIs to connect model clients to model instances (similar to EMF-Rest [ECG+16]). However, it additionally enables synchronization of changes and command-based modifications across multiple modelling editors that may run in parallel on a client. It also allows retrieving model instances in different formats, e.g. as JSON. This is enabled by another sub-component of EMF.cloud, the EMF to JSON converter. Basically, the model server is like a *ResourceSet* with an *EditingDomain* for a client-server scenario. Based on the model server and the Graphical Language Server Protocol (GLSP) [RCW+18], EMF.cloud hosts a browser-based version of the Ecore tools based on Eclipse Theia allowing creating Ecore models in the browser. This also includes a tree-based form editor similarly to what we can generate with EMF.

GLSP [RCW+18]. The Graphical Language Server Platform (GLSP) is a framework for building web-based diagram editors, running in the browser. The concept of GLSP is based on the Language Server Protocol (LSP), which is the de-facto standard for implementing textual code editors on the web [LSP20]. The general idea is to cleanly encapsulate the client and the server part of an editor via a defined protocol. The client is responsible for rendering and for executing time-critical operations such as drag and drop. The server is responsible for

providing any domain-specific business logic, e.g. what shapes to display, how they can be connected and how the domain model is updated on creating a node.

The diagram client of GLSP is largely generic, which means it can be re-used for custom diagram type by adding custom shapes if needed. To create a custom diagram for a domain-specific language, we need to create a custom "graphical language server". Similarly to LSP, a GLSP server can be written in any language, since the communication to the client is encapsulated in a defined protocol. This gives the user freedom of choice for new projects and, even more importantly, it allows to adapt any existing code in the user language server. For instance, it can connect any diagram logic that is already implemented in any language for the desktop.

To sum up, GLSP provides two high-level benefits. First, the architectural frame, i.e. the strong encapsulation, allows to build flexible solutions and also reuse existing business logic on the web. Second, GLSP provides ready-to-use components for the creation of web-based diagram editors, i.e. an adaptable and powerful diagram client, the communication protocol and a server framework to create custom domain-specific language servers.

EuGENia Live [RKP12] is a web-based tool for designing graphical DSLs. It encourages the construction and collaboration of models and meta-models in iterative and incremental development. The tool supports starting from a meta-model of the DSL, and then modify it based on examples. As a final result, EuGENIA live generates a GMF Eclipse based graphical modelling environment.

Altogether, we have analysed several tools to create graphical editors for the web. However, we are not aware of solutions supporting migration of existing editors to the web, to enable their integration with low-code development platforms.

2.2 Lowcomotive approach to DSL definition

Domain-Specific Languages (DSLs) are defined in terms of their abstract syntax (the primitives they support, their properties and their relations), concrete syntax (how the DSL is visualized, typically graphically or using text), and semantics (how the DSL is executed) [BCW17]. In model-driven development approaches, all these three parts are defined using models.

Our proposal for the Lowcomotive engine in the project is to follow such standard separation of concerns, as Figure 2 shows. Please note that this WP is only concerned with the DSL syntax, while its execution semantics is dealt with in WP5.

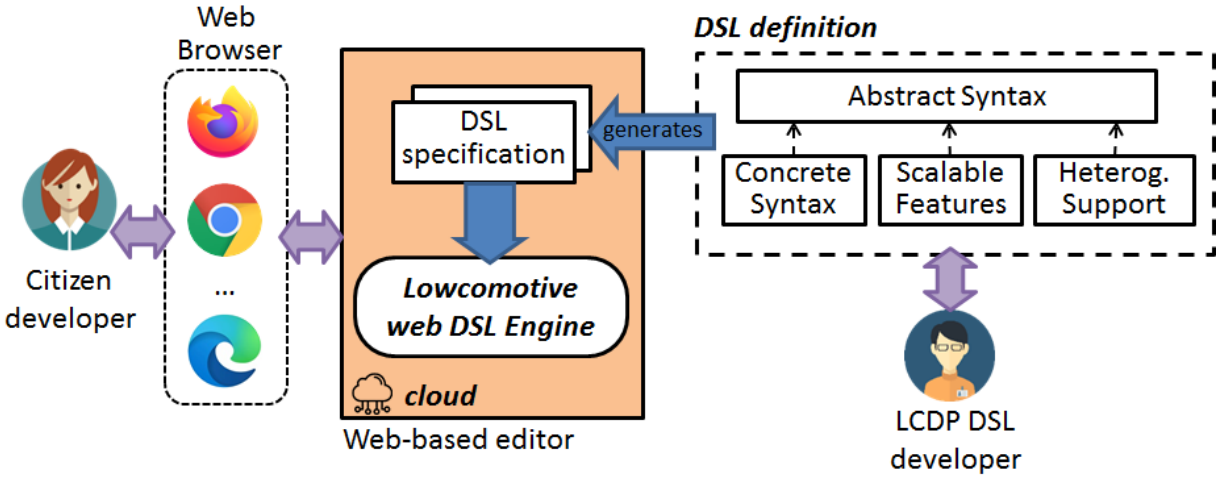


Figure 2.: Lowcomotive approach and architecture for defining DSLs for LCEPs

This way, the abstract syntax of the DSL will be defined via a meta-model [BCW17], typically a class diagram describing the elements of the language, their properties, relations and integrity constraints. We will consider graphical concrete syntaxes, which are given in reference (e.g., as annotations) to the abstract syntax model. Then, the approach will specify elements to enhance the DSL scalability, like views [BMD+20] (to display different concerns of a model in different diagrams), or abstraction patterns [JGL17][dLGS13] (to summarize parts of a model into a more abstract representation, which can be explored using hierarchical decomposition).

Finally, to cope with heterogeneity, we will provide a specification of how the DSLs can be connected to each other. For this purpose we plan on interfaces enabling different types of connections between DSLs:

- Open: From specific elements in the DSL to any element in any DSL
- Restricted: From elements in the DSL to concrete elements in other specific DSLs

Moreover, connections can be realized using different styles: using views (the connected model is displayed in a different view), or embedded (the connected model elements are displayed on the same diagram view).

The project goal is to enable this DSL definition both using web editors (from LCEPs) and from other means, like the Eclipse ecosystem, to enable migration of existing editors to the web for their integration into LCEPs.

2.3. Proof-of concept approach: enabling migration of editors to the web

We have performed a proof-of-concept preliminary prototype of the proposed architecture as shown in Figure 3. In this prototype, the abstract syntax is defined using the Eclipse Modelling Framework (EMF) [SBP+08], and the concrete syntax is defined using Eugenia [KGR+17]. This is an interesting scenario that enables migration of editors already created for the EMF/Eclipse ecosystem into the web. For the target, we use DPG/PSI, a JavaScript-based framework developed by UGROUND for diagramming in their LCEP ROSE [CJD17]. In this first prototype, we do not support the specification of scalable features and heterogeneity support.

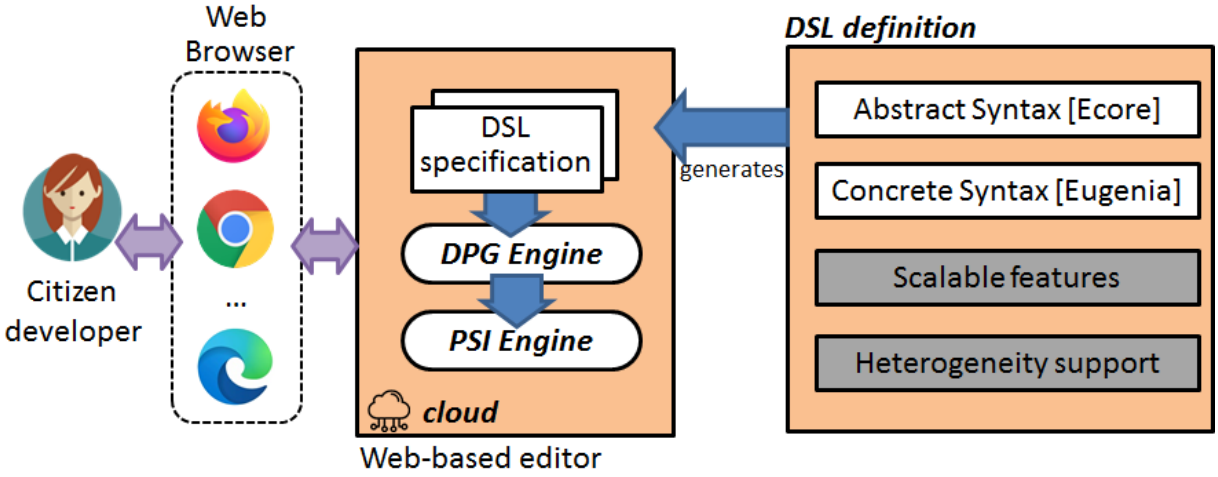


Figure 3.: Proof-of-concept approach to migrate DSLs to the web

In the following sections, we first provide more details on EMF and EuGENIA, describe the architecture and tool support, and a preliminary evaluation. These sections are based on our publication [RDC+20].

2.3.1 Eclipse Modeling Framework and EuGENia

The Eclipse Modeling Framework (EMF) is a widely used (partial) implementation of the meta-object facility (MOF) standard of the OMG [MOF16]. EMF is a meta-modelling framework integrated within Eclipse. It supports a notation (Ecore) for creating meta-models, and a code generation facility that produces Java code (enabling the construction of models and model transformations programmatically) and tree editors to create models interactively. EMF supports model serialization using the XML Metadata Interchange (XMI), an OMG standard for meta-data exchange.

As an example, assume we would like to create a modelling editor for Petri nets using EMF. Petri nets are a popular formalism to model concurrent systems, made of places, which may hold zero or more tokens, and that can be connected to transitions via input and output arcs. Figure 4 shows the Ecore meta-model of Petri nets with EuGENia annotations. The meta-model supports Petri nets with weighted arcs (attribute tokenNo in ArcPT and ArcTP) and time in transitions to model delays. By convention, most Ecore meta-models have a root class, which contains directly or indirectly all other classes via composition relations. This way, each class – except the root class – is contained into another one. This is especially useful to edit models using the generated tree editors since the instances of the contained classes appear as children of the container ones.

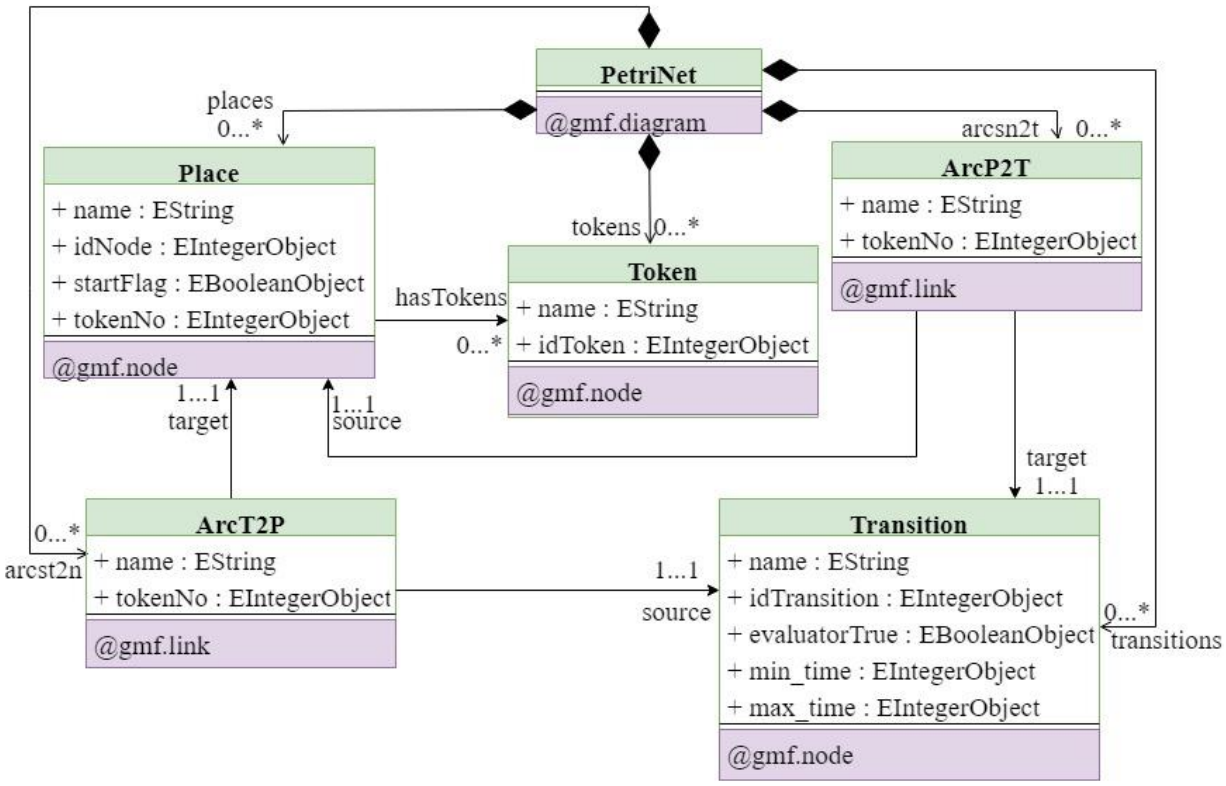


Figure 4: Ecore meta-model for Petri nets with EuGENia annotations

While EMF generates a tree editor by default, more user-friendly editors are typically required. They are normally either textual or graphical. Here, we focus on graphical DSLs. There are several technologies to help in the construction of graphical editors within Eclipse, like Graphiti, Sirius, GMF or EuGENia. We decided to support the latter technology due to its popularity, since it is extensively used by both researchers and practitioners [CJD17]. EuGENia simplifies the development of GMF-based graphical model editors by automatically generating the required models needed by the GMF editor construction framework from a single annotated Ecore meta-model.

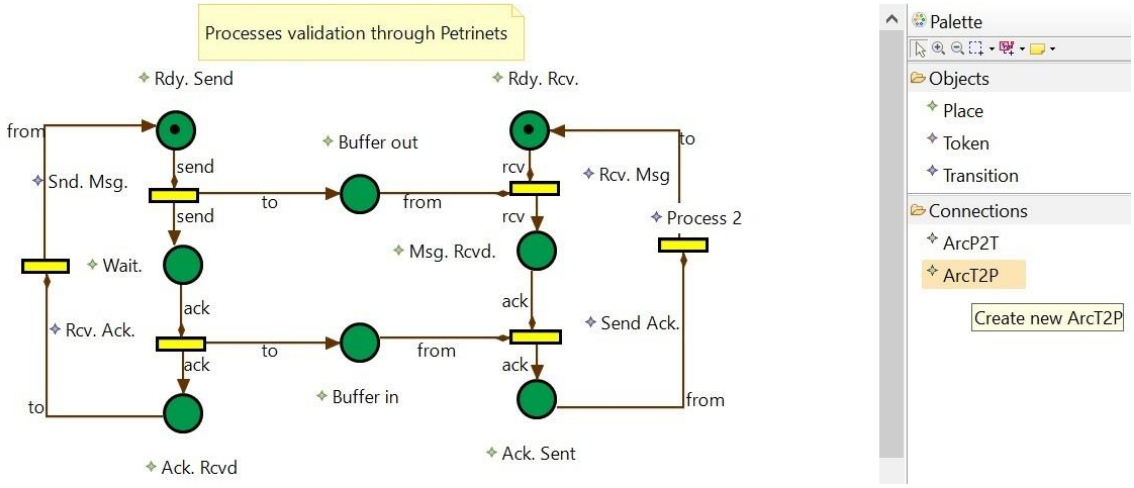


Figure 5: EuGENia graphical editor for Petri nets

As an example, Figure 5 shows some annotations indicating that Places, Transitions and Tokens are to be displayed as nodes (annotation *gmf.node*), while arcs are to be represented as links (annotation *gmf.link*), and all elements are to be placed in the diagram represented by the PetriNet (annotation *gmf.diagram*). Figure 5 shows the resulting editor. It contains a canvas to graphically edit the model, and a palette to create the objects. The object attributes can be changed from an Eclipse properties view.

2.3.2 DPG (Diagram Programming Generator)

The web technology that we target in this initial prototype is based on the Programmable Solutions Interpreter Engine, (PsiEngine) [CJD17]. PsiEngine implements, evaluates, interprets and executes DSLs described in XML within the web client. It uses HTML5, CSS3, JavaScript and DOM together with technologies, services and tools from Web 2.0 and the specification of XML-DSL grammars in order to build web components, widgets, and dynamic web sites to give the solution to specific web application problems or parts of them.

PsiEngine is a generic lightweight JavaScript framework which is a cross-browser platform that processes and evaluates programs written in Psi Language. On top of PsiEngine, a set of Psi languages (Graph Library Psi GLPsi, Diagram Psi DPsi, Visual Tool Psi TPsi and Data Form Psi DFPsi) were created to develop the functionality of a programmable diagram.

A programmable diagram is a set of graphical elements that define an SVG-based diagram, and each graphical element can have associated visual tools (like dialogue boxes, toolbars, pop-ups, floating menus, menus, and drag and drop), programming utilities (classes, scripts, functions,

and variables) and heterogeneous information data sources (XML/JSON) that determine their appearance and content.

The Diagram Programming Generator (DPG) is a layer on top of the PsiEngine that encodes DSLs as a JSON-based grammar, which specifies the elements necessary to create a programmable diagram in PsiEngine. The execution engine that interprets a DPG specification, is called DPG-PsiEngine. Its objective is to generate the code in the different Psi languages and to start the programmable diagram. Additionally, in DPG-PsiEngine a template engine was incorporated for the administration and generation of graphical elements and forms. It also includes components for connection via REST API, to obtain JSON information. These frameworks are used within UGROUND as a basis for the low-code solutions, and some examples are available at http://devrho.com/.

2.3.3 Architecture and Tool Support

This section presents our migration approach, architecture and tool support. First, we describe how the abstract syntax is migrated (Section 2.3.3.1), and then the graphical concrete syntax (Section 2.3.3.2).

2.3.3.1 Migrating the abstract syntax

Our approach starts reading the domain meta-model and transforming the classes into the DSL-JSON format of DPG. All the (non-abstract) classes in the EMF meta-model are placed inside a DPG table called "Elements": {...}. Then, inside such a table, they are placed into a key "Type": {...}. Later, classes need to be classified as nodes or connectors, as we will explain in the next section. All the attributes and references of the classes are stored into a key "Fields": {...}, and then attributes are associated with different types of DPG controls depending on their eType, as shown in Table 1. The source and target of references are specified using the JSON keys "Start": and "End":.

Table 1: Mapping EMF types to DPG Controls

EMF	DPG-PsiEngine
boolean	checkbox(disabled=0)
int, long, short, double, float	number
char	text(disabled=no)
String	textarea(disabled=no)
Date	date (configDate= 'format=dd/mm/yyyy'; configTime='autoclose=true')

In EMF meta-models, we may have abstract classes and inheritance hierarchies. However, neither abstract classes or inheritance are supported by DPG. To overcome this limitation, our mapping flattens the inheritance hierarchy, collecting the attributes and references of upper classes in the hierarchy, and replicating them in lower classes where they are required.

2.3.3.2 Migrating the concrete syntax

Our approach also translates EuGENia annotations to the DPG format, which distinguishes nodes and connectors. Hence, we have to decide on the basis of EuGENia annotations which classes or references of the EMF meta-model are going to become nodes, and which ones will be connectors. In addition, we need to consider the different graphical styles configured in these annotations.

Listing 1: Node and Link annotations in EuGENia

```
@gmf.node(
    label="name",figure="ellipse",border.styles="dash",
     color ="0,153,76",label.color = "0,0,0",border.color
     ="0,0,0",border.width="3"
    )
@gmf.link(
    label="name",source="source",target="target",style="
     dash",width="2",color="102,51,0",source.decoration="
     none",target.decoration="filledrhomb"
    )
```

An example of EuGENia annotations is shown in Listing 1. For the case of nodes, using key-value pairs, the DSL designer can specify the attribute to be shown as label of the node (key label), the figure representing the node (ellipse), the style of the figure border (border.styles), the colours of the figure background, label and border (color, label.color, border.color) and the border width (border.width).

Listing 2: DPG Code for nodes and connectors

```
Elements Classes
GraphLibraryNODE:
 {
 "GraphLibrary":
 {
    "Config": {
        "Shape": "Ball",
        "Styles":
            "boxFill=rgb(0,153,76);
            nameColorFont=rgb(0,0,0);
            boxStroke=rgb(0,0,0);
            boxStrokeWidth=3px"
    }
 }
GraphLibraryCONNECTOR:
 {
    "EndMarker": "rightarrow",
    "GraphLibrary":
        "class=line-CT-CT;
         stroke-width=2;
         stroke=rgb(102,51,0);
         minimum=none"
 }
```

For links attached to nodes, we need to define their source and target references (keys source and target), and we can also specify a label for the link (label) and graphical styles including colour, width, line style, and decorations for the source and targets of the link. Details on additional EuGENia styles can be found at https://www.eclipse.org/epsilon/doc/eugenia/. In DPG the graphical information is stored in JSON, where we need to define libraries of nodes and connectors, as Listing 2 shows. Please note that DPG uses CSS styles, e.g., for colour values.

Table 2: Mappings between EuGENia and DPG for figures and decorations

figure		source/target.decoration		border.styles	
EuGENia	**DPG-PsiEngine**	**EuGENia**	**DPG-PsiEngine**	**EuGENia**	**DPG-PsiEngine**
`rectangle ->`	Box	`arrow ->`	rightarrow	`solid ->`	".line-CT-CT ",
`ellipse ->`	Ball	`closedarrow ->`	rightarrow	`dash ->`	".line-CT-CT
`rounded(default) ->`	Box	`filledclosedarrow ->`	rightarrow		{stroke-dasharray:5,5;}",
`polygon ->`	Rhombus	`rhomb ->`	circlecross	`dot ->`	".line-CT-A
`svg (svg.uri)->`	"Url":"..." , "Doc": "..."	`filledrhomb ->`	circlecross		{stroke-dasharray:2,2;}",
`java class name ->`	"Shape":"Image"	`square ->`	circle		
	"Field":"IMG_URL"	`filledsquare ->`	circle		
	"Symbol":"name_symbol"	`java class ->`	java class		

The details of the mapping of different styles of nodes and links between EuGENia and DPG are described in Tables 2 and 3. The former table details the mappings between figures and decorations for links and the latter describes the mapping between the other annotation options. In DPG, when attributes of nodes or links are to be edited, a dialogue box is presented. Each attribute can be associated with a different control, depending on its eType, to properly edit its value. Finally, the information about the palette and the corresponding icons are also translated. Regarding the limitations of our translation, we do not currently support the EuGENia annotation @gmf.affixed to attach nodes to the border of another one. In addition, the @gmf.compartment annotation, which allows node nesting, is not natively supported either. However, we provide a workaround that allows inserting elements in the container objects.

Table 3: Mapping EuGENia and DPG styles for nodes/links

EuGENia	DPG-PsiEngine
@gmf.node	**NODE**
border.color	boxStroke or stroke
border.styles	as in Table 1
border.width	boxStrokeWidth or strokeWidth
color	boxFill or fill
figure	as in Table 1
label	name
label.color	nameColorFont or colorFont
tool...(fields)	icon (symbol & color)
@gmf.link	**CONNECTOR**
color	fill or stroke
incoming	"Starts":
label	name
source	"Starts":
source.decoration	as in Table 1
style	as in Table 1
target	"Ends":
target.decoration	as in Table 1
width	strokeWidth
tool...(fields)	icon (symbol & color)

2.3.4 Tool Support

We have created a tool called ROCCO (MigRatiOn towards Cloud Based GraphiCal EditOr) that implements the previous mapping. ROCCO is an Eclipse plugin that reads Ecore meta-models with EuGENia annotations and synthesises the necessary DPG files. For this purpose, it uses the Acceleo code generation language. The architecture of the tool is shown in Figure 6, which also shows the generation process.

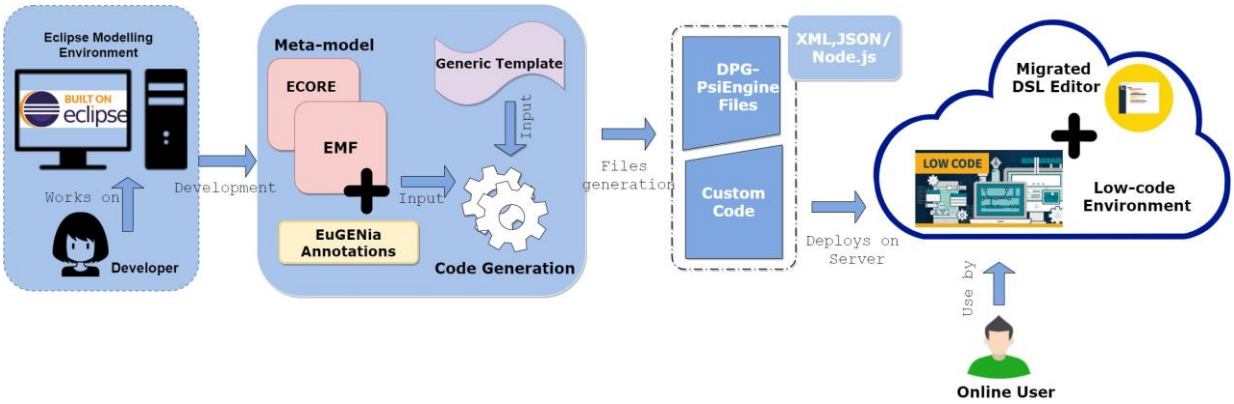


Figure 6: Code Generation Template Schema

Figure 7 shows the resulting DPG editor for the running example. It can be seen how the synthesized editor mimics well the original one defined with EuGENia (cf. Figure 5).

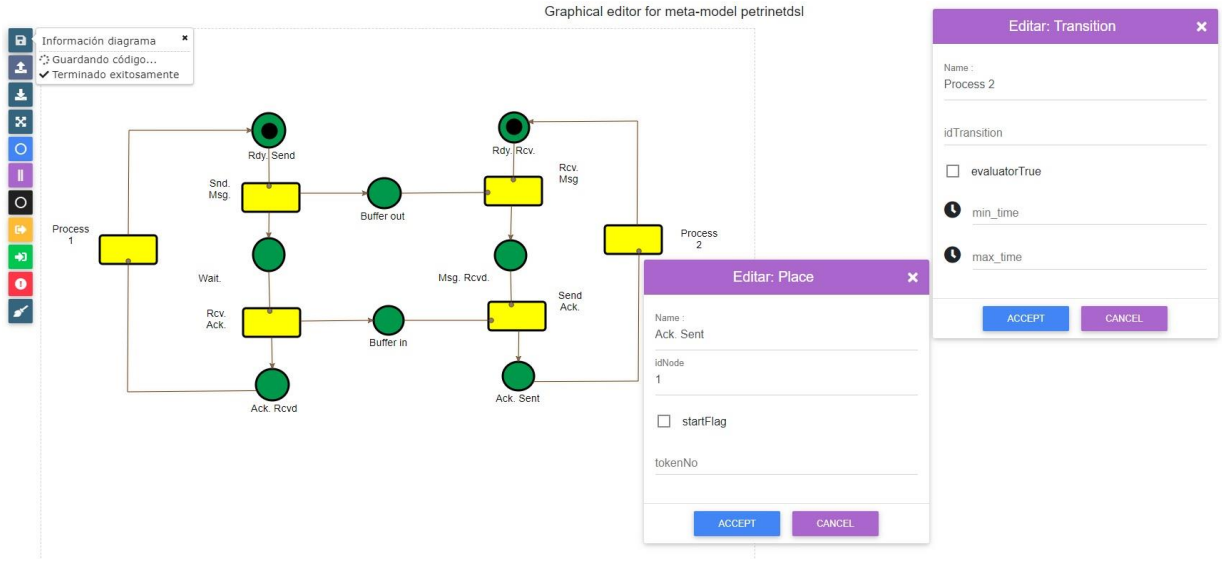


Figure 7: DPG-PsiEngine graphical editor for Petri nets

2.3.5 Evaluation

In this section, our goal is to answer the following research question: Can ROCCO migrate Eclipse-based graphical modelling editors (EuGENia) to the web, to facilitate their integration with low-code platforms?

For this purpose, we have evaluated our tool by migrating existing EuGENia editors into DPG-PsiEngine. We have taken nine EuGENia editors from the Epsilon online repository (taking all

meta-models with EuGENia annotations) and checked whether a complete DPG editor is obtained, required manual changes or had lacking functionality.

Table 4: Results of the evaluation

			EClass @gmf.node		EClass @gmf.link		EReference (non-containment) @gmf.link		EReference (containment) @gmf.compartment		EReference (containment) @gmf.affixed	
Meta-Models	MM Size	DPG LOCs	EuGENia	DPG	EuGENia	DPG	EuGENia	DPG	EuGENia	DPG	EuGENia	DPG
scl	3	100	2	Yes	1	Yes	1	Yes	1	Yes*	0	0
petrinetdsl	6	149	3	Yes	2	Yes	0	0	0	0	0	0
components	5	86	2	Yes	1	Yes	0	0	0	0	1	No
bpmn	21	352	10	Yes	4	Yes	0	0	2	Yes*	0	0
fed	4	134	1	Yes	0	0	2	Yes	1	Yes*	0	0
filesystem	6	161	3	Yes	1	Yes	0	0	0	0	0	0
friends	2	62	1	Yes	0	0	2	Yes	0	0	0	0
rcpapp	2	65	1	Yes	0	0	0	0	0	0	0	0
widgets	2	65	1	Yes	0	0	0	0	0	0	0	0

The evaluation results are shown in Table 4. The table shows the meta-model size in classes, and the lines of code of the generated DPG specifications. Then, it shows the number of nodes, links (applied to nodes and references), compartment and affixed annotations. We marked with Yes/No whether the different annotations were correctly translated between these two platforms. For containment, we used Yes*, since we emulate compartments as graphical areas with less functionalities than in EuGENia. Overall, we could fully migrate eight out of the nine editors. For the other one (components), we obtained a working editor, but with lacking functionality regarding affixed features. In the EuGENia editor ports can be affixed to components, to appear in their borders. Instead, they are connected via links in the DPG editor. Overall, we can answer our research question in a positive way, since in all cases we obtained a working editor, albeit with lacking functionality for affixed, and less sophistication for compartments. These limitations will be addressed in future work.

3. Recommendation support for modelling environments

In parallel to the work presented in Section 2, and attached to the work of ESR1, we are developing a generic model-driven framework capable of generating ad-hoc, task-oriented recommender systems (RSs) to assist in the modelling tasks. The framework provides a DSL where RSs designers can define the settings that they would like to have in the RS. Thus, the DSL describes the different aspects of recommender systems, including a description of the recommended items and their features, the profile and preferences of the users of the recommendations, the recommendation methods, and the evaluation procedures and metrics.

The following sections contain background information related to recommender systems (Section 3.1), the proposed approach (Section 3.2), the used domain-specific language (3.3), the data preparation (3.4), the considered recommendation engines (3.5) and the evaluation of our approach (3.6). These sections follow the article [ACG+20].

3.1 Background

Recommender Systems (RSs) are software tools and techniques that suggest items considered relevant for a particular user. "Item" is the prevalent word to refer to what the system recommends, e.g., the products to buy on an online retail store, or the songs to listen on a music streaming service provider platform. These systems support individuals to evaluate an overwhelming amount of item options [RRS15]. For this purpose, they may exploit item characterizations based on a range of item features (e.g., the genre in a movie recommender) [AT05].

RSs can be classified into the following broad categories based on how the recommendations are made: *content-based filtering*, where users are recommended items similar to the ones they preferred before; *collaborative filtering*, where users are recommended items that other people with similar preferences like; and *hybrid filtering*, which combines the previous two techniques to avoid the limitations of the content-based and collaborative methods [AT05]. Another way to classify RSs is based on the recommendation output. This can be either an estimation of user preference values (usually expressed in the form of numeric ratings) for items, or the generation of an ordered (ranked) list of the most relevant items for the target user. To measure the RS performance, there are different metrics for each of these types of approaches. Some metrics are based on the rating prediction error (e.g., MAE, RMSE), and others measure the item ranking quality (e.g., precision, recall, nDCG, MRR) [GS15].

Software development environments are starting to integrate RSs to assist developers in various software engineering activities, from reusing code to effective bug reports [RMW+14]. Examples of recommended items in these systems are method calls that can be useful in a certain context [TKO+05], software components that may be reused in a given situation [MCK05], and required software artefacts [MS10].

3.2 Proposed approach

The architecture of the proposed approach is shown in Figure 8. In the proposal we apply MDE techniques to develop RSs. In label 1, the recommender system designer provides a meta-model of the notation that will be the subject of the recommendation.

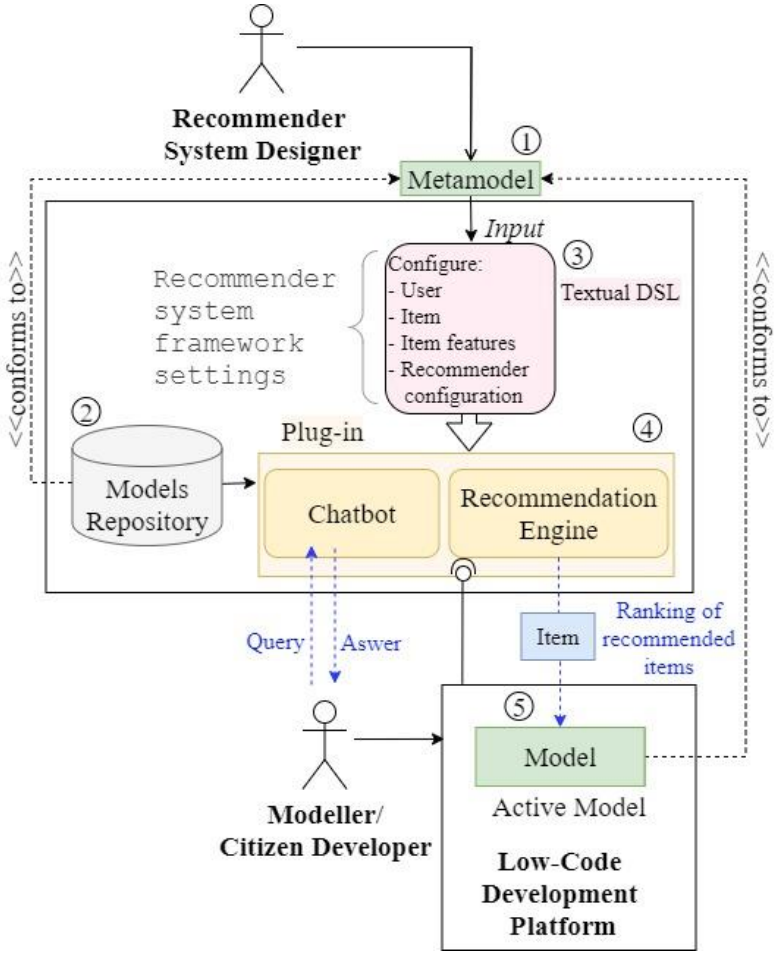


Figure 8: Overview of the proposed approach to define task-specific RSs

As a running example, a meta-model of class diagrams is used. Also, we assume the existence of a repository of models conformant to the metamodel, which will be used in label 2 for the recommendation. Afterwards, the RS designer with the use of a textual DSL (label 3) defines the RS configuration. Such configuration needs to define which of the meta-model elements will play the roles of user, item and item features, as in traditional RSs. The DSL also permits customising other aspects of the RS, such as the maximum number of recommended items, the applied recommendation method, and the recommendation format that best fits for the task at hand.

Using this information, the framework will generate a tailored RS available as a plug-in for the LCDP (label 4). The recommendations will be offered to the citizen developers within the environment in label 5. In this context, the citizen developers are the users of the LCDPs, which typically lack background in programming. Hence, it is important that LCDPs are able to integrate useful, easy-to-use mechanisms to assist these users in their development tasks. We foresee the provision of alternative ways to render the recommendations, such as tips over the diagram elements, example fragments, or by means of query-answer chatbots addressed in natural language.

Figure 9 includes an overview of the RS configuration process. In the first step, the RS designer needs to provide some data, specifically, the meta-model of the notation for which the RS is to be developed, and the set of instance models to be used for training the RS. In step 2, the RS designer uses the DSL to configure the desired features of the RS. From this information, the

RS designer can trigger the generation of the RS. This generation comprises steps 3 to 7, which are completely automated.

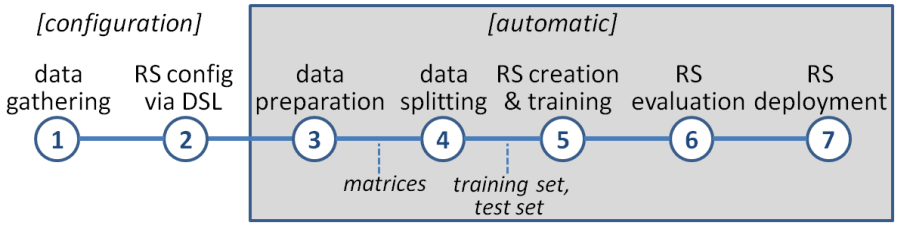


Figure 9: Overview of the process

In step 3, the data provided in step 1 are prepared to produce the user-item and item-feature matrices, considering the specific items and features indicated in the RS configuration. Then, the data are split into two sets (step 4): one is used for training the RS (step 5), and the other one is used for evaluating the accuracy of the RS after its training (step 6). Finally, in step 7, the resulting RS can be deployed and used to obtain lists of recommended items.

In the following subsections, we provide additional details of the DSL, the data preparation step and the recommendation engine.

3.3 Domain-specific language for configuring the RS

We have designed a DSL to offer a flexible configuration of the RS for arbitrary languages that are defined by a meta-model. The DSL allows configuring the recommendation method, the data splitting method, the evaluation method, and the kind of elements to be recommended. The DSL provides a high level syntax for this task, which avoids the RS designer's use of lower-level general-purpose programming languages like C or Java (typically more technical and complex) or the need to have deep expertise in libraries for RSs.

The meta-model that captures the main elements of the DSL is presented in Figure 10. The main class of the DSL is the *RecommenderConfiguration* class. This class contains the other classes, and allows the specification of the name of the recommender, the meta-model of the notation for which the RS is being defined, and a set of instance models conformant to this meta-model. The instance models will be used to train (build) the recommender.

The *RecommendationMethod* class permits selecting the recommendation methods of interest (e.g., item popularity, collaborative filtering, content-based) and configuring their parameters (e.g., the neighbourhood size for collaborative filtering methods). The *SplitMethod* class allows customising how to split the set of provided instance models for training and testing the RS. In particular, it defines the split type (e.g., cross-validation, random), the number of folds (if needed), the splitting method (per-user or per-item), and the percentage of data used for training the RS (the rest of the data will be automatically assigned for testing). The *EvaluationMethod* class defines all the configuration related to the evaluation of the RS, namely, the metrics used to evaluate the RS (e.g., precision, recall, F1), the maximum number of recommended items and the relevance threshold to consider in the evaluation. The *EvaluationResult* class represents the values of the evaluation metrics after executing each selected recommendation method. *DomainClass* allows specifying the type of the model elements that will play the role of user in the context of the RS. Likewise, *DomainProperty* is used to specify the type of the items to be recommended, which can be either features (attributes or references) of the specified *DomainClass* or derived features via expressions.

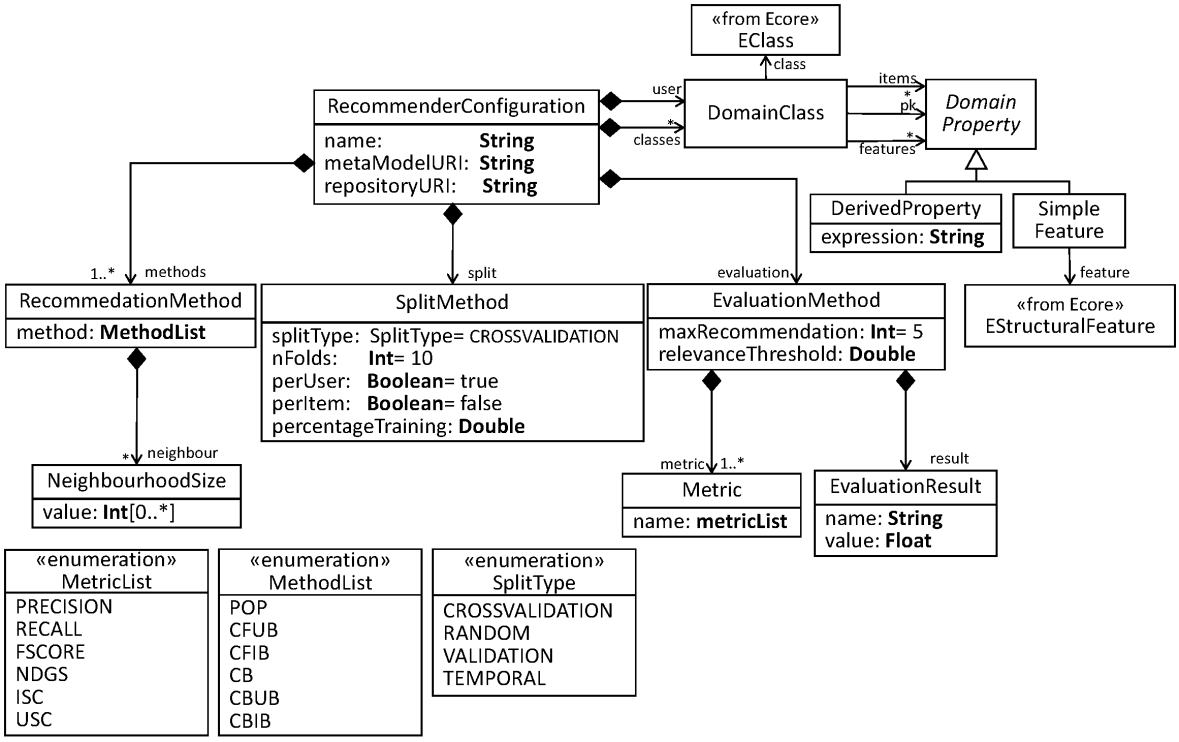


Figure 10: Meta-model of the DSL for RS con.

Listing 3 illustrates the textual concrete syntax that we have devised for the DSL. The listing configures the RS for our running example. For clarity, we assume our RS is to be developed for simple class diagrams, conforming to the simple meta-model shown in Figure 11. This meta-model allows the specification of *ClassDeclarations*, *AttributeDeclarations* and *MethodDeclarations*.

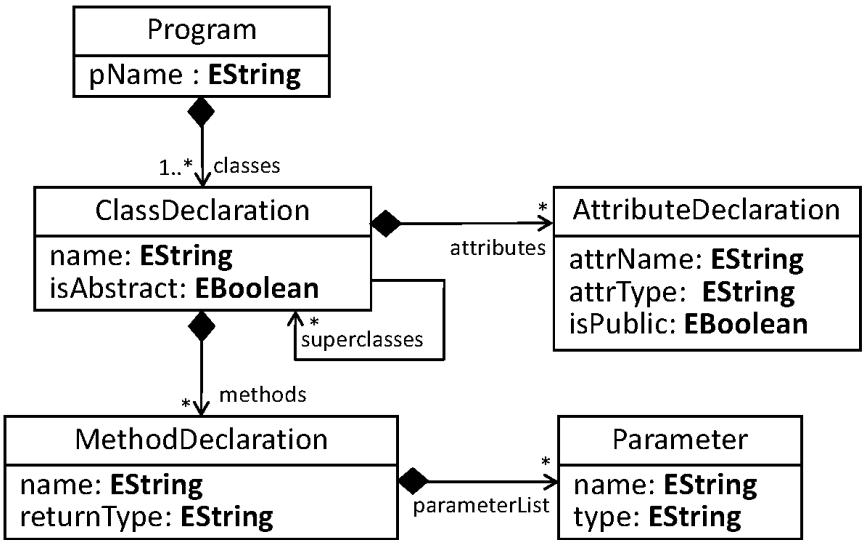


Figure 11: Simple meta-model for class diagrams.

```
1  Metamodel: "/SimpleOOPL.ecore"
2  Repository: "/Instances/"
3
4  //Definition of user and items
5  Users: ClassDeclaration {
6    Items: attributes, methods, superclasses; }
7
8  //Definition of primary keys (pks) and features
9  ClassDeclaration {
10   pk: name; }
11
12 AttributeDeclaration {
13   pk: attrName;
14   features: attrName, attrType; }
15
16 MethodDeclaration {
17   pk: name;
18   features: name, returnType; }
19
20 //Recommender preferences
21 Recommendations {
22   //split configuration
23   Split {
24     splitType: CrossValidation;
25     nFolds: 10;
26     perUser: true;
27     percentageTraining: 0.8; }
28
29   //methods configuration
30   Methods {
31     collaborativeFiltering: pop, cfub(2,3,5,10), cfib;
32     contentBased: cb;
33     hybrid: cbub(2,3,5,10), cbib(2,3,5,10); }
34
35   //evaluation configuration
36   Evaluation {
37     metrics: precision, recall, f1, ndgs, isc, usc;
38     maxRecommendations: 5;
39     relevanceThreholds: 0.5; }}
```

Listing 3: Example of recommender system configuration

In Listing 3, lines 1–2 identify the meta-model of the language the RS is built for (cf. Figure 8), and the URL of a repository of instances of this meta-model (step 1 in Figure 9).

The following lines configure the RS (step 2 in Figure 9). Lines 5–6 specify the meta-model elements that will play the roles of users and items in the RS. These elements must belong to the meta-model provided in line 1. The listing sets the class *ClassDeclaration* as the *User* of the RS, while its attributes, methods and superclasses are set as the Items of *ClassDeclaration*. This means that the RS will be able to recommend these three kinds of items for a given class. Then, lines 9–18 define the primary key used to identify each user and item in the RS, as well as the features used for comparing this information for the item *AttributeDeclaration*. In particular, its attribute *attrName* will be used as its primary key, and the features *attrName* and *attrType* will be used for the comparison of attribute declarations.

The remainder of the listing declares recommender preferences. The *Split* fragment (lines 23–27) configures the application of the cross-validation split method type with 10 folds, following a per user technique, and using 80% of the input data as training data. The *Methods* fragment (lines 30–33) selects the recommendation methods to apply and evaluate. Among others, the DSL designer has selected some collaborative filtering methods such as *pop* (item popularity) and *cfub* (collaborative filtering user base with 2, 3, 5 and 10 neighbours). Section 3.5 will describe these methods. Finally, the *Evaluation* fragment (lines 36–39) selects the evaluation protocol. In particular, line 37 chooses the metrics to be used for the evaluation, line 38 specifies the number of items to recommend, and line 39 defines a relevance threshold.

3.4 Data preparation

The step 3 of Figure 9, data preparation, is presented in this section. After the RS has been configured using the DSL the first step that our framework performs is preparing the data for building and evaluating the RS.

Figure 12 shows the steps followed. First, the framework retrieves the collection of models specified with the DSL (1). Then, it extracts the model objects corresponding to the configured types of users, items and item features (2). Lastly, it generates a user-item matrix and an item-feature matrix for them (3). The user-item (resp. item-feature) matrix contains the users (resp. items) as rows and the items (resp.features) as columns. Then, each cell is set to 1 if the user (resp. item) has the item (resp. feature), and to 0 otherwise.

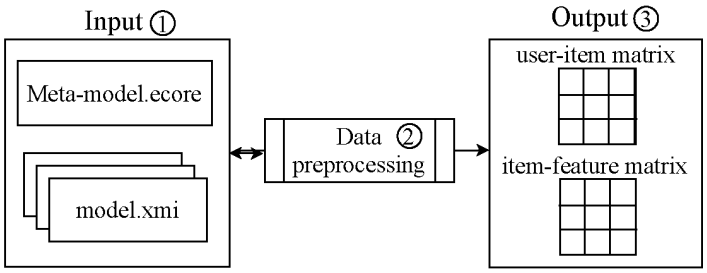
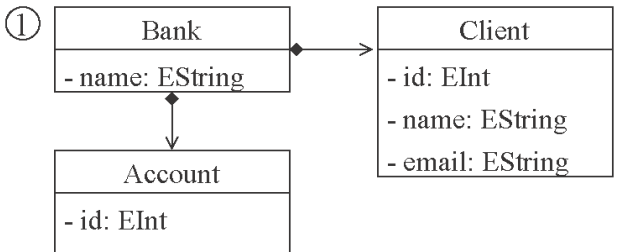


Figure 12: Data preparation steps.

Figure 13 shows an example of data preparation for the running example. To ease understanding, we assume that there is a single class diagram with three classes (1). The table with label 2 shows the extracted users, i.e., the three classes. The table with label 3 shows the extracted items, i.e., each different attribute and method declaration. The item comparison is based on the features selected in Listing 3 (e.g., *attrName* and *attrType* for attributes). The table with label 4 shows the value of those item features. From this information, the framework builds the user-item matrix shown to the right (5), where each row represents a class, and each column represents an attribute, method or superclass. The figure also shows the generated item-feature matrix (6).



②	user0	user1	user2
name (**pk**)	Bank	Account	Client

③	item0	item1	item2	item3
attrName (**pk**)	id	credit	name	email

④	feature0	feature1	feature2	feature3
attrName	id	credit	name	email
attrType	EInt	EDouble	EString	EString

	i0	i1	i2	i3
u0	0	0	1	0
u1	1	1	0	0
u2	1	0	1	1

User-item matrix ⑤

	f0	f1	f2	f3
i0	1	0	0	0
i1	0	1	0	0
i2	0	0	1	0
i3	0	0	0	1

Item-feature matrix ⑥

Figure 13: Example of data preparation.

3.5 Recommendation engine

The steps 4 to 7 of Figure 9; data splitting, the RS creation and training, the RS evaluation, and the RS deployment; are presented in this section. The matrices generated in the previous step (data preparation) are the inputs to build the RS. The steps followed are depicted in Figure 14.

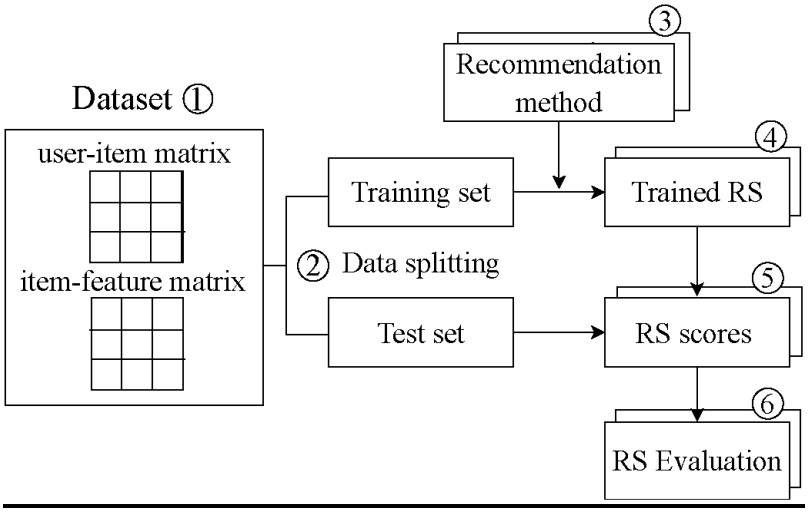


Figure 14: Steps to build the recommendation engine.

Our framework splits the provided data into two sets: one for training the RS, and the other to evaluate the quality of the resulting system (2) (step 4 in Figure 9). The splitting is made according to the specified protocol (see lines 23–27 in Listing 3). Next, the framework uses the configured recommendation methods (3) to train the RS with the training set (4) (step 5 in Figure 9). The RS designer may have configured several methods, as in lines 30–33 in Listing 3, and hence, several candidate RSs may be generated. Then, the test set is applied to each candidate system, and a score is computed based on the obtained results in each case (5). Finally, each candidate RS is evaluated (step 6 in Figure 9) according to the specified metrics (6, see lines 36–39 in Listing 3), and the results are made available for the designer inspection.

In the long term, we envision an intelligent framework that is able to suggest the best configuration for the target recommendation task and the available data (step 7 in Figure 9). This would free the RS designer from having to possess deep expertise in RS techniques.

3.6 Evaluation

As a proof of concept and to evaluate our framework we have designed an experiment. In this section we present the initial results of our envisioned framework. The framework presented is being developed using Java and the Eclipse Modeling Framework (EMF) [EMF20].

The framework presented in this evaluation is applicable to languages defined by an Ecore meta-model. For the moment, we have automated support for data preparation, data splitting, RS training and RS evaluation. The configuration data must be provided programmatically though, as the configuration DSL, while designed, is still under development. The deployment of the generated RS in an LCDP is also future work.

To have an initial assessment of our framework, we have applied it to the construction of the RS using Listing 3 as a running example. The RS would be integrated into an LCDP and would suggest attributes, methods and superclasses that may be added to new classes, based on the definition of other similar classes. By means of this experiment, we aim to answer the following research questions.

- **RQ1** *Can a recommender system help in class modelling tasks?*
- **RQ2** *Which recommendation method of relevant attributes, methods and superclasses has the best performance?*
- **RQ3** *Can hybrid approaches be beneficial for the recommendation of attributes, methods and superclasses?*
- **RQ4** *Which method performs better when considering user and item coverage in the recommendation of relevant attributes, methods and superclasses?*

We run the experiment on three datasets. Table 5 shows some size metrics of them (number of models, users, items and item features). The Synthetic dataset contains 29 models conformant to the running example meta-model (cf. Figure 4). The models were created manually using EMF [EMF20]. These models are based on class diagram examples from the internet.We have made sure that the models created have all the characteristics normally present on class diagrams, such as attributes, methods and inheritance hierarchies.

Table 5: Description of the datasets.

	Synthetic	SyntheticExtended	AtlanEcore
Num. models	29	58	300
Num. users	150	181	6555
Num. items	412	520	4338
Num. features	438	557	4867

The *SyntheticExtended* dataset extends the first one with further models which are similar to those in the Synthetic dataset but substituting the name of some model elements by synonyms. Finally, since meta-models are similar to class diagrams, our third dataset (AtlanEcore) is composed of 300 Ecore meta-models from the AtlanEcore Zoo (https://web.imt-atlantique.fr/x-info/atlanmod/index.php?title=Ecore). This is an open-source repository of Ecore meta-models, which are conformant to the Ecore.ecore meta-metamodel. With this last dataset, we want to validate the versatility of our proposal. The configuration of the RS for the first two datasets was the one shown in Listing 3. The configuration for the AtlanEcore dataset was similar but using types from *Ecore.ecore* (i.e., setting *EClass* as the user of the RS; *eAttributes*, *eOperations* and *eSuperTypes* as the items; and so on).

As data splitting methodology we used 10-fold cross-validation with 80% of the data as a training set, and the remainder 20% as a test set (cf. lines 23–27 in Listing 3). We followed a per-user method, whereby the training and test sets are built per available user (i.e., for each class, it takes 80% of its items for training and the rest for testing). Using 10-fold cross-validation avoids over-specialization. This is so as the training set is split into 10 subsets, the training is performed 10 times taking one of the subsets for testing and the others for training, and finally, the average performance of the 10 learned RSs is reported.

For the recommendations methods, we trained the RS using a variety of collaborative, content-based, and hybrid recommendation methods (cf. lines 30–33 in Listing 3). For reproducibility, we used the RankSys framework (https://github.com/RankSys/RankSys) to implement the methods. The recommendation consisted of the top 5 highest-rated items. In particular, the collaborative methods will recommend to users (i.e., to classes) items (i.e., attributes, methods, superclasses) that were rated (i.e., used) by like-minded users. The similarity between users and items is based on rating patterns.

In the experiment,we evaluated both user-based and item-based k-nearest neighbour (k-NN) heuristics. The item-based approach (*cfib*) creates neighbourhoods by exploiting the rating-based similarities between items, and the user-based approach (*cfub*) computes similarities between users. We have used the cosine as a similarity metric. We tested neighbourhoods of size *k = 2, 3, 5, 10*. To refer to a specific instance of a method, we concatenate the value of k to its name. For instance, *cfub3* refers to collaborative user-based k-NN size *k = 3*. As a baseline, we also evaluated the item popularity method (*pop*). The content-based method (*cb*) will recommend to users (i.e., to classes) items (i.e., attributes, methods, superclasses) similar to the ones liked (i.e., used) by the user. In this case, the similarity is computed based on profiles built from textual information.

The item features correspond to text features extracted from the items, and the recommendations are based on similarities in the text feature space. In the experiment, we used the name and data type as features of attribute declarations; the name and return type as features of method declarations; and the name of superclasses. Finally, the hybrid methods exploit both rating and text features by combining content-based and collaborative methods. We considered the methods *cbub* and *cbib*, which combine either user-based (*cfub*) or item-based (*cfib*) collaborative filtering with content-based similarity (*cb*). We tested neighbourhoods of size *k = 2, 3, 5, 10*. As before, we concatenate the name of the method and the value of k. For instance, *cbub5* refers to content-based user-based k-NN size *k = 5*.

We analysed the performance of the resulting RSs using some ranking-based, coverage and diversity metrics typically used in RSs (cf. lines 36–39 of Listing 3). The metrics were implemented in the RiVaL framework (https://github.com/recommenders/rival). The selection of metrics depends on the faced recommendation problem. In particular, since our RS should provide an ordered list of recommended items, we used the classical ranking metrics precision, recall and F1. Precision is the percentage of the recommended items that are relevant; recall is the percentage of relevant items included in the recommendation list; F1 is a harmonic means of precision and recall. To measure coverage, we used the metrics USC (User Space Coverage) and ISC (Item Space Coverage). USC measures the percentage of users that the RS can recommend, and ISC the diversity in terms of the popularity of what is recommended. Finally, to measure the quality of the recommended list, which should contain just the most relevant items, we used the metric nDCG (Normalized Discounted Cumulative Gain). This metric penalises when the most relevant items are not at the top of the list.

Table 6: Results of the experiment. The best values are shown in bold.

	Synthetic						SyntheticExtended						AtlanEcore					
Method	prec.	recall	F1	nDCG	ISC	USC	prec.	recall	F1	nDCG	ISC	USC	prec.	recall	F1	nDCG	ISC	USC
pop	0.048	**0.221**	0.079	**0.177**	0.015	**1.000**	0.046	0.207	0.076	0.170	0.015	**1.000**	0.018	0.083	0.029	0.055	0.002	**1.000**
cfub2	0.060	0.185	0.091	0.144	0.035	0.719	**0.093**	0.242	**0.135**	0.179	0.043	0.698	**0.241**	0.362	**0.289**	**0.323**	0.048	0.332
cfub3	0.054	0.190	0.084	0.145	**0.036**	0.802	0.088	0.256	**0.132**	0.188	0.046	0.802	**0.211**	**0.367**	**0.268**	**0.322**	0.055	0.372
cfub5	0.054	0.206	0.085	**0.165**	**0.036**	0.898	0.083	**0.276**	0.128	**0.202**	0.048	0.837	0.179	**0.368**	0.241	0.321	0.061	0.415
cfub10	**0.066**	0.193	**0.099**	0.146	0.032	0.600	0.074	**0.289**	0.118	**0.219**	**0.049**	0.919	0.140	0.347	0.200	0.297	**0.067**	0.482
cfib	0.053	**0.207**	0.085	0.147	**0.038**	0.901	0.064	0.239	0.101	0.172	**0.049**	0.921	0.092	0.273	0.138	0.225	0.063	0.627
cb	0.018	0.086	0.030	0.086	0.008	**1.000**	0.018	0.086	0.030	0.085	0.006	**1.000**	0.005	0.022	0.008	0.010	0.001	**1.000**
cbub2	0.016	0.015	0.016	0.016	0.002	0.968	**0.096**	0.176	0.125	0.124	0.033	0.633	0.200	0.246	0.221	0.220	0.035	0.311
cbub3	0.016	0.032	0.022	0.026	0.005	0.968	0.065	0.196	0.098	0.142	0.040	0.745	0.155	0.259	0.194	0.230	0.048	0.410
cbub5	0.016	0.057	0.025	0.039	0.010	0.968	0.057	0.203	0.089	0.147	0.043	0.896	0.113	0.276	0.160	0.238	0.058	0.483
cbub10	0.016	0.070	0.026	0.044	0.012	0.968	0.052	0.211	0.083	0.158	0.043	0.993	0.079	0.269	0.122	0.228	**0.065**	0.558
cbib2	**0.095**	0.199	**0.129**	0.131	0.026	0.539	0.014	0.010	0.012	0.011	0.002	0.973	0.001	0.001	0.001	0.001	0.000	0.697
cbib3	0.049	0.166	0.075	0.120	0.030	0.634	0.013	0.020	0.016	0.018	0.004	0.973	0.001	0.002	0.002	0.002	0.000	0.697
cbib5	0.036	0.131	0.056	0.098	0.033	0.864	0.013	0.039	0.019	0.027	0.008	0.973	0.001	0.005	0.002	0.003	0.001	0.697
cbib10	0.032	0.138	0.051	0.112	0.034	**1.000**	0.013	0.049	0.020	0.031	0.010	0.973	0.001	0.006	0.002	0.004	0.001	0.697

The results of the experiment are presented in Table 6. The rows contain the recommendation methods used to train the RS, and the columns show their performance metrics. We can see that the metric values are drastically different depending on the dataset. The AtlanEcore dataset has the best overall performance. This is a common situation, which is due to differences on the dataset sizes and user preference sparsity levels among the three. Hence, to evaluate the distinct recommendation methods, relative metric value differences should be considered for each particular dataset.

In the following we answer our research questions.

- **RQ1** *Can a recommender system help in class modelling tasks?* In order to answer this question, we analyse the performance of the recommendation methods. Specifically, we look at their precision, recall and F1 values in Table 6, as they give a measure of the ranking quality. With regards to our evaluation methodology, where the task is recommending the most relevant items in the test set, the obtained performance is relevant according to the literature [HHL+19],[NDD+19],[RRS15],[SGM17]. In the AtlanEcore dataset, the highest F1 value was 0.289 for the cfub2 method. This shows that we can build an RS that helps in class modelling by recommending valuable attributes, methods and superclasses for a given class.

- **RQ2** *Which recommendation method of relevant attributes, methods and superclasses has the best performance?* In the SyntheticExtended and AtlanEcore datasets, the collaborative filtering methods obtained the best performance. In particular, *cfub2* has the best results, as the F1 measure is 0.135 and 0.289 for the SyntheticExtended and AtlanEcore datasets, respectively. Conversely, among the collaborative methods, the baseline pop has the worst performance (e.g., 0.018 precision and 0.083 recall on the AtlanEcore dataset). When it comes to coverage, pop has a very high USC value (1.000), as it recommends the most popular items to all users. However, it possesses a low ISC (0.002), which suggests a very low diversity. As for the Synthetic dataset, the best result was obtained for the hybrid method *cbib2*, followed by all other collaborative methods.

- **RQ3** *Can hybrid approaches be beneficial for the recommendation of attributes, methods and superclasses?* When analysing the hybrid methods applied in this experiment, we observe that some performed very well. For instance, in the Synthetic dataset, *cbib2* performed even better than the collaborative methods, obtaining 0.095 precision, 0.199 recall, and an F1 value of 0.129.

- **RQ4** *Which method performs better when considering user and item coverage in the recommendation of relevant attributes, methods and superclasses?* We observe a compromise between the coverage metrics USC and ISC, and the ranking-based metrics. The methods with low precision and recall, like most of the hybrid ones, report high user coverage and low item coverage. A good user coverage comes at the cost of losing item diversity when compared to collaborative methods.

4. Summary, conclusions and further developments

In this document, we have described the initial design and prototypes for a system supporting the development of web-based graphical editors supporting scalable modelling, heterogeneity and recommendation. For this purpose, we have proposed an approach that follows software language engineering principles [BCW17], separating the definition of the abstract syntax, concrete syntax, scalability features and heterogeneity support. We have proposed an initial prototype specially targeting migration of existing editors to the web. For the recommender, we have proposed a DSL to configure the recommender system to arbitrary DSLs defined by a meta-model. We have performed an initial evaluation showing good results.

In relation to ESR2's work, we are currently working on providing support to migrate models (not only editors), and also from other popular frameworks for DSL definition, like Sirius. We also plan to work on the definition of modelling scalability features, including views and abstraction patterns able to summarize a large graph into a smaller one (e.g., aggregating parallel or sequential nodes) that is provided with a hierarchical visualization structure. As mentioned in Section 2.2, we also plan to include support for specifying how the models of heterogeneous DSLs can be connected, related and navigated.

In relation to ESR1's work, currently, the DSL works under Xtext within Eclipse, but we plan to migrate it to the web. We plan to apply our framework to more languages and modelling tasks. Moreover, we plan to provide support for deploying the synthesised recommenders in the LCDPs. Specifically, we foresee the provision of a chatbot -- integrated within the LCDP -- that citizen developers can address to access the recommendations.

5. References

[ACG+20] Lissette Almonte, Iván Cantador, Esther Guerra, and Juan de Lara. 2020. Towards automating the construction of recommender systems for low-code development platforms. In Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (MODELS '20): 66:1–66:10.

[AT05] Gediminas Adomavicius and Alexander Tuzhilin. 2005. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. IEEE Trans. on Knowl. and Data Eng. 17, 6 (2005), 734–749.

[BCW17] Marco Brambilla, Jordi Cabot, Manuel Wimmer: Model-Driven Software Engineering in Practice, Second Edition. Synthesis Lectures on Software Engineering, Morgan & Claypool Publishers 2017

[BEE+08] E. Biermann, K. Ehrig, C. Ermel, and G. Taentzer, "Generating eclipse editor plug-ins using Tiger," in Applications of Graph Transformations with Industrial Relevance, A. Schürr, M. Nagl, and A. Zündorf, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 583–584.

[BGS+10] Stefan Berger, Georg Grossmann, Markus Stumptner, Michael Schrefl: Metamodel-Based Information Integration at Industrial Scale. MoDELS (2) 2010: 153-167. See also: http://dome.ggrossmann.com/

[BMD+20] Hugo Brunelière, Florent Marchand de Kerchove, Gwendal Daniel, Sina Madani, Dimitris S. Kolovos, Jordi Cabot: Scalable model views over heterogeneous modeling technologies and resources. Softw. Syst. Model. 19(4): 827-851 (2020)

[CJD17] Enrique Chavarriaga, Francisco Jurado, and Fernando Díez. 2017. PsiLight: a Lightweight Programming Language to Explore Multiple Program Execution and Data-binding in a Web-Client DSL Evaluation Engine. J. UCS 23, 10 (2017), 953–968.

[dLGS13] Juan de Lara, Esther Guerra, Jesús Sánchez Cuadrado: Reusable abstractions for modeling languages. Inf. Syst. 38(8): 1128-1149 (2013)

[dLV02] Juan de Lara and Hans Vangheluwe. 2002. AToM^3: A Tool for Multi-formalism and Meta-modelling. In International Conference on Fundamental Approaches to Software Engineering. LNCS 2306, Springer, 174–188.

[DNF+20] Alfonso Diez, Nga Nguyen, Fernando Díez, Enrique Chavarriaga: MDE for Enterprise Application Systems. MODELSWARD 2013: 253-256

[EC20] EMF Cloud. https://www.eclipse.org/emfcloud/, (last accessed in Nov. 2020).

[ECG+16] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, Abel Gómez, Massimo Tisi, Jordi Cabot: EMF-REST: generation of RESTful APIs from models. SAC 2016: 1446-1453

[EMF20] Eclipse Modelling Framework. https://www.eclipse.org/modeling/emf/, (last accessed in Nov. 2020).

[EWD+96] J. Ebert, A. Winter, P. Dahm, A. Franzke, and R. Süttenbach, “Graph based modeling and implementation with EER/GRAL,” in 15th International Conference on Conceptual Modeling — ER ’96, Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 163–178.

[GB16] M. Gerhart and M. Boger, “Concepts for the model-driven generation of graphical editors in eclipse by using the graphiti framework,” International Journal of Computer Techniques, vol. 3, no. 4, 2016.

[GEF20] Eclipse Graphical Editing Framework, https://www.eclipse.org/gef/, (last accessed in Nov. 2020).

[GMF20] GMF, https://www.eclipse.org/gmf-tooling/, (last accessed in Nov. 2020).

[Gra20] Graphiti, https://www.eclipse.org/graphiti/, (last accessed in Nov. 2020).

[GS15] Asela Gunawardana and Guy Shani. 2015. Evaluating recommender systems. In Recommender Systems Handbook. Springer, 265–308.

[HHL+19] Bernd Heinrich, Marcus Hopf, Daniel Lohninger, Alexander Schiller, and Michael Szubartowicz. 2019. Data quality in recommender systems: The impact of completeness of item content data on prediction accuracy of recommender systems. Electronic Markets (2019), 1–21.

[JGL17] Antonio Jiménez-Pastor, Antonio Garmendia, Juan de Lara. Scalable model exploration for model-driven engineering. J. Syst. Softw. 132: 204-225 (2017)

[KGR+17] Dimitrios S. Kolovos, Antonio García-Domínguez, Louis M. Rose, Richard F. Paige: Eugenia: towards disciplined and automated development of GMF-based graphical model editors. Softw. Syst. Model. 16(1): 229-255 (2017)

[KEP+06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, Gabriele Taentzer. Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. An EATCS Series, Springer 2006

[KT08] Steven Kelly, Juha-Pekka Tolvanen: Domain-Specific Modeling - Enabling Full Code Generation. Wiley 2008, ISBN 978-0-470-03666-2, pp. I-XVI, 1-427

[LMK+02] A. Ledeczi, M. Maroti, G. Karsai, and G. Nordstrom, “Metaprogrammable toolkit for model-integrated computing,” in Proceedings of the IEEE Conference on Engineering of Computer-based Systems, ser. ECBS, Nashville, Tennessee: IEEE Computer Society, 1999, pp. 311–317.

[LSP20] Language Server Procotol. https://langserver.org/, (last accessed in Nov. 2020).

[M02] M. Minas, “Concepts and realization of a diagram editor generator based on hypergraph transformation,” Sci. Comput. Program., vol. 44, no. 2, pp. 157–180, Aug. 2002.

[MCK05] Frank McCarey, Mel Ó Cinnéide, and Nicholas Kushmerick. 2005. RASCAL: A recommender agent for agile reuse. Artificial Intelligence Review 24, 3-4 (2005), 253–276.

[MKK+14] Miklós Maróti, Tamás Kecskés, Róbert Kereskényi, Brian Broll, Péter Völgyesi, László Jurácz, Tihamer Levendovszky, Ákos Lédeczi: Next Generation (Meta)Modeling: Web- and Cloud-based Collaborative Tool Infrastructure. MPM@MoDELS 2014: 41-60. See also https://webgme.org/, (last accessed in Nov. 2020).

[MOF16] Meta Object Facility (OMG). http://www.omg.org/spec/MOF, 2016.

[MS10] Walid Maalej and Alexander Sahm. 2010. Assisting engineers in switching artifacts by using task semantic and interaction history. RSSE@ICSE (2010), 59–63.

[NDD+19] Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio, Lina Ochoa, Thomas Degueule, and Massimiliano Di Penta. 2019. FOCUS: A recommender system for mining API function calls and usage patterns. In ICSE. IEEE, 1050–1060

[RCW+18] Roberto Rodríguez-Echeverría, Javier Luis Cánovas Izquierdo, Manuel Wimmer, Jordi Cabot. Towards a Language Server Protocol Infrastructure for Graphical Modeling. MoDELS 2018: 370-380

[RDC+20] Fatima Rani, Pablo Diez, Enrique Chavarriaga, Esther Guerra, Juan de Lara. Automated migration of EuGENia graphical editors to the web. MODEProceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (MODELS '20): 71:1-71:7

[RKP12] L. M. Rose, D. S. Kolovos, and R. F. Paige. Eugenia live: A flexible graphical modelling tool. In XM @ MoDELS, pages 15–20. ACM, 2012.

[RMW+14] Martin P. Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann. 2014. Recommendation Systems in Software Engineering. Springer-Verlag Berlin Heidelberg 2014.

[RRS15] Francesco Ricci, Lior Rokach, and Bracha Shapira. 2015. Recommender Systems Handbook (2 ed.). Springer US.

[SBP+08] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. EMF: Eclipse Modeling Framework, 2nd Edition. Addison-Wesley Professional, 2008.

[SGM17] Ritu Sharma, Dinesh Gopalani, and Yogesh Meena. 2017. Collaborative filtering based recommender system: Approaches and research challenges. In ICICT. 1–6.

[Sir20] Sirius, https://www.eclipse.org/sirius/, (last accessed in Nov. 2020).

[Spr20] Eclipse Sprotty, https://github.com/eclipse/sprotty, (last accessed in Nov. 2020).

[SVM+13] Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Van Mierlo, S., Ergin, H.: AToMPM: A web-based modeling environment. In: Invited Talks,Demonstration Session, Poster Session, and ACMStudent Research Competition, MODELS'13, vol. 1115, pp. 21–25. CEUR-WS.org (2013). See also: https://atompm.github.io/

[The20] Eclipse Theia. https://theia-ide.org/, (last accessed in Nov. 2020).

[TKO+05] Masateru Tsunoda, Takeshi Kakimoto, Naoki Ohsugi, Akito Monden, and Kenichi Matsumoto. 2005. Javawock: A Java class recommender system based on collaborative filtering. SEKE, 491–497.