"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

D3.3. Cloud-Based Low-Code Engineering Editor - Final Version

---

Project GA: 813884

Project Acronym: Lowcomote

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

Project officer: Thomas Vyzikas

Work Package: WP3

Deliverable number: D3.3

Production date: September 30th 2022

Contractual date of delivery: September 30th 2022

Actual date of delivery: October 5th 2022

Dissemination level: Public

Lead beneficiary: Universidad Autónoma de Madrid

Authors: Lissette Almonte, Francisco Martínez Lasaca, Juan de Lara

Contributors: The Lowcomote partners

HISTORY OF CHANGES		
Version	**Publication date**	**Change**
0.1	September 19th 2022	Initial draft
0.2	October 3rd 2022	Revised draft
1.0	October 4th 2022	Revised document

Project Abstract

Low-code development platforms (LCDPs) 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 of expertise instead of implementation requirements.

Lowcomote will train a generation of experts that will upgrade the current trend of LCDPs 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 into 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 the agility of our future professionals between the academic and the industrial world.

Table of contents

1. Introduction **7**

2. Cloud-Based Domain-specific Graphical Modelling Environments **8**

2.1 Scalable web-based graphical modelling environments: state of the art 8

2.1.1 Web-based graphical modelling environments 8

2.1.2 Scalability in modelling environments 11

2.2 DSL definition in Dandelion 12

2.2.1 Overview 12

2.2.2 Neutral data model 13

2.2.3 Visual concrete syntax 14

2.2.4 Scalability configuration 15

2.2.5 Sensemaking strategies 16

2.3 Architecture and tool support 18

2.3.1 Architecture 19

2.3.1 Tool support 20

3. Recommendation Support for Modelling Environments **24**

3.1 Background 24

3.2 Related work 25

3.2.1 Recommenders for Modelling Languages 25

3.2.2 Recommender System Generation 26

3.3 Proposed approach 27

3.4 Domain-specific language for configuring the RS 29

3.4.1 Data pre-processing 33

3.4.2 Data splitting 34

3.4.3 Methods supported 35

3.4.4 Evaluation protocol 36

3.5 Tool support 36

3.5.1 Recommendation service 38

3.5.2 Client 38

3.6 Experiments 40
3.6.1 Offline experiment 40
3.6.2 Case study 43
4. Adding Recommendation Support to Low-code Editors 47
5. Summary, Conclusions and Further Developments 47
References 49

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, realisation, and experiments with a cloud-based low-code engineering editor, with unified support for heterogeneous technologies and customised recommendations.

Figure 1 presents a high-level structure of the architecture. Both the graphical editor and the recommender systems are the front-ends of the proposed LCEP of this project, 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, called Dandelion, is the focus of the work of ESR2. The recommender systems are generated using a model-driven solution called Droid, and is the focus of ESR1. Both components will be analysed in the next two sections.

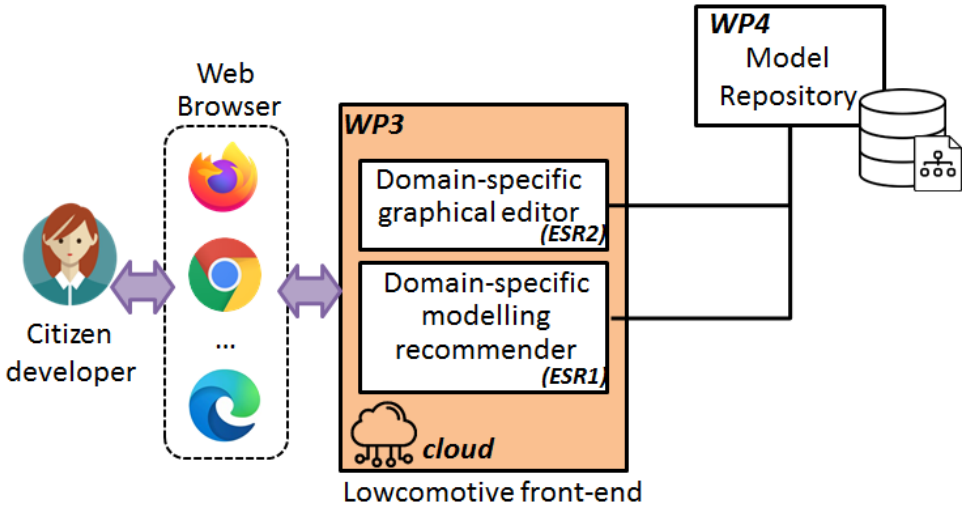


Figure 1. A high-level overview of the architecture of WP3 and WP4

The rest of the document is structured as follows. Section 2 reviews the state-of-the-art of graphical modelling environments and presents *Dandelion*, the approach from ESR2 to define graphical DSLs. Section 3 introduces recommender systems, reviews the state-of-the-art and describes *Droid*, a model-driven solution to generate domain-specific modelling recommenders by ESR1. Both works are integrated in Section 4. Finally, Section 5 concludes with a summary, conclusions and further development.

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, or do not enable rich modelling of editor aspects (e.g., domain-specific abstractions).

To alleviate these problems, the Lowcomote project proposes a novel approach – realised in the Dandelion tool – to ease the creation of graphical editors for the Cloud. Instead of relying on low-level JavaScript graphical frameworks, Dandelion is founded on language engineering principles. This way, all aspects of the editor (abstract and concrete syntax, scalability configuration and applicable abstractions) are described through models. Dandelion proposes a neutral data model, 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 provides a rich concept of model pagination, and will provide extensible libraries of model abstractions and graph summarization techniques to support creating more succinct model views. A Cloud-based modelling environment is ideal for this purpose, to provide enough computation power to perform complex abstractions (enabling better model comprehension and navigation) over large models.

In the following, Section 2.1 reviews works on web-based graphical modelling environments, Section 2.2 describes the Dandelion approach to define and use DSLs, and Section 2.3 details the architecture and tool support.

2.1 Scalable web-based graphical modelling environments: state of the art

This section presents a state-of-the-art revision of some of the current main frameworks and approaches and then reports on our approach (partly based on [RDC+20]).

2.1.1 Web-based graphical modelling environments

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 popularisation of the Eclipse framework. This led 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 LCDPs. However, because LCDPs are based on cloud infrastructure, they 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 LCEPs. We review 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 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 specifying DSL semantics using graph transformations [KEP+06]. It supports two types of collaboration mechanisms in real time. On the one hand, screen sharing 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, model sharing only shares the abstract syntax of a model between clients.

Eclipse Theia [The20] is an open-source IDE platform that runs on browsers and 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, etc. 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 a 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 support diagrammatic views. Sprotty’s reactive architecture makes it 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 synchronisation 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. 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 the creation of 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 and thus can be reused for custom diagram types by adding custom shapes if needed. To create a custom diagram for a DSL 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 adapting any existing code in the user language server. For instance, it can connect any diagram logic 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 the construction of flexible solutions and the reuse of existing business logic on the web. Second, GLSP provides ready-to-use components for creating web-based diagram editors, i.e. an adaptable and powerful diagram client, the communication protocol and a server framework to create custom DSL 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 modifying 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 unaware of solutions enabling the construction of web-based graphical editors using language engineering principles, supporting scalability mechanisms, and enabling their integration with low-code development platforms.

2.1.2 Scalability in modelling environments

Traditionally, the EMF ecosystem has relied on file-based persistence for models, often in a monolithic way. This approach, however, may yield large models for which file-based persistence may not be suitable. There are several proposals to overcome this problem. In [GGdL+19], the authors propose fragmenting model files by defining fragmentation strategies at the meta-model level, which results in smaller files that can be loaded and processed faster. A similar approach is followed in [JBD21] for storing models in multiple files. In [WKG+16], the authors propose a method to load the EMF models partially. To facilitate model management, some authors decompose complex models into smaller sub-models conformant to the same meta-model [MKG15]. For faster access to model elements, some authors have proposed model indexers [BK13] – similar to those existing in relational databases. Finally, caching techniques for large models and queries over them have been proposed in [D16, DSC19].

Traditional MDE settings based on EMF typically involve expert engineers operating on a desktop IDE where models are treated like any other software artefact and, thus, are persisted as files. These are amenable for asynchronous collaboration, e.g., via version control systems [FdRM+18]. LCDPs, on the contrary, target low-technical profile citizen developers and strive for multi-user (possibly synchronous) collaboration. Particularly, low-code platforms built atop MDE (i.e., LCEP) also have models as their backbone, with the difference that every model interaction occurs within a browser. Model persistence is, hence, transparent to the user. Therefore, there is comparatively more flexibility in addressing model persistence.

To overcome the limitations of file-based persistence, approaches for EMF, like CDO[1] or Teneo[2] have proposed a data persistence based on relational databases. Other approaches, like Morsa [ECM11] or NeoEMF [DSB+17] use non-relational databases to persist and query very large models.

The ultimate goal is devising persistence mechanisms that are *scalable*, to handle models on the millions of elements; *efficient*, to achieve reasonably reactive updates (e.g., no user interaction takes more than a few seconds to be processed); and *concurrent*, to support multi-user synchronous collaboration. We believe that traditional

[1] https://www.eclipse.org/cdo/
[2] https://wiki.eclipse.org/Teneo

file-based persistence defies these goals, but it is a legacy format from traditional MDE. File systems lack native model-specific caching mechanisms and multi-user handling, and XMI, the de-facto format for model persistence, has fixed language granularity, thus hampering scalability. Proposed solutions such as indexers [BK13] or partial loading [WKG+16] are effective given the historicity of vendor lock-in on Eclipse-based solutions. However, low-code solutions can address this problem differently. Our approach is based on database persistence, but we resort to a cloud-native database, with flexible storage and querying mechanisms, like Elasticsearch[3]. In contrast to the existing approaches that also use non-relational databases, like Morsa and NeoEMF, our proposal is not limited to EMF. Instead, it relies on a neutral data model to represent models, which might be defined using diverse technologies.

2.2 DSL definition in Dandelion

2.2.1 Overview

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 visualised, 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. Note that this WP is only concerned with the DSL syntax, while its execution semantics is dealt with in WP5.

The tool supports two roles: the language engineer, in charge of defining DSLs and customising ther modelling editors, and the citizen developers, who use the graphical editors within a LCDP.

The language engineer defines the abstract syntax of the DSL via a meta-model [BCW17], a class diagram describing the elements of the language, their properties, relations and integrity constraints. We consider graphical concrete syntaxes, which are given in reference to the abstract syntax model. Then, the approach considers elements to enhance the DSL scalability, in particular graphical pagination mechanisms, and abstraction patterns [JGL17][dLGS13] (to summarise parts of a model into a more abstract representation, which can be explored using hierarchical decomposition). In the backend, meta-models and models are persisted in Elasticsearch, a cloud-native database, with flexible storage and querying mechanisms.

Once defined, the citizen developers may use the editors developed by the language designers.

[3] https://www.elastic.co/

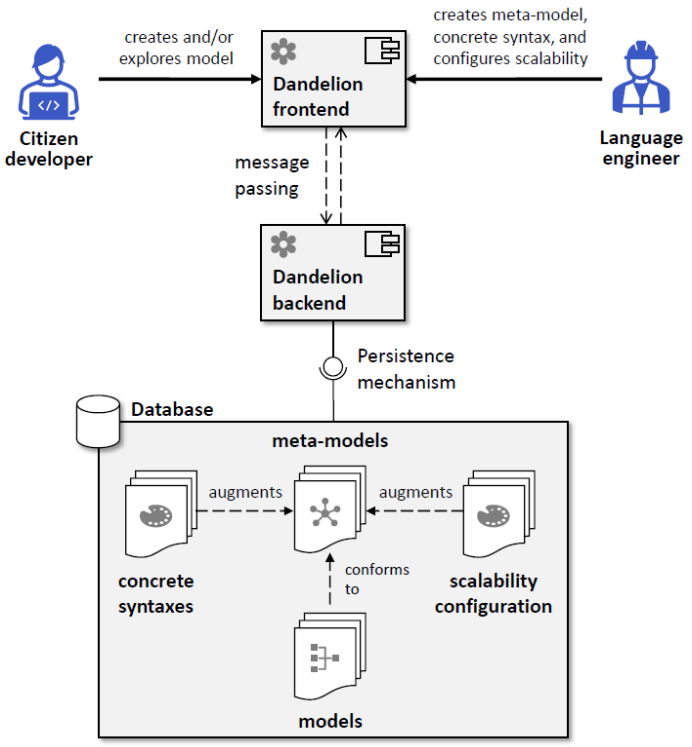


Figure 2. Dandelion approach and architecture for defining DSLs for LCEPs

In the following, we describe the main components of Dandelion.

2.2.2 Neutral data model

Given the plethora of LCDPs targeting different domains and purposes, we argue that a practical graphical DSL framework should be agnostic of the modelling techniques employed. This will facilitate its integration with other frameworks, enabling heterogeneous modelling. For this purpose, we propose a neutral data model that harmonises models from different platforms (see Figure 3).

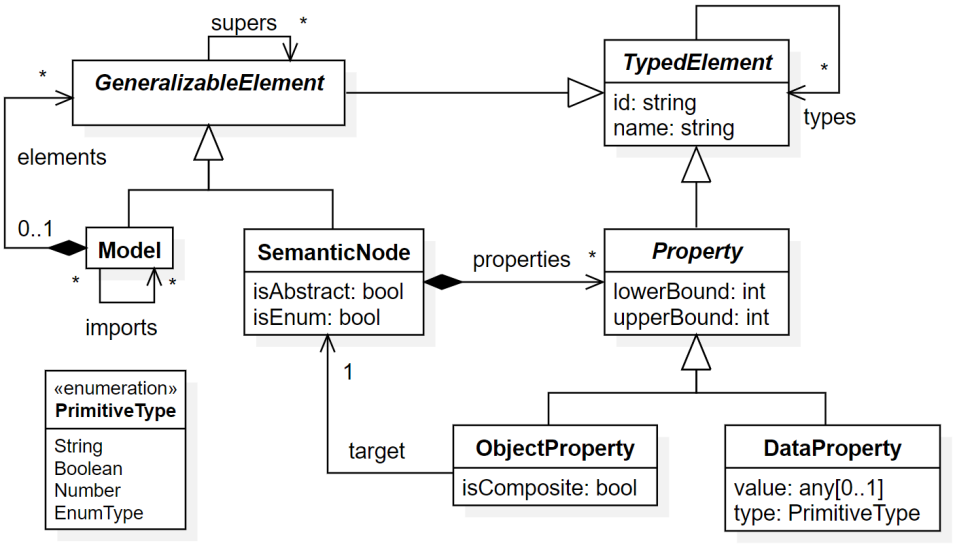


Figure 3. Dandelion's neutral data model

This data model represents objects as SemanticNodes and links as *ObjectProperty* objects. SemanticNodes have properties with a cardinality given by the *lowerBound..upperBound* interval. Properties can be either primitive data types (*DataProperty*) or reference types (*ObjectProperty*). The latter can be composite. *SemanticNodes* can be declared abstract and be used to represent enumerations (*isEnum*). In such a case, each *Property* is a literal of the enumeration.

The data model can represent both models (where objects/links are mapped to *SemanticNode* and *ObjectProperty*) and meta-models (where classes/associations are mapped to *SemanticNode* and *ObjectProperty*) in a uniform way. Therefore, this approach is level-agnostic [LA22] and – beyond traditional two-level modelling approaches like EMF – can represent an arbitrary number of meta-levels since both *Models* and *SemanticNodes* (at any meta-level) may have a type. Also beyond EMF, the approach explicitly reifies the notion of *Model*, enables multiple or no model types (relation *TypedElement.types*) and nested models (relation *Model.modelElements*). Both *Models* and *SemanticNodes* can be generalised. Models, in particular, can reference other *Models* via *Model.import*, possibly coming from different heterogeneous formats, thereby supporting interoperability. The current data model version does not support the definition of invariants expressed in OCL, but this is left as future work.

2.2.3 Visual concrete syntax

To visualise models conformant to the neutral data meta-model, it is necessary to provide them with a concrete syntax. Dandelion supports graphical concrete syntax, for which it introduces the visual concrete syntax meta-model of Figure 4, whose instantiations provide the models with graphical visualisation.

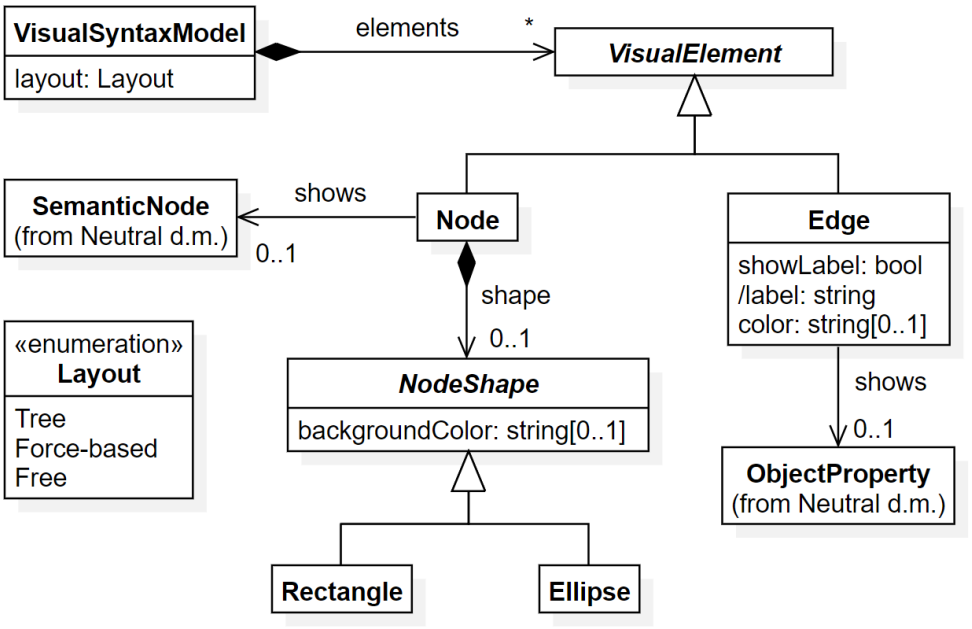


Figure 4. Dandelion's visual concrete syntax meta-model

The root element of the meta-model is *VisualSyntaxModel*, which contains several VisualElements which infuse visual representations into the elements of the neutral data model. In particular, *Nodes* and *Edges* are mapped to *SemanticNodes* and *ObjectProperties*, respectively.

On the one hand, *Nodes* can optionally feature a *NodeShape* (be it a rectangle or ellipse) with a background colour. On the other hand, *Edges* are depicted using arrows with optional labels and colours. The label, if shown (*showLabel*), is the name of the shown property (*Edge.shows.name*).

2.2.4 Scalability configuration

Large models become unwieldy and impractical. This calls for the introduction of scalable mechanisms to handle their size. In Dandelion, scalability is handled through a scalability meta-model, as seen in Figure 5. This meta-model works like the visual syntax meta-model in decorating the neutral data model elements. In particular, it permits defining fine-grained *ScalabilityMechanisms* over any *TypedElement* of the neutral data model.

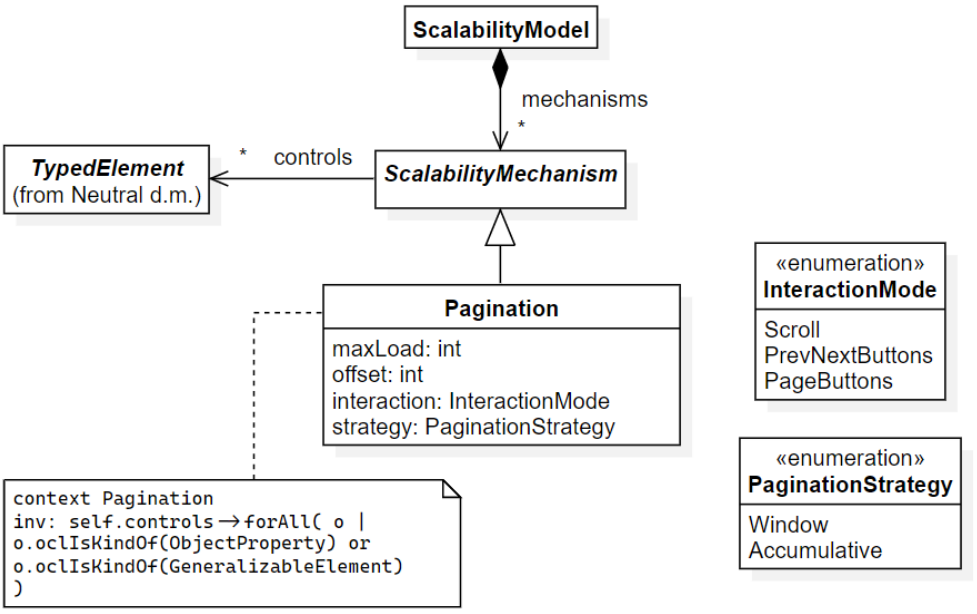


Figure 5. Dandelion's scalability meta-model

Currently, the meta-model supports pagination: elements are split into pages of maxLoad capacity, and the user can navigate between them. At every interaction, several elements are impacted within pages (*offset*). The interaction is also customisable with an *InteractionMode*. *Pagination* can be, thus, shown as a scroll bar, as previous and next buttons, or as the traditional pagination number buttons. Furthermore, pagination is governed by a *PaginationStrategy*. With *Window*, elements are discarded and loaded according to offset. Alternatively, *Accumulative* does not discard previously loaded elements.

2.2.5 Sensemaking strategies

Manipulating models is central in many disciplines, including software development in low-code platforms. Understanding models is, therefore, essential. This process of understanding, also called *cognition* or *sensemaking*, is highly complex since it involves numerous factors, from the user's prior knowledge to the tools' visual syntax [M09]. The process is also iterative and gradual. Users alternate between complementary actions to achieve their objectives, such as structure understanding, exploration, and visualisation [PAK+15]. Sensemaking strategies can be, thus, tailored to assist recurrent user goals within concrete domains.

These strategies have been applied mainly to graphs for their shared understanding purposes in data mining, network analysis, and recommender systems. Usually, they are applied to graphs devoid of semantics. That is, graphs whose interest lies in their structure and whose constituents (i.e., vertices and edges) carry no more information than unstructured metadata. Models, on the contrary, are infused with semantics both in their vertices (e.g., attributes and stereotypes) and in their edges (e.g., inheritance, multiplicities, navigability, and multiplicity of references). This renders conventional graph sensemaking strategies ineffective and calls for creating strategies adapted to particularised model patterns.

Most model sensemaking strategies are scattered across platforms and designed ad-hoc, thus hampering their reusability. In addition, sensemaking strategies frequently have different names despite having the same functionality. We propose introducing *language-agnostic model sensemaking strategies* to solve these issues, where strategies are defined on meta-model snippets and then mapped to specific DSL meta-models. This way, strategies can be specified once and applied to different DSLs. With their introduction, we intend to improve overall model comprehension.

Model sensemaking strategies impact how users interact with models. This can entail adapting features provided by the model editors (e.g., click, zoom, panning, or available tabs) to fulfil the purpose of the strategy. This differs from editor interaction mechanisms, like specifying how models are to be created [SSF19]. Instead, sensemaking strategies are independent of the targeted languages, ensuring their generalisation. They typically encapsulate a model exploration task (e.g., visualising the model connectivity or expanding the details of a node).

Figure 6 depicts a schema of the structure and application of a model sensemaking strategy. The strategy is defined by a (1) *context meta-model* and is applied to a (2) *target meta-model*. To do so, the user must specify a (3) *binding* that maps every element of the context meta-model to elements of the target meta-model. That way, whenever a (4) *model* conformant to the target meta-model is (5) *visualised*, it can benefit from applying the strategy.

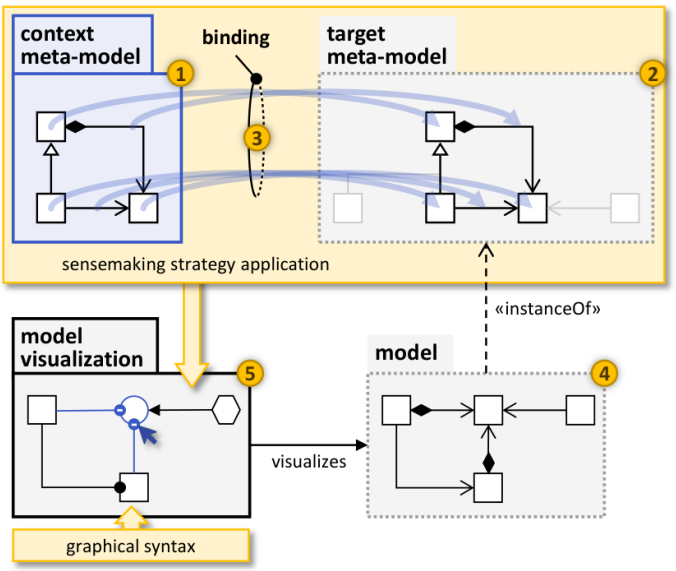


Figure 6. Scheme of definition and application of a model sensemaking strategy

The context meta-model shapes the sensemaking strategy, and it is its usage interface. Each element can be considered a conceptual “hole” to be filled by a meta-model element. To apply a strategy, there must be a structure-preserving mapping relating each class, attribute, and reference in the context meta-model to a class, attribute, and reference in the target meta-model, respectively [dLGS13]. This is called a *binding*.

We propose defining model sensemaking strategies using a consistent format similar to traditional software design patterns divided into sections [GHJ+94]. Namely: intent (i.e., what is the goal of the pattern), structure (i.e., the context meta-model of the strategy and its parts) and motivating example, applicability, consequences, and variants.

The following paragraphs exemplify these concepts with the *drill-down strategy*.

Intent. Composition is the most restrictive type of reference, and it involves two roles: the *container* and the *containees*, which are contained by the container. The containees' existence is contingent on that of the container: when the container is deleted, so are the containees. There is, therefore, a difference in knowledge expressivity between the container and the containees. This originated hierarchy can be exploited to create a *drill-down* navigation: composition can be traversed downwards to reach the containees, thus delving into the containees, or upwards to reach the container from the containees.

Structure and example. Figure 7 presents the drill-down strategy and an application example. The (a) context meta-model contains the involved elements in the strategy: a Container, a Containee, and a containment relation between them. The strategy is applied to an (b) example target meta-model on geography classification for

demonstration purposes. The strategy is (c) bound twice: one for the Planet-Country containment (binding 1), and another for Country-City (binding 2). The result of applying the strategy to a large model such as (d) (i.e., a model conformant to the targeted meta-model) is shown in (e).

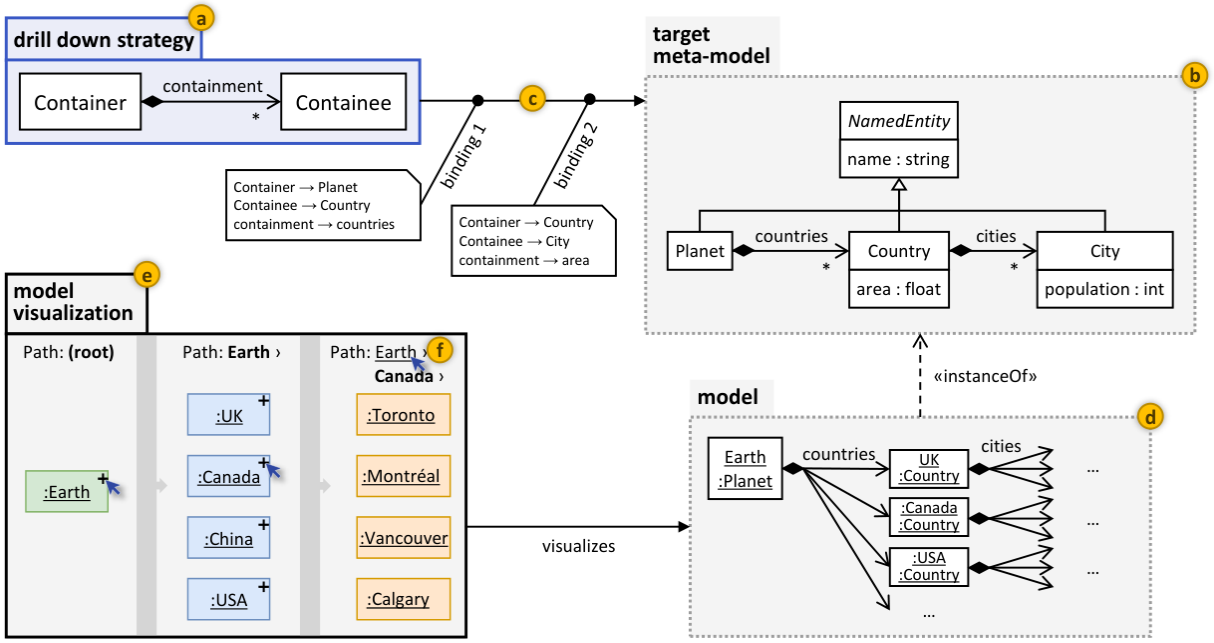


Figure 7. Drill-down sensemaking strategy

Applicability. The pattern can be applied to any containment relation between two classes. The strategy should be applied to the maximum number of containments to exploit the vertical nature of the exploration.

Consequences. The resulting interaction is a visualisation per layer. Containers hide all their Containees, and an explosion of the Containees takes place on user demand. Usually, it is implemented with an addition to the targeted language graphical syntax that triggers the action. In the example, a plus sign marks an exploitable containment.

Variants. The strategy can also feature *breadcrumbs* to situate the user within their exploration. Breadcrumbs are interactive in that they allow traversing back to upper layers (e.g., (f)).

2.3 Architecture and tool support

In this section, we provide details on the architecture and the tool support. Section 2.3.1 describes the architecture of the tool and the rationale behind it. Section 2.3.2 analyses the tool support.

2.3.1 Architecture

Figure 8 presents Dandelion's architecture, which is split into *frontend*, *backend*, and *persistence*.

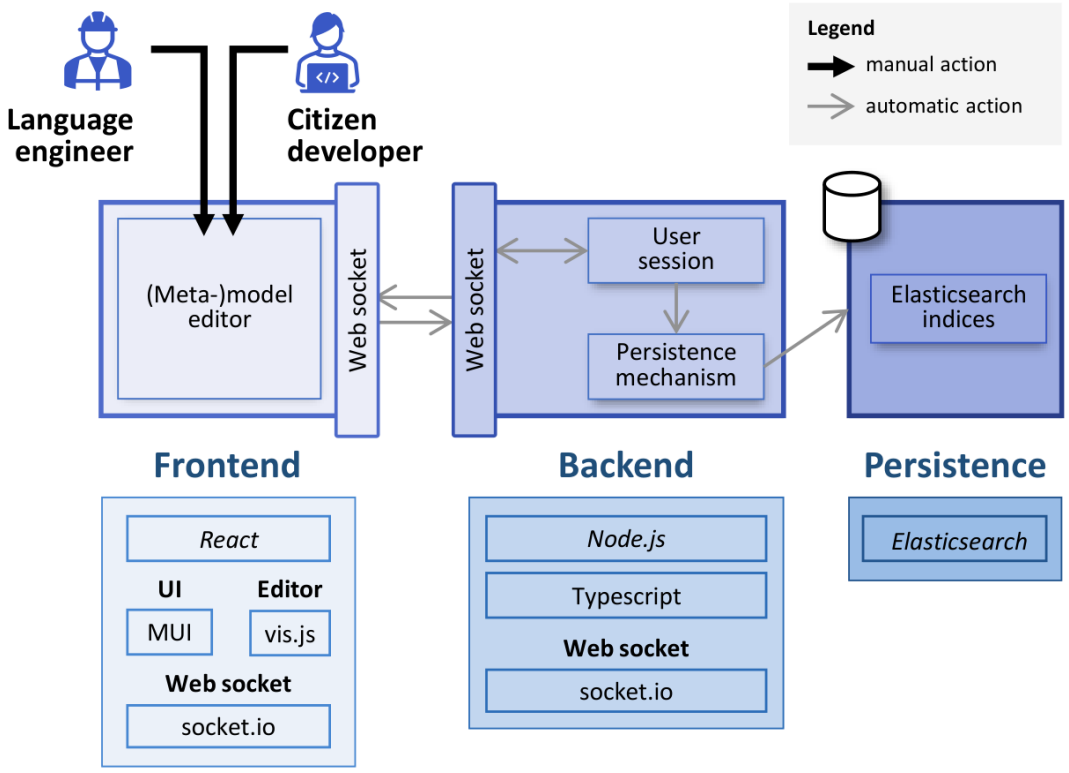


Figure 8. Dandelion's architecture

Both citizen developers and language engineers can use their browsers to access the application's frontend, which is deployed as a single-page web application (SPA). The frontend is developed in React,[4] and uses the vis.js[5] library to visualise data. The frontend and backend communicate back and forth using web sockets, which are implemented using the socket.io[6] library. This permits the frontend to receive real-time updates from the backend, and vice versa.

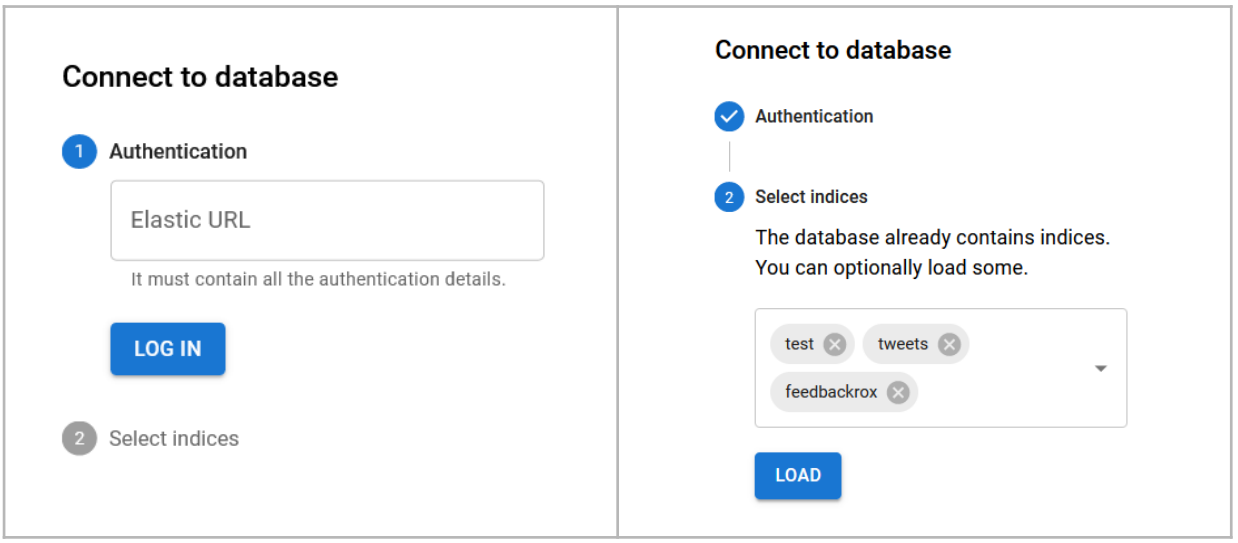
The backend is developed in Node.js[7] and is programmed in Typescript.[8] It keeps track of the connected users and their active sessions and implements a persistence mechanism to manage model loading.

[4] https://reactjs.org/
[5] https://visjs.org/
[6] https://socket.io/
[7] https://nodejs.org/
[8] https://typescriptlang.org/

The persistence is implemented in Elasticsearch, a document-based, distributed, scalable, open-source search engine. The basic unit of interaction with the database is the Elasticsearch index, which contains multiple *TypedElement*s (from Figure 3). Data in Elasticsearch indices must implement the neutral data meta-model in their mappings.[9]

2.3.1 Tool support

The first step for using the tool is connecting to an Elasticsearch database, as shown in Figure 9. The user provides the Elasticsearch entry point URL, from which Dandelion will extract its indices, which the user can optionally load for model manipulation.



a) Authentication details **b)** Indices selection

Figure 9. Connecting the Elasticsearch database

This grants access to the (meta-)model editor, which has three distinguished parts: the tree view on the left; the (meta-)model explorer, in the middle; and the selected element editor, as seen in Figure 10.

[9] https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping.html

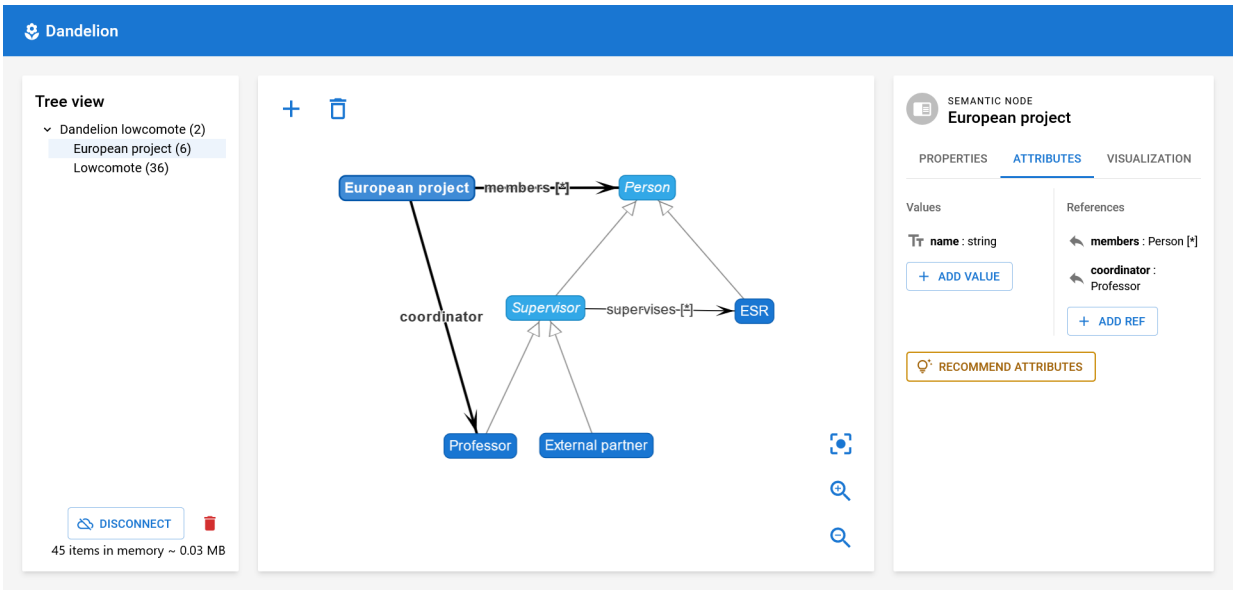


Figure 10. Dandelion's frontend (meta-)model editor.

Each loaded Elasticsearch index corresponds to an entry in the tree view. This display takes advantage of the multilevel nature of Dandelion's neutral data meta-model to represent nested models.

The central view visualises meta-models and their instances. The former displays meta-classes, their relations, and their inheritance. It also distinguishes abstract meta-classes with a different colour and name in italics. At the top of the view, the plus icon and the trash bin permit adding new elements (i.e., *SemanticNode*s or *Model*s), or deleting them, respectively. The bottom buttons permit re-centring the view and zooming in and out.

The right-most panel displays and permits the manipulation of the properties of the selected element. Depending on the type of selected element, its content changes. On the one hand, for *SemanticNode*s inside a meta-model (as the one in Figure 10), the panel exposes name, type, superclasses, is-abstract, is-enum, the attributes defined therein, and, optionally, a definition for the concrete syntax of the component. Instances of these *SemanticNode*s list and permit manipulating their attributes in this panel. On the other hand, *Model*s permit specifying if the model is an instance of another model, its scalability configuration, and its visual syntax configuration (cf. *VisualSyntaxModel* in Figure 4).

Models with a defined concrete syntax will display elements accordingly. For instance, in Figure 11, the central *Lowcomote* node is of type *European Project*; blue nodes are instances of *Supervisors*; and pink ones are of *PhD student*. Each displayed node has had its concrete syntax defined in the instanced meta-model.

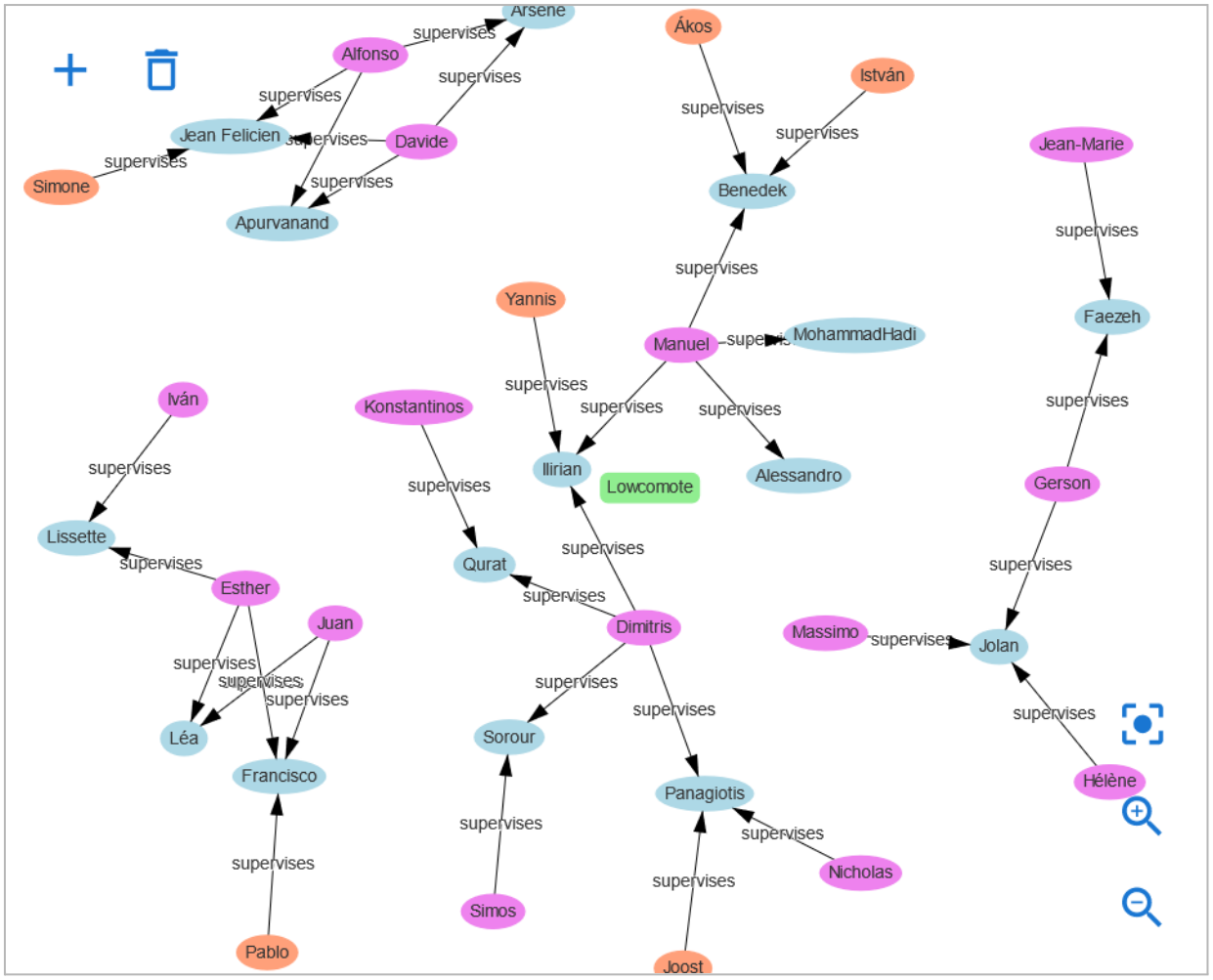
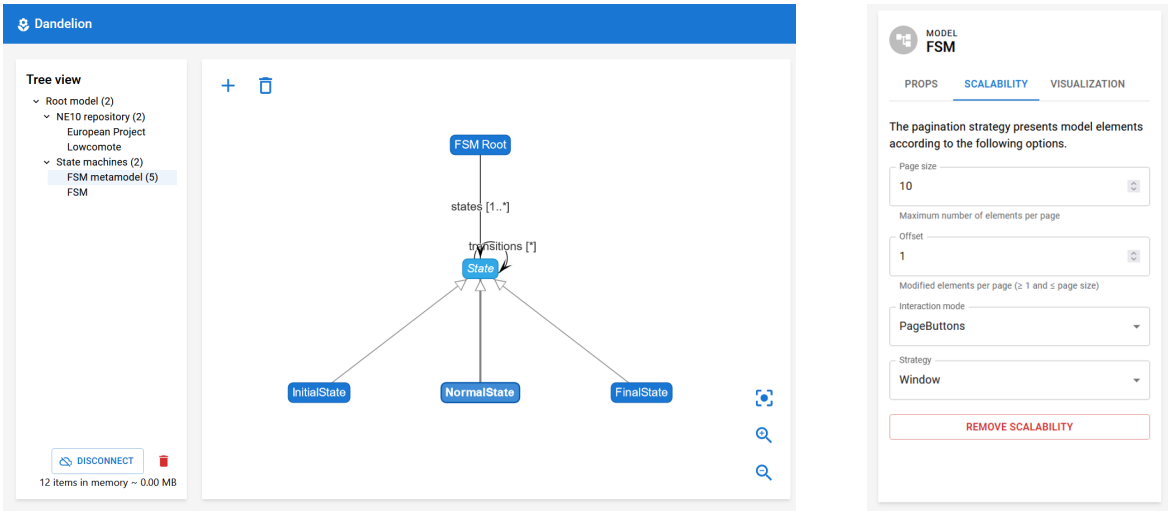


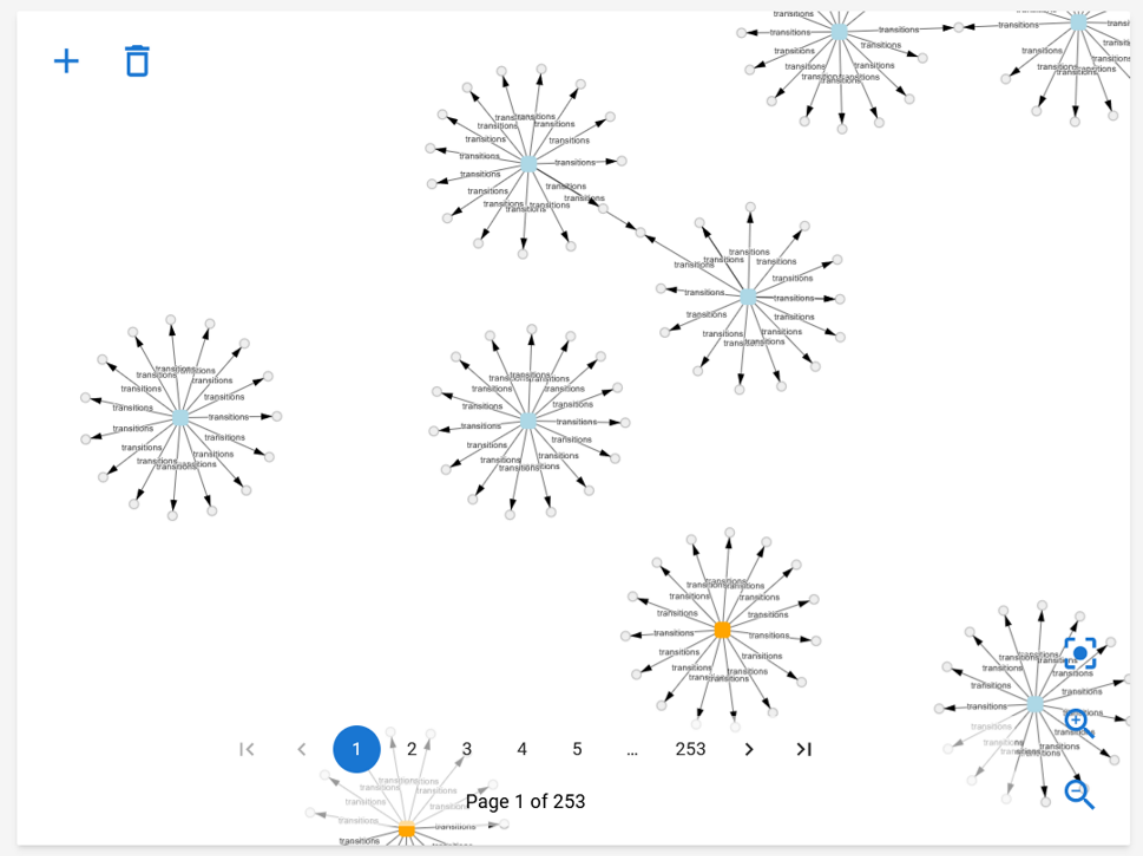
Figure 11. An example of a model with a defined concrete syntax.

Figure 12 is the representation of a synthetic finite state machine with around 2500 states. The FSM metamodel defines the following concrete syntax: initial states are green, normal states are blue, and final states are orange. It also demonstrates one scalability mechanism: pagination. Instead of showing every state on the same screen, the model is split into pages of the same size (10 elements in this case). To better understand the underlying models, pagination features *proxy nodes*, which are depicted as small grey circles. These are elements immediately adjacent to the loaded models but do not belong to the current page.

Finally, please note that sensemaking strategies are not yet included in the tool, but it is ongoing work.



(a) FSM metamodel and its associated scalability configuration.



(b) FSM instance showing pagination and proxy nodes.

Figure 12. Visualising a huge finite state machine with pagination

3. Recommendation Support for Modelling Environments

In parallel to the work presented in Section 2, and attached to the work of ESR1, we have developed a generic model-driven framework capable of generating task-oriented recommender systems (RSs) to assist in the modelling tasks using DSLs. The framework provides a DSL where RSs developers can define the settings that they would like to have in the RS. Thus, the DSL allows customising every aspect of an RS, including a description of the recommended items and their features, the profile and preferences of the target subjects of the recommendations, the pre-processing techniques to apply to the data, the data splitting configuration, the recommendation methods, and the evaluation procedures and metrics.

This work has resulted in publications in the LowCode'20 workshop at MoDELS [ACG+20], in the SLE'21 conference [APC+21], the SoSyM journal [ACG+22a], and the ASE'22 conference [ACG+22b].

The following sections contain background information related to recommender systems (Section 3.1), related work (Section 3.2), the proposed approach (Section 3.3), the used domain-specific language (3.4), tool support (3.5), and the experiments executed (3.6).

3.1 Background

Recommender Systems (RSs) are software tools and techniques that suggest items considered relevant for a particular target. The term "*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].

Recommender systems can be classified into the following broad categories based on how the recommendations are made: *content-based*, 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*, 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 target 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. 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 file 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]. Our goal is to facilitate the construction of RSs for modelling languages.

3.2 Related work

In this section the review the two main areas of related works: RSs for modelling languages, and automated approaches for the synthesis of RSs.

3.2.1 Recommenders for Modelling Languages

According to [ACG+22a], the most common usage purposes for recommenders in MDE are completion, finding, repair, reuse, and to a lesser extent, creation of modelling artefacts. The recommendations typically apply to models and meta-models, while recommenders for model transformations and code generators are scarce. Droid can be applied to any kind of artefact, provided that it is defined by a meta-model.

Most recommenders for modelling languages target UML, especially class diagrams. IPSE [G12] has a knowledge-based RS that guides students on creating class diagrams, and the recommendations build on Prolog constraints defined by the teacher. RapMOD [KM17] recommends relevant auto-completion actions for graphical UML class diagrams. REBUILDER [G04] relies on case-based reasoning, Bayesian networks and WordNet to recommend class diagrams similar to a given one. Elkamel et al. [EGB16] use similarity metrics to recommend similar classes to the ones in the current class diagram. Other researchers propose RSs for other UML diagrams: Cerqueira et al. [CRB16] propose a CB approach for recommending behavioural features for UML sequence diagrams, and Aquino et al. [RSV20] present a recommender of actors and use cases for use case diagrams. While these works tackle useful modelling tasks, they serve a specific modelling language and the recommendation method is fixed. Instead, Droid is not UML-specific but it permits customising the target modelling language, the kind of items to be recommended, and the recommendation algorithm.

Some approaches aim to provide semantically related terms and context-sensitive information for a modelling task. Burgueño et al. [BCL+21] propose a domain concept recommender based on the analysis of the textual information available on the domain model being constructed, as well as on general knowledge about the business domain. The domain modelling tool DoMoRe [AKS18] exploits a knowledge base of domain-specific terms and their relationships to provide context sensitive recommendations. Other tools, like Extremo [MdLN+18] or the assistant envisioned by Savary-Leblanc [S-L19], employ semantic similarity based on lexical databases like WordNet to recommend semantically related terms. While these tools target a specific

modelling task, our framework is generic and configurable for arbitrary modelling languages.

Recommenders have also been applied to business process modelling. For example, to recommend complete process models based on the user profile [KHM20], as well as finer-grained recommendations that pursue completing a process model with new fragments [KHO11], activity nodes [DWL+17, LCX+14], tasks [ECK15] or actor roles [ECK15]. Again, these works are specific to a modelling language, and the recommendation method is fixed.

In contrast to the previous language-specific approaches, others are language-independent. These are typically applicable to arbitrary modelling languages defined in a given meta-modelling framework, such as EMF. For example, PARMOREL [BRH20, IBR+20] uses reinforcement learning to repair malformed EMF models based on the user preferences and the experience gained from previous repairs. ReVision [OPK+18] suggests consistency-preserving model editing rules for model repair. SimVMA [S19] uses clone detection to help modellers find models or operations relevant to them. Finally, Kögel [K17] proposes to analyse the history of past model changes to suggest recommendations, and foresees the use of machine learning, heuristic search algorithms, association rules and decision trees. Altogether, even though these works plan on frameworks for different languages, the recommendation method is fixed, and the recommendations cannot be customised, as we can do using Droid.

3.2.2 Recommender System Generation

While we can find many RSs for modelling languages, most were developed by hand from scratch, which requires a high effort [36]. Hence, recent studies [ACG+22a] have identified the need for methods and tools automating the construction of recommenders for modelling languages. This work aims to fill this gap. Next, we compare with other related approaches.

Fellmann et al. [FMJ+18] define a reference model with the data perspective requirements of RSs for process modelling. The model can be instantiated as a guide for developing new process modelling recommenders, or to assess existing ones. While useful, the approach is specific to process modelling, and does not provide automation or code synthesis. Rojas et al. [RU13] present an MDE framework to create mobile RSs of geographic points of interest. The framework helps defining the structural, behavioural and navigational aspects of the RS, and customising the user preferences, similarity metrics and similarity formula. In [RFS09], a similar solution is used to recommend trips and tours. However, in both works, the target domain of the recommendation is fixed.

We also find MDE proposals to support non-expert users on applying data mining. For example, Espinosa at al. [EGZ+13, EGZ+19] reuse the past experiences of data

mining experts to compute the accuracy for a given new dataset and recommend the one with the best performance. The framework permits customising the data mining task to perform, the evaluation method and metrics, and the mining algorithm. Even though this solution offers the flexibility and benefits of MDE, the generated recommenders are data mining applications.

In a more general setting, Hermes [DGL14] is a generic framework to build recommenders for modelling environments. Its extensible architecture permits defining new recommendation strategies, new widgets to trigger and display the recommendations, and new contexts to adapt the recommendations to the modelling environment. These elements are coded as extensions of base classes, or registered in the case of resources like icons and labels. Hermes provides a dashboard to define the class extensions, and supports the manual testing of the recommender. In contrast, our DSL Droid does not require coding, but it provides a simple syntax to configure the kinds of recommended items, the recommendation method, and its evaluation based on standard metrics. Moreover, it automatically generates a tailored RS as a web service to make it available from arbitrary environments.

More similar to our proposal, the vision paper [dCdRN20] foresees a low-code development environment where end users can define RSs by using graphical interfaces, drag-and-drop utilities and forms. The authors aim to support the construction of arbitrary RSs, not specific for modelling languages. The low-code environment will build on a generic meta-model to provide components implementing recurring functionalities for RSs, such as data pre-processing, capturing context, and producing and presenting recommendations. The authors foresee having several DSLs to configure each aspect of the recommender. Our philosophy is similar, but we focus on RSs for modelling. This way, our DSL allows the fine-grained specification of the recommendation target and items, and our tooling generates a RS available as a REST API that can be integrated in other tools.

3.3 Proposed approach

The architecture of the proposed approach is shown in Figure 13. In the proposal, we apply MDE techniques to develop RSs. In label 1, the Droid editor is provided. The editor provides a wizard to pre-design a template of a Droid project. The RS developer provides, via the wizard, a meta-model of the notation that will be the subject of the recommendation.

Also, we assume the existence of a repository of models conformant to the meta-model. The wizard also provides an interface to explore and collect these models conformant to the meta-model (label E2). The models collected are used for the training and testing of the RSs.

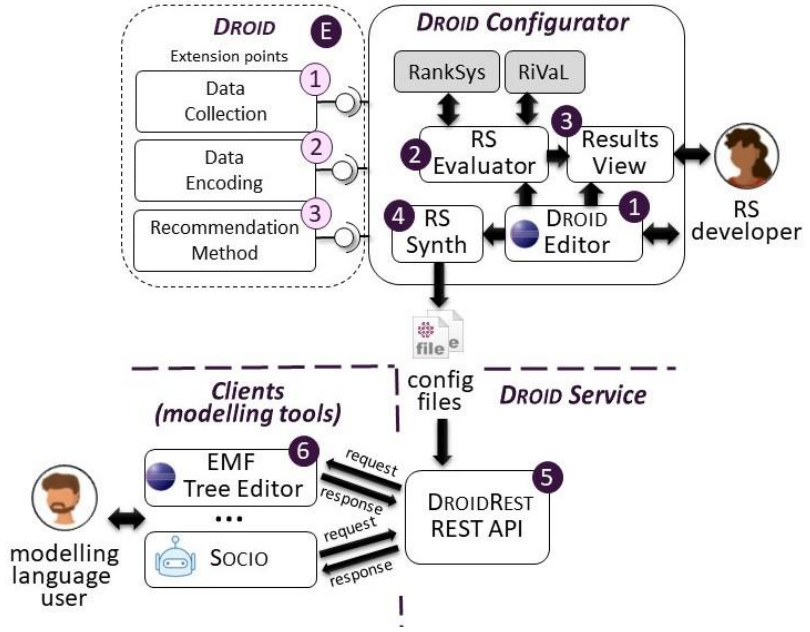


Figure 13. Overview of the proposed approach to define task-specific RSs

After collecting this data, the wizard generates a pre-designed template with the configuration of a RS. Such configuration is performed via a textual DSL that allows the definition of all aspects of a RS, including the meta-model elements that will play the roles of target, item and item features, as in traditional RSs. The DSL also supports the configuration of different types of data pre-processing and 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.

Label 3 presents the results view. This part of the Droid configurator contains two views 1) pre-processing results view and 2) the methods training results view. The first one contains the summary and statistics of the models used for the training of the RS and the results of applying the different pre-processing techniques. The second one presents the results of each method specified with its corresponding metrics result. Label 4 shows the synthesiser of tailored RS (label 4) that will be deployed on a REST API service (label 5).

DroidREST is a generic recommender service that computes the recommendations based on the configuration files generated by the RS Synthesizer. These configuration files store the trained recommender that knows which items to suggest based on the context information. Hence, there is no need to deploy a different service for each RS defined with Droid. Citizen developers within different modelling tools (label 6) can make POST requests to the service, which receives a recommender name together with a JSON file containing the target object of the recommendation and its context (i.e., the items that the target contains). The response to the request is a list of recommended items for the given target, using the recommendation method selected

by the designer. In this context, the citizen developers are the users of the LCDPs, which typically lack a background in programming. Hence, it is important that LCDPs can integrate useful, easy-to-use mechanisms to assist these users in their development tasks.

Figure 14 includes an overview of the RS configuration process. In the first step, the RS developer needs to define the Droid project, such as the name, (step 1) and provide some data (step 2), specifically, the meta-model of the notation for which the RS is to be developed. Additionally, the set of instance models is to be used for training the RS. These two steps are done using the Droid wizard.

In step 3, the RS developer uses the Droid DSL to configure the desired features of the RS. Using this information Droid produces the target-item and item-feature matrices, considering the specific items and features indicated in the RS configuration. Then, the data is pre-processed using each combination specified (label 4) and the data is split into two sets: 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.

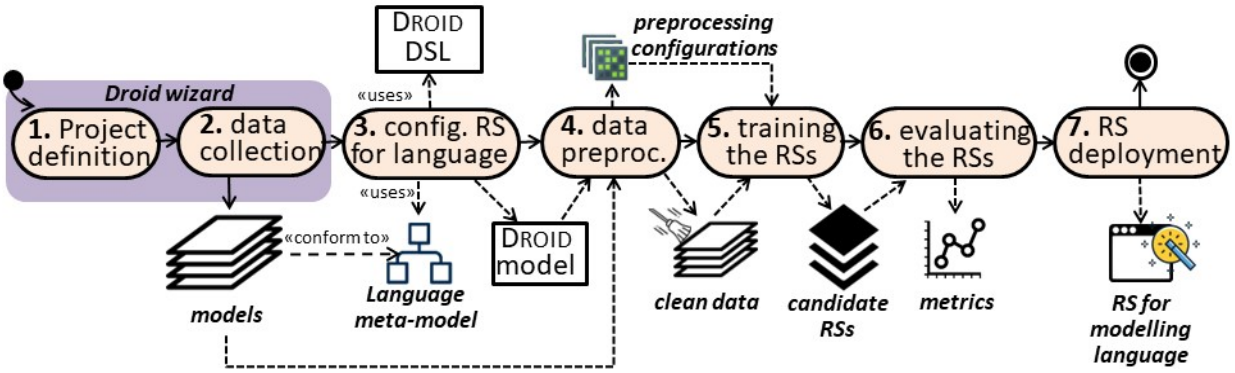


Figure 14. Overview of the process

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

3.4 Domain-specific language for configuring the RS

We have designed a textual DSL to configure and measure the performance of RSs for arbitrary modelling languages (as long as they are defined by a meta-model). The DSL allows configuring the recommendation method, the pre-processing technique, the data splitting and evaluation method, and the kind of elements to be recommended. The DSL provides a high level syntax for this task, which avoids the RS developer'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 15.

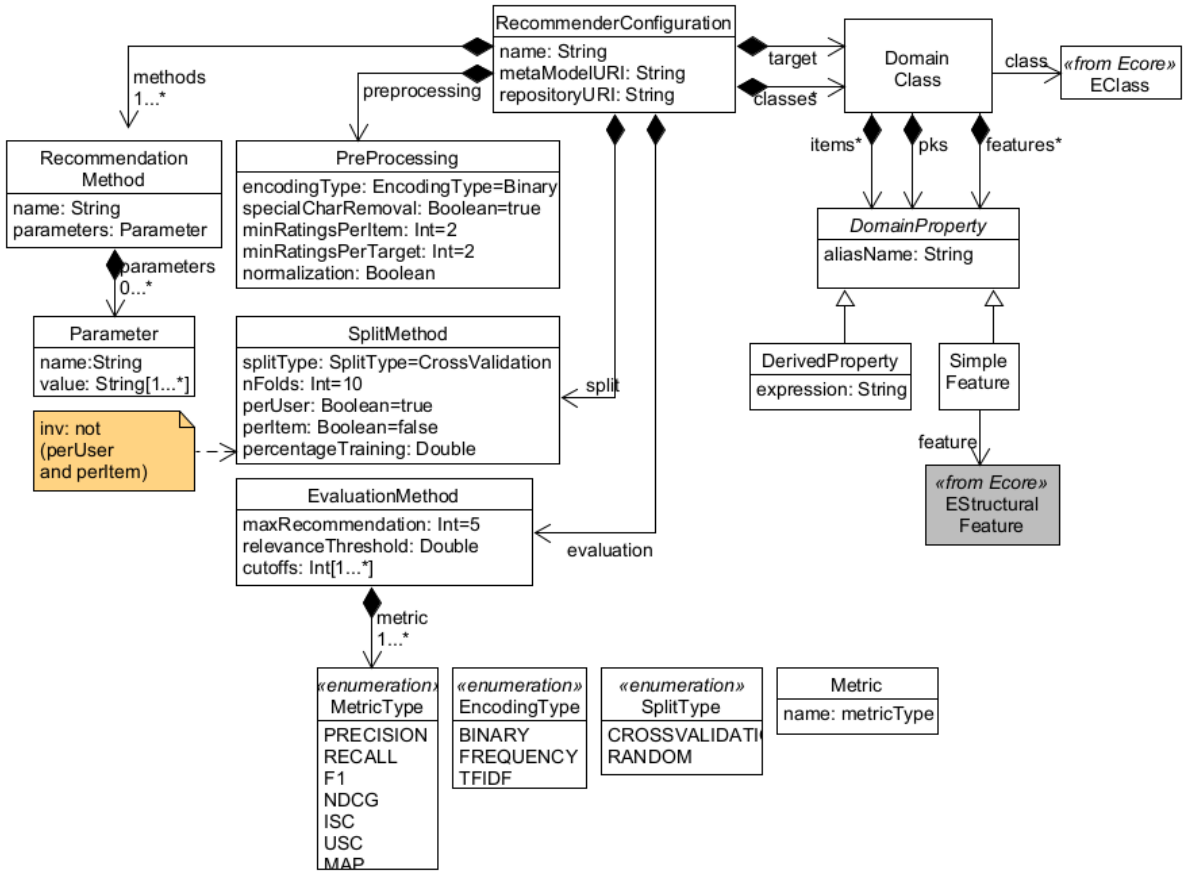


Figure 15. Meta-model of the DSL for RS configuration

The main class of the DSL is the *RecommenderConfiguration* class. This class is the container for 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 the location of a repository with 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 the *Parameter* class permits configuring their parameters (e.g., the neighbourhood size for collaborative filtering methods). The list of available recommendation methods as well as the parameters configuration is provided via extension point in Eclipse. Leveraging from the plugin-based architecture of Eclipse, Droid allows the RS language developer to define new recommendation methods and new methods to encode the data to be used for those recommendation engines.

The *PreProcessing* class allows defining the pre-processing techniques to be applied to the data. The configuration options include the encoding type for matrix generation (e.g., binary, frequency). If the encoding type is frequency it can be normalized or not; special character removal; minimum rating per target and per item.

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. *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.

Listing 1 illustrates the textual concrete syntax that we have devised for the DSL. The listing configures a RS for UML class diagrams, conforming to the meta-model shown in Figure 16.

In Listing 1, lines 1–3 define the name of the recommender, identify the meta-model of the language the RS is built for (cf. Figure 16), and the URL of a repository of instances of this meta-model (step 2 in Figure 14).

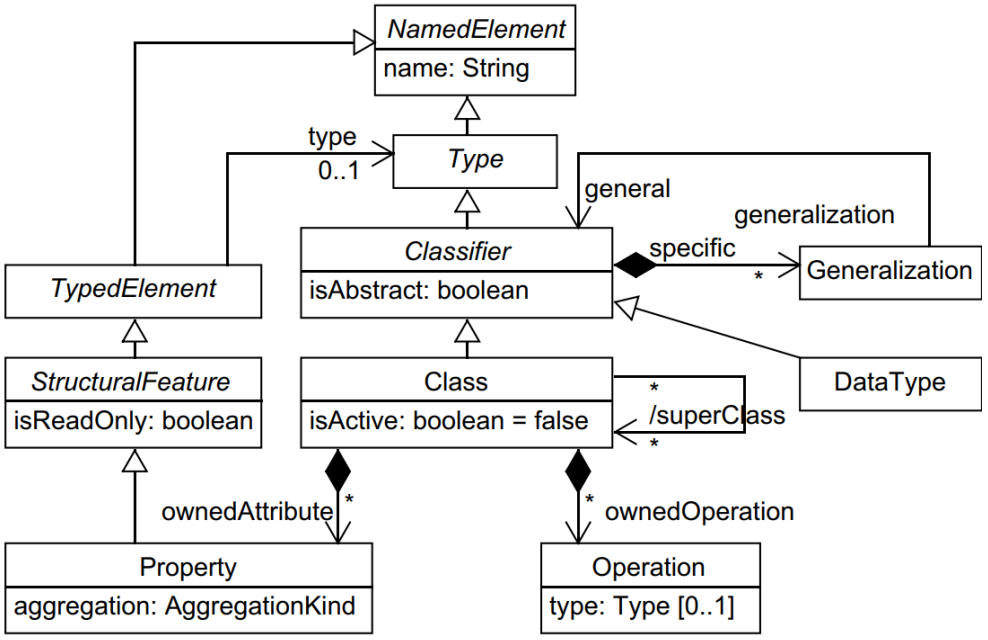


Figure 16. Simplified excerpt of the UML meta-model

The following lines configure the RS for the language (step 3 in Figure 14). Lines 5–9 specify the meta-model elements that will play the roles of target and items in the RS. These elements must belong to the meta-model provided in line 2. The listing sets the class *Class* as the *Target* of the RS, while its attributes, methods and superclasses are set as the *Items* of *Class*. This means that the RS will be able to recommend these three kinds of items for a given class. Then, lines 13–21 define the primary key used to

```
1  Recommender: "UMLRecommender"
2  Metamodel: "http://www.eclipse.org/uml2"
3  Repository: "/UMLRecommender/instances"
4
5  Target {
6    class Class {
7      item "attributes" : ownedAttribute;
8      item "methods" : ownedOperation;
9      item "super classes" : superClass;
10   }
11 }
12
13 Identifiers {
14   class Class {
15     pk feature name;
16   }
17   class Property {
18     pk feature name;
19   }
20   class Operation {
21     pk feature name;
22   }
23 }
24
25 PreProcessing {
26   encoding: binary;
27   specialCharRemoval: true,false;
28   editDistanceMerging: 2,3,4;
29   minRatingsPerItem: 1,2,3;
30   minRatingsPerTarget: 1,2,3;
31 }
32
33 Recommendations {
34   Split {
35     splitType: CrossValidation;
36     nFolds: 10;
37     perUser: true;
38   }
39   Methods {
40     ItemPop, CosineCB, CACF("5","10"), IBCF("5","10"), UBCF("5","10"),
41     CBIB("5","10"), CBUB("5","10");
42   }
43   Evaluation {
44     metrics: Precision, Recall, F1, NDCG, ISC, USC, MAP;
45     cutoffs: 5,10;
46     maxRecommendations: 5;
47     relevanceThreshold: 0.5;
48   }
49 }
```

Listing 1. Example of recommender system configuration using the Droid DSL

identify each target and item in the RS, as well as the features used for comparing this information for the item *Property*. In particular, its attribute *name* will be used as its primary key and for the comparison of attribute declarations.

Lines 25–30 allow the definition of different pre-processing techniques. Droid allows defining one or more values in each parameter and the system generates matrices with the specification of each combination. Line 26 describes the *encoding* types used

for Droid for matrix generation. These matrices capture the target and items interactions. Line 27 defines the *specialCharRemoval* option, which allows removing characters from the data (i.e. numbers, blank spaces, commas (,)). It can be specified as only one value (e.g. *true*); or both values (i.e. *true*, *false*). Line 28 describes the option *editDistanceMerging*, which allows specifying the editing distance below which two words are considered equal. Multiple numbers can be included in the form of a list and Droid computes each combination. Finally, lines 29–30 define the *minRatingsPerItem* and *minRatingsPerTarget* options, which are filters to select only those items or targets with a minimum number of ratings (e.g., a class with at least 2 attributes).

The remainder of the listing declares recommender preferences. The *Split* fragment (lines 34–37) 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 40–413) selects the recommendation methods to apply and evaluate. Among others, the DSL designer has selected some collaborative filtering methods such as *ItemPop* (item popularity) and *UBCF* (collaborative filtering user base with 5 and 10 neighbours). Section 3.5 will describe these methods. Finally, the *Evaluation* fragment (lines 43–47) selects the evaluation protocol. In particular, line 44 chooses the metrics to be used for the evaluation, line 45 define the cutoffs values, line 46 specifies the number of items to recommend, and line 47 defines a relevance threshold.

3.4.1 Data pre-processing

This section describes Step 4 of Figure 14, data pre-processing. 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. Data pre-processing is an essential technique in machine learning, which includes modifying or deleting irrelevant data or information from the original dataset [RRS15]. When it comes to model-driven engineering (MDE), data are models and meta-models. Droid offers five pre-processing techniques. Most options provided by Droid allow defining one or more values and the system generates matrices with the specification of each combination.

The pre-processing techniques supported for Droid are encoding types, special character removal, edit distance merging, minimum rating per target and minimum rating per item. The encoding types supported by Droid for matrix generation capture the target and item interactions. Droid supports multiple encoding types that vary depending on the recommendation engine that receives the matrix. At the moment Droid supports Ranksys [RS21] and MemoRec [dRdRdC22].

Figure 17 shows the supported encoding types: binary, frequency and normalised frequency. In each case, the matrix has as rows the targets (t), and as columns the items (i). In binary encoding (a), each cell is set to 1 if the target contains a particular item otherwise a 0 if it does not. Frequency encoding (b) calculates the times that a

particular target contains an item. Similarly, the frequency normalised encoding (5) counts the frequency of the item appearing on a particular item but uses a normalisation technique to map the rating to a particular scale. In this case, the values are normalised to a range between 0 and 1.

	i1	i2	i3	i4	i5
t1	0	1	1	1	1
t2	1	1	0	0	1
t3	1	1	1	1	1
t4	1	1	0	1	1
t5	1	1	0	0	1

(a) Binary matrix.

	i1	i2	i3	i4	i5
t1	0	1	5	6	3
t2	2	3	0	0	4
t3	4	1	1	3	5
t4	2	1	0	4	4
t5	8	5	0	0	2

(b) Frequency matrix.

	i1	i2	i3	i4	i5
t1	0.00	0.13	0.63	0.75	0.38
t2	0.25	0.38	0.00	0.00	0.50
t3	0.50	0.13	0.13	0.38	0.63
t4	0.25	0.13	0.00	0.50	0.50
t5	1.00	0.63	0.00	0.00	0.25

(c) Normalized frequency matrix.

Figure 17. Matrix encoding types for Ranksys library

The *specialCharRemoval* option removes characters from the data, like numbers, blank spaces and non-alphabetic characters such as exclamation points (!), commas (,), underscores (_), or symbols (@). The option accepts exclusively boolean values, and can be specified as only one value (e.g. *true*); or both values (i.e. *true*, *false*). The option *editDistanceMerging* permits the merge of words that may be similar or are the same. To calculate the similarity between two words we apply the Levenshtein distance algorithm. This algorithm measures how similar are two words using the number of deletions, insertions, or substitutions required to transform one word into the other [S+17]. For instance, if one word is “car” and the other one is “car” the distance is 0, as there was no required transformation. The distance between “car” and “cat” is 1 because only one substitution is needed to transform “car” into “cat” (change “r” to “t”). This configuration option allows specifying the integer number to use to merge two words based on the similarity within a dataset. Multiple numbers can be included in the form of a list and Droid computes each combination.

Finally, *minRatingsPerItem* and *minRatingsPerTarget* let defining filtering settings to include the items or target, depending on the option, that are present X times or more. For instance, if the minimum ratings per item specified is 3, the system will include only the items that appear at least 3 times within the whole dataset. This option also allows specifying one or multiple values in the form of a list.

3.4.2 Data splitting

Data splitting is the operation of partitioning the data into one or more subsets to perform an evaluation. These partitions are used for the training and testing of each defined RS [SB14a]. As the splitting configuration can impact notably the performance results, Droid provides the RS developer with different options and parameter setting configurations. The splitting techniques supported by Droid rely on the external toolkit

RiVal [SB14b]. Droid supports two data splitting types: Cross-validation and random. Cross-validation is a conventional approach depending on one parameter called *k* or *nFolds*. This parameter represents the number of groups that data will be split into. In each iteration, one of the groups or folds will be used for testing and the rest as training sets. Then, this process is repeated for each fold. The main purpose of this type of partitioning is to ensure good generalisation and to avoid over-training. Additionally, it is commonly used when evaluating multiple machine learning models with different algorithms or parameters.

Random splitting is a more common and easy to use technique. In the random splitting, the percentage of training needs to be specified and then the rest is reserved for testing. In this technique, the sampling for the test and training sets is done randomly following a uniform distribution.

For both split types the RS designer can choose between *perUser* and *perItem*. The per user technique refers to the splits built upon the available users. On the other hand, per item is split by the available items.

3.4.3 Methods supported

Droid supports the training and evaluation of multiple algorithms simultaneously, so that the designer can choose the most appropriate one for the given DSL. For this purpose, it provides two extension points. The first one (c.f. Figure 18 (a)) defines data encoding techniques for matrix generation. Recommendation methods use these matrices to train and evaluate their algorithms. Since each algorithm expects a specific encoding, data encoding details must be defined through this extension point. The extension point requires a name for the data encoding source, a Java class with the details of the data encoding and a description.

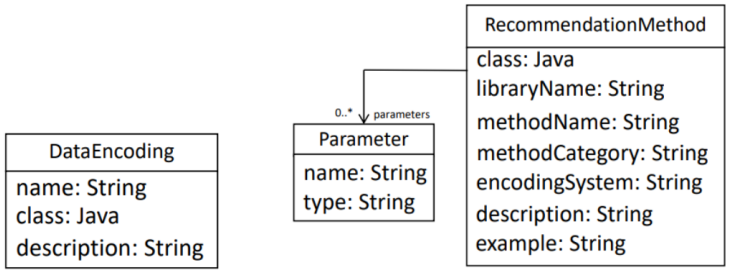


Figure 18. Extension point class diagrams

The second extension point (c.f. Figure 18 (b)) allows RS developers to register RS algorithms. This allows Droid to be algorithm-agnostic. Subsequently, the extension point requires a Java class with the details of the method source; the name of the library, method and category; a description and an example of use. Additionally, method parameters can be specified, which require a name and a data type. The

current version of Droid supports the following algorithms from two different libraries. From Memorec [dRdRdC22], we support Context-aware collaborative filtering (CACF), and from Ranksys [RS21] we support item popularity (ItemPop), content-based cosine similarity (CosineCB), user-based collaborative filtering (UBCF), item-based collaborative filtering (IBCF), user-based content-based (CBUB) and item-based content-based (CBIB).

3.4.4 Evaluation protocol

Research on algorithms for RSs has been receiving a lot of attention over the past decade. With the availability of a vast variety of algorithms comes the question of which is the most appropriate algorithm for a particular case. The decision on which is the best algorithm has commonly relied on experiments that compare the performance of algorithms using different metrics that typically provide a ranked list of candidates. The most common way to evaluate RSs is based on the ability to predict a user preference. But even though this is an important metric, it is insufficient to select and deploy a good RS [GS15].

To alleviate this problem, Droid supports the definition of multiple parameters to specify the evaluation protocol. The metrics supported by Droid to evaluate the candidate recommender systems (RSs) are: Precision, recall, F1, MAP (mean average precision), nDCG (normalised discounted cumulative gain), USC (user space coverage), and ISC (item space coverage). Additionally, Droid also supports the specification of the numbers for cut-offs (i.e., the number of most relevant items used to calculate the metrics); the number of maximum recommendations that the RS generate; and the threshold value that determines when a recommendation is deemed relevant.

3.5 Tool support

The Droid Configurator is an Eclipse plug-in designed to assist RS developers with the configuration and evaluation of RSs for modelling languages (https://droid-dsl.github.io/). It provides a wizard for the creation of Droid projects. The wizard requires that the RS developer provides a name for the RS being developed, the language of the modelling or meta-modelling technology that the RS will serve (e.g. ECORE, UML, XMI) and the data for the training and evaluation of the RSs. To facilities, the configuration of the RSs, the wizard, provides an option to automatically generate a default configuration setting.

Figure 19 shows the Droid configurator environment. The Droid editor (label 1) allows the configuration of multiple RSs through the DSL. This editor was built using Xtext, and includes syntax highlighting, auto-completion, and markers for errors and warnings. Suggestions for auto-completion (label 2) are presented in a pop-up window. This allows choosing elements of the meta-model of the modelling language the RS is created for (UML in our case). An excerpt of the UML meta-model is shown in the view with label 3, together with arrows to the Droid program where these are referenced.

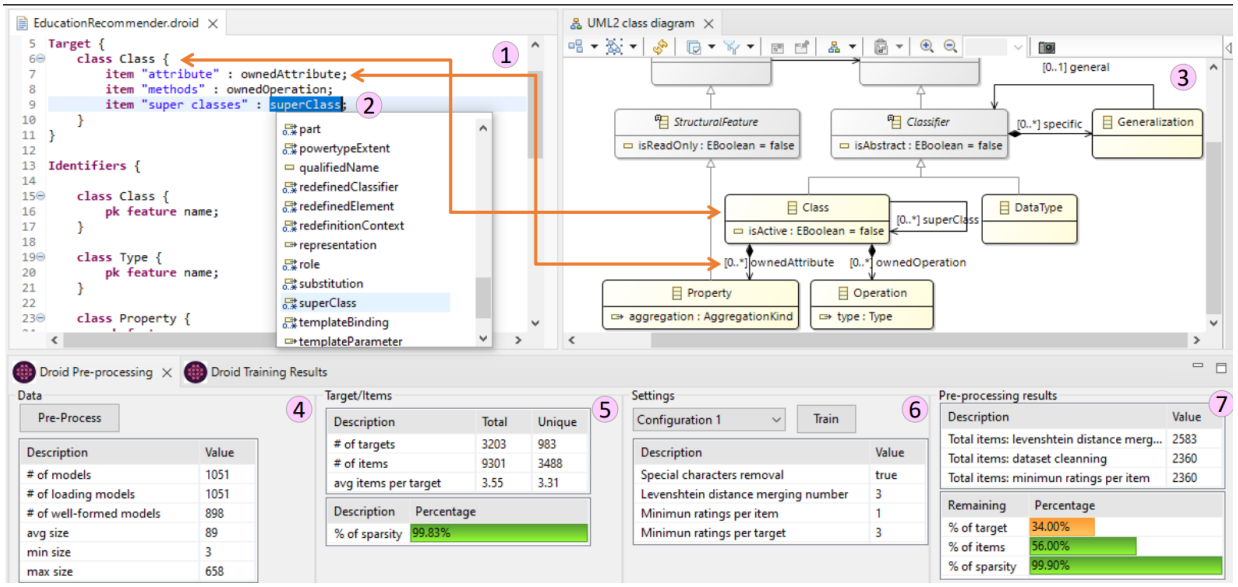
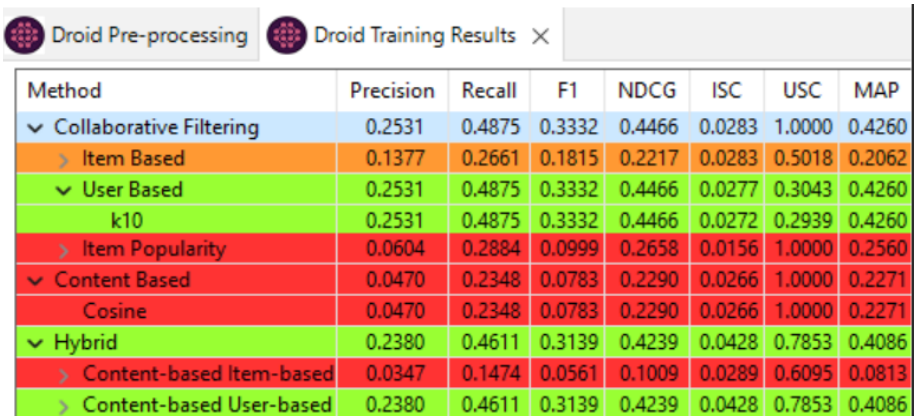


Figure 19. Screenshot of the Droid Configurator.

A Droid pre-processing view is available to inspect the result of each training configuration. The view is shown at the bottom of Figure 19. First, the data description section (label 4) is shown. This includes detailed information about the meta-models or models to be used as training and testing data. The panel includes information such as total number of models, number of loading models, number of well-formed models, and the minimum, maximum and average size of the models. The panel to the right (label 5) displays information about the targets and items, including the number of targets, items and average items per target in total and uniquely, and the percentage of sparsity in the raw data. The settings section (label 6) contains a combo-box with each pre-processing configuration, along with the details of their parameters (special character removal, Levenshtein distance, minimum rating per target and per item). Finally, the pre-processing results section (label 7) is displayed. The upper table describes the items left after each pre-processing technique has been applied. The bottom table displays the percentage of targets and items left after the cleaning of the data.

The evaluation results of the selected pre-processing configuration, for each selected method is shown in a different view. Figure 20 shows a screenshot for the example. Different colours are used to facilitate the understanding of the metric values using the F1 metric. Green is used to identify the top 20% methods, red to signal those RSs below the median, and the remaining methods are marked in orange. The results are displayed grouped by method category: Collaborative Filtering, Content-Based and Hybrid. Each category contains the different methods selected from that particular category. Within each method-category group, it contains a subsection per neighbourhood size, if applicable, as selected in the Droid configuration.



Method	Precision	Recall	F1	NDCG	ISC	USC	MAP
Collaborative Filtering	0.2531	0.4875	0.3332	0.4466	0.0283	1.0000	0.4260
Item Based	0.1377	0.2661	0.1815	0.2217	0.0283	0.5018	0.2062
User Based	0.2531	0.4875	0.3332	0.4466	0.0277	0.3043	0.4260
k10	0.2531	0.4875	0.3332	0.4466	0.0272	0.2939	0.4260
Item Popularity	0.0604	0.2884	0.0999	0.2658	0.0156	1.0000	0.2560
Content Based	0.0470	0.2348	0.0783	0.2290	0.0266	1.0000	0.2271
Cosine	0.0470	0.2348	0.0783	0.2290	0.0266	1.0000	0.2271
Hybrid	0.2380	0.4611	0.3139	0.4239	0.0428	0.7853	0.4086
Content-based Item-based	0.0347	0.1474	0.0561	0.1009	0.0289	0.6095	0.0813
Content-based User-based	0.2380	0.4611	0.3139	0.4239	0.0428	0.7853	0.4086

Figure 20. Results View of the Droid Configurator.

3.5.1 Recommendation service

A generic recommendation service called DroidREST has been constructed to facilitate the use and integration of Droid with Eclipse and Non-eclipse based modelling tools. This REST service is implemented in Java employing Jersey[10] and Tomcat[11]. DroidREST computes recommendations using the configuration files generated by the *RS Synthesizer* (cf. Figure 13). All the necessary information from the already trained recommenders is stored in these files, such as which items to recommend based on a target and the context data. This way, RSs defined with Droid would not be required to be re-deployed.

To make a POST request, clients have to specify the name of an already deployed recommender and include a JSON file containing the target object of the recommendation and its context (i.e., the items that the target contains). The service will respond with a list of the items recommended for that particular case. Additionally, optional parameters can also be passed in the POST request. These include the maximum number of recommended items to retrieve, the threshold for the ranking value, and the type of item.

The REST service is implemented with four main classes: *DroidMain*, which manages all the requests from the clients; *DroidView*, which collects all the necessary information from the trained recommender; *ContextItem*, which extracts the target and its items within the modelling context from the JSON files; and *Generator*, which given a target – and considering its context and query parameters – generates the recommendations.

3.5.2 Client

The Eclipse Modelling Framework (EMF) is a framework and code generator facility to build Java applications based on the definition of simple models [SBP+08]. It is one of

[10] https://eclipse-ee4j.github.io/jersey/
[11] https://tomcat.apache.org/

the major modelling technologies today, and the de-facto implementation of the MOF OMG's standard[12].

In principle, the recommender services can be used with any modelling technology, but we provide out-of-the-box integration with EMF editors. In EMF, the synthesis of a default modelling editor can be automated via the Ecore meta-model of a modelling language. The Ecore meta-model describes the model and the runtime support system. Consequently, this editor enables the creation of instances of a meta-model using a tree view. Droid incorporates an out-of-the-box generation and integration of the Droid recommendation service into the default EMF tree editor of a modelling language. Employing a model-to-text template language called Java Emitter Template (JET) Droid offers automatic deployment in the default tree editor of EMF. It supports the definition and execution of code generation templates from EMF models. A predefined set of JET templates are provided by EMF allowing the generation of the Java code needed for the implementation of the editor for a given Ecore meta-model. To support the out-of-the-box generation and integration of the Droid recommendation service we overwritten those JET templates.

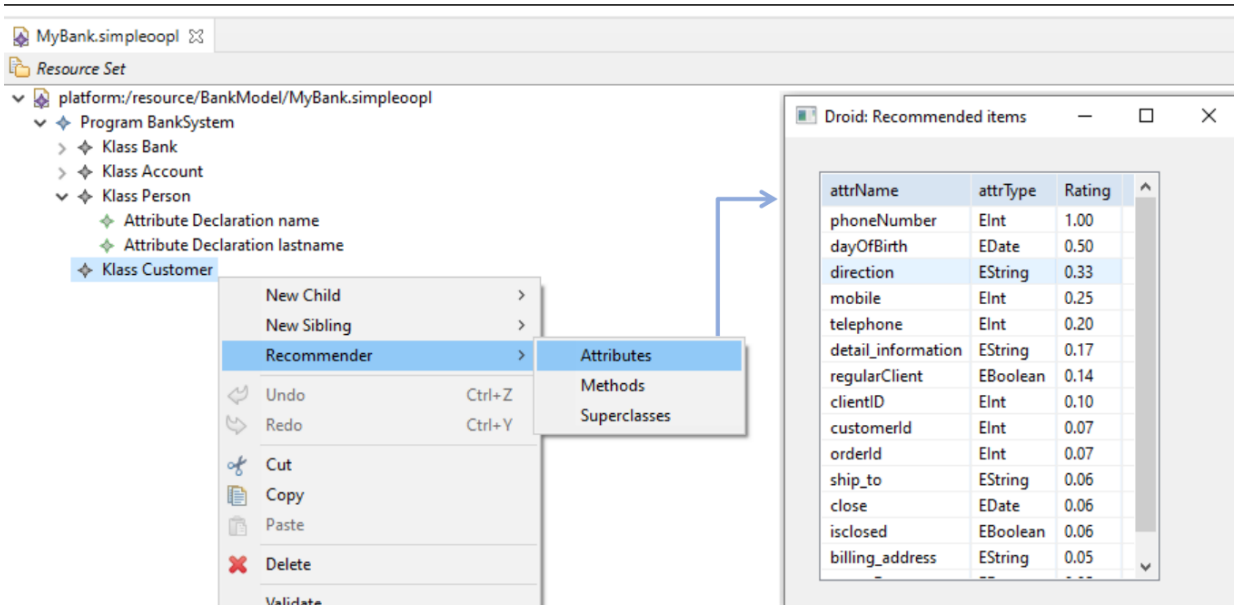


Figure 21. Results View of the Droid Configurator.

We included a "Recommender" pop-up menu (Figure 21) on the object target of recommendations. The menu displays a list of the types of objects that can be recommended followed by the list of items recommended for a particular item. A selection of a given item type triggers a request to the Droid service. This request includes the object, its context and the item type as parameters. The response is a list of recommendations displayed in a table ordered by their relevance. From this list, the

[12] https://www.omg.org/mof/

user can select the recommendations and apply them to the active model that is working on.

3.6 Experiments

We performed two different types of experiments. With the first one we wanted to study the usefulness of the recommendations provided by Droid RSs. For this purpose we performed an offline evaluation with UML class models. This experiment aimed to answer the following research questions

RQ1: *"How precise and complete are the recommendations provided by Droid recommenders?*"

RQ2: "*Can data preprocessing improve the recommendations provided by the previously developed Droid recommenders?*"

For the second experiment, we wanted to assess the feasibility of using Droid RSs outside Eclipse. With this aim, we present a case study that integrates a Droid RS with a modelling chatbot.

3.6.1 Offline experiment

We ran an offline experiment on two datasets from two different domains. The purpose was to analyse the performance of the RSs generated with Droid on distinct domains. The used datasets contain models extracted from MAR [HS20]. This is a structure-based search engine for models and meta-models, which can be queried via input keywords. In particular, we retrieved UML models, since they are the most numerous in MAR. As domains for our experiment, we chose Literature and Education. The keywords used to retrieve the models for the Literature domain were bibliography, book, author, journal and magazine. The keywords used for the Education domain were professor, teacher, student and alumn (as stem of other words like alumnus or alumni). The resulting datasets are available at https://github.com/Droiddsl/DroidConfigurator.

Table 1 shows, per each domain, the number of models, users (i.e., classes), items (i.e., attributes, methods and superclasses) and features (i.e., attributes describing users and items) in the datasets. The Literature and Education datasets have 1,447 and 1,051 UML models, respectively, conformant to the UML 2.0 class diagrams meta-model (cf. Figure 11). The table does not consider duplicate elements. Hence, if two models contain classes with the same name, they are considered to be the same class. This is more evident in the Education domain, which has more models than targets.

Table 1. Description of the datasets.

	Literature	Education
Num. models	1,447	1,051
Num. targets	1,740	905
Num. items	6,731	3,317
Num. features	6,497	3,231

We used Droid to configure the RSs for each domain, selecting all available recommendation methods with different parameters. Specifically, we used the Droid specification shown in Listing 1. We first executed without the pre-processing configuration and after that, we executed with the pre-processing configuration as we want to assess how much pre-processing techniques can improve the RS generated with Droid. We trained multiple RSs through a variety of collaborative, content-based and hybrid recommendation methods: item popularity (ItemPop), item-based collaborative filtering (IBCF), user-based collaborative filtering (UBCF), content-based with cosine similarity (CosineCB), content-based item-based (CBIB) and content-based user-based (CBUB). We parameterised the methods IBCF, UBCF, CBIB and CBUB with neighbourhood sizes k 10, 15, 20, 25, 50 and 100. In the listing, we show only k 5 and 10 for visualisation purposes. In the following, we refer to the methods that use neighbourhoods by concatenating the method name and the neighbourhood size k. For instance, IBCF50 refers to the IBCF k-NN method with 50 neighbours. In all cases, we used 10-fold cross-validation and a per-user technique to split the datasets. We analysed the performance of the RSs by means of the ranking quality metrics precision (p), recall (r), F1, MAP (Mean Average Precision) and nDCG (Normalized Discounted Cumulative Gain); and the coverage and diversity metrics USC (User Space Coverage) and ISC (Item Space Coverage). Additionally, in the experiment, we used a relevance threshold of 0.5, and cut-offs 5, 10, 15 and 20.

Table 2 shows the results of the experiment for each domain/dataset (Literature and Education). The rows show the selected recommendation methods, and the columns correspond to the metric values. For space constraints, the table omits the results of the recommendation methods IBCF and CBIB, as their performance is worse than that of UBCF and CBUB. We can observe that the order of magnitude of the metric values is the same in both domains. As studied in the RS field [BCC13], this magnitude depends on many factors, such as the dataset characteristics (e.g., the average number of preferences per user, or the rating sparsity, which is the proportion of existing ratings from the whole set of potential user-item preference relations), and the evaluation methodology (e.g., the method to split training and test data, or the test ratings for which the metrics are computed). In our experiment, we followed the Test Items methodology [BCC13] which, for a target user, evaluates recommendation lists that may contain test items from all users. This explains why the precision values are close to 0. For this reason, in general, the important aspect to consider is the relative difference of the metric values achieved by the different recommendation methods.

Table 2. Results of the experiment. The best values are shown in bold.

Method	Literature							Education						
	p	r	F1	MAP	nDCG	USC	ISC	p	r	F1	MAP	nDCG	USC	ISC
ItemPop	0.006	0.180	0.012	0.055	0.086	**1.000**	0.012	0.007	0.224	0.013	0.082	0.117	**1.000**	0.017
CosineCB	0.001	0.032	0.002	0.017	0.020	**1.000**	0.004	0.002	0.076	0.004	0.003	0.017	**1.000**	0.007
UBCF10	**0.033**	0.290	**0.060**	0.157	0.195	0.824	0.059	**0.035**	0.337	**0.064**	0.184	0.227	0.830	0.056
UBCF15	**0.026**	0.304	**0.048**	0.160	0.201	0.860	**0.060**	**0.026**	0.346	**0.048**	**0.184**	0.228	0.863	**0.057**
UBCF20	0.022	0.319	0.041	**0.161**	0.205	0.863	**0.060**	0.022	0.360	0.042	0.183	**0.232**	0.868	**0.057**
UBCF25	0.020	0.327	0.038	**0.163**	0.208	0.865	**0.060**	0.021	0.369	0.039	**0.184**	**0.235**	0.868	**0.057**
UBCF50	0.019	**0.348**	0.037	0.159	**0.211**	0.865	0.058	0.018	**0.383**	0.035	0.176	0.231	0.868	**0.057**
UBCF100	0.020	**0.360**	0.037	0.155	**0.210**	0.865	0.056	0.019	**0.387**	0.035	0.166	0.224	0.868	0.055
CBUB10	0.015	0.202	0.028	0.108	0.132	0.929	0.055	0.023	0.258	0.042	0.137	0.168	0.926	0.053
CBUB15	0.011	0.210	0.022	0.105	0.130	0.962	0.056	0.015	0.260	0.029	0.135	0.165	0.984	0.054
CBUB20	0.009	0.213	0.017	0.102	0.129	0.963	0.056	0.012	0.265	0.022	0.133	0.165	**1.000**	0.055
CBUB25	0.008	0.212	0.015	0.098	0.125	0.987	0.056	0.010	0.271	0.019	0.133	0.166	**1.000**	0.055
CBUB50	0.006	0.212	0.011	0.088	0.117	**1.000**	0.055	0.008	0.304	0.016	0.133	0.176	**1.000**	0.055
CBUB100	0.007	0.242	0.014	0.097	0.133	**1.000**	0.051	0.008	0.302	0.016	0.124	0.169	**1.000**	0.052

Analysing Table 2, a first conclusion is the fact that the content-based method CosineCB was the worst performing, being outperformed even by the ItemPop baseline. This is not surprising in our experiment. CosineCB estimates the preference of a target (class) for an item (attribute, method, or superclass) by means of the cosine of the angle between the target and item feature vectors. These feature vectors correspond to the names of the classes, attributes and methods in the models of the datasets. Since in this experiment we did not perform any text pre-processing on those names (e.g., to unify lowercase and uppercase, singular and plural, morphological deviations, misspellings, synonyms, ambiguities), there are different names that could have been considered the same, facilitating the cosine similarity. Moreover, we may have used finer-grained user and item profiles which capture the occurrence frequency of features.

By contrast, UBCF and CBUB were the best performing recommendation methods. The results of their item-based counterparts were worse, and are not reported in the table. UBCF with neighbourhoods of sizes 10 and 15 achieved the best F1 values in both domains. In terms of MAP and nDCG, which focus on the precision of the top positions in the recommendation lists, the best results were obtained with neighbourhoods of sizes 20 and 25 in the Education domain, and sizes 50 and 100 in the Literature domain. If we consider F1, MAP and nDCG all together, UBCF with neighbourhood size 15 seems the best choice for the available data and targeted task. This was expected, since CosineCB and ItemPop do not depend on user-item rating patterns, they have an USC of 1, which means that they are able to make recommendations for 100% of the users. In terms of ISC diversity, there is no significant difference between methods and domains, which reflects that both popular and unpopular items are recommended.

Answering RQ1. Our evaluation shows that standard recommendation methods are able to provide sensible recommendations for every class, starting from relatively small datasets that have not been pre-processed. These results are in-line with RSs specifically created for class diagrams [BCL+21]. Still, we have identified some aspects that would allow improving the generated recommendations, such as using

larger datasets, pre-processing the text features that the content-based methods exploit, or even incorporating more specific, task-oriented recommendation methods.

To anwer RQ2, we used the result of the best dataset without pre-processing (Education) and executed the experiment using the pre-processing configuration defined in Listing 1. The right of Table 3 compares the best RS configuration without pre-processing (user-based item-based k10), and the same configuration with pre-processing. F1 increased from 6.4% to 33.3%, precision from 3.5% to 25.3%, and recall from 33.7% to 48.8%. MAP and nDCG also improved about 50%. Instead, ISC and USC decreased, which was expected as there is a compromise between the precision-based and the diversity/coverage metrics.

Table 3. Results with pre-processing and without pre-processing

Dataset		Results		
Num. models	1051	**Metric**	**No preproc.**	**With preproc.**
Num. targets	983	Precision	0.035	0.253
Num. items	3488	Recall	0.337	0.488
Items per target	3.31	F1	0.064	0.333
Avg. model size	89	MAP	0.184	0.426
Min. model size	3	nDCG	0.227	0.447
Max. model size	658	USC	0.830	0.294
Sparsity	0.9982	ISC	0.056	0.027

Answering RQ2. We thus answer our question positively: pre-processing improves the precision-based metrics, maintaining a balance with diversity/coverage. While further experiments are needed to better characterise the preprocessing effects, the metrics show improvement w.r.t. existing hand-crafted RSs for class diagrams, like [BCL+21], which reports a precision around 4%, or MemoRec [dRdRdC22], with F1 scores around 17%.

3.6.2 Case study

For the second experiment, we developed a case study on the integration of a RS specified with Droid into a modelling chatbot called Socio. With this study, we aim to answer the following

RQ3: *How difficult is it to integrate a Droid-based RS with a non Eclipse-based modelling client?*

In addition, we wanted to experiment integrating a recommender within a natural-language interface.

Socio [PGdL18] is a chatbot or conversational agent that enables heterogeneous groups of domain and modelling experts to collaborate on modelling tasks. It works in social networks, like Telegram or Twitter, and facilitates the active participation of

domain experts with no technical background in building models (class diagrams) by using natural language (NL) as the modelling interface.

Figure 22(a) shows a user interaction with Socio in Telegram. The user can send messages expressing domain requirements in NL to the chatbot (labels 1 and 3). Socio interprets the messages and the current status of the model, infers the necessary modelling actions, updates the model, and sends back an image of the model with the modified elements in green (labels 2 and 4). For example, given the message "*School contains teachers and students*" (label 1), Socio infers that there must be three classes named School, Teacher and Student. Then, because of the contains verb, it infers that School should have two containment references with cardinality one to many (as teachers and students are plural), one called teachers and going to Teacher, and the other called students and going to Student. Since the model is empty at this moment, Socio creates all these elements (label 2).

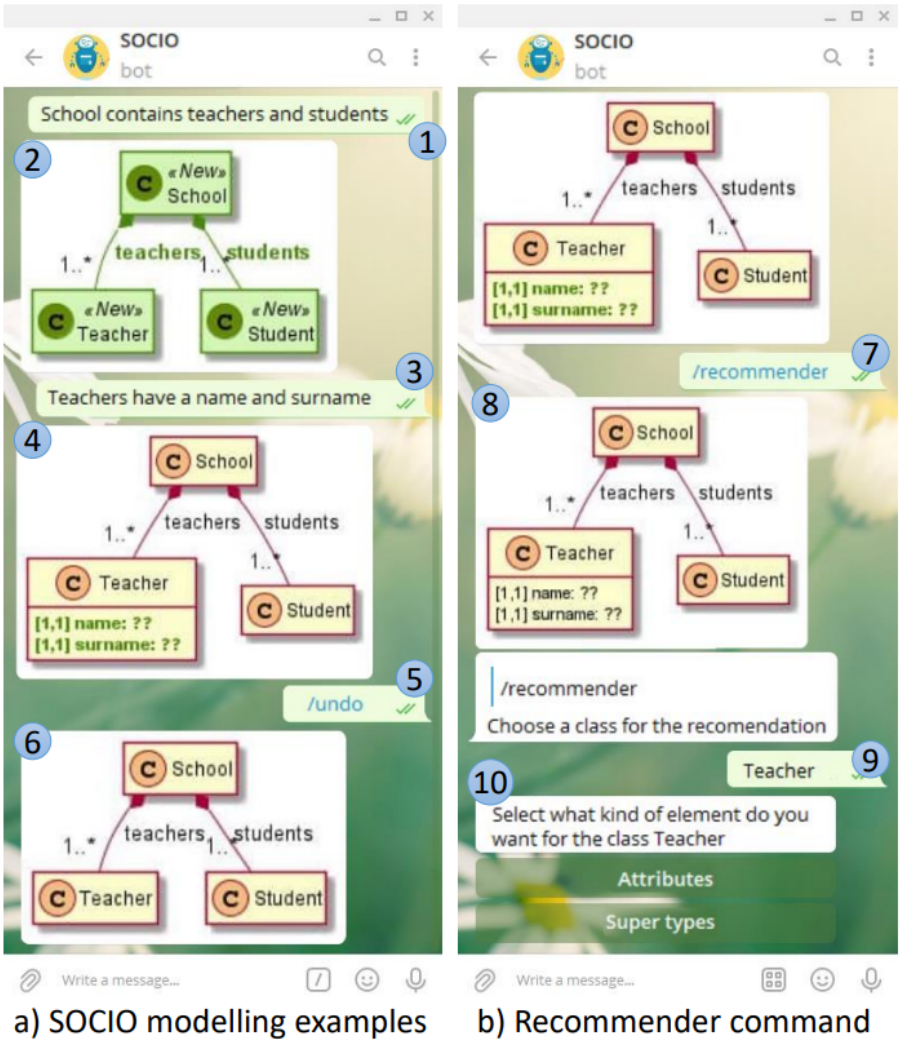


Figure 22. Interacting with Socio in Telegram.

Users normally do not provide all requirements in a single message, and so, Socio permits a model to be incomplete or incorrect. The interaction with label 3 illustrates this. When the user says "*Teachers have a name and surname*", Socio interprets that there must be a class named Teacher with two features, name and surname. Since the class already exists, it only adds the two features, but since there is no information about their types, their definition is incomplete (label 4). Besides model creation via NL processing, the chatbot has commands to manage, validate, download the model, or undo and redo the modelling actions. In Telegram, these commands start by a backslash followed by a keyword. Labels 5 and 6 in Figure 22(a) show an example of the undo command.

For this case study, we extended Socio with a RS specified with Droid and hence available as a service. Figure 23 illustrates the recommendations provided by Droid. It shows the recommended supertypes (label 1) and attributes (label 2) for the class Teacher. When the user presses the button with the recommendation Person, Socio creates a new class because it does not exist, and adds it as a supertype of Teacher. When the user presses the button with the recommendation name, Socio detects that Teacher already defines this attribute and only updates its type. This way, recommendations not only add new elements to the model, but sometimes also allow fixing incomplete elements.

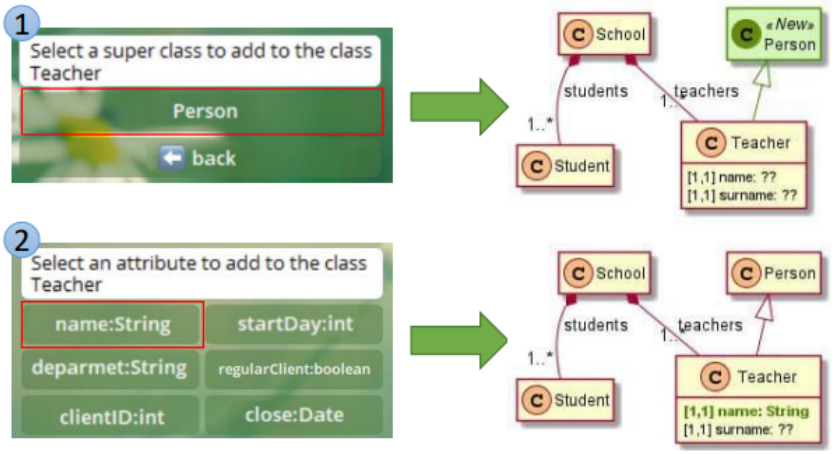


Figure 23. Droid recommendations in Socio.

Table 4 shows the LOC and number of Java classes developed to achieve the RS integration. The Interactive message handler is the largest component, which is normal as it handles several user interactions. We can observe that the integration did not require many changes in the Socio architecture, and the new components are not large.

Table 4. Metrics for integrating Droid with Socio.

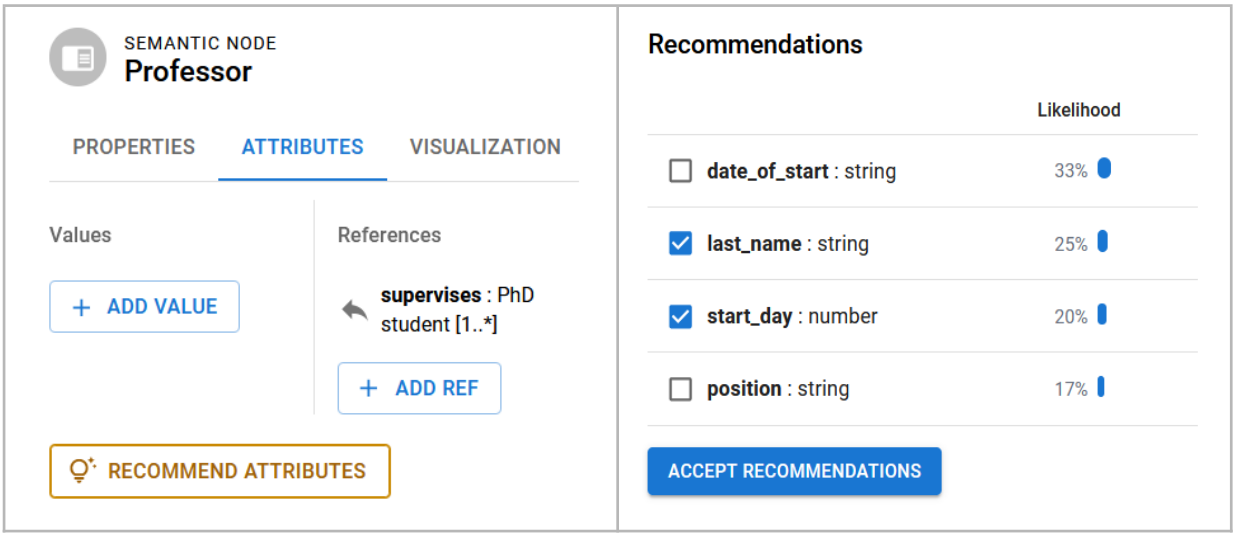
		LOC	Num. Classes
Back	Recommender handler	160	2
	Transformer	44	1
Front	Recommender command	128	2
	Interac. message handler	400	1
	Total	732	6

Answering RQ3. This case study proves that Droid-based RSs can be easily integrated with tools outside Eclipse. While the integration with Socio has not many LOC, we added code on both its front-end and its back-end. Moreover, more than 50% of the code was dedicated to the user interaction. These two circumstances can make a big difference in the effort required to integrate the RS with other modelling tools.

4. Adding Recommendation Support to Low-code Editors

One of the central activities when building an LCDP or an MDE solution is creating meta-models (i.e., models to be instantiated). Dandelion permits it via (meta-)model manipulation on a browser. This process can be eased by introducing a recommender system, such as the ones generated by Droid, as seen in the previous section. This section reports on the integration of a recommender system generated with Droid into Dandelion's frontend.

Dandelion has been extended with a *recommend attributes* button under the *attributes* section of a selected metaclass, which the citizen developer can click to ask for recommendations (Figure 24 a). Internally, a request is sent to DroidREST, the REST API exposed by Droid, with the name of the selected element and its already defined attributes.[13] The response is a list of new recommended attributes sorted in descending likelihood value, which is presented to the user as a selectable list (Figure 24 b).



a) Recommend attributes button **b)** List of recommendations

Figure 24. Suggested recommendations for a class *Professor*

Once the developer accepts the desired recommendations, they are added to the selected metaclass respecting the suggested types. Depending on the attribute's name, the multiplicity is set to 0..1 if the noun is singular, or 0..* otherwise.

In the future, we want to also provide recommendations for enumerations and multiplicities. To do so, Droid has to be trained on these features and provide recommendations accordingly. In addition, we envision global recommendations on entire meta-models, especially useful for large, unwieldy meta-models.

[13] The REST API is called with the parameter *newMaxRec* set to 7, which limits the number of recommended attributes to ease their visualisation.

5. Summary, Conclusions and Further Developments

In this document, we have described the design and realisation 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. For the recommender, we have proposed a DSL to configure the recommender system to arbitrary DSLs defined by a meta-model. We have performed evaluations showing good results both on the usefulness of the recommendations and the ease of integration with modelling tools.

Even though the work described in this document is fully functional, as any research work, it can be subject to improvement. For example, Dandelion can be increased with other scalability models (e.g., pagination defined on relations), and a full realisation of the sensemaking strategies. An application of the approach within the context of UGROUND's models and technologies is also expected in the short term. Regarding Droid, the aim is at providing out-of-the-box integration with Xtext and Sirius editors, and to experiment with other types of recommenders, like those based on pre-trained language models [WSS22].

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.

[ACG+22a] Lissette Almonte, Iván Cantador, Esther Guerra, and Juan de Lara. 2022. Recommender systems in model-driven engineering. Software and Systems Modelling (SoSyM), 21, 249–280.

[ACG+22b] Lissette Almonte, Iván Cantador, Esther Guerra, and Juan de Lara. 2022. Building recommender systems for modelling languages with Droid. In Proceedings of 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022). ACM, New York, NY, USA.

[AKS18] Henning Agt-Rickauer, Ralf-Detlef Kutsche, and Harald Sack. 2018. DoMoRe - A recommender system for domain modeling. In 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD). SciTePress, 71–82.

[APC+21] Lissette Almonte, Sara Pérez-Soler, Iván Cantador, Esther Guerra, and Juan de Lara. 2021. Automating the synthesis of recommender systems for modelling languages. In Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2021). ACM, New York, NY, USA, 22–35.

[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.

[BCC13] Alejandro Bellogín, Iván Cantador, and Pablo Castells. 2013. A comparative study of heterogeneous item recommendations in social systems. Information Sciences 221 (2013), 142–169.

[BCL+21] Loli Burgueño, Robert Clarisó, Shuai Li, Sébastien Gérard, and Jordi Cabot. 2021. An NLP-based architecture for the autocompletion of partial domain models. In CAiSE (LNCS, Vol. 12751'). Springer International Publishing, 91–106.

[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/

[BK13] Barmpis, K., & Kolovos, D. S. (2013). Hawk: towards a scalable model indexing architecture. In Proceedings of the workshop on scalability in model driven engineering (p. 6). ACM.

[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).

[BRH20] Angela Barriga, Adrian Rutle, and Rogardt Heldal. 2020. Improving model repair through experience sharing. Journal of Object Technology 19, 2 (2020), 13:1–21.

[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.

[CRB16] Thaciana Cerqueira, Franklin Ramalho, and Leandro Balby Marinho. 2016. A content-based approach for recommending UML sequence diagrams. In 28th International Conference on Software Engineering and Knowledge Engineering (SEKE). KSI Research Inc. and Knowledge Systems Institute Graduate School, 644–649.

[D16] Daniel, G. (2016). Efficient Persistence and Query Techniques for Very Large Models. Proceedings of the ACM Student Research Competition at MODELS 2016, 1775. CEUR-WS.org.

[DGL14] Andrej Dyck, Andreas Ganser, and Horst Lichter. 2014. A framework for model recommenders - Requirements, architecture and tool support. In 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD). SciTePress, 282–290.

[DSC19] Daniel, G., Sunyé, G., & Cabot, J. (2019). Advanced prefetching and caching of models with PrefetchML. *Softw. Syst. Model.*, *18*(3), 1773–1794.

[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.

[dRdRdC22] J. Di Rocco, D. Di Ruscio, D., Di Sipio, C. et al. 2022. MemoRec: a recommender system for assisting modelers in specifying metamodels. SoSyM (2022), in press.

[dCdRN20] Claudio Di Sipio, Davide Di Ruscio, and Phuong T. Nguyen. 2020. Democratizing the development of recommender systems by means of low-code platforms. In MODELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Esther Guerra and Ludovico Iovino (Eds.). ACM, 68:1–68:9

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

[DSB+17] Daniel, G., Sunyé, G., Benelallam, A., Tisi, M., Vernageau, Y., Gómez, A., & Cabot, J. (2017). NeoEMF: A multi-database model persistence framework for very large models. *Sci. Comput. Program.*, *149*, 9–14.

[DWL+17] ShuiGuang Deng, Dongjing Wang, Ying Li, Bin Cao, Jianwei Yin, Zhaohui Wu, and Mengchu Zhou. 2017. A recommendation system to facilitate business process modeling. IEEE Transactions on Cybernetics 47, 6 (2017), 1380–1394.

[EC20] EMF Cloud. https://www.eclipse.org/emfcloud/, (last accessed in Sep. 2022).

[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.

[ECK15] Mohammad Ehson Rangiha, Marco Comuzzi, and Bill Karakostas. 2015. Role and task recommendation and social tagging to enable social business process management. In BPMDS/EMMSAD@CAiSE (Lecture Notes in Business Information Processing, Vol. 214). Springer, 68–82.

[ECM11] Espinazo-Pagán, J., Cuadrado, J. S., & Molina, J. G. (2011). Morsa: A Scalable Approach for Persisting and Accessing Large Models. *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS*, *6981*, 77–92. Springer.

[EGB16] Akil Elkamel, Mariem Gzara, and Hanêne Ben-Abdallah. 2016. An UML class recommender system for software design. In 13th IEEE/ACS International Conference of Computer Systems and Applications (AICCSA). IEEE Computer Society, 1–8.

[EGZ+13] Roberto Espinosa, Diego García-Saiz, Marta E. Zorrilla, José Jacobo Zubcoff, and Jose-Norberto Mazón. 2013. Development of a knowledge base for enabling non-expert users to apply data mining algorithms, In SIMPDA. CEUR Workshop Proceedings 1027, 46–61.

[EGZ+19] Roberto Espinosa, Diego García-Saiz, Marta E. Zorrilla, José Jacobo Zubcoff, and Jose-Norberto Mazón. 2019. S3Mining: A model-driven engineering approach for supporting novice data miners in selecting suitable classifiers. Computer Standards and Interfaces 65 (2019), 143–158.

[EMF20] Eclipse Modelling Framework. https://www.eclipse.org/modeling/emf/, (last accessed in Sep. 2022).

[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.

[FdRM+18] Franzago, M., di Ruscio, D., Malavolta, I., & Muccini, H. (2018). Collaborative Model-Driven Software Engineering: A Classification Framework and a Research Map. *IEEE Trans. Software Eng.*, *44*(12), 1146–1175.

[FMJ+18] Michael Fellmann, Dirk Metzger, Sven Jannaber, Novica Zarvic, and Oliver Thomas. 2018. Process modeling recommender systems - A generic data model and its application to a smart glasses-based modeling environment. Bus. Inf. Syst. Eng. 60, 1 (2018), 21–38.

[G04] Paulo Gomes. 2004. Software design retrieval using Bayesian networks and WordNet. In 7th European Conf. on Advances in Case-Based Reasoning (ECCBR) (Lecture Notes in Computer Science, Vol. 3155). Springer, 184–197.

[G12] H. Garbe. 2012. Intelligent assistance in a problem solving environment for UML class diagrams by combining a generative system with constraints. In eLearning. IADIS, 412–416.

[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 Sep. 2022).

[GGdL+19] Garmendia, A., Guerra, E., de Lara, J., García-Domínguez, A., & Kolovos, D. S. (2019). Scaling-up domain-specific modelling languages through modularity services. *Inf. Softw. Technol.*, *115*, 97–118.

[GHJ+94] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design patterns: Elements of reusable object-oriented software. Addison-Wesley Professional, 1st edition.

[GMF20] GMF, https://www.eclipse.org/gmf-tooling/, (last accessed in Sep. 2022).

[Gra20] Graphiti, https://www.eclipse.org/graphiti/, (last accessed in Sep. 2022).

[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.

[HS20] José Antonio Hernández López and Jesús Sánchez Cuadrado. 2020. MAR: a structure-based search engine for models. In MoDELS '20. ACM, 57–67.

[IBR+20] Ludovico Iovino, Angela Barriga, Adrian Rutle, and Rogardt Heldal. 2020. Model repair with quality-based reinforcement learning. Journal of Object Technology 19, 2 (2020), 17:1–21.

[JBD21] Jahed, K., Bagherzadeh, M., & Dingel, J. (2021). On the benefits of file-level modularity for EMF models. *Softw. Syst. Model.*, *20*(1), 267–286.

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

[K17] Stefan Kögel. 2017. Recommender system for model driven software development. In 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE). ACM, 1026–102.

[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.

[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).

[KHM20] Hadjer Khider, Slimane Hammoudi, and Abdelkrim Meziane. 2020. Business process model recommendation as a transformation process in MDE: Conceptualization and first experiments. In 8th International Conference on Model-Driven Engineering and Software Development (MODELSWARD). SciTePress, 65–75.

[KHO11] Agnes Koschmider, Thomas Hornung, and Andreas Oberweis. 2011. Recommendation-based editor for business process modeling. Data & Knowledge Engineering 70, 6 (2011), 483–503.

[KM17] Tobias Kuschke and Patrick Mäder. 2017. RapMOD - in situ autocompletion for graphical models: poster. In 39th International Conference on Software Engineering (ICSE), Companion Volume. IEEE Computer Society, 303–304.

[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.

[LA22] Arne Lange, Colin Atkinson. Multi-level modeling with LML A Contribution to the Multi-Level Process Challenge. Enterp. Model. Inf. Syst. Archit. Int. J. Concept. Model. 17 (2022).

[LCX+14] Ying Li, Bin Cao, Lida Xu, Jianwei Yin, ShuiGuang Deng, Yuyu Yin, and Zhaohui Wu. 2014. An efficient recommendation method for improving business process modeling. IEEE Transactions on Industrial Informatics 10, 1 (2014), 502–513.

[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.

[M09] Moody, D. L. (2009). The physics of notations: Toward a scientific basis for constructing visual notations in software engineering. IEEE Trans. Software Eng., 35(6), 756–779.

[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.

[MCK+20] Gunter Mussbacher, Benoît Combemale, Jörg Kienzle, Silvia Abrahão, Hyacinth Ali, Nelly Bencomo, Márton Búr, Loli Burgueño, Gregor Engels, Pierre Jeanjean, Jean-Marc Jézéquel, Thomas Kühn, Sébastien Mosser, Houari A. Sahraoui, Eugene Syriani, Dániel Varró, and Martin Weyssow. 2020. Opportunities in intelligent modeling assistance. Softw. Syst. Model. 19, 5 (2020), 1045–1053.

[MdLN+18] Ángel Mora Segura, Juan de Lara, Patrick Neubauer, and Manuel Wimmer. 2018. Automated modelling assistance by integrating heterogeneous information sources. Computer Languages, Systems and Structures 53 (2018), 90–120.

[MKG15] Ma, Q., Kelsen, P., & Glodt, C. (2015). A generic model decomposition technique and its application to the Eclipse modeling framework. *Softw. Syst. Model.*, *14*(2), 921–952.

[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.

[S19] Matthew Stephan. 2019. Towards a cognizant virtual software modeling assistant using model clones. In 41st International Conference on Software Engineering: New Ideas and Emerging Results (NIER@ICSE). IEEE / ACM, 21–24.

[S-L19] Maxime Savary-Leblanc. 2019. Improving MBSE tools UX with AIempowered software assistants. In 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS), Companion Volume. IEEE, 648–652.

[OPK+18] Manuel Ohrndorf, Christopher Pietsch, Udo Kelter, and Timo Kehrer. 2018. ReVision: a tool for history-based model repair recommendations. In 40th International Conference on Software Engineering (ICSE), Companion Proceeedings. ACM, 105–108.

[PAK+15] Pienta, R., Abello, J., Kahng, M., & Chau, D. H. (2015). Scalable graph exploration and visualization: Sensemaking challenges and opportunities. *2015 International Conference on Big Data and Smart Computing (BIGCOMP)*, 271–278.

[PGdL18] Sara Pérez-Soler, Esther Guerra, and Juan de Lara. 2018. Collaborative modeling and group decision making using chatbots in social networks. IEEE Softw. 35, 6 (2018), 48–54.

[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.

[RFS09] Gonzalo Rojas, Francisco Dominguez, and Stefano Salvatori. 2009. Recommender systems on the Web: A model-driven approach. In ECommerce and Web Technologies, Tommaso Di Noia and Francesco Buccafurri (Eds.). Springer Berlin Heidelberg, 252–263.

[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.

[RS21] RankSys. RankSys/RankSys: Java 8 Recommender Systems framework for novelty, diversity and much more (github.com), (last accessed in Feb. 2021).

[RSV20] Erika Rizzo Aquino, Pierre de Saqui-Sannes, and Rob A. Vingerhoeds. 2020. A methodological assistant for use case diagrams. In 8th International Conference on Model-Driven Engineering and Software Development (MODELSWARD). SciTePress, 227–236.

[RU13] Gonzalo Rojas and Claudio Uribe. 2013. A conceptual framework to develop mobile recommender systems of points of interest. In SCCC. IEEE Computer Society, 16–20.

[S17] F. F. Shahare. 2017. Sentiment analysis for the news data based on the social media. International Conference on Intelligent Computing and Control Systems (ICICCS), 1365–1370.

[SB14a] A. Said and A. Bellogín. Comparative recommender system evaluation:Benchmarking recommendation frameworks. In Proceedings of the 8th ACM Conference on Recommender Systems, RecSys ’14, page 129–136, New York, NY, USA, 2014. Association for Computing Machinery.

[SB14b] A. Said and A. Bellogín. Rival: a toolkit to foster reproducibility in recommender system evaluation. In Eighth ACM Conference on Recommender Systems, RecSys ’14, pages 371–372. ACM, 2014. See also https://github.com/recommenders/rival.

[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 Sep. 2022).

[Spr20] Eclipse Sprotty, https://www.eclipse.org/sprotty/, (last accessed in Sep. 2022).

[SSF19] Sousa, V., Syriani, E., & Fall, K. (2019). Operationalizing the Integration of User Interaction Specifications in the Synthesis of Modeling Editors. *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering*, 42–54. Athens, Greece: ACM.

[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 Sep. 2022).

[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.

[WKG+16] Wei, R., Kolovos, D. S., García-Domínguez, A., Barmpis, K., & Paige, R. F. (2016). Partial loading of XMI models. *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, 329–339. ACM.

[WSS22] Martin Weyssow, Houari A. Sahraoui, Eugene Syriani. Recommending metamodel concepts during modeling activities with pre-trained language models. Softw. Syst. Model. 21(3): 1071-1089 (2022).